Ch8. Arrays and Matrices

© copyright 2006 SNU IDB Lab.

SNU
IDB Lab.

Bird's-Eye View

- In practice, data are often in tabular form
  - Arrays are the most natural way to represent it
  - Want to reduce both the space and time requirements by using a customized representation
- This chapter
  - Representation of a multidimensional array
    - Row major and column major representation
  - Develop the class Matrix
    - Represent two-dimensional array
    - Indexed beginning at 1 rather than 0
    - Support operations such as add, multiply, and transpose
  - Introduce matrices with special structures
    - Diagonal, triangular, and symmetric matrices
    - Sparse matrix

Table of Contents

The Abstract Data Type: Array

```
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 one (if any) with the same index
}
```

Indexing a Java Array

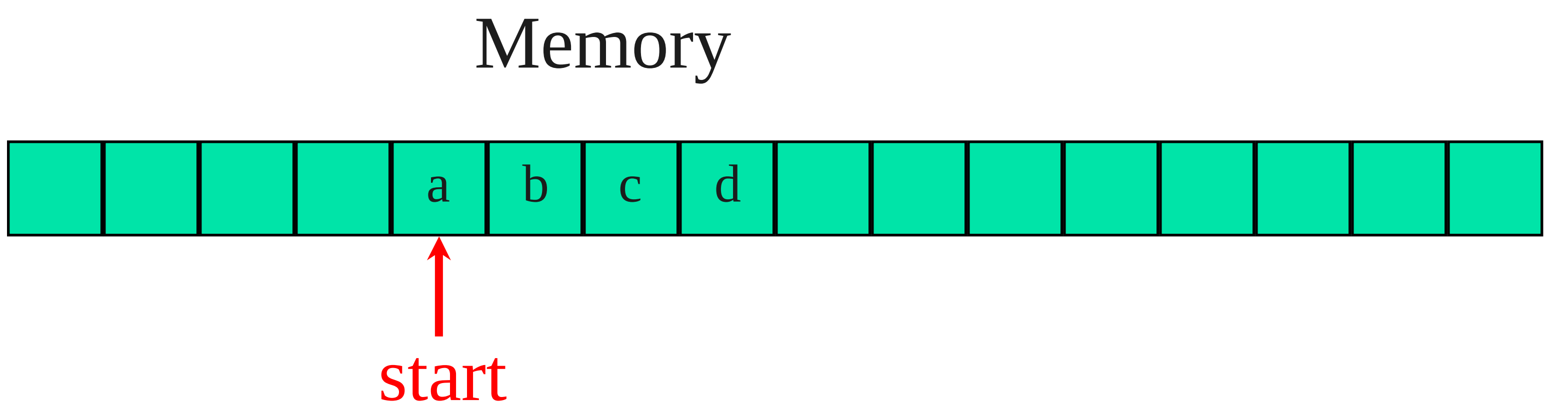
- Arrays are a standard data structure in Java

- The index (subscript) of an array in Java
  - $[\boldsymbol{i}_1]$ $[\boldsymbol{i}_2]$ $[\boldsymbol{i}_3]$… $[\boldsymbol{i}_k]$

- Creating a 3-dimensional array score
  - int [][][] score = new int $[\boldsymbol{u}_1][\boldsymbol{u}_2]$ $[\boldsymbol{u}_3]$

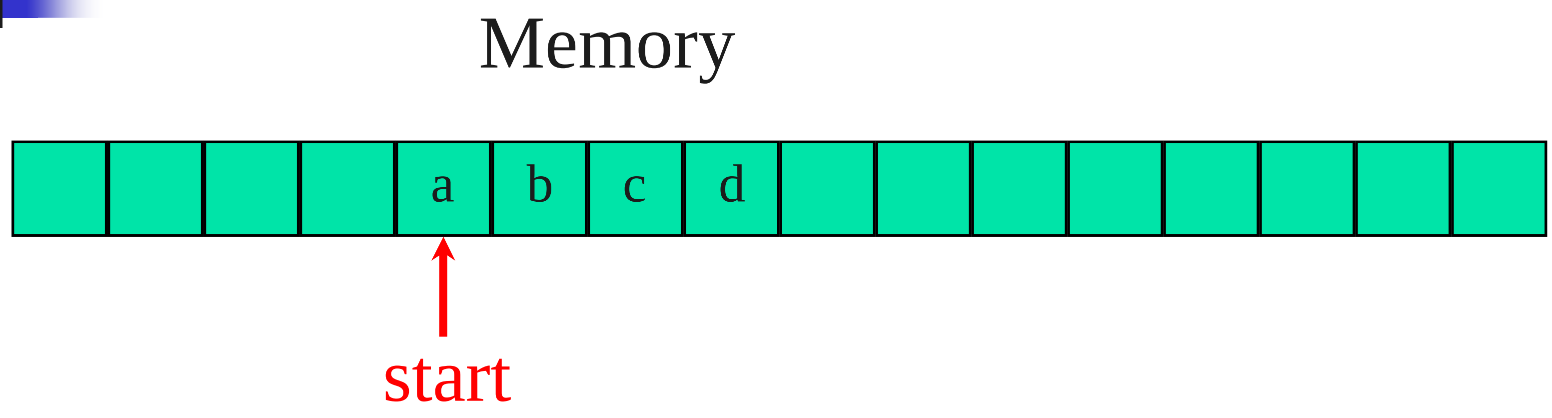
- Java initializes every element of an array to the default value for the data type of the array's components
  - Primitive data types vs. User-defined data types

