Arrays

Introduction to Data Structures

Kyuseok Shim

SoEECS, SNU.

Topics

- Abstract Data Types and the C++ Class
- Array As Abstract Data Type
- Polynomial Abstract Data Type
- Sparse Matrices
- Representation of Multidimensional Arrays
- String Abstract Data Type

Topics

- Abstract Data Types and the C++ Class
- Array As Abstract Data Type
- Polynomial Abstract Data Type
- Sparse Matrices
- Representation of Multidimensional Arrays
- String Abstract Data Type

C++ Class

- Represents an ADT
- Consists of four components
    - class name
    - data members
    - member functions
    - levels of program access
        - control the level of access to data members and member functions
        - *public* : anywhere
        - *private* : within its class
          a function or a class that is a *friend*
        - *protected* : within its class
          friend its subclasses

Definition of the C++ class 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 xlow, ylow, height, width;
    // (xlow, ylow) are the coordinates of
    // the bottom left corner of the rectangle
};
#endif
```

Data Abstraction and Encapsulation in C++

- Data encapsulation of C++ class
  - all data members are *private* (or *protected*)
  - external access to data members are by member functions
  - member functions
    - that will be invoked externally are *public*
    - all others are *private* (or *protected*)

Data Abstraction and Encapsulation in C++(cont.)

- Separation of specification and implementation of member functions
  - specification (function prototype)
    - name of functions
    - type of function arguments
- type of function result Separation of specification and implementation of member functions
  - specification (function prototype)
    - name of functions
    - type of function arguments
    - type of function result

Data Abstraction and Encapsulation in C++(cont.)

- specification characteristics
  - inside the public portion of the class
  - implementation-independent description using comments
  - placed separately in a header file
- implementation
  - placed in a source file of the same name
  - can be included inside its class definition: treated as an inline function

Implementation of operations on Rectangle

```cpp
// In the source file Rectangle.cpp
#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 height;}
int Rectangle::GetWidth() {return width;}
```

Declaring class objects

- in the same way as variables
- Invoking member functions
  - using component selection operators
    - dot(.) : direct selection
    - arrow : indirect selection through a pointer

A C++ code fragment demonstrating how Rectangle objects are declared and member functions invoked

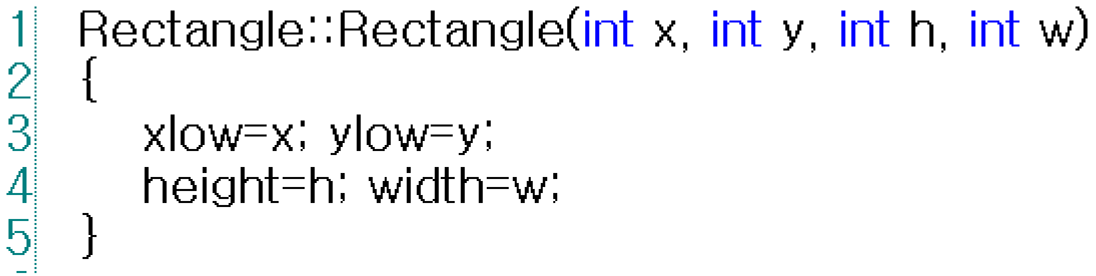
```
// In a source file main.cpp
#include <iostream>
#include "Rectangle.h"
main() {
    Rectangle r, s;    // r and s are objects of class Rectangle
    Rectangle *t = &s;    // t is a pointer to class object s
    .
    .
// use . to access members of class objects.
// use → to access members of class objects through pointers.

if (r.GetHeight()*r.GetWidth() > t→GetHeight() * t→GetWidth())
        cout << " r ";
    else cout << " s ";
        cout << "has the greater area" << endl;
}
```

Special Class Operations

- Constructor
  - a member function which initializes data members of an object
  - If provided for a class, automatically executed when an object of that class is created
  - must be public
  - the name must be identical to the name of the class
  - must not specify a return type or return a value

Definition of a constructor for Rectangle



