CSE221

Lecture 2:
Arrays

Hyungon Moon

Slide credits: The textbook authors, Won-ki Jeong,
Tsz-Chiu Au, and Myeongjae Jeon, and Kasey Champion (UW)

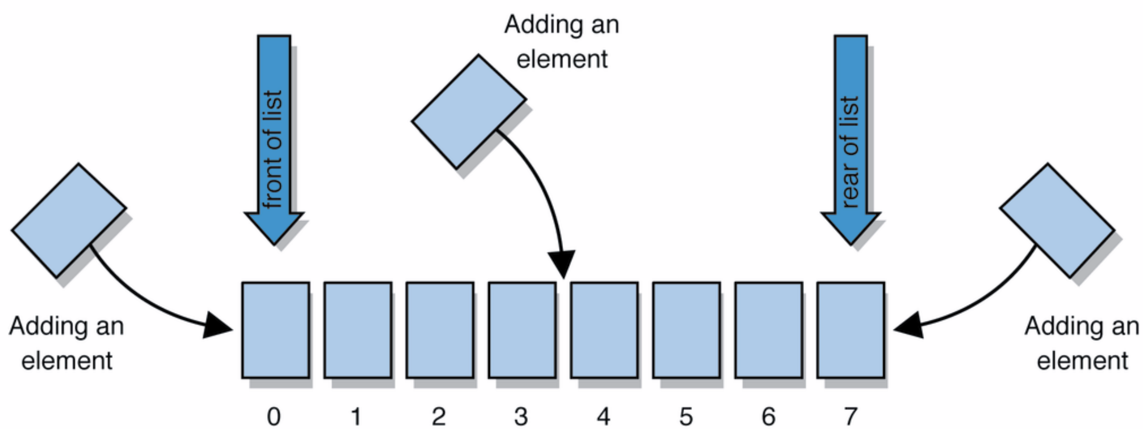
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

- Basic Concepts
- Arrays
- Examples of representing other ADTs
  - Polynomial
  - Sparse matrices

Abstract Data Type (ADT)

  - A definition for expected operations and behavior
- Examples: list - a collection storing an ordered sequence of elements.
  - Operations:



Data Structure

– How we organize/storing the data points.

– Determines how each operation would behave.

– One ADT can be implemented many ways.

– One data structure can be used for multiple ADTs.

ADT first, then data structure

- Wrong way:
  - I'll represent my data A as an Array.
- Right way:
  - I'll be frequently accessing a random data point of my data A: I need operation [].
  - I'll thus use Array.

Outline

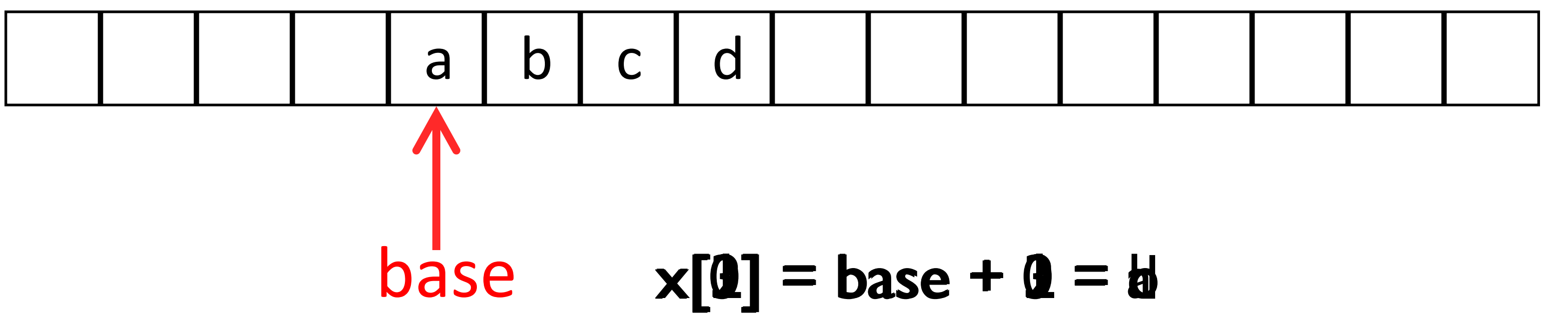
- Basic Concepts
- Arrays
- Examples of representing other ADTs
  - Polynomials
  - Sparse matrices