1-D Array Representation in Java, C, C++



- 1-dimensional array x = [a, b, c, d]
  - X[0], X[1], X[2], X[3]
- Map into contiguous memory locations
- location(x[i]) = start + i

Space Overhead



- Space overhead = 4 bytes for start + 4 bytes for x.length

  = 8 bytes

  (Excluding space needed for the elements of x)

2-D Arrays

- The elements of a 2-dimensional array "a" declared as
  - int [][] a = new int[3][4];

- May be shown as table

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

Rows of a 2-D Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]	→ row 0
a[1][0]	a[1][1]	a[1][2]	a[1][3]	→ row 1
a[2][0]	a[2][1]	a[2][2]	a[2][3]	→ row 2

Columns of a 2-D Array

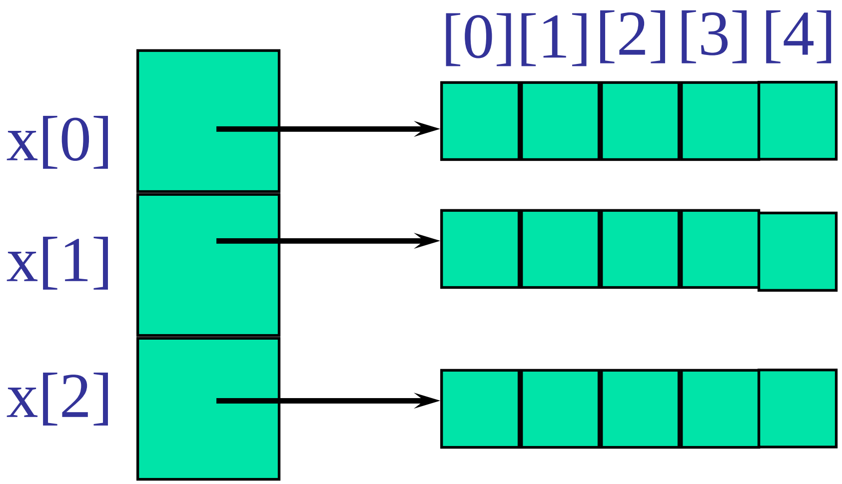
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
column 0	column 1	column 2	column 3

Array of Arrays Representation (1/5)

- Same in Java, C, and C++
- Two-dimensional array is represented as a one-dimensional array
- The one-dimensional array's each element is, itself, a one-dimensional array

Array of Arrays Representation (2/5)

- int [][] x = new int[3][5]
  - A one-dimensional array x (length 3)
  - Each element of x is a one-dimensional array (length5)



Array of Arrays Representation (3/5)

- 2-dimensional array x

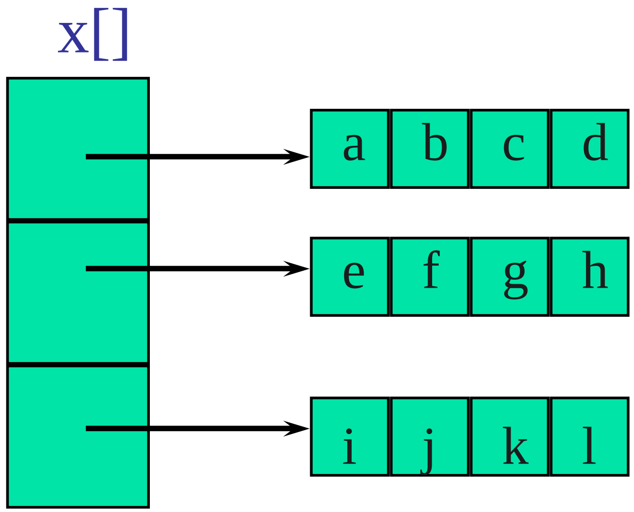
  a, b, c, d
  e, f, g, h
  i,   j, k, l

- View 2-D array as a 1-D arrays of rows

  x = [row0, row1, row2]
  row 0 = [a, b, c, d]
  row 1 = [e, f, g, h]
  row 2 = [i, j, k, l]

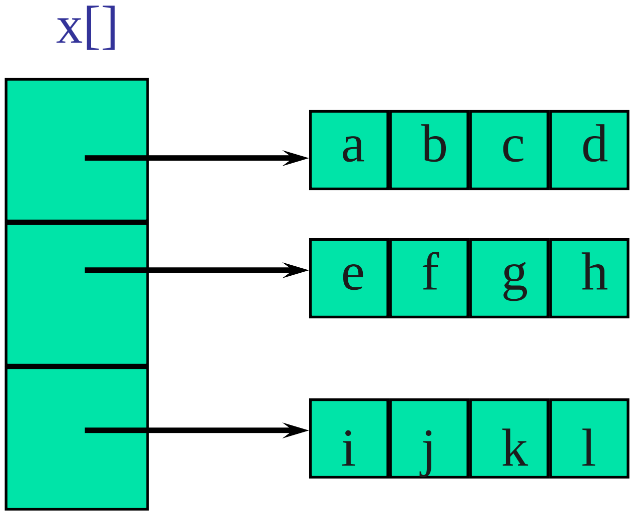
- So, store as 4 1-D arrays which require contiguous memory of size 3, 4, 4, and 4 respectively

Array of Arrays Representation (4/5)



- Array length
  - x.length = 3
  - x[0].length = x[1].length = x[2].length = 4

Array of arrays representation (5/5)



- space overhead = overhead for 4 1-D arrays
  = 4 * 8 bytes = 32 bytes
  = (num of rows + 1) x (start pointer + length variable)