```cpp
Rectangle::Rectangle(int x, int y, int h, int w)
{
    xlow=x; ylow=y;
    height=h; width=w;
}
```

Special Class Operations (cont.)

- initialize Rectangle object using constructor
  - Rectangle r(1, 3, 6, 6) ;
  - Rectangle *s = *new* Rectangle(0, 0, 3, 4) ;
- initialize using a default constructor
  - Rectangle r ;

A default constructor

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

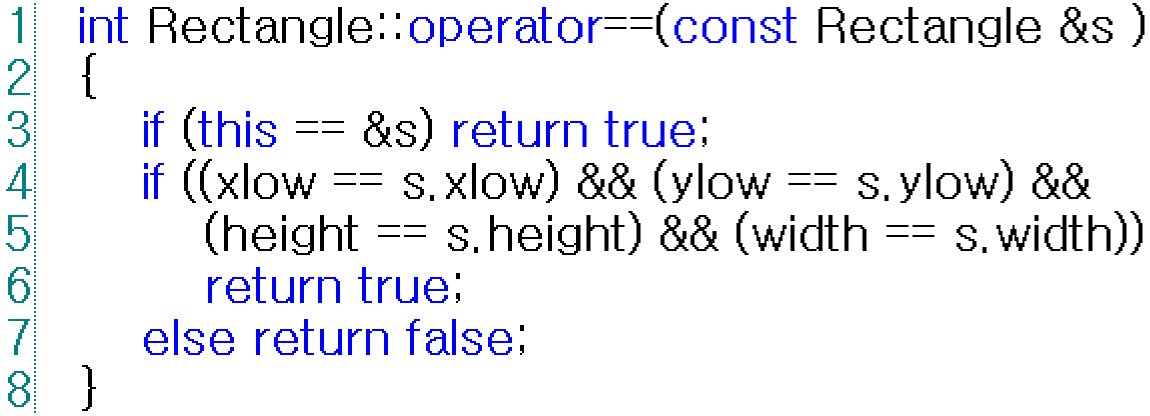
Special Class Operations (cont.)

- Destructor
    - a member function which deletes data members
    - automatically invoked when a class object goes out of scope or is deleted
    - must be public
    - its class name prefixed with ~
    - if a data member is a pointer, only the space of the pointer is returned

Special Class Operations (cont.)

- Operator overloading
  - polymorphism : same operator is used for different situations
  - for example, algorithm comparing two floats is different from algorithm comparing two ints
  - programmer can overload operators for user-defined data types

Overloading operator == for class Rectangle



```
int Rectangle::operator==(const Rectangle &s )
{
    if (this == &s) return true;
    if ((xlow == s.xlow) && (ylow == s.ylow) &&
        (height == s.height) && (width == s.width))
        return true;
    else return false;
}
```

Overloading operator << for class Rectangle

```
ostream& operator << (ostream& os, Rectangle &r)
{
    os<<“Position is:”<<r.xLow<<“ ”;
    os<<r.yLow<<endl;
    os<<“Height is:”<<r.height<< endl;
    os<<“Width is:”<<r.width<< endl;
    return os;
}
```

Special Class Operations (cont.)

- this
  - represents a pointer to the object that invoked a member function
  - *this represents the object

Miscellaneous Topics

- Union
  - reserves storage for the largest of its data members
  - only one of its data members can be stored, at any time
  - results in a more memory-efficient program
- static class data member
  - a global variable for its class
  - there is only one copy of a static data member and all class objects share it
  - declaration does not constitute a definition

ADTs and C++ classes

- They are similar
- Some operators in C++, when overloaded for user defined ADTs, are declared outside the C++ class definition of the ADT

Abstract data type Natural Number

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

Topics

- Abstract Data Types and the C++ Class
- Array As Abstract Data Type
- Polynomial Abstract Data Type
- Sparse Matrices
- Representation of Multidimensional Arrays
- String Abstract Data Type

Array As Abstract Data Type

- Array
  - a set of pairs <index, value>
  - ADT for array provides operations
  - retrieves a value
  - stores a value
- C++ Array
  - index starts at 0
  - C++ does not check bounds for an array index
  - Example
  - float example[n];
  - ith element: example[i] and *(example+i)

Abstract data type General Array (1/2)

```
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.
```

Abstract data type General Array (2/2)(cont.)

