2.4 Matrix

2.4 Matrix

- Denote a matrix consists of ***m rows*** and ***n columns*** as A_{m*n} (read A is a ***m by n*** matrix).
- Usually stored as a two-dimensional array, $a[m][n]$, in which element at i[th] row and j[th] column is accessed by $a[i][j]$.
- $$A_{5*3} = \begin{pmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{pmatrix}$$

Columns: col 0, col 1, col 2; rows: row 0, row 1, row 2, row 3, row 4

Matrix Operations

- Transpose
  - $C_{nxm} = A^T_{mxn}$
  - $c[i][j] = a[j][i]$
- Addition
  - $C_{mxn} = A_{mxn} + B_{mxn}$
  - $c[i][j] = a[i][j] + b[i][j]$
- Multiplication
  - $C_{mxp} = A_{mxn} + B_{nxp}$
  - $c[i][j] = \sum_{k=0}^{n-1} a[i][k] \times b[k][j]$

Matrix: ADT

```
class Matrix{
public:
    // Construct
    Matrix(int r, int c);
    // Return the transpose of (*this) matrix
    Matrix Transpose(void);
    // Return sum of *this and b
    Matrix Add(Matrix b);
    // Return the multiplication of *this and b
    Matrix Multiply(Matrix b);
private:
    // Array representation
     int **a, rows, cols;
};
```

18

Transpose : Code

```
Matrix Matrix::Transpose(void){
   Matrix c(cols, rows);
   for (i=0; i<rows; i++)          // O(rows)
      for (j=0; j<cols; j++)       // O(cols)
         c[j][i]=a[i][j];
   return c;
}
```

- Time complexity: O(rows · cols)

19

Add: Code

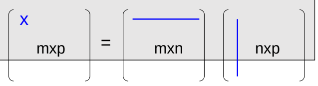
```
Matrix Matrix::Add(Matrix b){
   Matrix c(rows, cols);
   for (i=0; i<rows; i++)          // O(rows)
      for (j=0; j<cols; j++)       // O(cols)
         c[i][j]=a[i][j]+b[i][j];
   return c;
}
```

- Time complexity: O(rows · cols)

20

Multiply: Code

```
Matrix Matrix::Multiply(Matrix b){
   Matrix c(rows, b.cols);
   for (i=0; i<rows; i++) {            // O(rows)
      for (j=0; j<b.cols; j++) {       // O(b.cols)
          sum=0;
          for (k=0; k<cols; k++)       // O(cols)
              sum += a[i][k]*b[k][j];
          c[i][j]=sum;
      }
  }
  return c;
}
```



- Time complexity: O(rows · cols · b.cols)

21

2.4.2 Sparse Matrix

a[6][6] =

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

- A matrix has few non-zero elements.
- 2D array representation is inefficient.
  - Wasteful **memory** and **computing time**
  - Consider a matrix $A_{5000X5000}$ with only 100 nonzero elements!

22

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 =

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

One Linear List Per Row

0 0 3 0 4 row1 = [(3, 3), (5,4)]
0 0 5 7 0 row2 = [(3,5), (4,7)]
0 0 0 0 0 row3 = []
0 2 6 0 0 row4 = [(2,2), (3,6)]

Sparse Matrix Representation

- We use an array, ***smArray[]***, of ***triple*** **<*row, col, value*>** to store those nonzero elements.
- Triples are stored in a ***row-major*** order.

a[6][6] =

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

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

25

2.4.2
ADT2.4

Sparse Matrix: ADT

```
class SparseMatrix{
public:
    // Construct, t is the capacity of nonzero terms
    SparseMatrix(int r, int c, int t);
    // Return the transpose of (*this) matrix
    SparseMatrix Transpose(void);
    // Return sum of *this and b
    SparseMatrix Add(SparseMatrix b);
    // Return the multiplication of *this and b
    SparseMatrix Multiply(SparseMatrix b);
private:
    // Sparse representation
     int rows, cols, terms, capacity;
     MatrixTerm *smArray;
};
```

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

26

Approximate Memory Requirements

- 5000 x 5000 matrix with 100 nonzero elements, 4 bytes per element
- 2D array
  - 5000 x 5000 x 4 = 100 million bytes
- Class SparseMatrix
  - 100 x 4 x 3 + 4 = 1204 bytes

2.4.3

Trivial Transpose

- $c[i][j] = a[j][i]$

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

Transpose

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

- Problem: the nonzero terms in A^T are no longer stored in row major order!

28

Smart Transpose

Because the row and column are swapped, we trace the nonzero terms in a **column-major** order.

```
For(all non-zero elements in column j)
  Store a(i,j,value) as aT(j,i,value)
```

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

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

29

Smart Transpose: Code

```
SparseMatrix SparseMatrix::Transpose()
{ // Return the transpose of (*this) matrix
  // b.smArray has the same number of nonzero terms
  SparseMatrix b(cols, rows, terms);
  if (terms > 0) // has nonzero terms
  {
    int currentB = 0;
    for(int c=0; c<cols; c++)      // O(cols)
      for(int i=0; i<terms; i++)  // O(terms)
        if(smArray[i].col == c)
        {
          b.smArray[currentB].row = c;
          b.smArray[currentB].col = smArray[i].row;
          b.smArray[currentB++].value = smArray[i].value;
        }
  }
  return b;
}
```

30

Fast Transpose

- Examine all terms only twice!
- Use additional space to store
  - rowSize[i]: # of nonzero terms in i^{th} row of A^T
  - rowStart[i]: location of nonzero term in i^{th} row of A^T
  - For i>0, rowStart[i]=rowStart[i-1]+rowSize[i-1]
- Copy element from A to A^T one by one.
- Time complexity: O(terms + cols)!