2-D to 1-D: Row-Major Mapping

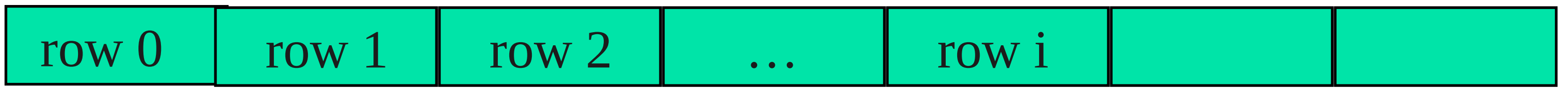
- Example 3 x 4 array

  a, b, c, d
  e, f, g, h
  i, j, k, l

- Convert into 1-D array y by collecting elements by rows
- Within a row elements are collected from left to right
- Rows are collected from top to bottom
- We get y[] = {a, b, c, d, e, f, g, h, I, j, k, l}

row 0	row 1	row 2	…	row i		

Locating Element x[i][j]



- Assume x has r rows and c columns
- Each row has c elements
- There are i rows to the left of row i starting with x[i][0]
- So i * c elements to the left of x[i][0]
- So x[i][j] is mapped to position of i * c + j of the 1D array

Space Overhead for 2D array

row 0	row 1	row 2	…	row i		

- Assume x has r rows and c columns
- 4 bytes for start of 1D array +
  4 bytes for length of 1D array +
  4 bytes for c (number of columns) = 12 bytes
- number of rows r = length / c
- Disadvantage: should have contiguous memory of size r * c

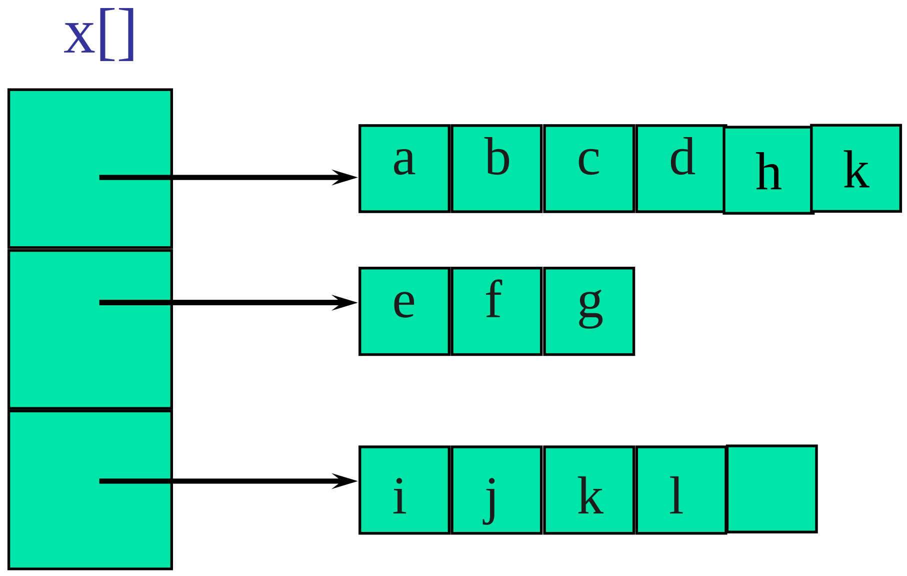
2-D to 1-D: Column Major Mapping

a, b, c, d
e, f, g, h
i, j, k, l

- Convert into 1D array y by collecting elements by columns
- Within a column elements are collected from top to bottom
- Columns are collected from left to right
- We get y = {a, e, i, b, f, j, c, g, k, d, h, l}

Irregular Two-Dimensional Arrays

- Arrays with two or more rows that have a different number of elements
- Size[i] for i (i is the row number)



Creating and Using an Irregular Array

```
// declare a two-dimensional array variable
// and allocate the desired number of rows
int [][] irregularArray = new int [numberOfRows][];

// now allocate space for the elements in each row
for (int i = 0; i < numberOfRows; i++)
        irregularArray[i] = new int [size[i]];

// use the array like any regular array
irregularArray[2][3] = 5;
irregularArray[4][6] = irregularArray[2][3] + 2;
irregularArray[1][1] += 3;
```

Table of Contents

- Arrays
- Matrices
- Special Matrices
- Sparse Matrices

Matrix

- Table of values
  - has as rows and columns like 2-D array, but numbering begins at 1 rather than 0

a b c d	row 1
e f g h	row 2
i j k l	row 3

- Use notation x(i, j) rather than x[i][j]
- Sometimes, we may use Java's 2-D array to represent a matrix

Pitfalls of using a 2D Array for a Matrix

- A[0,*] and A[*,0] of 2D array cannot be used
- Java arrays do not support matrix operations such as add, transport, multiply, and so on
  - i.e. Suppose that x and y are 2D arrays, we cannot do x + y, x – y, x * y, etc. directly in java
- So, need to develop a class Matrix for object-oriented support of all matrix operations