Today's focus: Array

- Continuous memory space with a same data type
- Accessing by index is efficient.
- `type name [elements];` C/C++
- Example
  - int a[100];
  - float b[10][20];
  - mytype c[5]; // user-defined data type

1D Array Representation In C

1D Memory Space



- 1-dimensional array x = [a, b, c, d]
- map into contiguous memory locations
- location(x[i]) = base + i

Array as a data structure for List

- Operations
  - Insert/Delete
  - Access by index
  - Access by value
  - Sort
  - Swap
- Memory: no/small metadata

2D Arrays

- The elements of a 2-dimensional array x declared as:
- int x[3][4];
- may be shown as a table

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

Rows Of A 2D Array

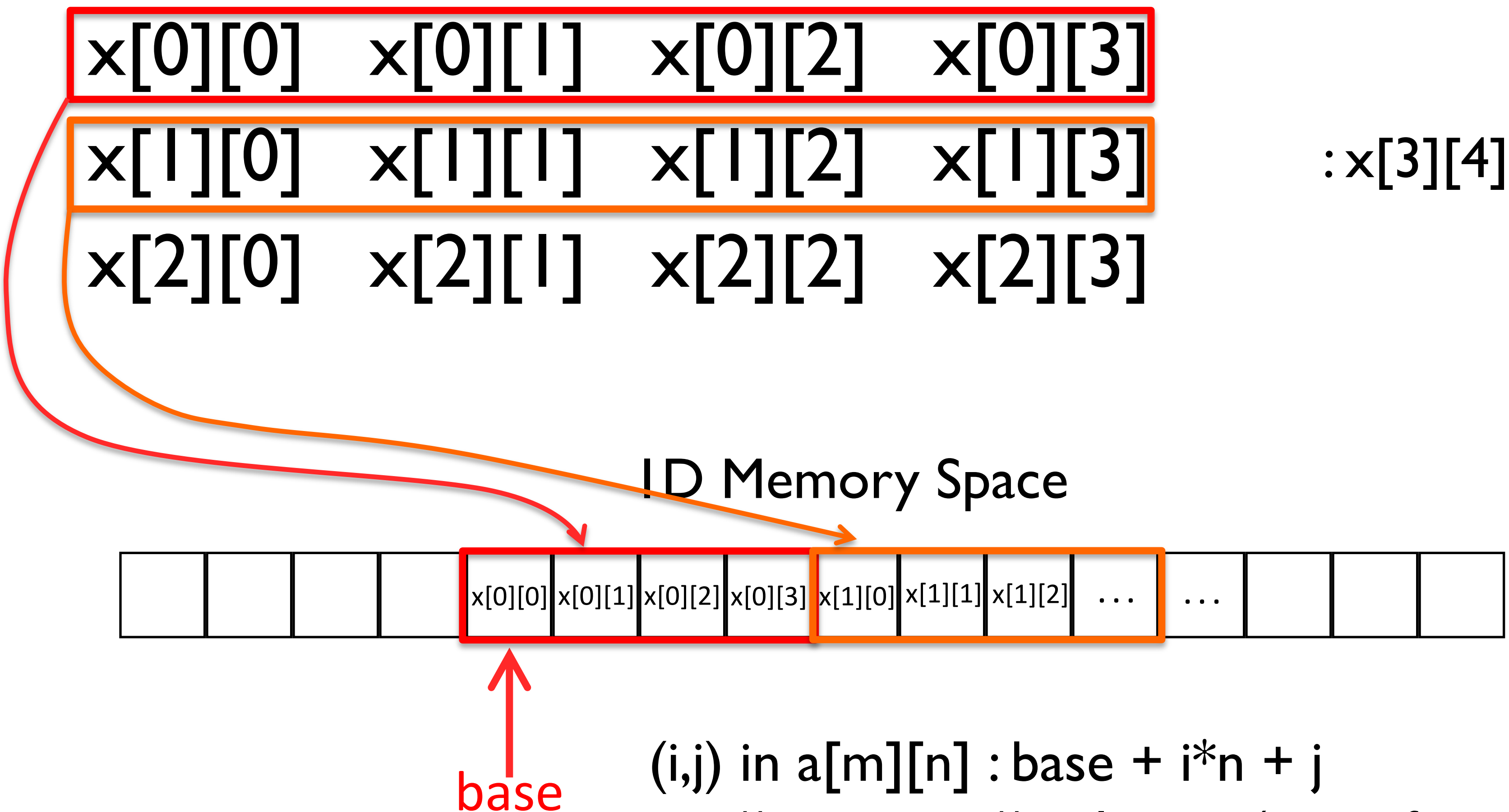
x[i][j]: jth element of an array x[i]