```

        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, y>; else signal an error.
};  // end of GeneralArray
```

Topics

- Abstract Data Types and the C++ Class
- Array As Abstract Data Type
- Polynomial Abstract Data Type
- Sparse Matrices
- Representation of Multidimensional Arrays
- String Abstract Data Type

Polynomial Abstract Data Type

- Ordered (or linear) list
  - days of the week : (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
  - years Switzerland fought in WWII : ()
- Operations on lists (a_0, a_1, …, a_{n-1}) :
  - find the length, n, of the list
  - read the list from left to right (or reverse)
  - retrieve the i-th element, $0 \leq i < n$
  - store a new value into the i-th position, $0 \leq i < n$
  - insert a new element at the position i, $0 \leq i < n$
  - delete the element at position i, $0 \leq i < n$

Polynomial Abstract Data Type (cont.)

- Polynomial
  - requires ordered lists
  - the largest exponent is called degree
  - sum and product of polynomials
  - $A(x) = \sum a_i x_i$ and $B(x) = \sum b_i x_i$
    - $A(x) + B(x) = \sum (a_i + b_i) x_i$
    - $A(x) \cdot B(x) = \sum (a_i x_i \cdot \sum (b_j x_j))$

Abstract data type Polynomial

```
Class Polynomial {
    // p(x) = a0xe0 + ··· anxen ; a set of ordered pairs of <ei, ai>,
    // where ai is a nonzero float coefficient and ei is
    // a non-negative integer exponent.
public:
      Polynomial();
      // return the polynomial p(x)=0

      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.
}
```

Polynomial Representation

- Principle
  - unique exponents are arranged in decreasing order
- Representation 1
  - define the private data members of Polynomial
    - private :
    - int degree ; // degree≤MaxDegree
    - float coef[MaxDegree+1] ;
  - for Polynomial object a, n≤MaxDegree
    - a.degree=n
    - a.coef[i]=a_{n-i}, $0 \le i \le n$
  - a.coef[i] is the coefficient of x^{n-i}
    - $A(x)=a_n x^n + a_{n-1} x^{n-i} + \ldots + a_1 x + a_0$
  - leads to a very simple algorithms for many of the operations on polynomials
  - wastes computer memory
    - for example, if a.degree≪MaxDegree

Polynomial Representation (cont.)

- Representation 2
  - define coef with size a.degree+1
  - declare private data members
    - private :
    - int degree ;
    - float *coef ;
  - add a constructor to Polynomial
    - Polynomial::Polynomial(int d)
    - {
    - degree=d ;
    - coef=new float[degree+1] ;
  - wastes space for sparse polynomials
    - for example, $x^{1000}+1$

Polynomial Representation (cont.)

- Representation 3
  - previously, exponents are represented by array indices
  - now, (non-zero) exponents are stored
  - all Polynomials will be represented in a single array called termArray
    - termArray is shared by all Polynomial objects
    - it is declared as static
    - each element in termArray is of type term
  - Class Term {
  - friend Polynomial ;
  - private :
  -     float coef ; // coefficient
  -     int exp ; // exponent
  - } ;

  - Class Polynomial {
  - Private:
  - Term *termArray; //size of nonzero terms
  - int capacity; //size of termArray
  - int terms; //number of nonzero terms
  - }

Adding two polynomials (1/2)

```
Polynomial Polynomial::Add(Polynomial b)
{//Return the sum of of the polynomials *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++;
        }
```

Adding two polynomials (2/2) (cont.)

```
        //add in remaining terms of *this
        for (; aPos<terms; aPos++)
            c.NewTerm(termArray[aPos].coef,          termArray[aPos].exp);
        //add in remining terms of b(x)
        for (;bPos<b.terms;b++)
            c.NewTerm(b.termArray[bPos].coef,
            b.termArray[bPos]exp);
        return c;
}
```

Adding a new term

```
void Polynomial::NewTerm(const float theCoeff, const int theExp)
// Add a new term to the end of termArray
{
    if (terms==capacity)
    {//double capacity of termArray
        capacity*=2;
        term *temp = new term[capacity]; //new array
        copy(termArray, termArray+terms, temp);
        delete[] termArray; //deallocate old memory
        termArray=temp;
    }
    termArray[terms].coef=theCoeff;
    termArray[terms++].exp=theExp;
}
```

Polynomial Addition (cont.)

- Analysis of Add
  - m and n are number of nonzero-terms in A and B, respectively
  - the while loop of line 5 is bounded by m+n-1
  - the for loops of lines 20 and 23 are bounded by O(n+m)
  - summing up, the asymptotic computing time is O(n+m)
- Disadvantages of representing polynomials by arrays
  - space must be reused for unused polynomials
  - linked lists in chapter 4 provide a solution

Topics

- Abstract Data Types and the C++ Class
- Array As Abstract Data Type
- Polynomial Abstract Data Type
- Sparse Matrices
- Representation of Multidimensional Arrays
- String Abstract Data Type

Introduction

- Matrix with m rows and n columns
  - m×n (m by n)
  - mn elements
  - when m=n, the matrix is square
- Storing a matrix
  - two dimensional array A[m][n]
    - an element is A[i][j]
  - sparse matrix
    - store only the nonzero elements
- Matrix operations
  - creation
  - transposition
  - addition
  - multiplication

Two matrices

$$\begin{array}{cc}
 & \begin{array}{rrr} 0 & 1 & 2 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} & \left[\begin{array}{rrr} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{array}\right]
\end{array}$$

(a)

$$\begin{array}{cc}
 & \begin{array}{rrrrrr} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \left[\begin{array}{rrrrrr} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{array}\right]
\end{array}$$

(b)

Abstract data type SparseMatrix (1/2)

