Arrays

Dr. Megha Ummat

Arrays: Abstract Data Type

- Each instance of the data object array is a set of pairs of the form (index, value).
- No two pairs in this set have the same index.
- The functions performed on the array are as follows:
  - **Sets an Element** - Adds a pair of the form (index, value) to the set, and if a pair with the same index already exists, deletes the old pair.
  - **Gets an Element** – Retrieves the value of the pair that has a given index.

Arrays: Abstract Data Type

AbstractDataType *array*
{
instances
set of (index, value) pairs, no two pairs have the same index

operations
$get(index)$: return the value of the pair with this index

$set(index, value)$: add this pair to set of pairs, overwrite existing pair (if any) with the same index
}

Example

- The high temperature (in degrees Farenheit) for each day of last week may e represented by the following array:

High ={(Sunday, 82), (Monday,79), (Tuesday,45), (Wednesday, 92), (Thursday, 88), (Friday, 89), (Saturday, 91)}

- We can change the high temperature recorded for Monday to 83 by performing the operation **set(Monday,83)**.
- We can determine the high temperature of Friday by performing the operation **get(Friday)**.
- An alternative way to represent the daily high temperature is High ={(0, 82), (1,79), (2,45), (3, 92), (4, 88), (5, 89), (6, 91)}

Indexing a C++ Array

- The index of an array in C++ must be of the form $[i_1][i_2][i_3]\ldots[i_k]$, where each i_j is a non-negative integer.
- A 3-dimensional array score, whose values are of type integer, can be created as int score $[u_1]$ $[u_2]$ $[u_3]$, where u_i's are positive constants or positive expressions derived from constants.
- Index i_j should be in the range $0 \le i_j \le u_j$ and $1 \le j \le k$.
- The array can hold a maximum of $\boldsymbol{n = u_1.\ u_2.\ u_3}$ values
- The memory size of score is **n * sizeof(int) bytes**. This memory begins at byte **start** (say) and extends up to and including byte **start + sizeof(score)-1**.

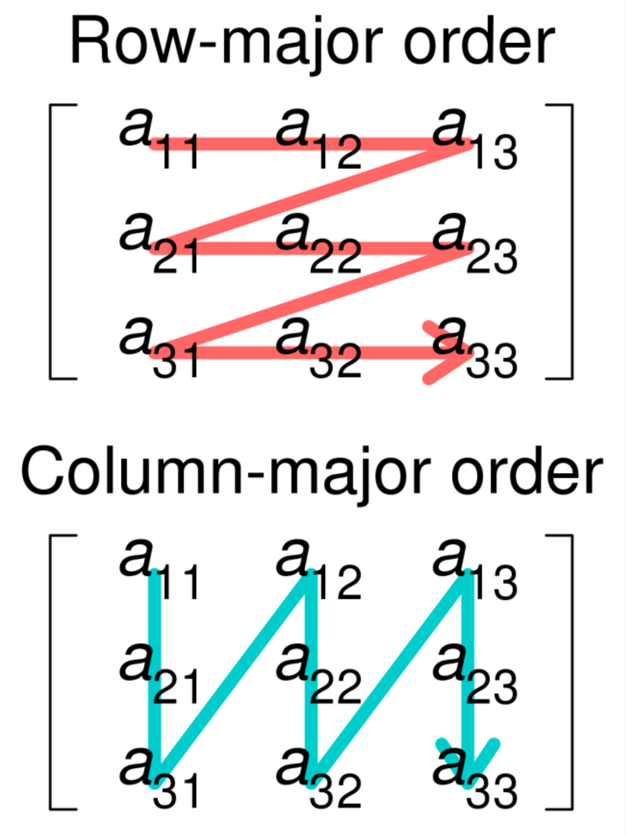
Row Major and Column Major Mappings

- Let n be the number of elements in a k-dimensional array.
- The serialization of the array is done using a mapping function, which maps the array index $[i_1][i_2[i_3]...[i_k]$ into a number map $(i_1,i_2,i_3...i_k)$, in the range [0 ,n-1] such that array element with index $[i_1][i_2[i_3]...[i_k]$ is mapped to position map $(i_1,i_2,i_3...i_k)$ in the serial order.
- When the number of dimensions is 1 (i.e. k=1), the function

$$map(i_1) = i_1$$

Row Major and Column Major Mappings

- In a 2D array there are two possible ways to store the array:
  - Row Major
  - Column Major



