Chapter Two

Abstract Data Types & Linked lists

Abstract Data Types

data object

set or collection of instances

integer = {0, +1, -1, +2, -2, +3, -3, …}

daysOfWeek = {S,M,T,W,Th,F,Sa}

Data Object

instances may or may not be related

myDataObject = {apple, chair, 2, 5.2, red, green, Jack}

Data Structure

Data object +

relationships that exist among instances

and elements that comprise an instance

Among instances of integer

$369 < 370$

$280 + 4 = 284$

Data Structure

Among elements that comprise an instance

369

3 is more significant than 6

3 is immediately to the left of 6

9 is immediately to the right of 6

Data Structure

The relationships are *usually specified by specifying operations on one or more instances*.

add, subtract, predecessor, multiply

Linear (or Ordered) Lists

instances are of the form

$(e_0, e_1, e_2, \ldots, e_{n-1})$

where e_i denotes a list element

$n >= 0$ is finite

list size is n

Linear Lists

$L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})$

relationships

e_0 is the zero'th (or front) element

e_{n-1} is the last element

e_i immediately precedes e_{i+1}

Linear List Examples/Instances

Students in MyClass =

(Jack, Jill, Abe, Henry, Mary, …, Judy)

Exams in MyClass =

(exam1, exam2, exam3)

Days of Week = (S, M, T, W, Th, F, Sa)

Months = (Jan, Feb, Mar, Apr, …, Nov, Dec)

Linear List Operations—Length()

determine number of elements in list

$$L = (a,b,c,d,e)$$

length = 5

Linear List Operations—Retrieve(theIndex)

retrieve element with given index

$$L = (a,b,c,d,e)$$

Retrieve(0) = *a*

Retrieve(2) = *c*

Retrieve(4) = *e*

Retrieve(-1) = error

Retrieve(9) = error

Linear List Operations—IndexOf(theElement)

determine the index of an element

$L = (a,b,d,b,a)$

$IndexOf(d) = 2$

$IndexOf(a) = 0$

$IndexOf(z) = -1$

Linear List Operations—Delete(theIndex)

delete and return element with given index

$L = (a,b,c,d,e,f,g)$

Delete(2) returns *c*

and *L* becomes *(a,b,d,e,f,g)*

index of *d,e,f,* and *g* decrease by *1*

Linear List Operations—Delete(theIndex)

delete and return element with given index

$L = (a,b,c,d,e,f,g)$

Delete(-1) => error

Delete(20) => error

Linear List Operations—Insert(theIndex, theElement)

insert an element so that the new element has a specified index

$L = (a,b,c,d,e,f,g)$

$Insert(0,h)$ => $L = (h,a,b,c,d,e,f,g)$

index of $a,b,c,d,e,f,$ and g increase by 1

Linear List Operations—Insert(theIndex, theElement)

L = (a,b,c,d,e,f,g)

Insert(2,h) => *L = (a,b,h,c,d,e,f,g)*

index of *c,d,e,f,* and *g* increase by *1*

Insert(10,h) => error

Insert(-6,h) => error

Data Structure Specification

- Language independent
  - Abstract Data Type
- C++
  - Class

Linear List Abstract Data Type

AbstractDataType *LinearList*

{

instances

ordered finite collections of zero or more elements

operations

IsEmpty(): return true iff the list is empty, false otherwise

Length(): return the list size (i.e., number of elements in the list)

Retrieve(index): return the *index*th element of the list

IndexO f(x): return the index of the first occurrence of *x* in the list, return -1 if *x* is not in the list

Delete(index): remove and return the *index*th element, elements with higher index have their index reduced by 1

Insert(theIndex, x): insert *x* as the *index*th element, elements with theIndex >= *index* have their index increased by 1

}

Linear List As A C++ Class

- To specify a general linear list as a C++ class, we need to use a template class.
- We shall study C++ templates later.
- So, for now we restrict ourselves to linear lists whose elements are integers.

Linear List As A C++ Class

```
class LinearListOfIntegers
{
  bool IsEmpty() const;
  int length() const;
  int Retrieve(int index) const;
  int IndexOf(int theElement) const;
  int Delete(int index);
  void Insert(int index, int theElement);
}
```

Data Structures In Text

Generally specified as a C++ (template) class.