The Class Matrix

- Uses 1-D array element to store a matrix in row-major order
- The CloneableObject interface has clone() and copy()

```
public class Matrix implements CloneableObject {
    int rows, cols;             // matrix dimensions
    Object [] element;          // element array

    public Matrix(int theRows, int theColumns) {
            rows = theRows;
             cols = theColumns;
            element = new Object [ rows * cols];
    }
}
```

clone() & copy() of Matrix

```
public Object clone() {  // return a clone of the matix
    Matrix x = new Matrix(rows, cols);
    for (int i=0; i < rows * cols; i++)
        x.element[i] = ((CloneableObject) element[i]).clone();
    return x;
}
public void copy(Matrix m) { // copy the references in m into this
    if (this != m) {
        rows = m.rows;
        cols = m.cols;
        element = new Object[rows * cols];
        for (int i=0;  i < rows  * cols; i++)
            element[i] = m.element[i]; // copy each reference
    }
}
```

get() & set() of Matrix

```
/**@return the element this[i, j]
  * @throws IndexOutOfBoundsException when i or j invalid */
public Object get(int i, int j) {
    checkIndex(i, j); // validate index
     return element[(i – 1) * cols + j -1];
}

/**set this(i, j) = newValue
  * @throws IndexOutOfBoundsException when i or j invalid */
public void set(int i, int j, Object newValue) {
     checkIndex(i, j);
     element[(i – 1) * cols + j – 1] = newValue;
}
```

add() of Matrix

```
/**@return the this + m
  * @throws IllegalArgumentException when matrices are incomputible */
public Matrix add(Matrix m) {
    if (rows != m.rows || cols != m.cols)
    throw new IllegalArgumentException(“Imcompatible”);

    // create result matrix w
    Matrix w = new Matrix(rows, cols);
    int numberOfTerms = rows * cols;
    for (int i=0; i < numberOfTerms; i++)
            w.element[i] = ((Computable) element[i]).add(m.element[i]));
    return w;
}
```

Complexity of Matrix operations

- Constructor: O(rows * cols)
- Clone(), Copy(), Add(): O(rows * cols)
- Multiply():
  - Program 8.6 at pp 270
  - O(this.row * this.cols * m.cols)

Table of Contents

Special Matrix Definitions

- Diagonal → $M(i, j) = 0$ for $i = j$
- 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, j

Diagonal Matrix

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

- An n x n matrix in which all nonzero terms are on the diagonal
- x(i, j) is on diagonal iff i = j
- Number of diagonal elements in an n x n matrix is n
- Non diagonal elements are zero
- Store diagonal only vs store n^2 whole

The Class DiagonalMatrix

```
public class DiagonalMatrix {
    int rows;                    // matrix dimension (no cols!)
    Object zero;                 // zero element
    Object [] element;           // element array

    public DiagonalMatrix (int theRows, Object theZero) {
        if (theRow < 1)
            throw new IllegalArgumentException("row >0");
        rows = theRows;
        zero = theZero;
        for (int i=0; i<rows; i++)
         element[i] = zero; //construct only the diagonal elements
    }
}
```

get() and set() for diagonal matrix

```
public Object get(int i, int j) {
    checkIndex(i, j); // validate index
    if (i == j) return element[i – 1]; // return only the diagonal element
    else return zero;                   // nondiagonal element
}

public void set(int i, int j, Object newValue) {
    if (i == j) element[i – 1] = newValue; // save only the diagonal element
    else   // nondiagonal element, newValue must be zero
        if (!((Zero)newValue).equalsZero())
          throw new IllegalArgumenetException(“must be zero”);
}
```

Tridiagonal Matrix

- The nonzero elements lie on only the 3 diagonals
  - Main diagonal: M(i, j) where i = j
  - Diagonal below main diagonal: M(i, j) where i = j + 1
  - Diagonal above main diagonal: M(i, j) where j = j - 1

1	1	0	0	0	0
2	2	2	0	0	0
0	3	3	3	0	0
0	0	4	4	4	0
0	0	0	5	5	5
0	0	0	0	6	6

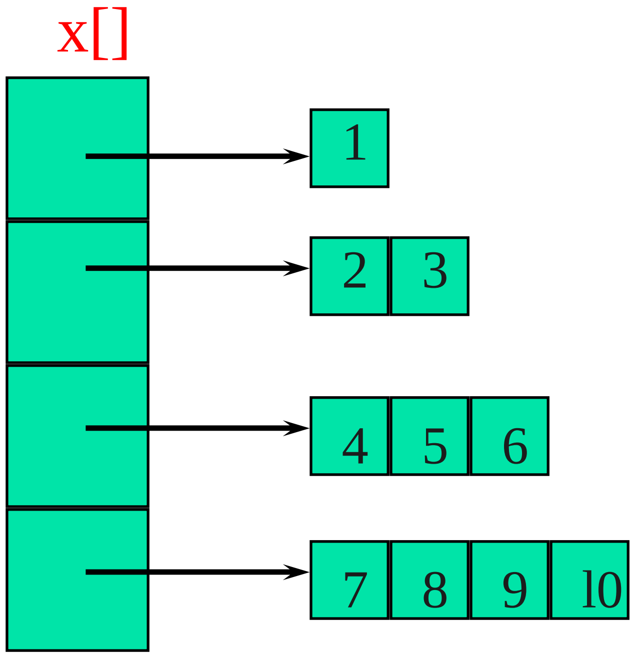
Lower Triangular Matrix (LTM)

- An n x n matrix in which all nonzero terms are either on or below the diagonal.

$$
\begin{matrix}
1 & 0 & 0 & 0 \\
2 & 3 & 0 & 0 \\
4 & 5 & 6 & 0 \\
7 & 8 & 9 & 10
\end{matrix}
$$

- x(i, j) is part of lower triangular iff i>= j
- Number of elements in lower triangle is 1+ 2+ 3+ … + n = n(n+1) / 2
- Store only the lower triangle

LTM: Array of Arrays Representation



- Use an irregular 2D array: length of rows is not required to be the same

Map LTM into a 1D Array

- Use row-major order, but omit terms that are not part of the lower triangle
- For the matrix

