國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 2 Arrays

Outline

- ➡ Abstract Data Types and Class
- The Array as an Abstract Data Type
- The Polynomial
- Sparse Matrix
- The String

A Class

- **Four components of a class**
  - **A class name**
  - **Data members**
  - **Member functions**
  - **Levels of program access**
    - **Public: data or functions can be accessed from anywhere**
    - **Protected: accessed from within its class, from its sub-class, or from a friend class**
    - **Private: accessed from within its class or by a friend class**

Definition of a Class for Rectangle

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
// In the header file Rectangle.h
class Rectangle {
public:        //  the following members are public
   // The next four members are member functions
   Rectangle();       // constructor
   ~Rectangle();      // destructor
   int GetHeight();   // returns the height of the rectangle
   int GetWidth();    // returns the width of the rectangle
private:           // the following members are private
   // the following members are data members
   int x1, y1, h, w;
   // (x1, y1) are the coordinates of the bottom left corner of the rectangle
   // w is the width of the rectangle; h is the height of the rectangle
};
#endif
```

Special Class Operations

- **Constructor**
  - Is a member function which initializes data members of an object
  - If provided, it is automatically executed when an object of that class is created
  - If not provided, data members are not properly initialized
- **Destructor**
  - is member function which deletes data members immediately before the object disappears
  - Invoked automatically when a class object goes out of scope or explicitly deleted

---

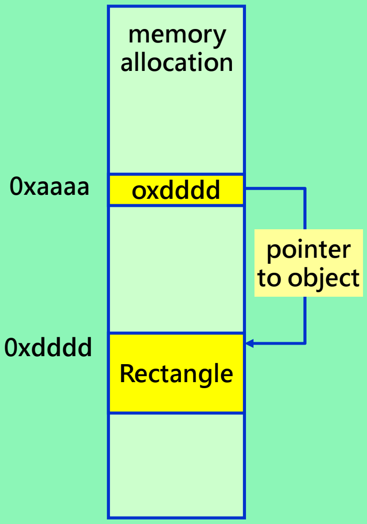
Implementation of Operations on Rectangle

```
// In the source file Rectangle.C
#include “Rectangle.h”

// The prefix “Rectangle::” identifies GetHeight() and GetWidth()
// as member functions belonging to class Rectangle. It is required
// because the member functions are implemented outside their
// class definition

int  Rectangle::GetHeight() { return h; }
int  Rectangle::GetWidth() { return w; }
```

Example: Object Usage

```
// In a source file main.C
#include <iostream.h>
#include "Rectangle.h"

main() {
   Rectangle r, s;      // r and s are objects of class Rectangle
   Rectangle *t = &s;   // t is a pointer to class object s
    .
    .
   // use · to access members of class objects.
   // use → to access members of class objects through pointers.
   if (r.GetHeight () * r.GetWidth () > t→GetHeight () * t→GetWidth ())
      cout << " r ";
   else cout << " s ";
   cout << "has the greater area" << endl;
}
```

Object vs. Pointer



```
Rectangle r, s;
Rectangle *t = &s;
```

symbol table

name	type	address
s	Rectangle	0xdddd
t	pointer	0xaaaa

Example: Constructor

```
Rectangle::Rectangle(int x, int y, int height, int width)
{
  x1 = x;   y1 = y;
  h = height;   w = width;
}
→ a constructor must be public, and has no return type
```

Example: initializes Rectangle objects:
Rectangle r(1, 3, 6, 6);
Rectangle *s = new Rectangle(0, 0, 3, 4)

Example: Illegal declaration
Once a constructor is defined, proper input arguments for initialization must be provided
→ otherwise, it is a compile-time error

Efficient Yet Sophisticated Constructor

```
Rectangle::Rectangle(int x=0, int y=0, int height=0, int
width=0): x1(x), y1 (y), h (height), w (width)
{}
```

The data members are initialized by using a member initialization list
(i.e., colon followed by a list of data members and the arguments to
which they are to be initialized in parentheses)
→ Directly initializes the data members in a single step

Operator Overloading

- **Overload operators for user-defined data types**
  - **Is allowed in C++**
  - **Takes the form of a class member function or an ordinary function**

```
int Rectangle::operator==(const Rectangle& s)
{
   if(this == &s) return(1);  // check if two objects are the same
   if( (x1 == s.x1) && (y1 == s.y1) && (h == s.h) (w == s.w) {
                 return(1);
   }
          else  return(0);
}
```

"*this*" is the pointer to the data object upon which the operator is performed

---

Example: Overload Operator==

```
#include <iostream.h>
class complex{
   public:
      complex(int re, int im){ real = re; imaginary = im; }
      int         get_real(){            return(real); }
      int         get_imaginary(){   return(imaginary); }
     int          operator == (complex x){
                       if(real == x.get_real() && imaginary == x.get_imaginary())
                             return(1);
                       else  return(0);
                  }
   private:
      int   real;      int  imaginary;
};

main(){
      int         i;
      complex  a(1, 2), b(3, 4);
     cout << (a == b);
}
```

Example: Overload Operator<<

```
ostream& operator<<(ostream& os, Rectangle& r)
{
    os << “Position is: “ << r.x1 << “ “;
    os << r.y1 << endl;
    os << “Height is: “ << r.h << endl;
    os << “Width is: “ << r.w << endl;
    return os;
}
```

Operator<< accesses private data members of class Rectangle
→ therefore, it must be made a friend of Rectangle.
Note that: friend is an exception of data encapsulation, should be avoided in most cases. But sometimes it is necessary as in this case.

Example: cout << r;
→ Position is: 1 3
→ Height is: 6
→ Width is: 6

Example: Overload Operator<<