Sparse Matrices

Matrix➔ table of values

Sparse Matrices

Matrix➔ table of values

$$\begin{array}{cccccc} 0 & 0 & 3 & 0 & 4 & \\ 0 & 0 & 5 & 7 & 0 & \text{Row 2} \\ 0 & 0 & 0 & 0 & 0 & \\ 0 & 2 & 6 & 0 & 0 & \\ & & & \text{Column 4} & & \end{array}$$

4 x 5 matrix

4 rows

5 columns

20 elements

Sparse Matrices

Sparse matrix → #nonzero elements/#elements is small.

Examples:

- Diagonal
  - Only elements along diagonal may be nonzero
  - n x n matrix → ratio is $n/n^2 = 1/n$
- Tridiagonal
  - Only elements on 3 central diagonals may be nonzero
  - Ratio is $(3n-2)/n^2 = 3/n - 2/n^2$

Sparse Matrices

- Lower triangular (?)
  - Only elements on or below diagonal may be nonzero
  - Ratio is $n(n+1)(2n^2) \sim 0.5$

These are structured sparse matrices. Nonzero elements are in a well-defined portion of the matrix.

Sparse Matrices

An n x n matrix may be stored as an n x n array.

This takes $O(n^2)$ space.

The example structured sparse matrices may be mapped into a 1D array so that a mapping function can be used to locate an element quickly; the space required by the 1D array is less than that required by an n x n array (next lecture).

Unstructured Sparse Matrices

Airline flight matrix.

- airports are numbered 1 through n
- flight(i,j) = list of nonstop flights from airport i to airport j
- n = 1000 (say)
- n x n array of list pointers => 4 million bytes
- total number of nonempty flight lists = 20,000 (say)
- need at most 20,000 list pointers => at most 80,000 bytes

Unstructured Sparse Matrices

Web page matrix.

- web pages are numbered 1 through n
- web(i,j) = number of links from page i to page j

Web analysis.

- authority page … page that has many links to it
- hub page … links to many authority pages

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 Unstructured Sparse Matrices

Single linear list in row-major order.

- scan the nonzero elements of the sparse matrix in row-major order (i.e., scan the rows left to right beginning with row 1 and picking up the nonzero elements)
- each nonzero element is represented by a triple

  (row, column, value)

- the list of triples is stored in a 1D array

Single Linear List Example

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

list =

$$\begin{array}{l} \text{row} \\ \text{column} \\ \text{value} \end{array} \left[\begin{array}{cccccc} 1 & 1 & 2 & 2 & 4 & 4 \\ 3 & 5 & 3 & 4 & 2 & 3 \\ 3 & 4 & 5 & 7 & 2 & 6 \end{array} \right]$$

One Linear List Per Row

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

Single Linear List

- Class SparseMatrix
  - Array of triples of type MatrixTerm
    - ✓ row, col, value
  - rows, // number of rows

    cols, // number of columns

    terms, // number of nonzero elements

    capacity; // size of
- Size of generally not predictable at time of initialization.
  - Start with some default capacity/size (say 10)
  - Increase capacity as needed

Approximate Memory Requirements

500 x 500 matrix with 1994 nonzero elements, 4 bytes per element

2D array	500 x 500 x 4 = 1million bytes
Class SparseMatrix	3 x 1994 x 4 + 4 x 4
	= 23,944 bytes

Array Resizing

```
   (newSize < terms)          “Error”;
MatrixTerm *temp =        MatrixTerm[newSize];
copy(smArray, smArray+terms, temp);
        [] smArray;
smArray = temp;
capacity = newSize;
```

Array Resizing

- To avoid spending too much overall time resizing arrays, we generally set newSize = c * oldSize, where c >0 is some constant.
- Quite often, we use c = 2 (array doubling) or c = 1.5.
- Now, we can show that the total time spent in resizing is O(s), where s is the maximum number of elements added to smArray.

***Matrix Transpose

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

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

Matrix Transpose

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

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

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

2	3	3	3	4	5
4	1	2	4	2	1
2	3	5	6	7	4

Matrix Transpose

Step 1: #nonzero in each row of transpose.

= #nonzero in each column of

original matrix

= [0, 1, 3, 1, 1]

Step2: Start of each row of transpose

= sum of size of preceding rows of

transpose

= [0, 0, 1, 4, 5]

Step 3: Move elements, left to right,

Matrix Transpose

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

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

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

Step 1: #nonzero in each row of transpose.