1	0	0	0
2	3	0	0
4	5	6	0
7	8	9	10

- We get

  1, 2, 3, 4, 5, 6, 7, 8, 9, 10

LTM: Index of Element[i][j]

- Suppose we store the LTM using 1D array
- Order is: row 1, row 2, row 3, …
- Row i is preceded by rows 1, 2, …, i-1
- Size of row i is i
- Number of elements that precede row i is
  $1 + 2 + 3 + \ldots + (i-1) = i(i-1)/2$
- So element (i,j) is at position $i(i-1)/2 + j - 1$ of the 1D array

Table of Contents

- Arrays
- Matrices
- Special Matrices
- Sparse Matrices

Sparse Matrices

- Sparse matrix ➔ Many elements are zero
- Dense matrix ➔ Few elements are zero
- The boundary between a dense and a sparse matrix is not precisely defined

- Structured sparse matrices
  - Diagonal
  - Tridiagonal
  - Lower triangular

- May be mapped into a 1D array so that a mapping function can be used to locate an element

Unstructured Sparse Matrices (USM) (1/2)

- Airline flight matrix
  - airports are numbered 1 through n (say 1000 airports)
  - flight(i,j) = list of nonstop flights from airport i to airport j
  - 1000 X 1000 matrix ➔ 1 million possible flights
  - n x n array of list references ➔ need 4 million bytes
  - However, only total number of flights = 20,000 (say)
  - need at most 20,000 list references ➔ at most 80,000 bytes
  - We need an economic representation!

Unstructured Sparse Matrices (USM) (2/2)

- Web page matrix
  - web pages are numbered 1 through n
  - Millions of trillions of web pages
  - web(i,j) = number of links from page i to page j
  - The number of links is very very smaller than the number of web pages
- Web analysis
  - authority page … page that has many links to it
  - hub page … links to many authority pages

Facts of Web Page Matrix

- n = 2 billion (and growing by 1 million a day)
- n x n array of ints → $16 * 10^{18}$ bytes ($16 * 10^{9}$ GB)
- Each page links to 10 (say) other pages on average
- On average there are 10 nonzero entries per row
- Space needed for nonzero elements is approximately 20 billion x 4 bytes = 80 billion bytes (80 GB)

Representation of USM

- Single linear list in row-major order
  - Scan the nonzero elements of the sparse matrix in row-major order
  - Each nonzero element is represented by a triple

    (row, column, value)
  - The list of triples may be an array list or a linked list (chain)

USM is viewed as Single Linear List

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0

list =

$$\begin{matrix} \text{row} \\ \text{column} \\ \text{value} \end{matrix} \begin{bmatrix} 1 & 1 & 2 & 2 & 4 & 4 \\ 3 & 5 & 3 & 4 & 2 & 3 \\ 3 & 4 & 5 & 7 & 2 & 6 \end{bmatrix}$$

USM is implemented using Array Linear List

list =	row	1	1	2	2	4	4
	column	3	5	3	4	2	3
	value	3	4	5	7	2	6

element[]	0	1	2	3	4	5
row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

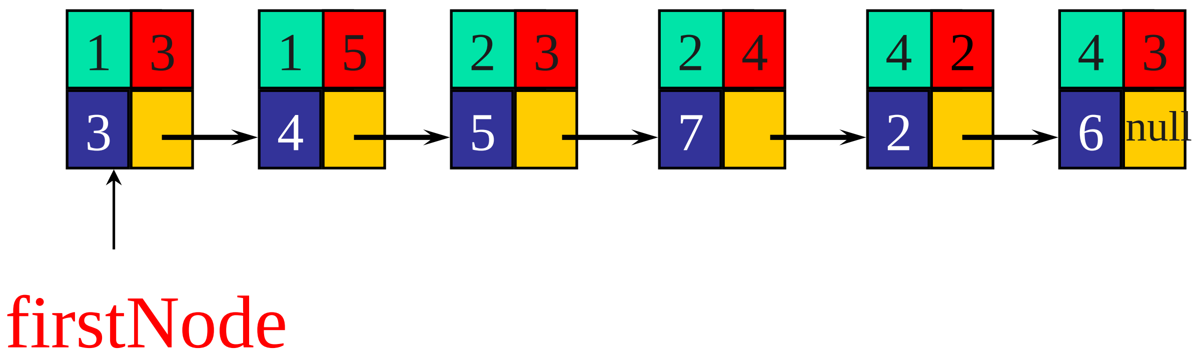
USM implementation using array linear list

- Node Structure

row	col
value	next

USM Implementation using array linear list

$$\text{list} = \begin{matrix} \text{row} \\ \text{column} \\ \text{value} \end{matrix} \begin{bmatrix} 1 & 1 & 2 & 2 & 4 & 4 \\ 3 & 5 & 3 & 4 & 2 & 3 \\ 3 & 4 & 5 & 7 & 2 & 6 \end{bmatrix}$$



One Linear List Per Row

- USM is viewed as array of linear list

➔ Synonym: Array of row chains

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0

row1 = [(3, 3), (5,4)]

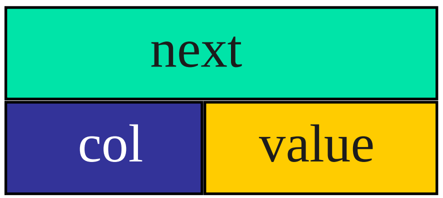
row2 = [(3,5), (4,7)]

row3 = []

row4 = [(2,2), (3,6)]