```
#include "iostream.h"
class complex{
public:
       complex(int re, int im){ real = re; imaginary = im; }
       int         get_real(){            return(real); }
       int         get_imaginary(){  return(imaginary); }
       friend     ostream& operator<<(ostream&, complex);
private:
       int   real;     int  imaginary;
};
ostream& operator<<(ostream& os, complex x){
       os << x.get_real() << endl;
       os << x.get_imaginary() <<endl ;
      return(os);
}
main(){
       int          i;
       complex  a(1, 2), b(3, 4);
      cout << a << b;   // will be illegal if the return type of << is not ostream&
}
```

declare operator<< as a friend operator

Miscellaneous Topics

- **Struct**
  - **Is identical to a class, except that the default level of access is public.**
  - **In a class, the default is private**
- **Union**
  - **A struct that reserves storage for the largest of its data members**
  - **Useful for applications where only one of many possible data items need to be stored at any time**
- **Static class data member**
  - **May be thought of as a global variable for its class**
  - **A definition of the data member outside the class definition is required**

Example of Union

```
struct pair {
      int   n1; // 32 bits
      float  n2; // 32 bits
}
```

A *pair* requires 2x32 bits = 64 bits

```
struct exclusive_pair {
      union {
          int   n1;  // 32 bits
          float  n2; // 32 bits
      };
}
```

An *exclusive_pair* contains only
an integer or a floating point number
→ requires only 32 bits

Example of Using Union

```
#include <iostream.h>
typedef  struct  _exclusive_pair {
     union {
          int      n1;
          int      n2;
     }
} ex_pair;

main()
{
    ex_pair p;
    p.n1 = 1;   p.n2 = 10;
    cout << p.n1 << “  “ << p.n2 ;
}

→(Result):  10  10
```

C++ ADT for Natural Numbers

```
class NaturalNumber {
// An ordered subrange of the integers starting at zero and ending at
// the maximum integer (MAXINT) on the computer
public:
    NaturalNumber Zero( );
    // returns 0

    Boolean IsZero( );
    // if *this is 0, return TRUE; otherwise, return FALSE

    NaturalNumber Add(NaturalNumber y);
    // return the smaller of *this + y and MAXINT;

    Boolean Equal(NaturalNumber y);
    // return TRUE if *this == y; otherwise return FALSE

    NaturalNumber Successor( );
    // if *this is MAXINT return MAXINT; otherwise return *this + 1

    NaturalNumber Subtract(NaturalNumber y);
    // if *this < y, return 0; otherwise return *this – y
};
```

Outline

- Abstract Data Types and Class
- **The Array as an Abstract Data Type**
- The Polynomial
- Sparse Matrices
- The String

Traditional Array

- Array
  - Is often viewed as a consecutive set of memory locations
  - Is a set of ordered pair, i.e., <index, value>
  - Provides two standard operations
    - Store a value to a given index
    - Retrieve a value corresponding to a given index
  - Store and Retrieve are performed in constant time
- A more robust array is needed
  - To avoid out-of-bound access

ADT GeneralArray

```
class GeneralArray {
// objects: A set of pairs <index, value> where for each value of index in IndexSet there
// is a value of type float . IndexSet is a finite ordered set of one or more
// dimensions, for example, {0, ··· , n – 1} for one dimension,
// {(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)} for two dimensions, etc.
public:
   GeneralArray(int j, RangeList list,float initValue = defaultValue);
   // The constructor GeneralArray creates a j dimensional array of floats; the range
   // of the kth dimension is given by the kth element of list. For each index i in the
   // index set, insert <i, initValue> into the array.

   float Retrieve(index i);
   // if (i is in the index set of the array) return the float associated with i
   // in the array; else signal an error.