```
Class SparseMatrix
{
// objects : A set of triples, <row, column, value>, where row
// and column are integers and form a unique combination; value
// is also an integer.
    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 triple in *this
        SparseMatrix Add(SparseMatrix b);
        // if the dimensions of a(*this) and b are the same, then
        // the matrix produced by adding corresponding items,
        // namely those with identical row and column values is
        // returned else error.
```

Abstract data type SparseMatrix (2/2)(cont.)

```

        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 the formula
        // d[i] [j]=∑(a[i] [k]·b[k] [j]),
        // where d[i] [j] is the (i, j)th element, is returned.
        // k ranges from 0 to the number of columns in a-1
        // else error.
};
```

Sparse Matrix Representation

- Representation
  - use the triple <row, col, value> to represent an element
  - store the triples by rows
  - for each row, the column indices are in ascending order
  - store the number of rows, columns, and nonzero elements

Sparse Matrix Representation (cont.)

- C++ code
  - class SparseMatrix; // forward declaration
  - class MatrixTerm {
  - friend class SparseMatrix
  - private:
  -     int row, col, value;
  - };
- in class SparseMatrix
  - private:
  -     int Rows, Cols, Terms;
  -     MatrixTerm smArray[MaxTerms];

Sparse matrix and its transpose stored as triples

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

(b)

	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

(a)

	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

(b)

Transposing a Matrix

- An element at [i][j] will be at [j][i]

```
for (each row i)
    take element (i, j, value) and store it in (j, i, value) of the transpose;
```

  - example
    - (0, 0, 15) → (0, 0, 15)
    - (0, 3, 22) → (3, 0, 22)
    - (0, 5, −15) → (5, 0, −15)
    - (1, 1, 11) → (1, 1, 11)
  - need to insert many new triples, elements are moved down very often
- Find the elements in the order

```
for (all elements in column j)
    place element (i, j, value) in position (j, i, value);
```

Transposing a matrix

```
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;
                    CurrentB++;
                }
    } // end of if (Terms > 0)
    return b;
} // end of transpose
```

Transposing a Matrix (cont.)

- Analysis of transpose
  - the number of iterations of the for loop at line 9 is terms
  - the number of iterations of the for loop at line 8 is columns
  - total time is O(terms·columns)
  - total space is O(space for a and b)
- Using two-dimensional arrays

```
for(int j=0; j<columns; j++)
    for(int i=0; i<rows; i++)
        B[j][i]=A[i][j];
```

  - total time is O(rows·columns)
- Comparison
  - $O(\text{terms}\cdot\text{columns}) = O(\text{rows}\cdot\text{columns}^2) > O(\text{rows}\cdot\text{columns})$
  - space-time trade-off

Transposing a Matrix (cont.)

- FastTranspose algorithm
  - determine the number of elements in each column of a

    →the number of elements in each row of b

    →starting point of each of b's rows
  - move elements of a one by one into their correct position in b

Transposing a Matrix (cont.)

	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

- values for Figure matrix

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

- asymptotic complexity
  - there are four unnested for loops
  - O(columns+terms)
  - if terms → rows·columns, O(rows·columns)
  - if terms << rows·columns, less than O(rows·columns)
- requires space for RowSize and RowStart

Transposing a matrix faster (1/2)

```
SparseMatrix SparseMatrix::FastTranspose()
{//Return the transpose of *this in O(terms+cols) time.
    SparseMatrix b(cols,rows,terms);
    if (terms>0)
    {//nonzero matrix
        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);//initialize
        for (i=0; i<terms; i++) rowSize[smArray[i].col]++;
        //rowStart[i]=starting position of row i in b
        rowStart[0]=0;
        for (i=1; i<cols; i++) rowStart[i]=rowStart[i-1]+rowSize[i-1];
```

Transposing a matrix faster (2/2) (cont.)