Row Major Order: Locating element A[i][j]

- **Location** of A[I][J] = Base Address (A) + W * [N (I-L_r) + (J-L_c)]
- Where **Base Address** is the address of the first element in an array.

- **W**= Weight (size) of a data type.
- **N=** Total No of Columns.
- **I=** Row Number
- **J=** Column Number of an element whose address is to find out.
- **L_r** = Lower limit of row/ if not given assume 0
- **L_c** = Lower limit of column/ if not given assume 0

Column Major Order: Locating element A[i][j]

– **Location** of A[I][J] = Base Address (A) + W * [M (J-L_c) + (I-L_r)]
– Where **Base Address** is the address of the first element in an array.
- **W**= Weight (size) of a data type.
- **M=** Total No of Rows.
- **I=** Row Number
- **J=** Column Number of an element whose address is to find out.
- **L_r** = Lower limit of row/ if not given assume 0
- **L_c** = Lower limit of column/ if not given assume 0
- **Number of Rows = (U_r – L_r) + 1**
- **Number of Columns = (U_c-L_c) + 1**

Mapping Functions

– When row major order is used, the mapping function is:

$$map(i_1, i_2) = i_1u_2 + i_2$$

where u_2 is the number of columns in the array.

- Note that by the time index $[i_1][i_2]$ is numbered in the row major scheme, i_1u_2 elements from the rows 0, …, i_1-1 as well as i_2 elements from row i_1 have been numbered.

Mapping Functions

– A sample array of 3 X 6 is shown in figure. Since the number of columns in 6, u_2 becomes

$$map(i_1, i_2) = 6i_1 + i_2$$

So map(1,3) = 6 + 3 = 9 and map(2,5) = 6 * 2 + 5 = 17

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

(a) Row-major mapping

0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17

(b) Column-major mapping

Mapping in a 3D Array

- The discussed scheme can be extended to obtain mapping functions for arrays with more than 2 dimensions.
- In row major order, we first list all indexes with the first coordinate equal to 0, then those with this coordinate equal to 1 and so on.
- For instance, the indexes of score [3] [2] [4] in row major order are

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

Mapping in a 3D Array

- The mapping function for 3D array is

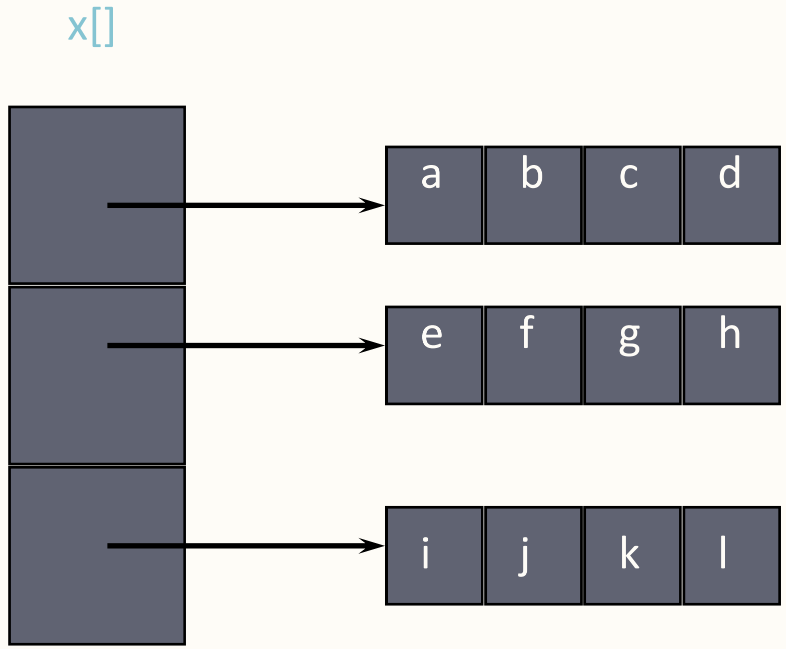
$$map(i_1, i_2, i_3) = i_1u_2u_3 + i_2u_3 + i_3$$