   void Store(index i, float x);
   // if (i is in the index set of the array) delete any pair of the form <i, y> present
   // in the array and insert the new pair <i, x>; else signal an error.
}; // end of GeneralArray
```

Multi-dimensional Array

- **An n-dimensional array**
  - **Is usually implemented as a one-dimensional array**
  - **A mapping mechanism is needed**
  - **Assumption of $A[u_1][u_2]\ldots[u_n]$**
    - **First index range: $[p_1 .. q_1]$**
    - **Second index range: $[p_2 .. q_2]$**
    - **…**
    - **n-th index range: $[p_n .. q_n]$**
  - **The total number of elements in this n-dimensional array is**

$$\prod_{i=1}^{n} (q_i - p_i + 1)$$

Row Major

- **Row Major Order**
  - Is also called lexicographic order
- **Example**
  - A[4..5] [2..4] [1..2] [3..4]
  - Total number of elements:2*3*2*2 = 24
  - Element order

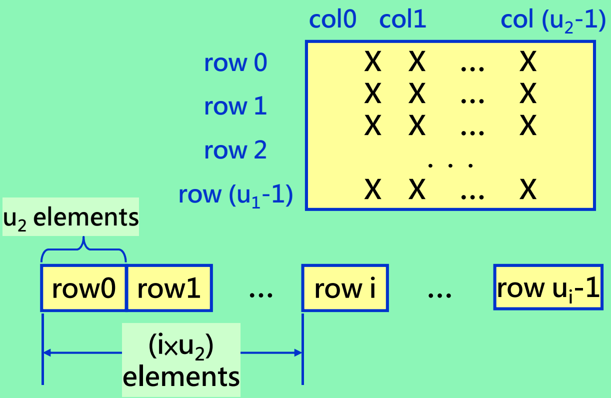
    A[4][2][1][3], A[4][2][1][4], A[4][2][2][3], A[4][2][2][4],
    A[4][3][1][3], A[4][3][1][4], A[4][3][2][3], A[4][3][2][4],
    A[4][4][1][3], A[4][4][1][4], A[4][4][2][3], A[4][4][2][4],
    …

Demonstrations

Sequential representation of $A[u_1]$

Array element:	A[0]	A[1]	A[2]	…	A[i]	…	$A[u_1-1]$
Address:	α	α +1	α+2		α+i-1		α+i

Sequential representation of $A[u_1]\ [u_2]$



Address Mapping Formula

- Assume α is the address of A[0][0][0]
- Address for $A[i][0][0] = \alpha + (i \times u_2\, u_3)$
- Address for $A[i][j][0] = \alpha + (i \times u_2\, u_3) + (j \times u_3)$
- Address for $A[i][j][k] = \alpha + (i \times u_2\, u_3) + (j \times u_3) + k$

$A[i_1]\,[i_2],\ \ldots,\ [i_n]$

(Row major order)

Mapped to

$$\alpha + \sum_{j=1}^{n} i_j\, a_j$$

$$\text{where } a_j = \prod_{k=j+1}^{n} u_k \quad 1 \leqq j \leqq n$$

$$a_n = 1$$

Outline

- Abstract Data Types and Class
- The Array as an Abstract Data Type
- **The Polynomial**
- Sparse Matrices
- The String

ADT of Polynomial

class *Polynomial* **{**
//**objects**: $p(x) = a_0x^{e_0} + \cdots + a_nx^{e_n}$; a set of ordered pairs of $<e_i, a_i>$,
// where $a_i \in$ *Coefficient* and $e_i \in$ *Exponent*
// We assume that *Exponent* consists of integers ≥ 0
public:

Polynomial();
//return the polynomial $p(x) = 0$

int operator!();
//**if *this** is the zero polynomial, return 1; **else** return 0;

Coefficient Coef(*Exponent e*);
//return the coefficient of *e* in ***this**

Exponent LeadExp();
//return the largest exponent in ***this**

Polynomial Add(*Polynomial poly*);
// return the sum of the polynomials ***this** and *poly*

Polynomial Mult(*Polynomial poly*);
// return the product of the polynomials ***this** and *poly*

float *Eval*(**float** *f*) ;
// Evaluate the polynomial ***this** at *f* and return the result.

}; // end of *Polynomial*

Polynomial Representation (1)

- **Fixed-size array representation**

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

- **Example**
  - Assume that *a* is a polynomial class object and *n* ≦ MaxDegree
  - E.g., $a_nx^n + \ldots + a_1x^1 + a_0$
  - Then, *a.degree* = *n* and *a.coef*[*i*] = a_{n-i}, $0 \leqq i \leqq n$

- **Disadvantage of using static array**
  - **Wasteful** in its use of computer memory

Polynomial Representation (2)

- **Array with dynamic size**

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

- **Constructor**

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

- **Advantage**
  - **The size of array can be dynamically decided, leading to a more efficient memory usage**

Polynomial Representation (3)

- **For sparse polynomial**
  - **E.g., $x^{100} + 1$ → as <coef, exp> list → {<1, 100>, <1, 0>}**
- **Shared single array is used**
  - **All polynomials, (if there are many in the program), will be put together as a shared single array**

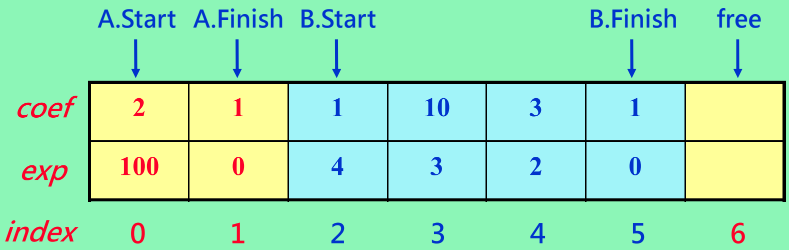
```
class Polynomial; // forward declaration

class term {
   friend Polynomial;
   private:
      float  coef; // coefficient
      int exp; // exponent
};
```

```
// private data members of Polynomial

private:
   static term termArray[MaxTerms];
   static int free;
   int  Start, Finish;
```