= #nonzero in each column o

original matrix

= [0, 1, 3, 1, 1]

Step2: Start of each row of transpos

= sum of size of preceding rows of

transpose

= [0, 0, 1, 4, 5]

Step 3: Move elements, left to right

Runtime Performance

Matrix Transpose

500 x 500 matrix with 1994 nonzero elements

Run time measured on a 300MHz Pentium II PC

2D array	210 ms
SparseMatrix	6 ms

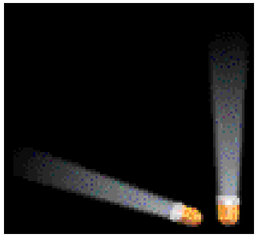
Performance

Matrix Addition.

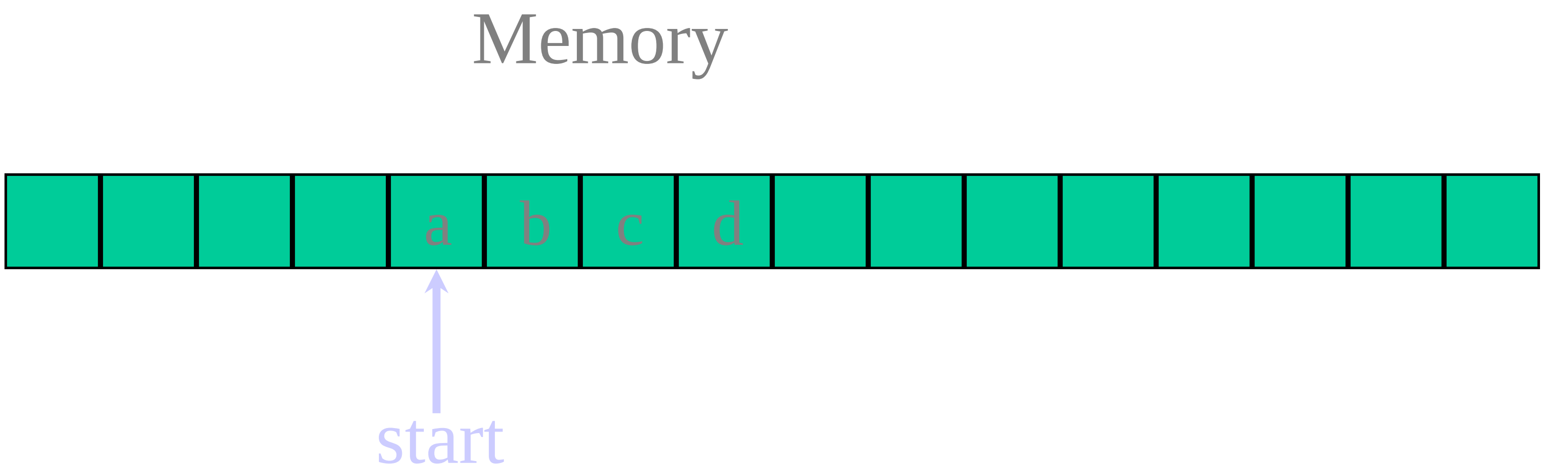
500 x 500 matrices with 1994 and 999 nonzero elements