- Note that the elements with the first coordinate i_1 are preceded by all the elements whose first coordinate is less than i_1. There are u_2u_3 elements that have the same first coordinate. So there are $i_1u_2u_3$ elements with the first coordinate less than i_1.
- The number of elements with the first coordinate equal to i_1 and the second coordinate less that i_2 is i_2u_3, and the number with the first coordinate equal to i_1, the second equal to i_2, and the third less than i_3 is i_3.

Array of Arrays Representation

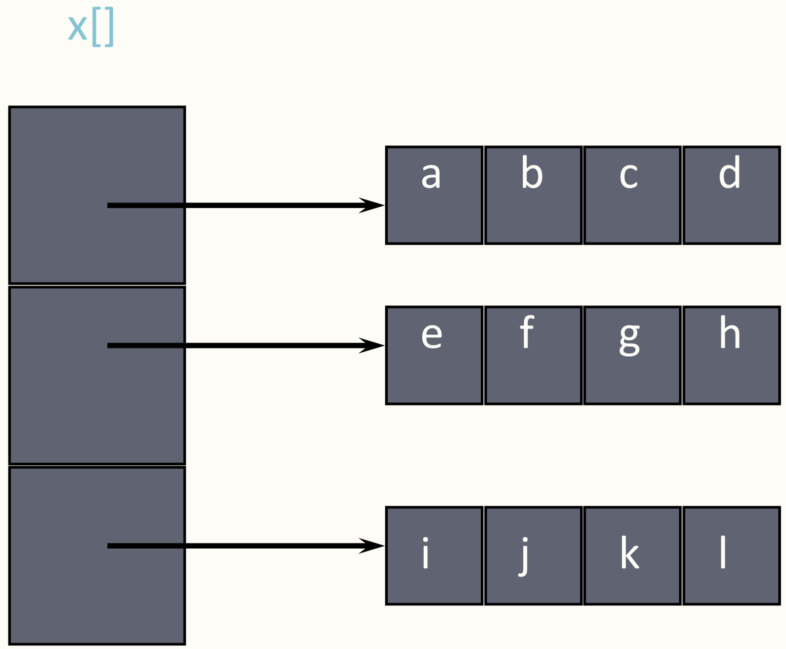
C++ uses array of arrays representation to represent a multidimensional array.

In this representation, a two-dimensional array is represented as a one-dimensional array in which each element is, itself, a one-dimensional array.