```
// static data outside class declaration
term  Polynomial::termArray[MaxTerms];
int Polynomial::free = 0;
          // next free location in termArray
```

Example: Array for Two Polynomials

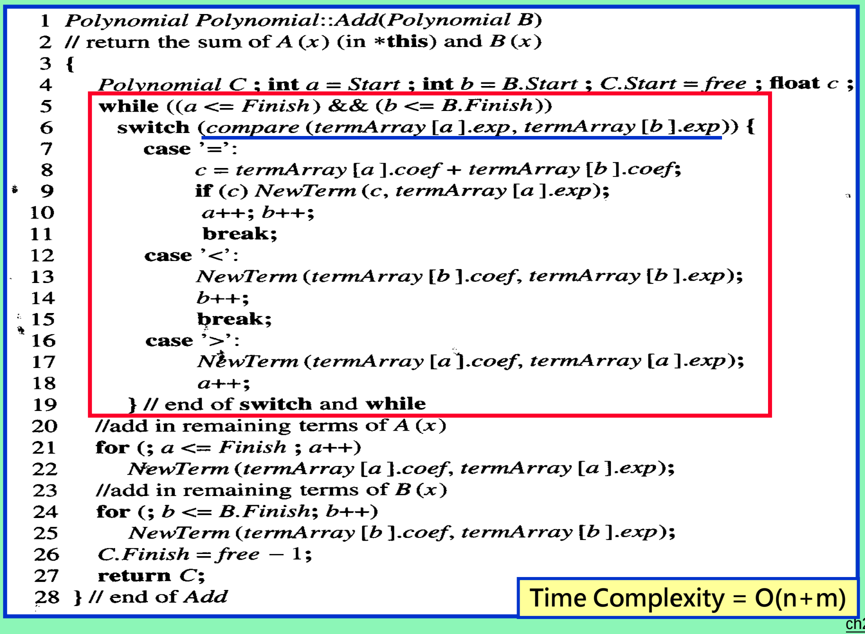
- A(x) = $2x^{100} + 1$
  - A.Start = 0, A.Finish = 1
- B(x) = $x^4 + 10x^3 + 3x^2 + 1$
  - B.Start = 2, B.Finish = 5



coef	2	1	1	10	3	1	
exp	100	0	4	3	2	0	
index	0	1	2	3	4	5	6

A zero polynomial $Z(x)=0$ → *Z.Finish* = *Z.Start*-1

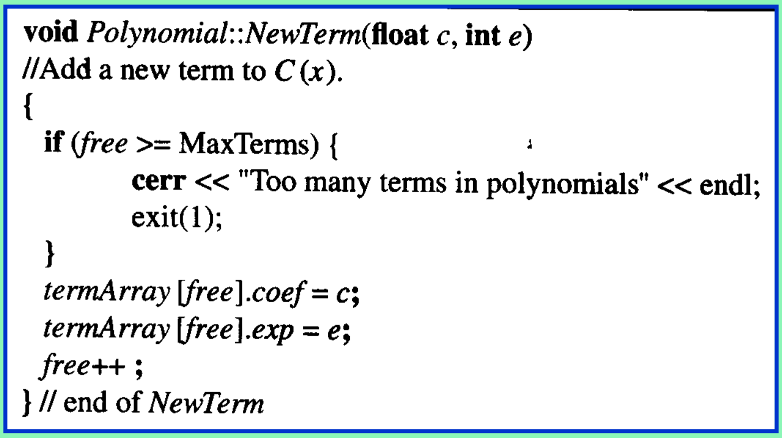
Polynomial Addition



```
Polynomial Polynomial::Add(Polynomial B)
// return the sum of A (x) (in *this) and B (x)
{
    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)) {
         case '=':
                c = termArray [a ].coef + termArray [b ].coef;
                if (c) NewTerm (c, termArray [a ].exp);
                 a++; b++;
                 break;
         case '<':
                NewTerm (termArray [b ].coef, termArray [b ].exp);
                b++;
                break;
         case '>':
                NewTerm (termArray [a ].coef, termArray [a ].exp);
                a++;
      } // end of switch and while
    //add in remaining terms of A (x)
    for (; a <= Finish ; a++)
       NewTerm (termArray [a ].coef, termArray [a ].exp);
    //add in remaining terms of B (x)
    for (; b <= B.Finish; b++)
       NewTerm (termArray [b ].coef, termArray [b ].exp);
    C.Finish = free - 1;
    return C;
} // end of Add
```

Time Complexity = O(n+m)

Adding A New Polynomial Term



```
void Polynomial::NewTerm(float c, int e)
//Add a new term to C(x).
{
  if (free >= MaxTerms) {
           cerr << "Too many terms in polynomials" << endl;
           exit(1);
  }
  termArray[free].coef = c;
  termArray[free].exp = e;
  free++ ;
} // end of NewTerm
```

Disadvantages of Representing Polynomials by Arrays

- **What could happen when the array is used up?**
  - **Recycle certain polynomials no longer needed**
  - **A sophisticated compaction routine is required, involving a lot of data movement**
- **Another Choice**
  - **Use a single array of terms for each polynomial**
  - **Each array is created by using "new"**
  - **This will requires us to know the size of the polynomial prior to its creation**
    - **A potential trouble to run the addition algorithm twice**

Outline

- Abstract Data Types and Class
- The Array as an Abstract Data Type
- The Polynomial
- Sparse Matrices
- The String

How To Store Matrix ?

- Two dimensional array
  - int A[m][n]
  - may require huge memory space, e.g., A[5000][5000]
- Sparse matrix
  - Is a matrix in which most elements are zero
- Use two-dimensional array for sparse matrix is a big waste of memory space

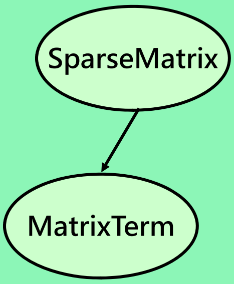
Example of Sparse Matrix

SparseMatrix	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5
Row 0	15	0	0	22	0	-15
Row 1	0	11	3	0	0	0
Row 2	0	0	0	-6	0	0
Row 3	0	0	0	0	0	0
Row 4	91	0	0	0	0	0
Row 5	0	0	28	0	0	0

Sparse Matrix Representation