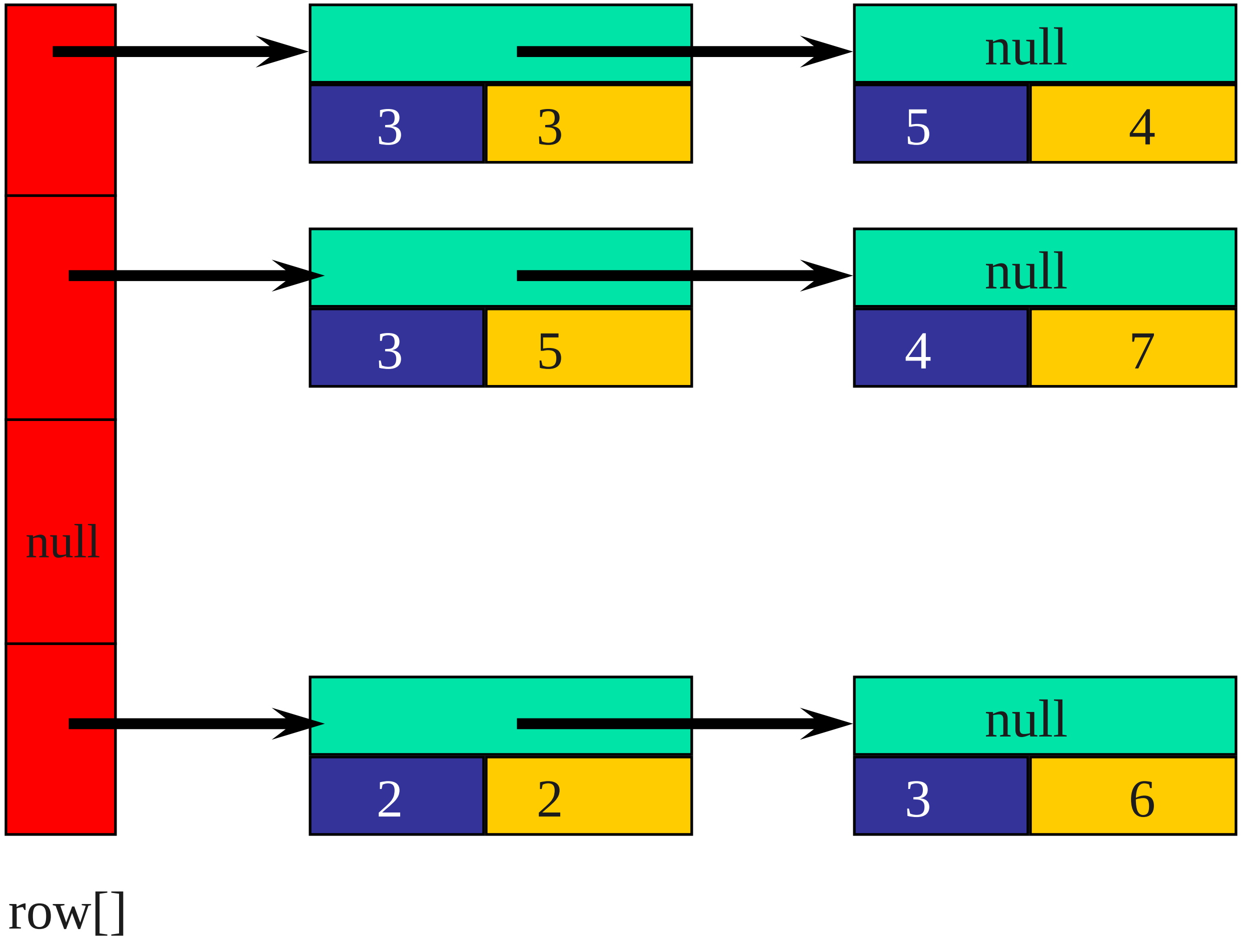
USM implementation using array of row chains (1/2)

- Each row has a chain of the following node structure



USM implementation using array of row chains (2/2)

