Chapter 2
Arrays

Data Objects

- Data object: A collection of instances is a data object.
  - Order of the instances doesn't matter.
  - E.g., {1,2,3} and {3,2,1} are the same data object.
  - E.g., Integer = {0, +1, -1, +2, -2, +3, -3, …}
  - E.g., daysOfWeek = {S,M,T,W,Th,F,Sa}

Data Objects

- Instances may or may not be related
- myDataObject = {man, chairs, apple, 100.005, @green, Jackie, 3}

Data Structures

- Data Structure =

  data object +

  relationships that exist among instances

  and/or elements that comprise an instance

- Among instances of integer

  369 < 370

Data Structures

- 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 Structures

- Relationships may exist by specifying operations on one or more instances.
  - Examples of operations: add, subtract, predecessor, multiply, …., etc.

Linear (or Ordered) Lists

- One form of the simplest data structures
- Instances are of the form

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

where e_i denotes a list element and $n-1 >= 0$ 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

Students in MyClass =

(Jackie, Abs, Jason, Hank, 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 (-1)* = error

Linear List Operations —IndexOf(theElement)

- Determine the index of an element
- E.g., L = (*a*,*b*,*d*,*b*,*a*)
  - *IndexOf*(*d*) = *2*
  - *IndexOf*(*a*) = *0*
  - *IndexOf*(*z*) = *-1* (if the index of a nonexistent element is defined to be –1)

Linear List Operations — Delete(theIndex)

- Delete an element with given index and return its value

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

  - *Delete*(2) returns *c*
    - and L becomes (a,b,d,e,f,g)

  - Indices of *d,e,f,* and *g* decreased by *1*

  - *Delete*(*-1*) => 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, g* increased by *1*

- *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, g* increased by *1*
  - *Insert(10,h)* => error

Data Structure Specification

- Abstract Data Type
  - I.e., implementations are language-independent
- Use classes for implementations of data structures in C++

Linear List as a Abstract Data Type

AbstractDataType *LinearList*

{

  instances

    ordered finite collections of 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