2D array	880 ms
SparseMatrix	18 ms

Arrays



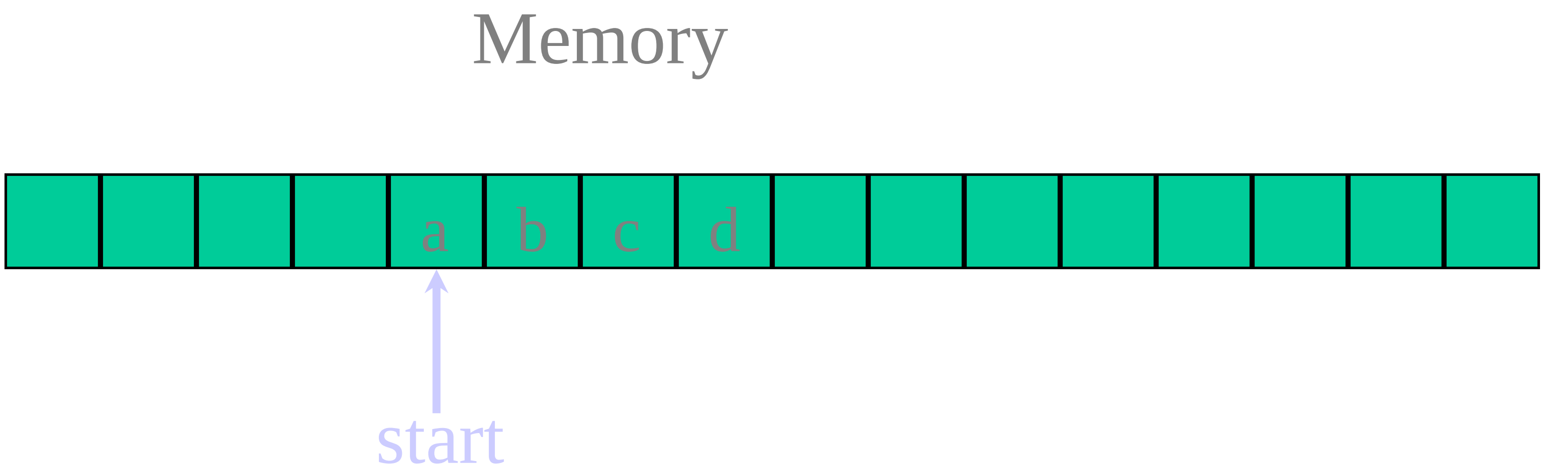
1D Array Representation In C++



- 1-dimensional array x = [a, b, c, d]
- map into contiguous memory locations

- location(x[i]) = start + i

Space Overhead



space overhead = 4 bytes for start

(excludes space needed for the elements of x)

2D Arrays

The elements of a 2-dimensional array a declared as:

```
int [][]a = new int[3][4];
```

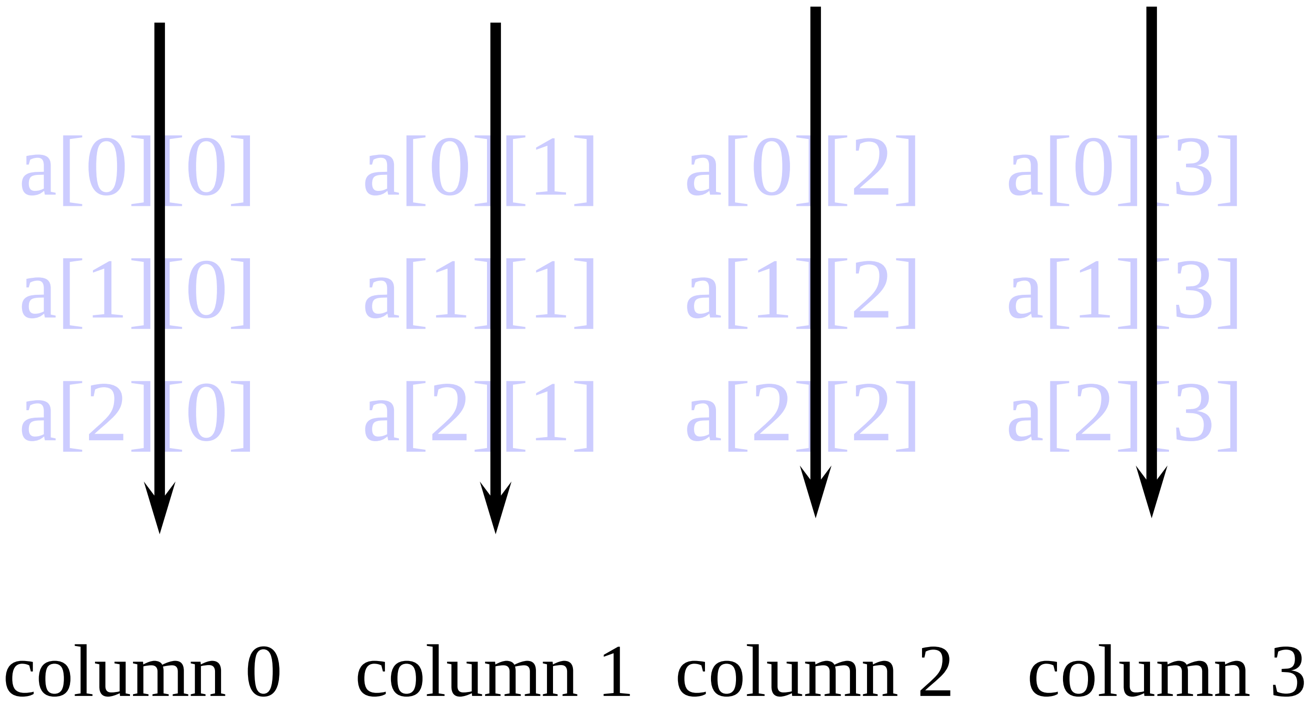
may be shown as a table

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 2D 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 2D Array