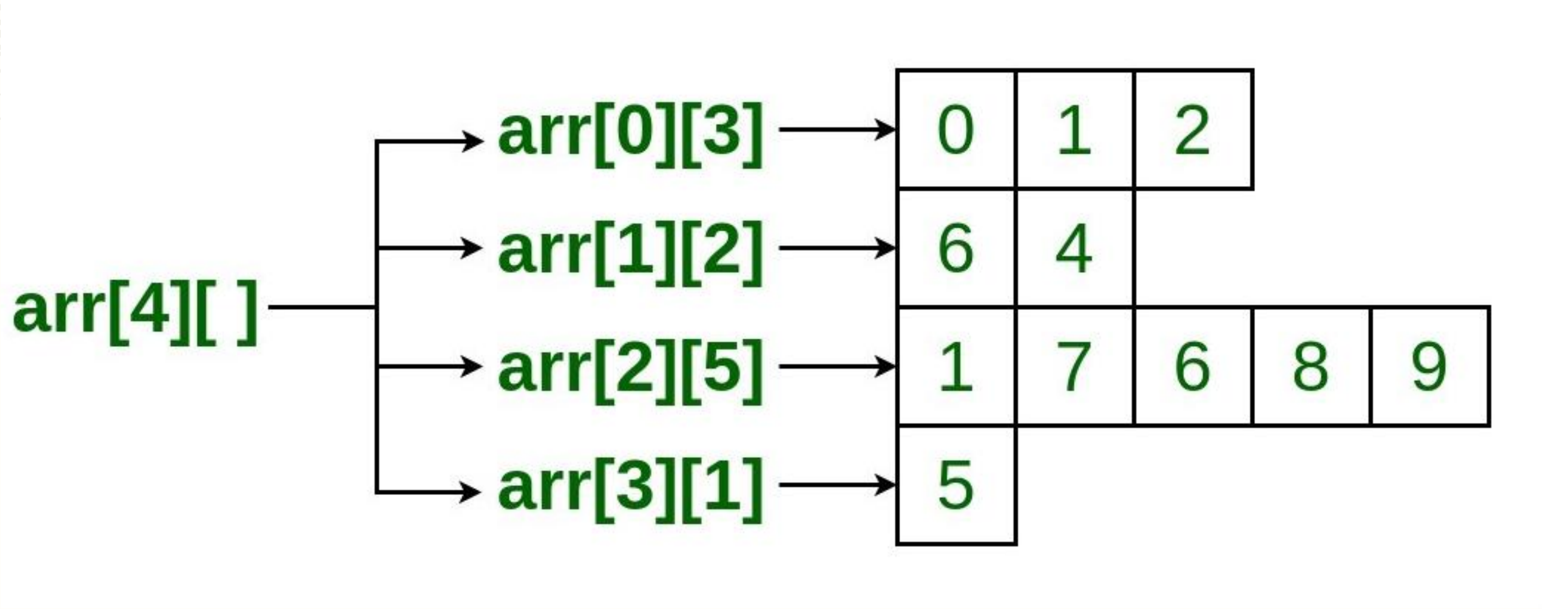
Array of Arrays Representation

- This representation is called the array-of-arrays representation.
- Requires contiguous memory of size 3, 4, 4, and 4 for the 4 1D arrays.
- 1 memory block of size **number of rows** and **number of rows blocks** of **size number of columns.**