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

Columns Of A 2D Array

x[0][0]	x[0][1]	x[0][2]	x[0][3]
x[1][0]	x[1][1]	x[1][2]	x[1][3]
x[2][0]	x[2][1]	x[2][2]	x[2][3]
col 0	col 1	col 2	col 3

1D Indexing of 2D Array



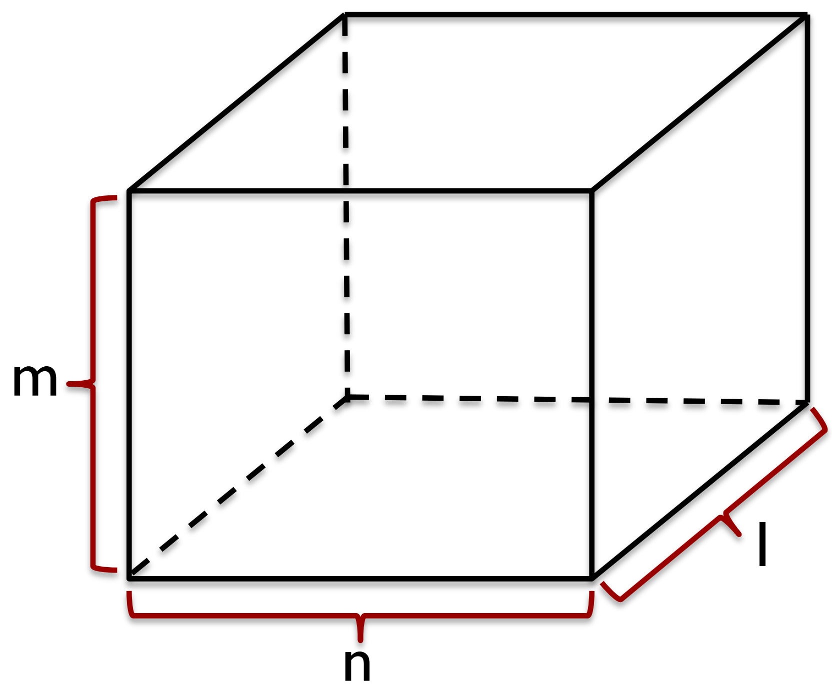
(i,j) in a[m][n] : base + i*n + j

m : # rows, n : # columns (size of a row)

i : row index, j : column index

1D Indexing of 3D Array

- (i,j,k) in x[l][m][n] : base + i*m*n + j*n + k



Array can also be an ADT

- An ADT defined with array-like operations.
- Doesn't need to be implemented with pure array!
- Mapping between index and value
- Operators to retrieve and store value
- We can use operator overloading to make it looks and feels like a raw array.

```
class GeneralArray {
// A set of pairs <index, value> where for each
// value of index in index set there is a value of
// type float
public:
   // Constructor.
   // j : dimension, list : range of each dimension.
   // initValue : initial value
   GeneralArray(int j,
                RangeList list,
                float initValue);
   // Return the float associate with i if i is in
   // the index set. Otherwise throw an exception
   float Retrieve(const index i);
   float& operator[] (const int index);
   // Replace the old value associate with i if i is
   // in the index set. Otherwise throw an exception
   void Store(index I, float x)
}
```

Outline

- Basic Concepts
- Arrays
- Examples of representing other ADTs
  - Polynomial
  - Sparse matrices

Polynomial Abstract Data Type

- Data
  - Set of ordered pairs of <e,*c*> where e is exponent and *c* is coefficient of terms of the polynomial

coefficient exponent

$$a(x) = 3x^2 + 2x + 4$$

Degree: 2, Order : 3

- Operations
  - Addition, subtraction, multiplication, evaluation, …

Polynomial Representation

- Version 1: plain-static
  - Exponent is implicitly represented
  - Max degree has to be known, waste space if degree << MaxDegree

$$a(x) = \sum a_i x^i$$

```
private:
   int degree;
   float coef[MaxDegree + 1];
```

```
a.degree = n
a.coef[i] = a_{n-i} , 0<= i <= n
```