2D Array Representation In C++

2-dimensional array x

a, b, c, d

e, f, g, h

i, j, k, l

view 2D array as a 1D array of rows

x = [row0, row1, row 2]

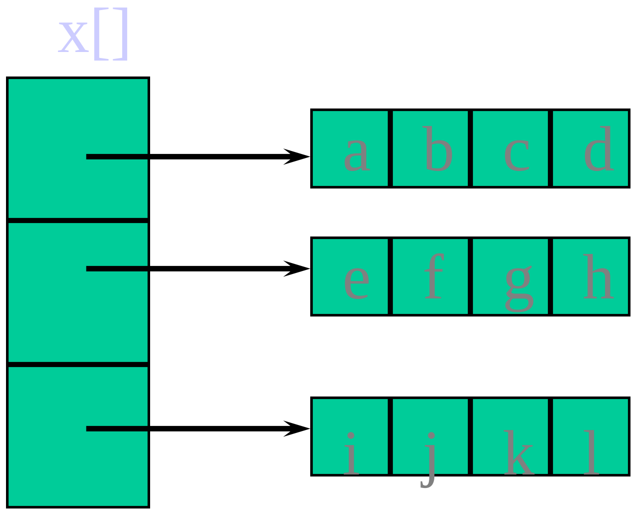
row 0 = [a,b, c, d]

row 1 = [e, f, g, h]

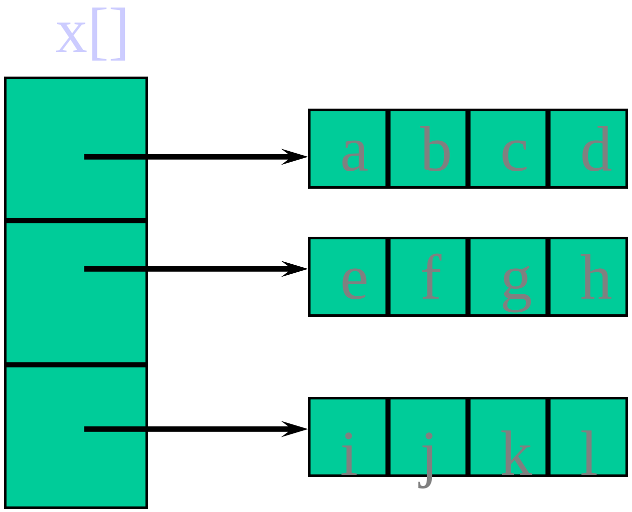
row 2 = [i, j, k, l]

and store as 4 1D arrays

2D Array Representation In C++



Space Overhead



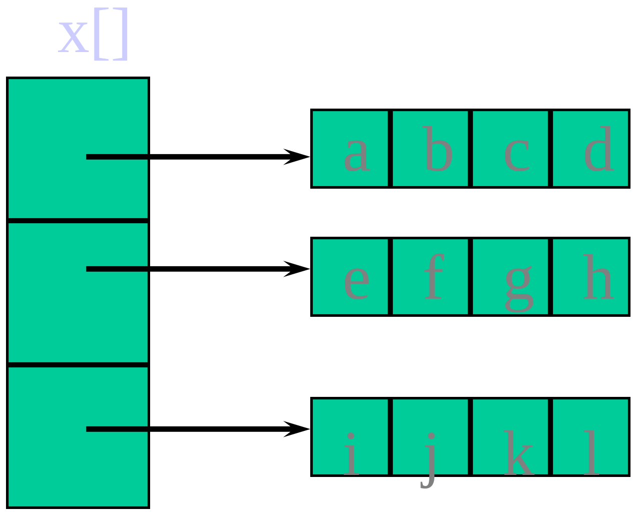
space overhead = overhead for 4 1D arrays

= 4 * 4 bytes

= 16 bytes

= (number of rows + 1) x 4 bytes

Array Representation In C++



- 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

Row-Major Mapping

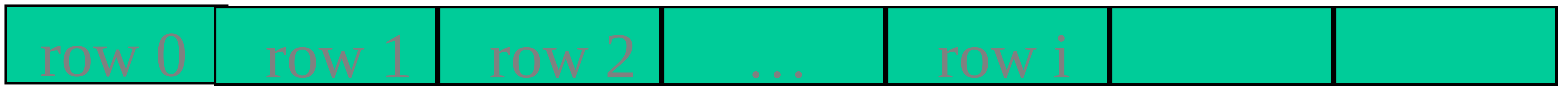
- Example 3 x 4 array:

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

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

row 0	row 1	row 2	…	row i		

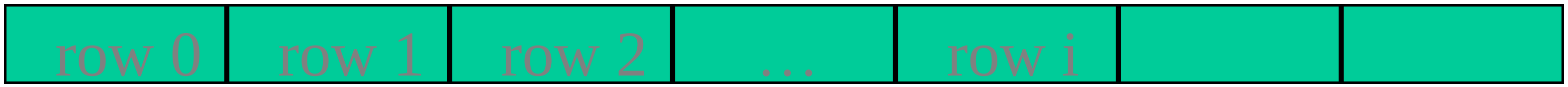
Locating Element x[i][j]



- assume x has r rows and c columns
- each row has c elements
- i rows to the left of row i
- so ic elements to the left of x[i][0]
- so x[i][j] is mapped to position

  ic + j of the 1D array

Space Overhead



4 bytes for start of 1D array +

4 bytes for c (number of columns)

= 8 bytes

Disadvantage

Need contiguous memory of size rc.

Column-Major Mapping

a b c d

e f g h

i j k l

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

Matrix

Table of values. Has rows and columns, 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].
- May use a 2D array to represent a matrix.

Shortcomings Of Using A 2D Array For A Matrix

- Indexes are off by 1.
- C++ arrays do not support matrix operations such as add, transpose, multiply, and so on.
  - Suppose that and are 2D arrays. Can't do , , , etc. in Java.
- Develop a class Matrix for object-oriented support of all matrix operations.

Diagonal Matrix

An n x n matrix in which all nonzero terms are on the diagonal.

Diagonal Matrix

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

- 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 n^2 whole

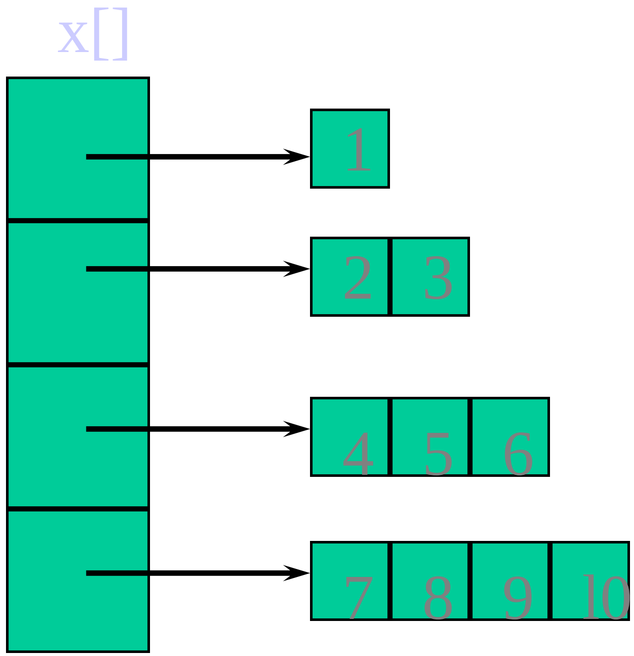
Lower Triangular Matrix

An $n \times n$ matrix in which all nonzero terms are either on or below the diagonal.

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

- $x(i,j)$ is part of lower triangle iff $i >= j$.
- number of elements in lower triangle is $1 + 2 + \ldots + n = n(n+1)/2$.
- store only the lower triangle

Array Of Arrays Representation



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

Creating And Using An Irregular Array

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

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

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

Map Lower Triangular Array Into A 1D Array

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

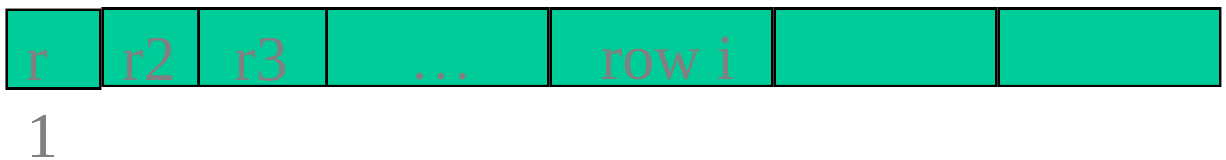
For the matrix

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

we get

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

Index Of Element [i][j]



- 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.