```
        for (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[] rosStart;
    }//end of if
    return b;
}
```

Matrix Multiplication

- Definition : Given a and b, where a is m×n and b is n×p, the product matrix d has dimension m×p.

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

for 0<=i<m and 0<=j<p.

Change the size of a 1-dimensional array

```
void ChangeSize1D(const int newSize)
{//Change the size of smArray to newSize.
    if (newSize<terms)
        throw “New size must be >= number of terms”;
    MatrixTerm *temp = new MatrixTerm[newSize];
    //new Array
    copy(smArray, smArray + terms, temp);
    delete[] smArray; //deallocate old memory
    smArray=temp;
    capacity = newSize;
}
```

Storing a matrix term

```
void SparseMatrix::StoreSum ( const int sum, const int r, const int c)
{//If sum != 0, then it along with its row and column
 //position are stored as the last term in *this.
if (sum!=0){
    if (terms==capacity)
        ChangeSize1D(2*capacity); //double size
    smArray[terms].row=r;
    smArray[terms].col=c;
    smArray[terms++].value=sum;
    }
}
```

Matrix Multiplication (cont.)

- currRowA ： currently being multiplied with the columns of b
- currRowBegin ： position in a of the first element of currRowA
- currColB ： currently being multiplied with currRowA
- currRowIndex, currColIndex ： examine successive elements of currRowA, currColB

Multiplying sparse matrices (1/3)

```
SparseMatrix SparseMatrix::Multiply(SparseMatrix b)
{//Return the product of the sparse matrices *this and b.
    if (cols!=b.rows) throw “Imcompatible matrices”;
    SparseMatrix bXpose = b.FastTranspose();
    SparseMatrix d(rows,b.cols,0);
    int currRowIndex=0, currRowBegin=0, currRowA=smArray[0].row;
    //set boundary conditions
    if (terms==capacity) ChangeSize1D(terms+1);
    bXpose.ChangeSize1D(bXpose.terms+1);
    smArray[terms].row=rows;
    bXpose.smArray[b.terms].row=b.cols;
    bXpose.smArray[b.terms].col=-1;
    int sum=0;
    while (currRowIndex<terms)
    {//generate row currentRowA of d
        int currColB=bXpose.smArray[0].row;
        int currColIndex=0;

```

Multiplying sparse matrices (2/3) (cont.)

```
        while (currColIndex <= b.terms)
        {//multiply row currRowA of *this by column currColb of b
            if (smArray[currRowIndex].row!=currRowA)
            {//end of row currRowA
                d.storeSum(sum, currRowA, currColB);
                sum=0; //reset sum
                currRowIndex = currRowBegin; //advance to next column
                while (bXpose.smArray[currColIndex].row==currColB)
                    currColIndex++;
                currColB=bXpose.smArray[currColIndex].row;
            }
            else if (bXpose.smArray[currColIndex].row!=currColB)
            {//end of column currColB of b
                d.StoreSum(sum,currRowA,currColb);
                sum=0; //reset sum
                //set to multiply row currRowA with next Column
                currRowIndex = currRowBegin;
                currColB=bXpose.smArray[currColIndex].row;
            }
```

Multiplying sparse matrices (3/3) (cont.)

```
            else
                if (smArray[currRowIndex].col <
                    bXpose.smArray[currColIndex].col)
                        currRowIndex++; //advance to next term in row
                else if (smArray[currRowIndex].col ==
                    bXpose.smArray[currColIndex].col)
                {//add to sum
                        sum += smArray[currRowIndex].value *
                            bXpose.smArray[currColIndex].value;
                        currRowIndex++; currColIndex++;
                }
                else currColIndex++; // next term in currColB
        }//end of while(currColindex <= b.terms)
        while (smArray[currRowIndex].row == currRowA)
            //advance to next row
            currRowIndex++;
        currRowBegin = currRowIndex;
        currRowA = smArray[currRowIndex].row;
    }// end of while (currRowIndex < terms)
return d;
}
```

Analysis of Multiply

- Line 3–13 : O(b.cols+b.terms)
- Line 14–56 : O(a.rows)
- Line 19–50 : O(b.cols+b.cols*t_r+b.terms.)

  t_r=number of terms in row r of a
- Line 51–53 : O(t_r)
- Line 14–56 : O(b.cols*t_r+b.terms)
- Overall time for this loop : O($\sum_r$(b.cols*t_r+b.terms))=O(b.cols*a.terms+ a.rows*b.terms)

Analysis of Multiply (cont.)

- Arrays are used.