    IndexOf(*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 *theIndex*-th element, elements with *index* >= *theIndex* 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.
- Will study C++ templates later.
- 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);
}
```

- “Lists” can be implemented by a type of data structures — array. So let’s talk about “array” now!

Array

- Array:
  - a set of index and value
- Data structure:
  - For each index, there is a value associated with that index.
- Can be implemented by consecutive memory
- Example: int a[5]
  - a[0], …, a[4] each contains an integer

Operations on Ordered List

(1) Find the length, n, of the list.

(2) Read the items from left to right (or right to left).

(3) Retrieve the i-th element.

(4) Replace the old value by a new one at the i-th position.

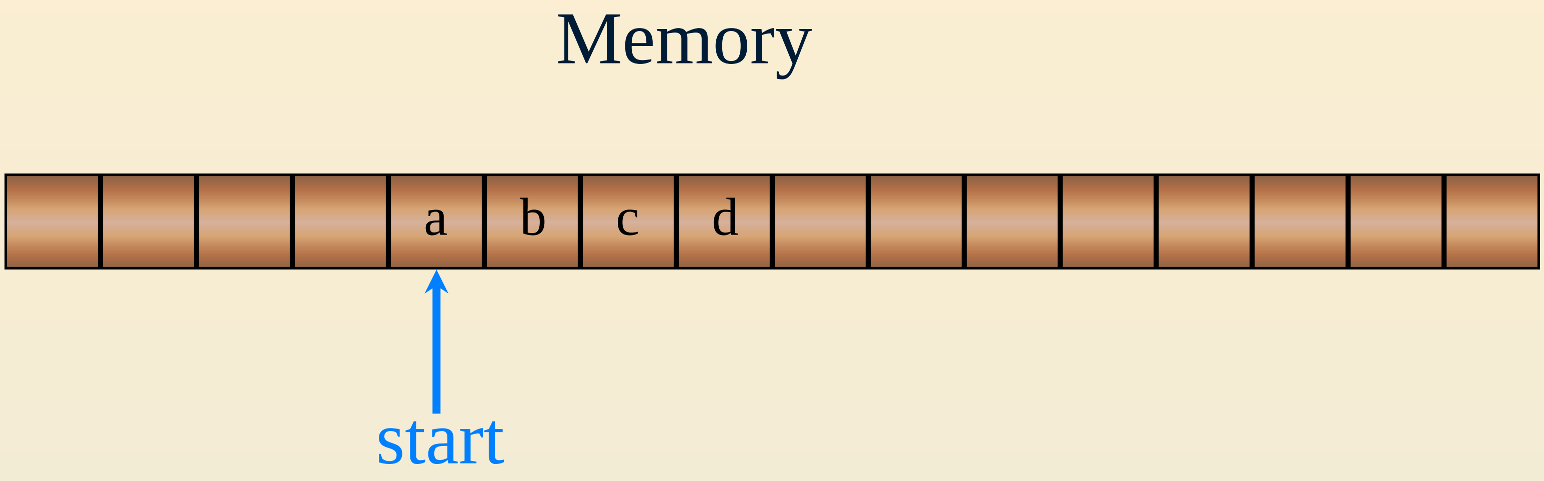
(5) Insert a new element at the position i ,causing elements numbered i, i+1, ..., n to be numbered i+1, i+2, ..., n+1

(6) Delete the element at position i , causing elements numbered i+1, ..., n to be numbered i, i+1, ..., n-1

Implementation on Ordered List by Array

- Implementing ordered list by array
  - Sequential mapping
  - Operations (1)~(4): OK
  - But operations 5 and 6 need data movement
    - this may be costly
- This overhead motivates us to consider non-sequential mapping of order lists in Chapter 4
  - Linked list (we will talk about this later in this course)

Example of Linear List – 1D Array Representation



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

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

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

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

Row-Major Mapping

- Example of 3 x 4 array:

  a b c d

  e f g h

  i j k l

- To store 2D arrays by contiguous memory locations, use 1D array as follows:
  - Convert into 1D 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]

0	c	2c	3c …	ic		
row 0	row 1	row 2	…	row i		

- Assum 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

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}
- We'll talk more about multi-dimensional arrays later if time permitted.

Matrix

- An entity consists of a table of values
  - Has rows and columns, but numbering begins at 1 rather than 0.
  - E.g.

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 offset by 1
- C++ arrays do not support matrix operations such as add, transpose, multiply, etc.
  - Suppose that x and y are 2D arrays. Can't do x + y, x – y, x * y immediately.
- Need to develop a class Matrix for OO features to support all matrix operations.
- Problems get worse when a matrix is sparse
  - Talk about this later, but let's first see some sparse matrix examples

Example – Diagonal Matrix

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

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

Example – Diagonal Matrix

- $x(i,j)$ is on diagonal iff $i = j$
- Number of diagonal elements in an $n \times n$ matrix is n
- Non-diagonal elements are zero
- Store diagonal only v.s. n^2 for the whole

Example – Lower Triangular Matrix

- 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 triangle iff $i >= j$.
  - Number of elements in lower triangle is $1 + 2 + \ldots + n = n(n+1)/2$.
  - Store only the lower triangle

Example – Sparse Matrix

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

- 4 x 5 matrix
- 20 elements
