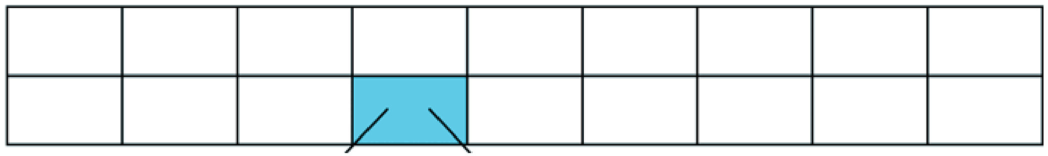
Arrays

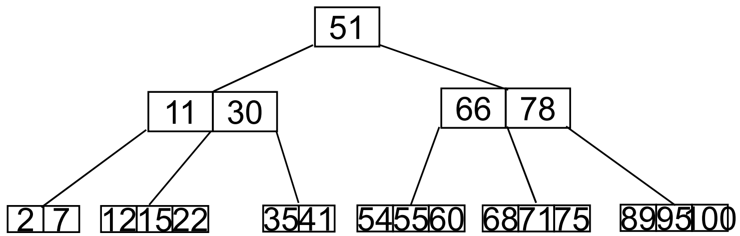
Yi-Shin Chen
Institute of Information Systems and Applications
Department of Computer Science
National Tsing Hua University
yishin@gmail.com

Basic Data Structures

- Homogeneous/Heterogeneous array



- List
  - Stack
  - Queue
- Tree



Definition of Array

- A data structure representing a linear list
  - Elements could be the same or different data types
- Examples:
  - Days of the week: {Sunday, Monday, …, Saturday}
  - Deck of cards: {Ace, 2, 3, …, King}
  - Phone Book: {(James, 31212), (Claire, 31213), …, (Tony, #99999)}

Common Operations

- ADT array[n]={a_0, a_1 ,…, a_{n-1}}
  - Find the length, n, of the array.
  - Read the array from left to right (or reverse).
  - Retrieve the i^{th} element, $0 \le i < n$.
  - Store a new element into i^{th} position , $0 \le i < n$.
  - Insert / delete the element at position i , $0 \le i < n$.

Array Representations

- Sequential mapping
  - Element a_i is stored in the location i of the array
  - The most commonly used
  - Efficient random access
- Non sequential mapping
  - Carry out insertion and deletion efficiently
  - E.g. Linked Lists in chapter 4

Building an ADT for Polynomials

$$p(x) = a_0 x^{e_0} + a_1 x^{e_1} + \cdots + a_n x^{e_n} = \sum_{i=0}^{n} a_i x^{e_i}$$

- Each $a_i x^{e_i}$ is called a term with coefficient a_i
- The **degree** of $p(x)$ is the largest exponent from among the non-zero terms
- Example:
- Ex. $p(x) = x^5 + 4x^3 + 2x^2 + 1$
  - Has 4 terms with coefficients 1, 4 ,2 and 1
  - The degree of $p(x)$ is 5
- Array representation
  - Store (a_i, e_i) as (array[n-i], i) pair and n is the degree

Polynomial Operations

$$a(x) = \sum a_i x^i \;\; and \;\; b(x) = \sum b_i x^i$$

- Polynomial addition
  - $a(x) + b(x) = \sum (a_i + b_i) x^i$
- Polynomial multiplication
  - $a(x) \cdot b(x) = \sum \left(a_i x^i \cdot \sum \left(b_j x^j \right) \right)$
- Examples
  - $a(x)=x^5+4x^3+2x^2+1$ (degree = 5)
  - $b(x)=3x^6+4x^3+x$ (degree = 6)
  - $a(x) + b(x) = 3x^6+x^5+8x^3+2x^2+x+1$ (degree = 6)

Polynomial : ADT

```
class Polynomial {
public:
    // Construct p(x) = 0
    Polynomial(void);
    // Destructor
    ~Polynomial(void);
    // Return the sum of *this and poly
    Polynomial Add(Polynomial poly);
    // Return multiplication of *this and poly
    Polynomial Mult(Polynomial poly);
    // Return the evaluation result
    float Eval(float f );
private:
    // Array representation
     …
};
```

Polynomial: 1^{st} Representation

```
// in class Polynomial
public:
    // degree ≤ MaxDegree
    int degree;
    // coefficient array
    float coef[MaxDegree+1];
```

```
Usage:
    Polynomial a;
     a.degree = n;
     a.coef[i] = a_{n-i}
```

- Coefficients are stored in order of decreasing exponents
- Advantages:
  - Easy to implement operations
- Disadvantages:
  - Waste memory in a sparse polynomial

Polynomial: 2^{nd} Representation

```
class Term {
 friend Polynomial;
 float coef;
 int exp;
};
```

```
// in class Polynomial
private:
  // array of nonzero terms
  Term* termArray;
  int capacity; // size of termArray
  int terms; // number of nonzero terms
```

- Store only nonzero terms
  - Each nonzero term holds an exponent and its corresponding coefficient
- Advantages:
  - If polynomial is sparse, 2^{nd} representation is better
- Disadvantages:
  - If polynomial is full, 2^{nd} one has double size of 1^{st}