```
for (int i=0; i<a.rows; i++)
    for (int j=0; j<b.cols; j++){
        sum=0;
        for (int k=0; k<a.cols; k++)
            sum+=a[i][l]*b[k][j];
        c[i][j]=sum;
```

- Time for this : O(a.rows*a.cols*b.cols)

Topics

- Abstract Data Types and the C++ Class
- Array As Abstract Data Type
- Polynomial Abstract Data Type
- Sparse Matrices
- Representation of Multidimensional Arrays
- String Abstract Data Type

Representation of Multidimensional Arrays

- $A[p_1 \ldots q_1][p_2 \ldots q_2] \ldots [p_n \ldots q_n]$, Where $p_i \ldots q_i$ is the range of index values in dimension i
  - the number of elements : $\prod_{i=1}^{n}(q_i - p_i + 1)$
  - an element $A[i_1][i_2] \ldots [i_n]$ is mapped onto a position in a one-dim C++ array

Representation of Multidimensional Arrays (cont.)

- Row major order example
  - A[4..5][2..4][1..2][3..4]
  - 2*3*2*2 = 24 elements
  - stored as A[4][2][1][3], A[4][2][1][4], …, A[5][4][2][3], A[5][4][2][4]
  - indices are increasing : lexicographic order
  - translate to locations in the one-dim array
    - A[4][2][1][3] → position 0
    - A[4][2][1][4] → position 1
    - A[5][4][2][4] → position 23

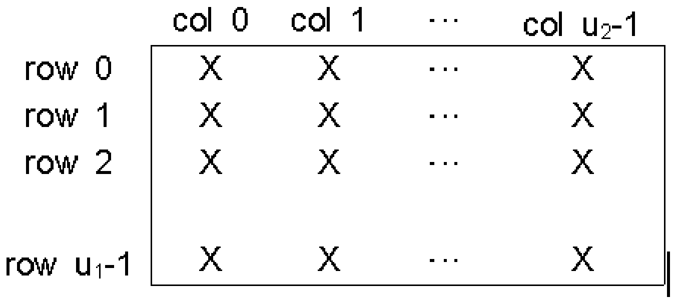
Representation of Multidimensional Arrays (cont.)

- Translation for an n-dim array
  - assume $p_i=0$ and $q_i=u_i-1$
  - one-dim array $A[u_1]$

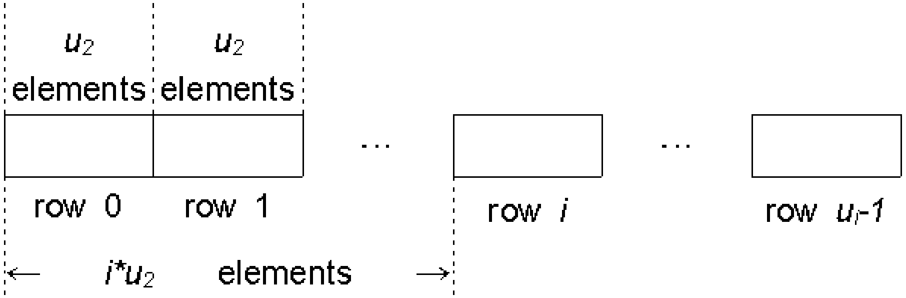
array element:	A[0]	A[1]	A[2]	…	A[i]	…	$A[u_1-1]$
address:	α	$\alpha+1$	$\alpha+2$	…	$\alpha+i$	…	$\alpha+u_1-1$

Representation of Multidimensional Arrays (cont.)

- two−dim array $A[u_1][u_2]$
  - let α be the address of A[0][0]
  - $A[i][0]$: $\alpha + i * u_2$
  - $A[i][j]$: $\alpha + i * u_2 + j$

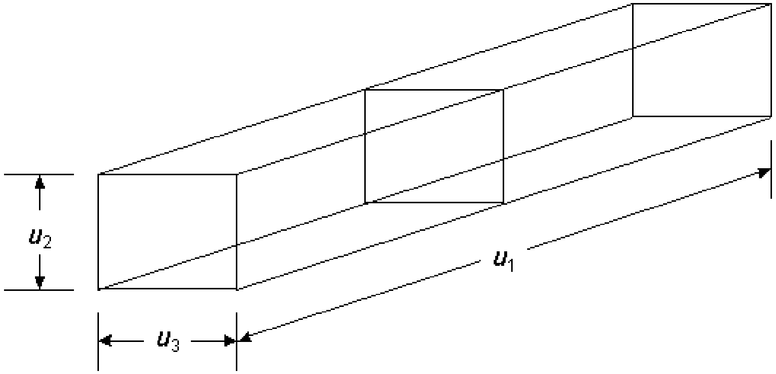


(a)

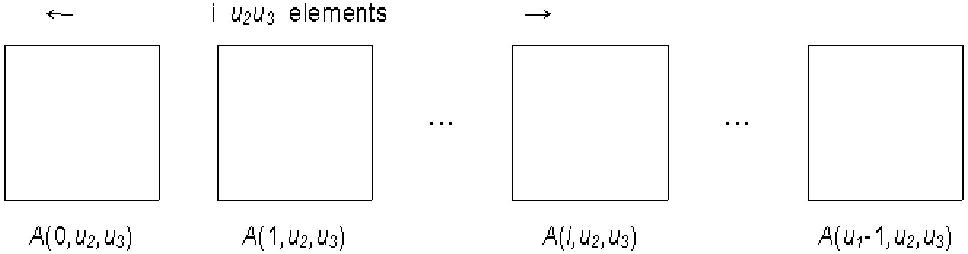