31

Fast Transpose

- Count the # of nonzero terms in each row of A^T
- Calculate the location of 1^{st} nonzero term i^{th} row of A^T

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

col	rowSize	rowStart
[0]	2	0
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7

32

Fast Transpose

- Copy element from A to A^T one by one

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

col	rowSize	rowStart
[0]	2	0
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			
smArray[5]			
smArray[6]			
smArray[7]			

33

Fast Transpose

- Copy element from A to A^T one by one

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

col	rowSize	rowStart
[0]	2	1
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			
smArray[5]			
smArray[6]			
smArray[7]			

34

Fast Transpose

- Copy element from A to A^T one by one

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

col	rowSize	rowStart
[0]	2	1
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			
smArray[5]			
smArray[6]			
smArray[7]			

35

Fast Transpose

- Copy element from A to A^T one by one

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

col	rowSize	rowStart
[0]	2	1
[1]	1	2
[2]	2	3
[3]	2	6
[4]	0	7
[5]	1	7

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			
smArray[5]	3	0	22
smArray[6]			
smArray[7]			

36

Fast Transpose

- Copy element from A to A^T one by one

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

col	rowSize	rowStart
[0]	2	1
[1]	1	3
[2]	2	4
[3]	2	7
[4]	0	7
[5]	1	8

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]	0	4	91
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]			
smArray[5]	3	0	22
smArray[6]	3	2	-6
smArray[7]	5	0	-15

37

Fast Transpose

- Copy element from A to A^T one by one

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

col	rowSize	rowStart
[0]	2	2
[1]	1	3
[2]	2	4
[3]	2	7
[4]	0	7
[5]	1	8

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]	0	4	91
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]			
smArray[5]	3	0	22
smArray[6]	3	2	-6
smArray[7]	5	0	-15

38

Fast Transpose

- Copy element from A to A^T one by one

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

col	rowSize	rowStart
[0]	2	2
[1]	1	3
[2]	2	4
[3]	2	7
[4]	0	7
[5]	1	8

A^T	row	col	value
smArray[0]	0	0	15
smArray[1]	0	4	91
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]	2	5	28
smArray[5]	3	0	22
smArray[6]	3	2	-6
smArray[7]	5	0	-15

39

Fast Transpose: Codes

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

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

    for(int i=0; i<terms; i++)
    { // copy terms 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]++; // Increase the start pos by 1
    }
    delete [] rowSize;
    delete [] rowStart;
  }
  return b;
}
```

40

Computation Time Comparison

Trivial Transpose	Smart Transpose	Fast Transpose
$O(rows \cdot cols)$	$O(cols \cdot terms)$	$O(terms + cols)$

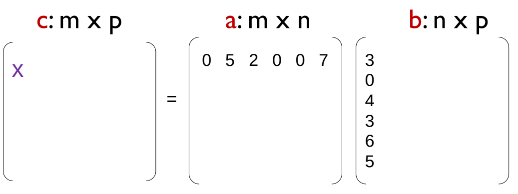
- For a dense matrix (terms $= rows \cdot cols$)
  Fast equals to **Trivial**: $O(rows \cdot cols)$
  Smart is the slowest: $O(rows \cdot cols^2)$
- For a sparse matrix (terms $\ll rows \cdot cols$)
  - **Fast** is faster than **Trivial** and **Smart** ones

41

2.4.4

Sparse Matrix Multiplication

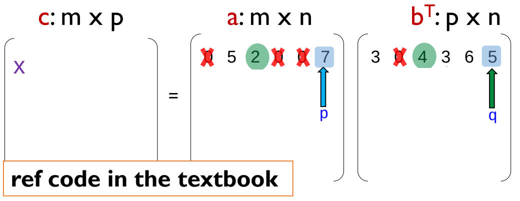
- Compute the transpose of b



42

Sparse Matrix Multiplication

- Use approach similar to "**Polynomial Addition**" to compute the X!



ref code in the textbook

x =(2)(4) + (7)(5)= 43

43

Time Complexity

```
SparseMatrix SparseMatrix::Multiply(SparseMatrix b)
{ // Compute the transpose of b
  SparseMatrix bT = b.FastTranspose(); // O(b.terms+b.cols)

  for ith row in smArray                    // O(rows)
    for jth row in bT.smArray               // O(b.cols)
       Perform "Polynomal Addition"         // O(Terms[i]+b.Terms[j])
}
```

- Complexity:
  - O(rows · b.cols · (Terms[i] + b.Terms[j]))
  - rows · Terms[i] = a.terms and b.cols · b.Terms[j] = b.terms
  - O(rows · b.terms + b.cols · a.terms)

44