Polynomial Addition: Codes

```
Polynomial Polynomial::Add(Polynomial b)
{ // Return sum of polynomial *this and b
  Polynomial c;
  int aPos = 0, bPos = 0;
  while((aPos < terms) && (bPos < b.terms))
    if(termArray[aPos].exp == b.termArray[bPos].exp){
        float t = termArray[aPos].coef + b.termArray[bPos].coef;
        If(t) c.NewTerm(t, termArray[aPos].exp);
        aPos++; bPos++;}
    else if(termArray[aPos].exp < b.termArray[bPos].exp){
        c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
        bPos++;}
    else{
        c.NewTerm(termArray[aPos].coef,termArray[aPos].exp);
        aPos++;}
  // add in remaining terms of *this
  for(; aPos < terms; aPos++)
    c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
  // add in remaining terms of b
  for(; bPos < b.terms; bPos++)
    c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
  return c;}
```

An Running Example

$$a(x) = x^5 + 9x^4 + 7x^3 + 2x$$

$$b(x) = x^6 + 3x^5 + 6x + 3$$

$$\begin{aligned} c(x) &= x^6 + (1+3)x^5 + 9x^4 + 7x^3 + (2+6)x + 3 \\ &= x^6 + 4x^5 + 9x^4 + 7x^3 + 8x + 3 \end{aligned}$$

Time Complexity of Analysis

- Inside the while loop: every statement has O(1) time
- How many times the "while loop" is executed in the worst case ?
  - Let a(x) have m terms, and
    b(x) have n terms.
  - In each iteration, we access next element
    in a(x) or b(x), or both.
  - Worst case: m + n.
    eg. It happens when
    $A(x) = 7x^5 + x^3 + x$; $B(x) = x^6 + 2x^4 + 6x^2 + 3$
    Access remaining terms in A(x): O(m)
    Access remaining terms in B(x): O(n)
- Hence, total running time = O(m + n)

Matrix

- A matrix A_{mxn} (read A is a *m by n* matrix) consists of
  - *m rows*
  - *n columns*
- Stored as a two dimensional array, a[m][n]
  - element at i^{th} row and j^{th} column could be accessed by a[i][j]

$$
\begin{array}{c}
\begin{array}{ccc} \text{col 0} & \text{col 1} & \text{col 2} \end{array} \\
\left(\begin{array}{rrr} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{array}\right)
\begin{array}{l} \text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \end{array}
\end{array}
$$

Matrix Operations

- Transpose
  - $C_{n \times m} = A^t_{m \times n}$
  - $c[i][j] = a[j][i]$
- Addition
  - $C_{m \times n} = A_{m \times n} + B_{m \times n}$
  - $c[i][j] = a[i][j] + b[i][j]$
- Multiplication
  - $C_{m \times p} = A_{m \times n} + B_{n \times p}$
  - $c[i][j] = \sum_{k=0}^{n-1} a[i][k] \times b[k][j]$

For more information, check the videos on the course webpage
Or, click https://youtu.be/kYB8IZa5AuE

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;
};
```

Transpose : Codes

```
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)

Add: Codes

```
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)

Multiply: Codes

```
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)

Sparse Matrix

a[6][6] =

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

- A matrix has many zero elements
  - E.g., a large matrix $A_{5000X5000}$ which has only 100 nonzero elements
- 2D array representation is inefficient
  - Waste both memory and running time to store and compute those zero elements

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;
};
```

Trivial Transpose

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

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

Transpose →

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

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

Smart Transpose

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

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

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

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

Smart Transpose : Codes

```
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;
}
```

- Time complexity: O(cols · terms).
- It can be faster!

Fast Transpose

- We need to examine all terms only once!
- 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)!

Fast Transpose

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

Count the # of nonzero terms in each row of A^T

Calculate rowstart[i]=rowSize[i-1]+rowStart[i-1]

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

A^T	rowSize	rowStart
[0]		
[1]		
[2]		
[3]		
[4]		
[5]		

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

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;}
```

Running Time Comparison

Trivial Transpose	Smart Transpose	Fast Transpose
O(rows · cols)	O(cols · terms)	O(cols + terms)

- For a dense matrix (terms = rows·cols)
  - Fast equals to trivial: O(rows · cols)
  - Smart is slowest: O(rows · $cols^2$)
- For a sparse matrix (terms << rows·cols)
  - Fast transpose is faster than trivial and smart ones

Sparse Matrix Multiplication

- Compute the transpose of b

$$
\underbrace{\begin{pmatrix} x & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix}}_{c:\ m \times p}
=
\underbrace{\begin{pmatrix} 0 & 5 & 2 & 0 & 0 & 7 \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix}}_{a:\ m \times n}
\underbrace{\begin{pmatrix} 3 & & & & & \\ 0 & & & & & \\ 4 & & & & & \\ 3 & & & & & \\ 6 & & & & & \\ 5 & & & & & \end{pmatrix}}_{b:\ n \times p}
$$

Sparse Matrix Multiplication

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

c: m x p	a: m x n	b^T: p x n
$\begin{pmatrix} x & & & & & \\ & & & & & \\ & & & & & \end{pmatrix}$	$\begin{pmatrix} 0 & 5 & 2 & 0 & 0 & 7 \\ & & & & & \\ & & & & & \end{pmatrix}$	$\begin{pmatrix} 3 & 0 & 4 & 3 & 6 & 5 \\ & & & & & \\ & & & & & \end{pmatrix}$

Please refer textbook for codes!

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