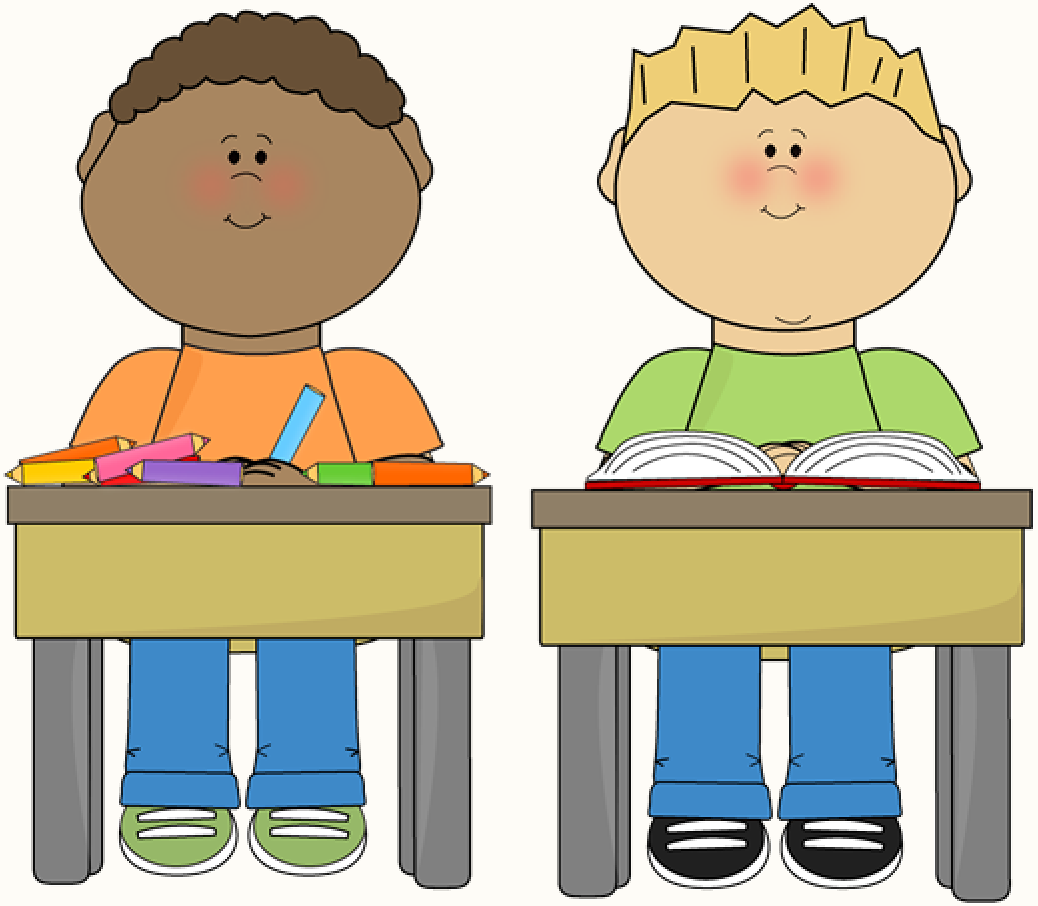


Irregular Two Dimensional Array

- When two or more rows have different number of columns.



Section 7.2.2 – The Class Matrix
Self Study



Special Matrices

– A square matrix has the same number of rows and columns. Some special forms of square matrices are:

- **Diagonal**: M is diagonal iff $M(i,j) = 0$ for $i \neq j$
- **Tridiagonal**: $M(i, j) = 0$ for $|i-j| > 1$
- **Lower Triangular**: $M(i,j) = 0$ for $i < j$
- **Upper Triangular**: $M(i,j) = 0$ for $i > j$
- **Symmetric**: $M(i,j) = M(j,i)$ for all i and j

Special Matrices

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 6 \end{matrix}$$

(a) Diagonal

$$\begin{matrix} 2 & 1 & 0 & 0 \\ 3 & 1 & 3 & 0 \\ 0 & 5 & 2 & 7 \\ 0 & 0 & 9 & 0 \end{matrix}$$

(b) Tridiagonal

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 4 & 2 & 7 & 0 \end{matrix}$$

(c) Lower triangular

$$\begin{matrix} 2 & 1 & 3 & 0 \\ 0 & 1 & 3 & 8 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 \end{matrix}$$

(d) Upper triangular

$$\begin{matrix} 2 & 4 & 6 & 0 \\ 4 & 1 & 9 & 5 \\ 6 & 9 & 4 & 7 \\ 0 & 5 & 7 & 0 \end{matrix}$$

(e) Symmetric

Diagonal Matrices

```
template<class T>
class diagonalMatrix
{
   public:
      diagonalMatrix(int theN = 10);
      ~diagonalMatrix() {delete [] element;}
      T get(int, int) const;
      void set(int, int, const T&);
   private:
      int n;        // matrix dimension
      T *element;   // 1D array for diagonal elements
};

template<class T>
diagonalMatrix<T>::diagonalMatrix(int theN)
{// Constructor.
   // validate theN
   if (theN < 1)
       throw illegalParameterValue("Matrix size must be > 0");

   n = theN;
   element = new T [n];
}
```

Add input() and display()