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0



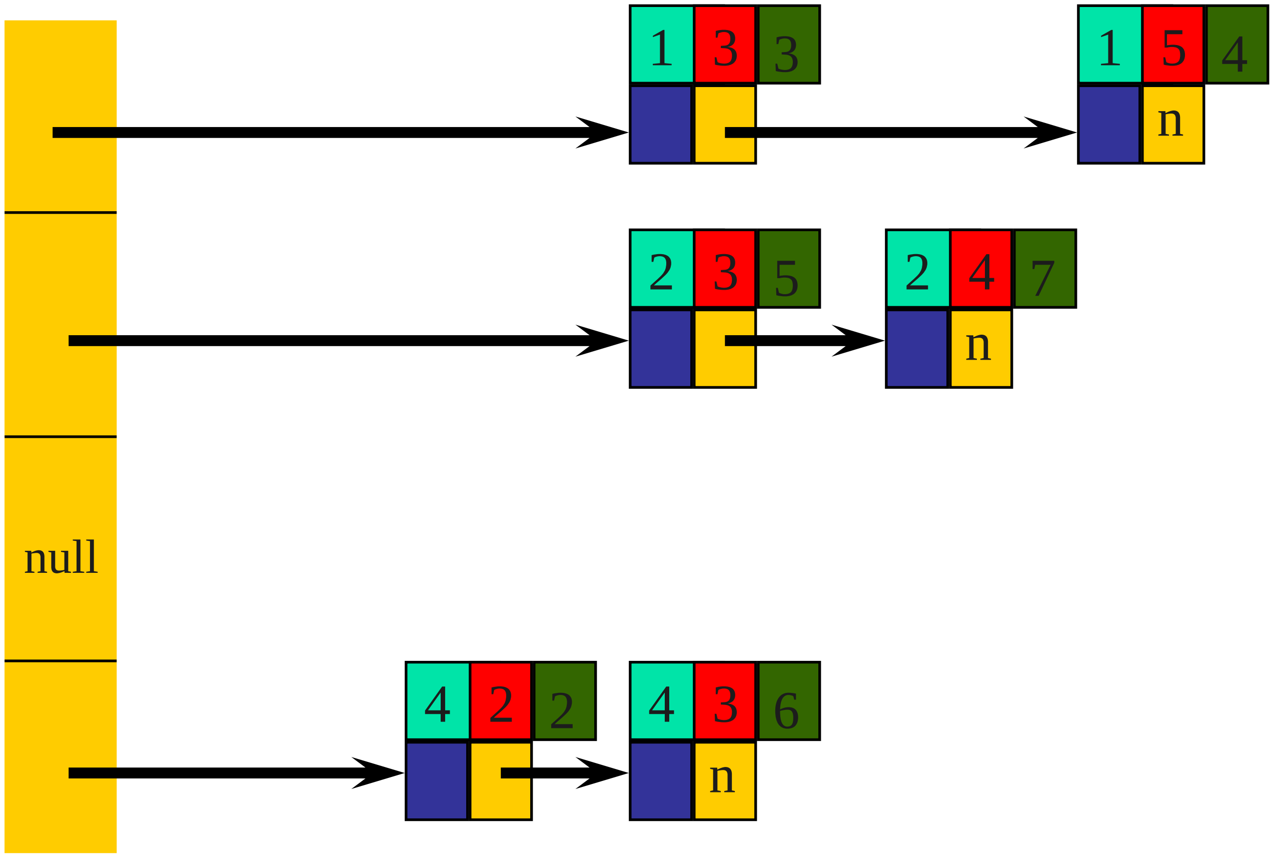
USM implementation using Orthogonal Lists

- Both row and column lists
- More expensive than array of row chains
- More complicated implementation
- Not much advantage!
- Node structure