Polynomial Representation

- Version 2: plain-dynamic
  - Dynamically allocate coefficients, no need to know max degree
  - Waste memory if most of coefficients are zero

$$a(x) = \sum a_i x^i$$

```
private:
   int degree;
   float *coef;

Polynomial:Polynomial(int d)
{
   degree = d;
   coef = new float[degree+1];
}
```

Polynomial Representation

- Version 3: nonzero-dynamic
  - Dynamically allocate coefficients & exponents
  - Store only nonzero terms
  - Waste memory if most of coefficients are nonzero

$$a(x) = \sum a_i x^i$$

```
class Term {
private:
   float coef;
   int exp;
};
private:
   Term *termArray;
   int capacity, terms;
```

Example

- $a(x) = 2x^{1000} + 1$
  - Plain-dynamic uses 1 int & 1001 float = 1002 words
  - Nonzero-dynamic uses 4 int (2+2) & 2 float = 6 words
- $a(x) = 3x^4 + 6x^3 - 5x^2 + 2x - 1$
  - Plain-dynamic uses 1 int & 5 float = 6 words
  - Nonzero-dynamic uses 7 int (2+5) & 5 float = 12 words
  - If all terms are nonzero, nonzero-dynamic uses about twice more memory (b/c exponent is stored explicitly)

Polynomial Add

- C = A + B
- Add terms that have the same exponent
- Only store nonzero terms to output

```
Polynomial Polynomial::Add(Polynomial B){
   Polynomial C;  int a = Start; int b = B.Start;
   C.Start = free;  float c;
  while ((a<=Finish) && (b<=B.Finish))
       switch (compare(termArray[a].exp, termArray[b].exp))  // compare exponent
       {
             case '=':
                 c = termArray[a].coef + termArray[b].coef;
                  if (c) C.NewTerm(c, termArray[a].exp); // if sum is nonzero
                    a++; b++;          break;
             case '<':
                  C.NewTerm(termArray[b].coef, termArray[b].exp);
                  b++;               break;
             case '>':
                  C.NewTerm(termArray[a].coef, termArray[a].exp);
                  a++;
       }
       for (; a<=Finish; a++)
          C.NewTerm(termArray[a].coef, termArray[a].exp);
       for (; b<=B.Finish; b++)
          C.Newterm(termArray[b].coef, termArray[b].exp);
       C.Finish = free - 1;    return C;
}
```

Matrices

- A[m][n]
  - m x n matrix A
  - m : # of rows
  - n : # of column
  - m*n : # of elements
- Accessing an element at i-th row and j-th col
  - A[i][j]
- 2D array is often used to represent matrices

Row Major vs. Column Major

- 2D->1D layout
- Row major
  - Store each row in contiguous memory
- Column major
  - Store each column in contiguous memory

Sparse Matrices

- Mostly zero

$$
\begin{array}{c}
\begin{array}{cccc} & 0 & 1 & 2 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}
\end{array}
$$

(a) Dense Matrix

$$
\begin{array}{c}
\begin{array}{ccccccc} & 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}
\end{array}
$$

(b) Sparse Matrix

Sparse Matrix Representation

- Array of triples <row, col, value> for nonzero elements

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

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

Sparse Matrix Representation

- Space requirement for nxn matrix
  - Dense : n^2
  - Sparse diagonal matrix : 3*n
  - Sparse tridiagonal matrix : 9*n - 6

$$f_n = \begin{vmatrix} a_1 & b_1 & & & \\ c_1 & a_2 & b_2 & & \\ & c_2 & \ddots & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & c_{n-1} & a_n \end{vmatrix}.$$

Example

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 15 & 0 & 0 & 22 & 0 & -15 \\ 1 & 0 & 11 & 3 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & -6 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 91 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 28 & 0 & 0 & 0 \end{array}$$

smArray		row	col	val
	[0]	0	0	15
	[1]	0	0	0
	[2]	0	0	0
	[3]	0	0	0
	[4]	0	0	0
	[5]	0	0	0
	[6]	0	0	0
	[7]	0	0	0

Row-major ordering

Example

$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & & & & & & \\
1 & & & & & & \\
2 & & & & & & \\
3 & & & & & & \\
4 & & & & & & \\
5 & & & & & &
\end{array}
\quad
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
$$

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

Row-major ordering

Sparse Matrix Transpose

- Exchange (i,j) to (j,i)
- For each row i, store (i,j) to (j,i)
  - Problem

    (0, 0, 15) -> (0, 0, 15)