Diagonal Matrices get Method

```
template <class T>
T diagonalMatrix<T>::get(int i, int j) const
{// Return (i,j)th element of matrix.
   // validate i and j
   if (i < 1 || j < 1 || i > n || j > n)
       throw matrixIndexOutOfBounds();

   if (i == j)
      return element[i-1];   // diagonal element
   else
      return 0;              // nondiagonal element
}
```

Diagonal Matrices set Method

```
template<class T>
void diagonalMatrix<T>::set(int i, int j, const T& newValue)
{// Store newValue as (i,j)th element.
   // validate i and j
   if (i < 1 || j < 1 || i > n || j > n)
       throw matrixIndexOutOfBounds();

   if (i == j)
     // save the diagonal value
      element[i-1] = newValue;
   else
      // nondiagonal value, newValue must be zero
      if (newValue != 0)
         throw illegalParameterValue
               ("nondiagonal elements must be zero");
}
```

Diagonal Matrices input Method

```
template<class T>
void DiagonalMatrix<T>::input()
{
	T val;
	cout<<"Enter the elements of the Diagonal metric array:\n";
	for(int i=1;i<=n;i++)
	{
		for(int j=1;j<=n; j++)
		{
			cin>>val;
			set(i,j, val);
		}
	}
}
```

Diagonal Matrices display Method

```
template<class T>
void DiagonalMatrix<T>::display()
{
    cout<<"\nElements of the Diagonal metric array are:\n";
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=n; j++)
        {
            T val=get(i,j);
            cout<<val<<"\t";
        }
        cout<<"\n";
    }
}
```

TriDiagonal Matrices

- In an n X n tridiagonal matrix T, the nonzero elements lie on one of the 3 diagonals
  - **Main Diagonal** → *i = j*
  - **Diagonal below main diagonal** → *i = j + 1*
  - **Diagonal above main diagonal** → *i = j - 1*
- The total number of elements on these 3 diagonals is **3n-2**.
- Only the elements on the 3 diagonals are explicitly stored in one-dimensional array with 3n-2 positions.

TriDiagonal Matrices

- Elements can be mapped into one dimensional array
  - **By rows** = [2, 1, 3, 1, 3, 5, 2, 7, 9, 0]
  - **By columns** = [2, 3, 1, 1, 5, 3, 2, 9, 7, 0]
  - **By Diagonals** = [3, 5, 9, 2, 1, 2, 0 , 1, 3, 7]

$$\begin{matrix} 2 & 1 & 0 & 0 \\ 3 & 1 & 3 & 0 \\ 0 & 5 & 2 & 7 \\ 0 & 0 & 9 & 0 \end{matrix}$$

TriDiagonal Matrices get()

```
template <class T>
T tridiagonalMatrix<T>::get(int i, int j) const
{// Return (i,j)th element of matrix.

   // validate i and j
   if ( i < 1 || j < 1 || i > n || j > n)
       throw matrixIndexOutOfBounds();

   // determine lement to return
   switch (i - j)
   {
      case 1: // lower diagonal
              return element[i - 2];
      case 0: // main diagonal
              return element[n + i - 2];
      case -1: // upper diagonal
              return element[2 * n + i - 2];
      default: return 0;
   }
}
```

Triangular Matrices

- In an n-row lower triangular matrix, the nonzero region has one element in row 1, two in row 2,…., and n in row n.
- In an n-row upper triangular matrix, the nonzero region has n elements in row 1, n-1 in row 2 ,… and one in row n.
- In both the cases the total number of elements in the nonzero region is

$$\sum_{i=1}^{n} i = n(n+1)/2$$

- Both kind of triangular matrices can be stored in an array of size **n (n + 1) / 2.**

Triangular Matrices

- Consider element L(i,j) of a lower-triangular matrix.
- If i < j, the element is in the zero region.
- If i ≥ j, the element is in the non-zero region.
- In row mapping, the element L(i,j), i ≥ j is precede by $\sum_{k=1}^{i-1} k$ nonzero region elements that are in rows 1 through i-1 and j-1 such elements from row i.
- The total number of nonzero region elements that precede L(i, j) in a row mapping is ***i (i-1)/2 + j-1***. This expression also gives the position L(i,j) in element.