- 6 nonzero elements

Sparse Matrices

- This example is a sparse matrix
- Sparse matrix ➔

  (#nonzero elements)/(# all elements) is small.
- Why sparse matrices?
  - Because they are frequently seen in large scale computations
  - Regular matrices are easier to deal with
  - But for various algorithms to work on, sparse matrices usually require special data structures

Sparse Matrices

- Examples:
  - Diagonal matrix
    - Only elements along diagonal may be nonzero
    - n x n matrix → ratio is $n/n^2 = 1/n$
  - Lower triangular matrix
    - Only elements on or below diagonal may be nonzero
    - Ratio is ~ 0.5
- These are structured sparse matrices
  - Nonzero elements are in a well-defined portion of the matrix.

Sparse Matrices

- A general mxn matrix consists of m rows and n columns of numbers.
  - Space complexity is O(mxn).
- Natural to store a matrix in a two dimensional array, but…
  - it wastes a lot of memory — most of 0's don't need to be stored.
  - Computational cost could be huge — most of computations involving 0's could be completely avoided.
  - So we'll study special data structures for sparse matrices to make computation and storage requirement more efficient.
  - Before this, let's talk more about real-world examples for sparse matrix.

Unstructured Sparse Matrices – Another Example

- Biological molecular interactions:
  - Biological molecules are numbered from 1 through *n*
  - Matrix(*i*,*j*) = 1 → molecules *i* and *j* interact
  - Matrix(*i*,*j*) = 0 → molecules *i* and *j* don't interact
  - Typically, *n* is 1000~10000, so lots of 0's are in the matrix because a molecule may interact with only a few other molecules
  - Important problem in Biology and Medicine, such as drug discovery

Unstructured Sparse Matrices – More Examples

- 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
  - Other pages may have only few links
  - See IEEE Computer, August 1999, p. 60, for a paper that describes Web analysis based on such a matrix. Operations such as matrix transposition and multiplication are used.

Web Page Matrix

- Assume n = 1 billion (billion = 10^9) webpges
- n x n array of ints => 4 * 10^{18} bytes (4 * 10^9 GB) (if an integer uses 4 bytes)
- 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 only approximately 10 billion x 4 bytes = 40 billion bytes (40 GB)

Sparse Matrix Example in Textbook’s Fig. 2.2

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

Sparse Matrix Representation

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

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

Need to store 6*6 = 36 elements

Only need to store 8 elements now

Sparse Matrix Representation

- Ideally, use triplet <row, column, value>
- Store triplets row by row
- For all triplets within a row, their column indices are in ascending order
- Rows are in ascending order, too

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

Sparse Matrix: Abastract Data Type

class SparseMatrix

// objects: A set of triplets, <row, column, value>, where row and
//column are integers and form a unique combinations

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

// The constructor function creates a SparseMatrix with r rows, c
// columns, and a capacity of t nonzero terms

SparseMatrix Transpose();

// Returns the SparseMatrix obtained by interchanging the row
// and column value of every triplet in *this

Sparse Matrix Abastract Data Type