row	col	value
down	next	

Row Lists

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0



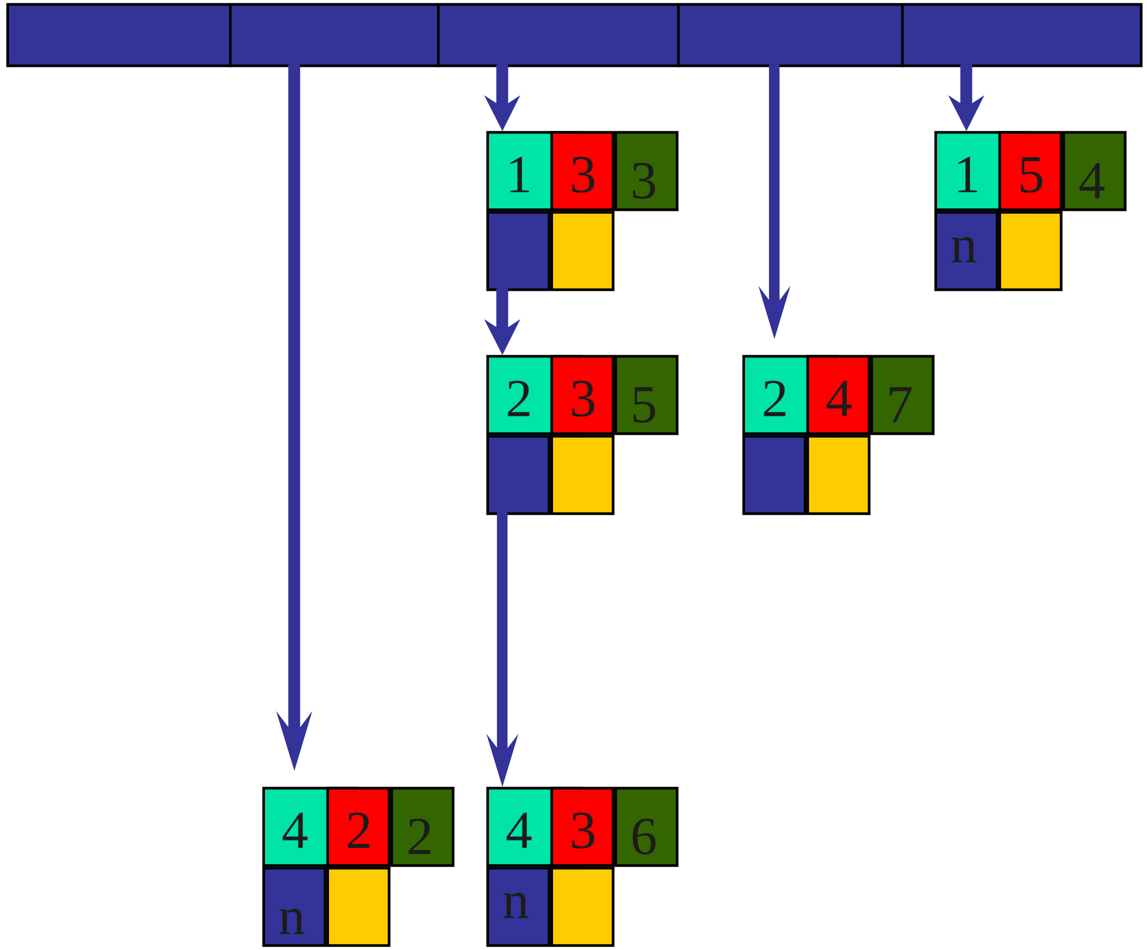
Column Lists

0 0 3 0 4

0 0 5 7 0

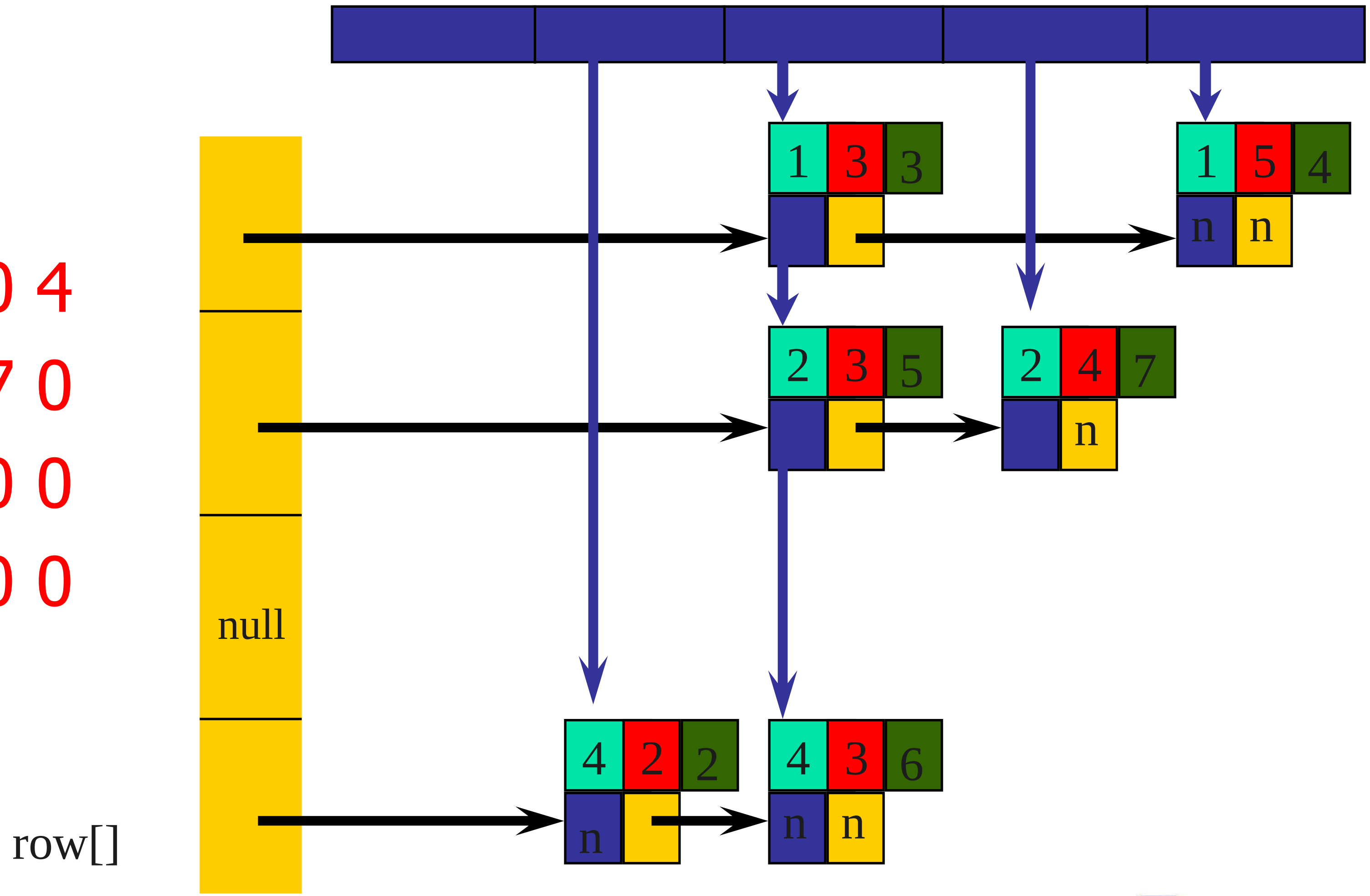
0 0 0 0 0

0 2 6 0 0



Orthogonal Lists

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0



Approximate Memory Requirements

- 500 x 500 matrix with 1994 nonzero elements
  - 2D array: 500 x 500 x 4 = 1million bytes
  - Single Array Linear List: 3 x 1994 x 4 = 23,928 bytes
  - One Chain Per Row: 23928 + 500 x 4 = 25,928 bytes
  - Orthogonal List: your job!

Runtime Performance (1/2)

- Matrix Transpose operation
- 500 x 500 matrix with 1994 nonzero elements
  - 2D array 210 ms
  - Array Linear List 6 ms
  - One Chain Per Row 12 ms

Runtime Performance (2/2)

- Matrix Addition operation
- 500 x 500 matrices with 1994 and 999 nonzero elements
  - 2D array 880 ms
  - Array Linear List 18 ms
  - One Chain Per Row 29 ms

Summary

- In practice, data are often in tabular form
  - Arrays are the most natural way to represent it
  - Reduce both the space and time requirements by using a customized representation

- This chapter
  - Representation of a multidimensional array
    - Row major and column major representation
  - Develop the class Matrix
    - Represent two-dimensional array
    - Indexed beginning at 1 rather than 0
    - Support operations such as add, multiply, and transpose
  - Introduce matrices with special structures
    - Diagonal, triangular, and symmetric matrices
    - Sparse matrix