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 4 & 2 & 7 & 0 \end{matrix}$$

Lower Triangular Matrix set method

```
template<class T>
void lowerTriangularMatrix<T>::set(int i, int j, const T& newValue)
{// Store newValue as (i,j)th element.
   // validate i and j
   if ( i < 1 || j < 1 || i > n || j > n)
       throw matrixIndexOutOfBounds();


   // (i,j) in lower triangle iff i >= j
   if (i >= j)
      element[i * (i - 1) / 2 + j - 1] = newValue;
   else
      if (newValue != 0)
         throw illegalParameterValue
               ("elements not in lower triangle must be zero");
}
```

Upper Triangular Matrix

- Similar to Lower Triangular Matrix, the mapping of Upper triangular matrix elements can be obtained. However, in this case, we use column major mapping.
- L(i, j) in a column mapping is ***j (j-1)/2 + i-1***

$$\begin{matrix} 2 & 1 & 3 & 0 \\ 0 & 1 & 3 & 8 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 \end{matrix}$$

Symmetric Matrices

- An n X n symmetric matrix can be represented using a one-dimensional array of size n(n+1)/2 by storing either the lower or upper triangle matrix using one of the schemes for a triangular matrix.
- The elements that are not explicitly stored may be computed from those that are.

$$\begin{matrix} 2 & 4 & 6 & 0 \\ 4 & 1 & 9 & 5 \\ 6 & 9 & 4 & 7 \\ 0 & 5 & 7 & 0 \end{matrix}$$

Sparse Matrices

- An m X n matrix is said to be sparse if many of its elements are zero.
- A matrix that is not sparse is dense.
- The boundary between a dense and a sparse matrix is not defined.
- Assumption: Sparse matrices with number of nonzero terms less than $n^2/3$ and in some cases less than $n^2/5$.
- Example: A supermarket study, with 1000 customers and 10,000 items. The Purchase (i,j) matrix stores the quantity of item i purchased by customer j. If an average customer buys 20 different items, only about 20,000 of the 10,000,000 matrix entries are nonzero. It may be noted that the distribution of these non-zero entries do not fall into any well defined structure.

Representation using Linked List

- The nonzero entries of an irregular sparse matrix may be mapped into a linear list in row major order.
- To reconstruct the matrix structure, we need to record the originating row and column of each nonzero entry.
- So each element of the array into which the sparse matrix is mapped needs to have 3 fields:
  - Row (the row of the matrix entry)
  - Col (the column of the matrix entry)
  - Value (the value of the matrix entry)

Representation using Linked List

– For this purpose, we define the struct matrixTerm that has three data members. The data type of row and col is int and that of value is T.

– In addition to storing the nonzero entries of the matrix, we need to store the number of rows and columns in the matrix.

0	0	0	2	0	0	1	0
0	6	0	0	7	0	0	3
0	0	0	9	0	8	0	0
0	4	5	0	0	0	0	0

(a) A 4 × 8 matrix

terms	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7	3	9	8	4	5

(b) Its linear list representation

Space Complexity

- For the example discussed, we need
  - 8 bytes for storing the number of rows and columns (2 ints)
  - 9 * 12 bytes for storing each nonzero element as term
  - 4 bytes (for a reference to the array terms.elements)
  - Total: 128 bytes
- Conventionally, the array purchase would have required 10,000,000 * 4 = 40,000,000 bytes

Time Complexity

- The get operation takes
  - O (log [number of nonzero entries]) time when an array linear list and binary search are used.
- The set operation takes
  - O (number of nonzero entries) time because we may need to move this many entries to make room for the new term.
- Each of these operations take $\theta(1)$ time using the standard two-dimensional array representation.