```
SparseMatrix Add(SparseMatrix b);
 // If the dimensions of a (*this) and b are the same, then
//  the matrix produced by adding corresponding items,
//  i.e., those with identical row and column values is returned
//  else error.
 SparseMatrix Multiply(SparseMatrix b);
// If number of columns in a (*this) equals number of rows
// in b, then the matrix d produced by multiplying a by b
// according to d[i][j] = Σ(a[i][k] * b[k][j]), where
// d[i][j] is the (i, j)-th element, is returned
// else error.
```

Matrix Transpose – Some More Codes

- class MatrixTerm {

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

- In class SparseMatrix, define:

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

(where terms: # non-zeros; capacity: size of smArray)

- We need smArray to represent a sparse matrix in a compact form using sets of the triplets.
- How to put these altogether is your homework. Refer to the homework assignment.

Matrix Transpose

a and b are smArray in the previous page

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

transpose

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

Matrix Transpose

- Naturally, one will do
  - For each row i,
    take element <i, j, value> and store it in element
    <j, i, value> of the transpose.
- E.g.

  (0, 0, 15) ====> (0, 0, 15)

  (0, 3, 22) ====> (3, 0, 22)

  (0, 5, -15) ====> (5, 0, -15)

  (1, 1, 11) ====> (1, 1, 11)
- But the method above has some problems because

  (1, 1, 11) must be moved above (3, 0, 22) (see p. 43)

Matrix Transpose

- So using rows to transpose may encounter some unforeseen problems like above.
- So the textbook suggests the following new method:
  - Instead of working on rows, just work on columns to make sure after transposing the rows (the columns in the original matrix) are in ascending order.
  - I.e.: “For all elements in column j,
    place element <i, j, value> in element
    <j, i, value>”

Matrix Transpose – The Transpose Method

- I.e., “find all elements in column 0 and store them in row 0, and
  - find all elements in column 1 and store them in row 1, etc.”
  - Since the rows are already in ascending order, this can locate elements in the correct column order
- See example next
- This is the “Transpose” method in the textbook

The Transpose Method

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

- Goal: Transpose b to a (not transposing a to b, because the textbook transposes b to a in a later example)
- Iteration 0: scan the array b and process the entries with col=0

The Transpose Method

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

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

- Iteration 1: scan the array b and process the entries with col=1
- Iteration 2: scan the array b and process the entries with col=2,

…, etc.

Matrix Transpose – using 2-Dimensional Array Representation

- So for each column, scan "#terms" times
  - Each iteration scans "#terms" times

- Since there are #columns, time complexity = O(cols×terms)

The Transpose Method – the Code