```
class SparseMatrix; // forward declaration

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

class SparseMatrix {
public: // member functions …
private:
   int  Rows, Cols, Terms;
   MatrixTerm smArray[MaxTerms];
}
```

sparseMatrix is modeled as a linear array



Example of Sparse Matrix and Its Transpose

The sparse matrix is ordered by rows, and within rows by columns

Original	Row	Col	value
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

Transposed	Row	Col	value
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

(0, 3, 22) → (3, 0, 22) after being transposed

Transposing A Matrix

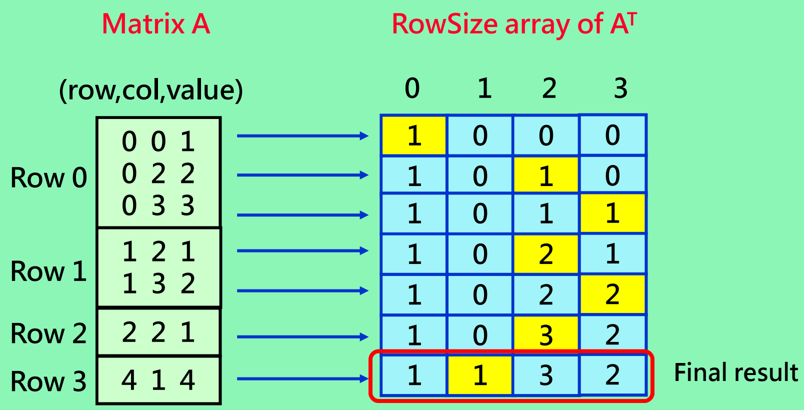
```
SparseMatrix SparseMatrix::Transpose()
// return the transpose of a (*this)
{
  SparseMatrix b;
  b.Rows = Cols;    b.Cols = Rows;   b.Terms = Terms;
  if(Terms > 0) {  // nonzero matrix
    int CurrentB = 0;
    for(int c=0; c < Cols; c++){
        for(int i=0; i<Terms; i++){ // find elements in 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;
               CurrentB++; }
        }
     }
  }
  return(b);
}
```

Time Complexity = O(columns x Terms)

Demo of Transposing a Matrix (1)

Step 1: Scan Matrix A to construct the RowSize array of A^T

(記錄 A^T 每一列有多少 “非零元素”)



Matrix A

(row,col,value)

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

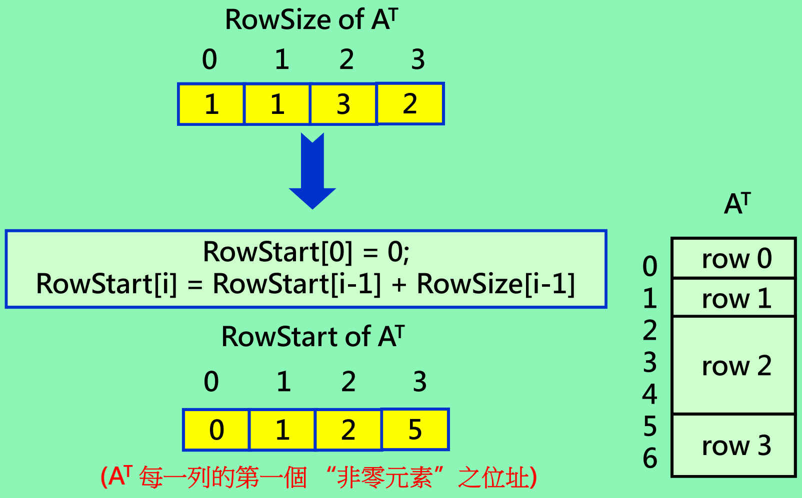
RowSize array of A^T

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

Final result

Demo of Transposing a Matrix (2)

Step 2: Scan RowSize array to construct the RowStart array of A^T



RowSize of A^T

0	1	2	3
1	1	3	2

RowStart[0] = 0;
RowStart[i] = RowStart[i-1] + RowSize[i-1]

RowStart of A^T

0	1	2	3
0	1	2	5

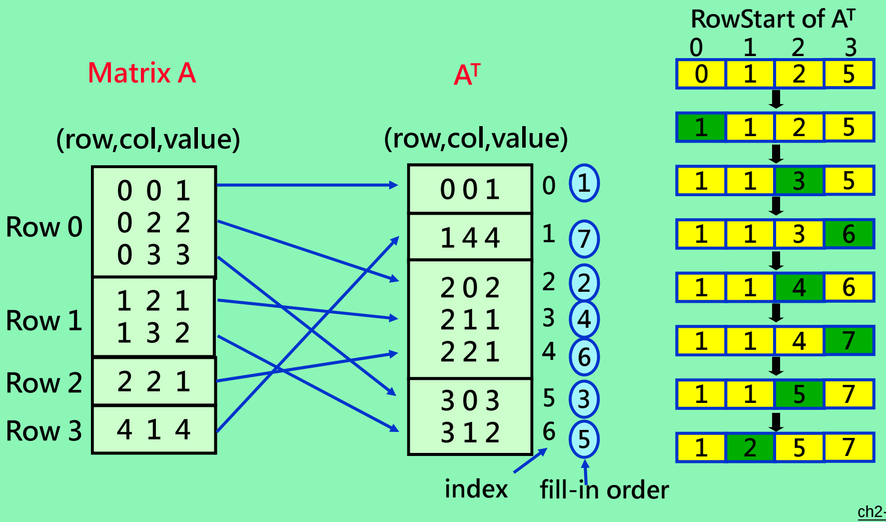
(A^T 每一列的第一個 “非零元素” 之位址)

A^T

0	row 0
1	row 1
2 3 4	row 2
5 6	row 3

Demo of Transposing a Matrix (3)

Step 3: Move each element of Matrix A to Matrix A^T



Matrix A

(row,col,value)

Row 0
0 0 1
0 2 2
0 3 3

Row 1
1 2 1
1 3 2

Row 2
2 2 1

Row 3
4 1 4

A^T

(row,col,value)

index		fill-in order
0	0 0 1	①
1	1 4 4	⑦
2	2 0 2	②
3	2 1 1	④
4	2 2 1	⑥
5	3 0 3	③
6	3 1 2	⑤

index fill-in order

RowStart of A^T

0	1	2	3
0	1	2	5
1	1	2	5
1	1	3	5
1	1	3	6
1	1	4	6
1	1	4	7
1	1	5	7
1	2	5	7

Transposing A Matrix Faster