Representation of Multidimensional Arrays (cont.)

- three–dim array $A[u_1][u_2][u_3]$
  - the address of $A[0][0][0]$: α
  - $A[i][0][0]$: $\alpha+iu_2u_3$
  - $A[i][j][k]$: $\alpha+iu_2u_3+ju_3+k$



(a) 3-dimensional array $A[u_1][u_2][u_3]$ regarded as u_1 2-dimensional array



(b) Sequential row major representation of a 3-demensional array. Each 2-dimensional array is represented as in Figure 2.6

Representation of Multidimensional Arrays (cont.)

- n-dim array $A[u_1][u_2] \ldots [u_n]$
  - the address of $A[0][0] \ldots [0]$: α
  - $A[i_1][0],\cdots,[0]$: $\alpha + i_1 u_2 u_3 \cdots u_n$
  - $A[i_1][i_2][0],\cdots,[0]$: $\alpha + i_1 u_2 u_3 \cdots u_n + i_2 u_3 u_4 \cdots u_n$
  - $A[i_1][i_2],\ldots,[i_n]$:

$$
\begin{aligned}
&\alpha + i_1 u_2 u_3 \cdots u_n \\
&\quad + i_2 u_3 u_4 \cdots u_n \\
&\quad + i_3 u_4 u_5 \cdots u_n \\
&\quad \cdot \\
&\quad \cdot \\
&\quad + i_{n-1} u_n \\
&\quad + i_n
\end{aligned}
$$

$$
\alpha + \sum_{j=1}^{n} i_j a_j \quad \textit{where} \quad \begin{cases} a_j = \prod_{k=j+1}^{n} u_k \,, & 1 \le j < n \\ a_n = 1 \end{cases}
$$

Topics

- Abstract Data Types and the C++ Class
- Array As Abstract Data Type
- Polynomial Abstract Data Type
- Sparse Matrices
- Representation of Multidimensional Arrays
- String Abstract Data Type

String Abstract Data Type

- String ADT
  - $S=s_0, \ldots, s_{n-1}$
    where s_i are characters and n is the length of the string
    - if n=0, S is an empty or null string
  - operations for strings
- C++ string
  - string literal constants (e.g., "abc")
  - array of chars : string characters + null character
  - assigned to variables of type char* (e.g., char* str="abc";)
  - ith character of str : str[i]

Abstract data type String (1/2)