    (0, 3, 22) -> (3, 0, 22)

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

    (1, 1, 11) -> (1, 1, 11) X

Violate row-major order

Sparse Matrix Transpose

- Algorithm
  - Use column major order traversal
    - After transpose, matrix is row-major order

```
for (all element in column j)
  store (i,j,value) of the original matrix as (j,i,value) of transposed matrix
```

Example

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

		row	col	val
smArray	[0]	0	0	15
	[1]	0	4	91
	[2]			
	[3]			
	[4]			
	[5]			
	[6]			
	[7]			

for Column 0

Example

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

		row	col	val
smArray	[0]	0	0	15
	[1]	0	4	91
	[2]	1	1	11
	[3]			
	[4]			
	[5]			
	[6]			
	[7]			

for Column 1

Example

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

		row	col	val
smArray	[0]	0	0	15
	[1]	0	4	91
	[2]	1	1	11
	[3]	2	1	3
	[4]	2	5	28
	[5]			
	[6]			
	[7]			

for Column 2

Example

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

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

for Column 3

Example

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

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

for Column 4 : No Match

Example

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

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

for Column 5

```
SparseMatrix SparseMatrix::Transpose(){
    SparseMatrix b;        // b : output (transposed matrix)
    b.Rows = Cols;         // b rows = a cols
    b.Cols = Rows;         // b cols = a rows
    b.Terms = Terms;       // # of elements is same

    if (Terms > 0){        // non empty matrix
        int CurrentB = 0;
cols+1                for (int c = 0; c < Cols; c++)      // per each column
cols*(terms+1)         for (int i = 0; i < Terms; i++)   // for all nonzero elements in sparse
cols*terms             if (smArray[i].col ==c) {         // matrix
cols*(0 ~ terms)          b.smArray[CurrentB].row = c;
                          b.smArray[CurrentB].col = smArray[i].row;
                          b.smArray[CurrentB].value = smArray[i].value;
                          CurrentB++;
        }
    }
    return b;
}
```

Complexity?

Sparse Matrix Transpose

- Fast algorithm ($A^T=B$)
  - Count elements in each column in A (# of rows in B)
  - Compute starting position of rows in B
  - Copy A from B

	ROW_SIZE	ROW_START
[0]	2	0
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7
	# of nonzero terms in B's row (A's col)	starting position of B's row

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

Example

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

	ROW_SIZE	ROW_START
[0]	2	0
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7
	# of terms in B's row (A's col)	starting position of B's row

RowStart[0] = 0;
RowStart[0]++;

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

for Column 0

Example

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

	ROW_SIZE	ROW_START
[0]	2	1
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7
	# of terms in B's row (A's col)	starting position of B's row

RowStart[3] = 5;
RowStart[3]++;

	row	col	val
[0]	0	0	15
[1]			
[2]			
[3]			
[4]			
[5]	3	0	22
[6]			
[7]			

for Column 0

Example

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

	ROW_SIZE	ROW_START
[0]	2	1
[1]	1	2
[2]	2	3
[3]	2	~~5~~ 6
[4]	0	7
[5]	1	7
	# of terms in B's row (A's col)	starting position of B's row

RowStart[5] = 7;
RowStart[5]++;

	row	col	val
[0]	0	0	15
[1]			
[2]			
[3]			
[4]			
[5]	3	0	22
[6]			
[7]	5	0	-15

for Column 0

```
          SparseMatrix SparseMatrix::FastTranspose(){
           // transpose a(*this) to b, O(terms+columns).
            int *RowSize = new int[Cols];
            int *RowStart = new int[Cols];
            SparseMatrix b;
            b.Rows = Cols;       b.cols = Rows;   b.Terms = Terms;

            if (Terms>0) {
cols+1          for (int i = 0; i<Cols; i++)    RowSize[i] = 0;     // initialize
terms+1         for (i = 0; i<Terms; i++)      RowSize[smArray[i].col]++;
1               RowStart[0] = 0;
cols+1          for (i =1; i<Cols; i++)   RowStart[i] = RowStart[i-1] + RowSize[i-1];

terms+1         for (i = 0; i<Terms; i++) {      // transpose
                     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]++;
              }
              delete [] RowSize;  delete [] RowStart;
              return b;
          }
```

Complexity?

Questions?