```
SparseMatrix SparseMatrix::FastTranspose()
{
  int *RowSize = new int[Cols];
  int *RowStart = new int[Cols];
  SparseMatrix b;
  b.Rows = Cols;    b.Cols = Rows;    b.Terms = Terms;
  if(Terms > 0) {  // nonzero matrix

    // compute no. of elements in each row of B
    for(int i=0; i<Cols; i++){ RowSize[i]=0; }
    for(i=0; i<Terms; i++){ RowSize[smArray[i].col]++; }

    // compute the starting position of each row of B
    RowStart[0]=0;
    for(i=1; i<Cols; i++){
          RowStart[i] = RowStart[i-1]+RowSize[i-1];
    }
    TO BE CONTINUED …
```

Transposing A Matrix Faster (con't)

```
SparseMatrix SparseMatrix::FastTranspose()
{
  if (Terms > 0) {
    … (in previous page)
    for(i=0; i<Terms; i++){ // move from a 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]++;
     }
  }
  delete [ ] RowSize;    delete [ ] RowStart;     return(b);
}
```

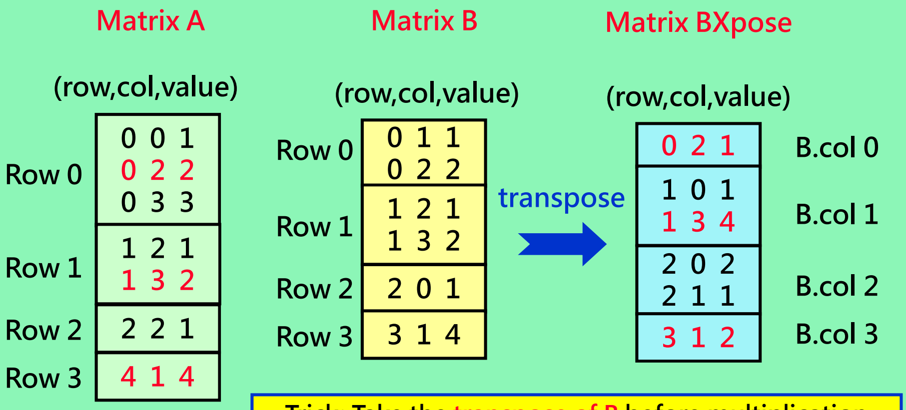
Time Complexity = O(Terms + Columns) ~ O(Terms)

Principles In Fast Transpose

- **Look Before You Leap**
- **A Stitch Beforehand Saves Nine**
- **Quick pre-computation of certain information pays back often**

Matrix Multiplication

$$C_{ij} = (A \times B)_{ij} = \Sigma_k A_{ik} \times B_{kj}$$



Matrix A

(row,col,value)

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

Matrix B

(row,col,value)

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

transpose

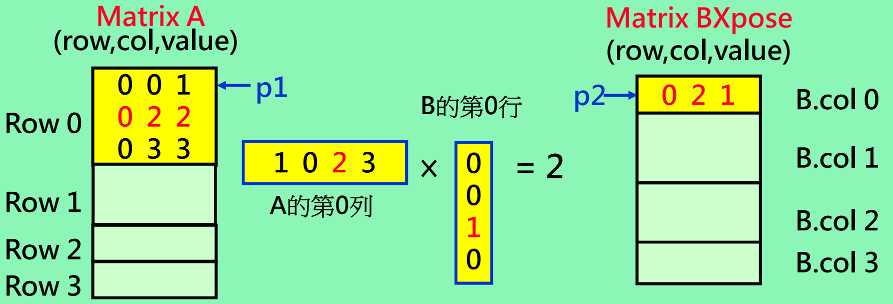
Matrix BXpose

(row,col,value)

0 2 1	B.col 0
1 0 1 1 3 4	B.col 1
2 0 2 2 1 1	B.col 2
3 1 2	B.col 3

Trick: Take the transpose of B before multiplication

Computing C_{00}



Matrix A

(row,col,value)

Row 0	0 0 1 0 2 2 0 3 3
Row 1	
Row 2	
Row 3	

B的第0行

$$\begin{bmatrix} 1 & 0 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = 2$$

A的第0列

Matrix BXpose

(row,col,value)

0 2 1	B.col 0
	B.col 1
	B.col 2
	B.col 3

```
/*--------- for computing one cij ---------*/
   result = 0
   switch( compare(A[p1].col, Bxpose[p2].col) ){
     case ‘<‘: p1++; break;
     case ‘=‘: result += A[p1].value * Bxpose[p2].value;
             p1++; p2++; break;
     case ‘>’: p2++; break;
   }
```

Big-O of Matrix Multiplication

- **Notation**
  - $A_{n \times n} \times B_{n \times n}$
  - A_i: the no. of non-zero elements in i-th row of A
  - B_j: the no. of non-zero elements in j-th column of B
- **Upper Bound Complexity**

$$= O(\sum_{i=0}^{n} \sum_{j=0}^{n} (A_i + B_j)) = O(\sum_{i=0}^{n} (n \cdot A_i + Terms\_of\_B))$$

$$= O(n \cdot Terms\_of\_A + n \cdot Terms\_of\_B)$$

$$= O(n \cdot max(Terms\_of\_A, Terms\_of\_B)$$

Outline

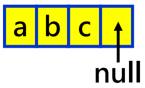
- Abstract Data Types and Class
- The Array as an Abstract Data Type
- The Polynomial
- Sparse Matrices
- The String
  - String Pattern Matching

Abstract Data Type *String*