```
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 element are those *this followed by
       // those of t.
```

Abstract data type String (2/2) (cont.)

```

        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
};
```

String Pattern Matching

- Function Find
  - two strings s and pat : pat is searched for in s
  - invocation : s.Find(pat)
  - return i : pat matches s beginning at position i
  - return -1 : pat is empty or is not a substring of s
- Implementation
  - representation of strings

    ```
    private
        char* str
    ```
  - sequentially consider each position of s
  - positions to the right of position LengthS-LengthP need not be considered

Exhaustive pattern matching

```
int String::Find(String pat)
{//Return -1 if pat does not occur in *this;
 //otherwise return the first position in *this, where pat begins.
    for (int start=0; start<=Length()-pat.Length(); start++)
    {//check for match beginning at str[start]
        int j;
        for (j=0; j<pat.Length()&&str[start+j]==pat.str[j];j++)
        if(j==pat.Length())return start; //match found
        //no match at positino start
    }
    return -1;//pat is empty of does not occur in s
}
```

String Pattern Matching (cont.)

- the complexity is O(LengthP·LengthS)
  - the number of execution of *while* loop to check *p==*s ≤ LengthP
  - the number of execution of *while* loop by incrementing i < LengthS

String pattern Matching : The Knuth–Morris–Pratt Algorithm

- We would like an algorithm that works in O(*lengthP*+*lengthS*) time.
- Knuth, Morris, and Pratt have developed a pattern matching algorithm that has linear complexity.

String pattern Matching : The Knuth–Morris–Pratt Algorithm (cont.)

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

$$f(j) = \begin{cases} \text{largest } k < j \text{ such that } p_0 \ldots p_k = p_{j-k} \ldots p_j \\ \qquad\qquad \text{(if such a } k \geq 0 \text{ exists)} \\ -1 \qquad\qquad\qquad\qquad \text{(otherwise)} \end{cases}$$

String pattern Matching : The Knuth–Morris–Pratt Algorithm (cont.)

$$f(j) = \begin{cases} \text{largest } k < j \text{ such that } p_0 \ldots p_k = p_{j-k} \ldots p_j & \\ \qquad \text{(if such a } k \geq 0 \text{ exists)} & \\ -1 & \text{(otherwise)} \end{cases}$$

j	0	1	2	3	4	5	6	7	8	9
Pat	a	b	c	a	b	c	a	c	a	b
f	−1	−1	−1	0	1	2	3	−1	0	1

String pattern Matching : The Knuth–Morris–Pratt Algorithm (cont.)

j	0	1	2	3	4	5	6	7	8	9
Pat	a	b	c	a	b	c	a	c	a	b
f	−1	−1	−1	0	1	2	3	−1	0	1

- If a partial match is found such that $s_{i-j} \cdots s_{i-1} = p_0 \cdots p_{j-1}$ and $s_i \neq p_j$ then matching may be resumed by comparing s_i and $p_{f(j-1)+1}$ if $j \neq 0$.
- If $j=0$, then we may continue by comparing s_{i+1} and p_0

Pattern-matching with a failure function

```
int String::FastFind (String pat)
{//Determine if pat is a substring of s.
    int posP=0, posS=0;
    int lengthP=pat.Length(), lengthS=Length();

    while( (posP<lengthP) && (posS<lengthS) )
        if ( pat.str[posP] == str[posS] ){//character match
            posP++; posS++;
        }
        else
            if(posP==0)
                posS++;
            else posP=pat.f[posP-1]+1;
    if (posP<lengthP) return -1;
    else return posS-legnthP;
}
```

String pattern Matching : The Knuth–Morris–Pratt Algorithm (cont.)

- We can compute the failure function in O(lengthP) time, then the entire pattern–matching process will have a computing time proportional to the sum of the lengths of the string and pattern.

$$f(j) = \begin{cases} -1 & (\text{if } j = 0) \\ f^{m}(j-1)+1 & \\ \quad (\text{where } m \text{ is the least integer } k \text{ for which } p_{f^{k}(j-1)+1} = p_j) & \\ -1 & (\text{if there is no } k \text{ satisfying the above}) \end{cases}$$

(Note that $f^{1}(j)=f(j)$ and $f^{m}(j)=f(f^{m-1}(j))$).

Computing the failure function

```
void String::FailureFunction()
{//Compute the failure function for the pattern *this.
    int lengthP=Length();
    f[0]=-1;
    for (int j=1; j<lengthP; j++) //compute f[j]
    {
        int i=f[j-1];
        while ( (*str+j) != *(str+i+1) ) && (i>=0) )
            i=f[i];
        if ( *(str+j) == *(str+i+1))
            f[j]=i+1;
        else f[j]=-1;
    }
}
```