```
SparseMatrix  SparseMatrix::Transpose()
// Return the transpose of a (*this) {
     SparseMatrix b(cols, rows, terms); // capacity of b.smArray is terms
    if (terms > 0)
    {// nonzero matrix
        int currentB = 0;
        for (int c=0; c<cols; c++)  // transpose by columns
            for (int i = 0; i < terms; i++)  // find and move terms in column c
                if (smArray[i].col == c) {
                    b.smArray[currentB].row=c;
                    b.smArray[currentB].col=smArray[i].row;
                    b.smArray[currentB++].value = smArray[i].value;
                }
    } // end of if (terms > 0)
    return b;
} // end of transpose
```

Time complexity = O(cols*terms)

Matrix Transpose – using 2-Dimensional Array Representation

- Traditional way for matrix transpose using 2-dimensional array representation

```
for (int j=0; j< cols; j++)
    for (int i=0; i< rows; i++)
        b[j][i]=a[i][j];
```

- Time complexity = $O(columns \times rows)$

Compare the "Transpose" Method with 2-Dimensional Array Representation

- $O(columns \times terms)$ vs. $O(columns \times rows)$

2D array representation

Transpose in textbook

  - If the matrix is non-sparse,
    - #terms $\rightarrow$ columns$\times$rows,
    - Time complexity $\rightarrow$ $O(columns^2 \times rows)$
    - This is even worse than $O(columns \times rows)$

- So which method is better depends on if the matrix is sparse.

Matrix Transpose – The FastTranspose Method

- A clever solution – FastTranspose:
  - Key: Find out which row of the transpose matrix to put the non-zero element of the original matrix.
  - How to do this?
  - Determine # elements in each column of the original matrix → store this as "RowSize"
    - I.e., how many rows in the transpose will be needed to store all the elements from the same column of the original matrix
  - For each column, determine its starting row position in the transpose → RowStart

Matrix Transpose – The FastTranspose Method

- The key: compute RowStart so that we know where non-zero elements should be transposed to the new matrix.

- Thus, this method requires more space than Transpose because it have to store the two variables
  - RowSize
  - RowStart

Matrix Transpose – The FastTranspose Method

- Initialization

	row	col	value
a[0]			
a[1]			
a[2]			
a[3]			
a[4]			
a[5]			
a[6]			
a[7]			

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

Index		[0]	[1]	[2]	[3]	[4]	[5]
RowSize	=	3	2	1	0	1	1
RowStart	=	0	3	5	6	6	7

For col 0, array b contains 3 elements

For col 1, array b contains 2 elements

For col 2, array b contains 1 element

- Step 1: Calculate RowSize by scanning array b's col
- **Step 2:** Calculate: RowStart[i] = RowStart[i-1] + RowSize[i-1], where RowStart[0] is set to 0.

Matrix Transpose – The FastTranspose Method

- So in the initialization step, first compute
  - RowSize, then compute
  - RowStart

Matrix Transpose – The FastTranspose Method

- Transpose begins

	row	col	value
a[0]	0	0	15
a[1]			
a[2]			
a[3]			
a[4]			
a[5]			
a[6]			
a[7]			

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

Index		[0]	[1]	[2]	[3]	[4]	[5]
RowSize	=	3	2	1	0	1	1
RowStart	=	0	3	5	6	6	7

RowStart[0] needs to increase by 1, so the next element from b's col 0 will be placed at the right location of a

RowStart[0]++

Matrix Transpose – The FastTranspose Method

	row	col	value
a[0]	0	0	15
a[1]			
a[2]			
a[3]			
a[4]			
a[5]			
a[6]			
a[7]			

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

Index	[0]	[1]	[2]	[3]	[4]	[5]
RowSize =	3	2	1	0	1	1
RowStart =	1	3	5	6	6	7

RowStart[0] becomes 1 now

Matrix Transpose – The FastTranspose Method

- Transpose continues

	row	col	value
a[0]	0	0	15
a[1]			
a[2]			
a[3]			
a[4]			
a[5]			
a[6]	4	0	91
a[7]			

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

Index	[0]	[1]	[2]	[3]	[4]	[5]
RowSize =	3	2	1	0	1	1
RowStart =	1	3	5	6	6	7

RowStart[4]++

RowStart[4] needs to increase by 1, so the next element from b's col 4 will be placed at a's right location

Matrix Transpose – The FastTranspose Method

- Transpose finishes

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

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

(not easy to grasp, so spend time trying to understand the whole process)

Summary – The FastTranspose Method

- So for FastTranspose
  - In initialization we compute
  - RowSize, then compute
  - RowStart
- Then do the transpose

FastTranspose – the Code

```
SparseMatrix SparseMatrix::FastTranspose( )
{// Return the transpose of *this in O(terms + cols) time
  SparseMatrix b(cols , rows , terms);
  if (terms > 0)
  {// non-zero matrix
    int *rowSize = new int[cols];
    int *rowStart = new int[cols];
    // compute rowSize[i] = number of terms in row i
    ...
    for (int i = 0 ; i < terms ; i ++) rowSize[smArray[i].col]++;

    // rowStart[i] = starting position of row i
    rowStart[0] = 0;
    for (int i = 1 ; i < cols ; i++) rowStart[i] = rowStart[i-1] + rowSize[i-1];
```

O(terms)

O(columns)

FastTranspose – the Code (cont.)

```
 // start the transpose process
for (int i = 0 ; i < terms ; i++)
    {// copy from *this to b
      int j = rowStart[smArray[i].col];
      b.smArray[j].row= smArray[i].col;
      b.smArray[j].col = smArray[i].row;
      b.smArray[j].value = smArray[i].value;
      rowStart[smArray[i].col]++;
    } // end of for
    delete [] rowSize;
    delete [] rowStart;
  } // end of if
  return b;
}
```

O(terms)

Overall complexity = O(columns+terms)

Representation of Multi-dimensional Arrays by 1-dimensional Arrays

- Multidimensional arrays are usually implemented by 1 dimensional array via either row major order or column major order.

Array Applications — String

- Usually string is represented as a character array.
- General string operations include comparison, string concatenation, copy, insertion, string matching, printing, etc.

H	e	l	l	o		W	o	r	l	d	\0

Array Applications — More

- Examples:
  - Stacks
  - Queues
  - String matching
  - Binary string representations for Genetic Algorithms
  - …
- All these are important subjects in mid/high-level algorithm course, but we shall cover only some of them in this course.

Homework Assignment

- Go to Elearning to get the homework assignment
- Note the deadline.