```
class String
{
// objects: A finite ordered set of zero or more characters.
public:
   String(char *init, int m);
   //Constructor that initializes *this to string init of length m
   int operator==(String t);
   // if (the string  represented by *this equals t) return 1 (TRUE)
   // else return 0 (FALSE);
   int operator!();
   // if *this is empty then return 1 (TRUE); else return 0 (FALSE);
   int Length();
   // return the number of characters in *this
   String Concat(String t);
   // return a string whose elements are those of *this followed by those of t.
   String Substr(int i, int j);
   // return a string containing j characters of *this at positions i, i+1, ..., i+j-1
   // if these are valid positions of *this; otherwise, return the empty string.
   int Find(String pat);
   // return an index i such that pat matches the substring of *this that begins at position i.
   // Return -1 if pat is either empty or not a substring of *this
};
```

→concatenation

string “abc” as an array



String Pattern Matching

- **Problem**
  - **Finding if a pattern is contained in another string**
- **Example**
  - **Pattern: “data structure”**
  - **String: “The course identifier of data structure is EE2410”**

    ↑ 26
  - **Result = 26**
  - **If Pattern does not exist in the string, return –1**

Simple String Pattern Matching

Pattern: data structure (14 characters)
String: data encapsulation is an important concept in data structure

A sliding widow

- Comparing the sub-string within the window with the pattern
  → A mismatch at the sixth character

- Slide the window forward by one step

Pattern: data structure (14 characters)
String: data encapsulation is an important concept in data structure

- A mismatch again !
- Continue to slide the window forward …

Simple String Matching – con't

Pattern: data structure (14 characters)
String: data encapsulation is an important concept in data structure

47

A sliding widow

(1) A match !
(2) Return the starting index of the sliding window = 47

(Time Complexity)
The worst-case complexity of this algorithm = O(m · n)
where m is the length of the pattern
and n is the length of the string
(The reason)
Window matching is performed for n times and each of them may take m steps to complete in the worst case

Algorithm – Simple Matching

```
int String::Find(String pat)
// i is set to –1 if pat does not occur in s (*this);
// otherwise i is set to point to the first position in *this, where pat begins.
{
   char *p = pat.str, *s = str ;
   int i = 0 ;  // i is starting point
   if (*p && *s)
     while (i <= Length() – pat.Length())
       if (*p++ == *s++) { //characters match, get next char in pat and s
          if (!*p) return i ; // match found
       }
       else { // no match
          i++ ; s = str + i ; p = pat.str ;
       }
   return –1 ; //pat is empty or does not occur in s
} // end of Find
```

i is the starting index of the window being compared

end of “pattern” reached

move window forward by 1

Optimal Pattern Matching
Knuth-Morris-Pratt Algorithm

- **The problem of the simple algorithm**
  - **The sliding window moves forward one step at a time**
- **Idea**
  - **Can we move the window several steps forward when a partial match occurs in a window matching ?**

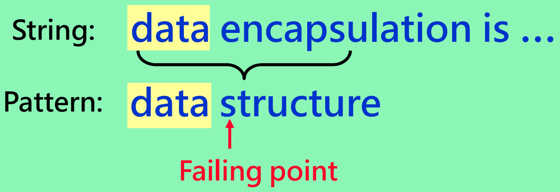
Pattern: data structure (14 characters)
String: data encapsulation is an important concept in data structure

Failed at 6th characters, a partial match

Can the window leap forward by 6 ?

Pattern: data structure (14 characters)
String: data encapsulation is an important concept in data structure

What Happens When Window Matching Fails Partially ?

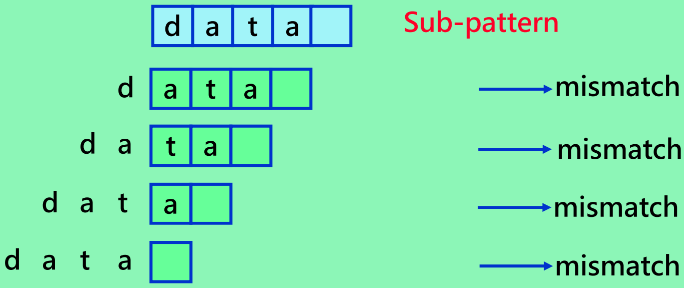


String: data encapsulation is …

Pattern: data structure

Failing point

The following information if pre-computed may enable the leap:



Failure Function

f(j) 是 partial match 後 pattern 的新比較點
i.e., string 指標不變 , pattern 移至 f(j) 這個位置

- **Definition**
  - If $p=p_0p_1\ldots p_{n-1}$ is a pattern, then its failure function, f, is defined as

    $f(j)$ =

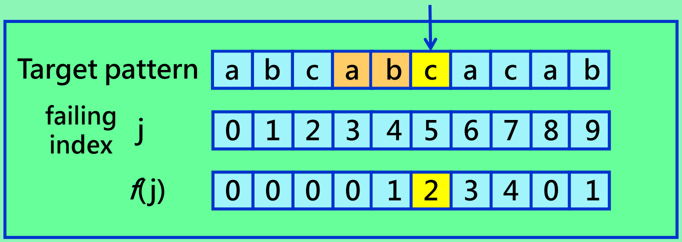
    (1) largest k<j such that $(p_{j-k}p_{j-k+1}\ldots p_{j-1})=(p_0p_1\ldots p_{k-1})$ if such a k≧0 exists

    (2) otherwise $f(j) = 0$;

k有點 max. self-matching length 的意思

- **Example:**

Length-2 prefix matches
length-2 subpattern p[0:1]



Target pattern	a	b	c	a	b	c	a	c	a	b
failing index j	0	1	2	3	4	5	6	7	8	9
f(j)	0	0	0	0	1	2	3	4	0	1

Usage of Failure Function

Failure function decides the leap size after a partial matching

Rule of pattern matching:
If a partial match is found such that
(s_{i-j} … s_{i-1}) = ($p_0 p_1$ … p_{j-1}) and $s_i \neq p_j$
then matching may be resumed by comparing
(1) s_{i+1} and p_0 if $j=0$;
(2) s_i and $p_{f(j)}$ if $j \neq 0$

s_i does not go backward after a partial mismatch

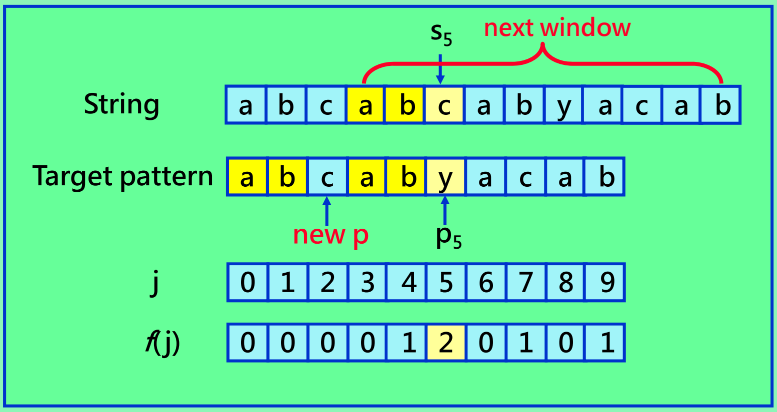
s_i

String: data encapsulation is an important concept in data structure

Pattern: p_j data structure (14 characters)

f(5) = 0 → next character in pattern checked is p_0

Example of Using Failure Function

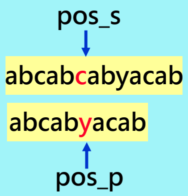


String	a	b	c	a	b	c	a	b	y	a	c	a	b
Target pattern	a	b	c	a	b	y	a	c	a	b			
j	0	1	2	3	4	5	6	7	8	9			
f(j)	0	0	0	0	1	2	0	1	0	1			

Next step: comparing s_5 with p_2
Now, the window of String being compared starts from s_3

FastFind Algorithm

```
int FastFind(String s, String pattern)
{
  // Determine if "pattern" is a substring of s
  int pos_p =0; pos_s=0;
  int length_p = pattern.Length();
  int length_s = s.Length();
  while (pos_p < length_p) && pos_s < length_s) {
     if(pattern.str[pos_p] == s.str[pos_s]) { // matched a character
         pos_p++; pos_s++;
     }
     else { // no match
           if(pos_p == 0) pos_s++;
           else pos_p = pattern.failure[pos_p];
     }
  }
  if(pos_p < length_p) return(-1);
  else return(pos_s – length_p);
}
```


pos_s

abcabcabyacab

abcabyacab

pos_p

String 是一個 class
(1) str() 取出其字串
(2) Length() 取其長度

Time = O(length_s)

取sliding window 起點當作 matching point

Computing the Failure Function

If the failure function can be computed in O(Length_of_Pattern)
→Then the FastFind has an optimal complexity of
O(Length_of_pattern + Length_of_String)

Target pattern	a	b	c	a	b	a	b	c	a	b	c	c
j	0	1	2	3	4	5	6	7	8	9	10	11
f(j)	0	0	0	0	1	2	1	2	3	4	5	?

Rule:
$f(0) = f(1) = 0$
$f(j) = f^m(j-1)+1$, where m is the least positive integer for
which pattern$(f^m(j-1))$ = pattern$(j-1)$
where $f^m(j) = f(f^{m-1}(j))$
$f(j) = 0$ if no such m exists above

Example: 求 f(11) → f(10)=5 , 但是 pattern(5) ≠pattern(10) 所以失敗
→ 試下一個 self-match length k = $f^2(10)$ = ff(10) = f(5) = 2
→ f(5)=2 and pattern(2)=pattern(10)
Therefore, f(11) = 2 + 1 = 3

Algorithm – Failure Function

Note: 這是一個 Class *String* 的 member function

```
void String::compute_failure_function()
{
  failure[0] = 0;  failure[1] = 0;
  for(int j=1; j<Length(); j++){  // find failure function for each element
      int k = failure[j-1];
      while(1) {
          if (str[k] == str[j-1]) {   // found a match !
                failure[j] = k+1;
                break;
          }
          else if(k != 0) {   // apply the rule recursively here
                k = failure[k];
          }
          else{ failure[j] = 0;  break; }
      }
  }
}
```

Using Vector in STL

C++ reference 網頁: http://www.cppreference.com/wiki/start

STL 網頁: http://www.cppreference.com/wiki/stl/start

STL Vector 網頁: http://www.cppreference.com/wiki/stl/vector/start

Example: create an array of string

```
vector<string> words;  // words is an array of string
string str; // str is a string
while( cin >> str ) words.push_back(str);
vector<string>::iterator iter;
for( iter = words.begin(); iter != words.end(); iter++ ) {
        cout << *iter << endl;
}
```

The End of
Chapter 2: Arrays !

Next Topic:

Stacks and Queues