# 2.1.5 Miscellaneous Topics

## struct definition of a data type

### not specify whether a given member has public, private, protected

## union

### no way to know what the data type is until runtime

## static class data member

### a global variable for its class

### each class object(instance object) does not have its own exclusive copy

### only one copy of a static data member and all class objects must share it

#### see the page 81-82

#### class Polynomial {

#### private:

####  static term termArray[MaxTerms];

####  static int free;

####  int Start, Finish;

#### };

#### main(){

####  term Polynomial::termArray[MaxTerms];

####  int Polynomial::free = 0;

#### }

# 2.1.6 ADTs and C++ classes

## use the C++ class to define an ADT instead of the notation of ADT 1.1



# 2.2 The Array as an Abstract Data Type

## an array is a set of pairs, <index, value>, no two pairs in this set have the same index

### we call the index a correspondence or a mapping

## concerned with the operations that can be performed on an array

### retrieve a value

### store a value



## the advantage of ADT definition

### the array is a more general structure than "a consecutive set of memory locations"

### more general than the C++ array

#### more flexible about the composition of the index set

### the C++ array

#### require the index set to be a set of consecutive integers starting at 0

#### do not check an array index

## a C++ array example of floats

###  float exampIe[n];

#### accessed in two ways: example[i], or \*(example+i)

* Figure 2.8 Lower and upper triangular matrices
* Figure 2.9 Tridiagonal matrix

# 2.3 The Polynomial Abstraction Data Type

## arrays

### data structures in their own right

### use them to implement other abstract data types: the ordered, or linear list

### data structure, list, (a0, a1, …, an-1)

### many operations on lists

### find the length, n, of the list

### read the list from left to right

### retrieve the ith element

### store a new value into the ith position

### insert a new element at the position i

### delete the element at position i

## build an ADT for the representation and manipulation of symbolic polynomials

###  A(x) = 3x2 + 2x + 4

###  B(x) = x4 + 10x3 + 3x2 + 1

#### degree : the largest exponent of a polynomial

###  A(x) + B(x), A(x) \* B(x)



# 2.3.1 Polynomial Representation

\* to arrange unique exponents in decreasing order

## Representation 1: to define the private data members of Polynomial

###  private:

###  int degree;

###  float coef[MaxDegree + 1];

### represent the polynomial A(x) as:

###  a.degree = n

###  a.coef[i] = an-i, 0 <= i <= n

#### a.coef[i] = the coefficient of xn-i

#### store the coefficients in order of decreasing exponents

## Representation 2:

### if a.degree « MaxDegree, then a.coef[0], ..., a.coef[N] are unused

### define coef so that its size is a.degree + 1

###  private:

###  int degree;

###  float \*coef;

####  adding the following constructor to Polynomial:

###  Polynomial::Polynomial(int d)

###  {

###  degree = d;

###  coef = new float [degree + 1];

###  }

## Representation 3:

### consider polynomials that have many zero terms, called sparse polynomials

###  x1000 + 1

###  bmxem + bm-1xem-1 + . . . b0xe0

###  b0 = a nonzero coefficient of A

###  ei decreasing em > em-1> ... > e0 ≥ 0

## represent polynomials in a single array called termArray

### the singe array termArray is to be shared by all polynomials: a static class data member of Polynomial

###  class Polynomial; // forward declaration

###  class term {

###  friend Polynomial;

###  private:

###  float coef;

###  int exp;

###  }

### define the private data members of polynomial as follows:

####  private:

####  //A, B에 의해 공유됨

####  static term termArray[MaxTerms];

####  static int free;

####  int Start, Finish

### the required definitions of the static class members outside the class definition:

####  term Polynomial::termArray[MaxTerms];

####  int Polynomial::free = 0;

### A(x) = 2x1000 + 1

### B(x) = x4 + 10x3 + 3x2 +1



# 2.3.2 Polynomial Addition

## C = A + B



#### merging the terms of the two polynomials

#### Prog 2

## Analysis of Add:

### let m and n be # of nonzero terms in A and B

### # of iterations the while loop: m + n + 1

### O(n + m) since non of the exponents are the same in A and B

# 2.3.3 Disadvantage of Representing Polynomials by Arrays

## C = A + B

### C를 새로 create

### Free space가 필요

## if there are some polynomials that are no longer needed

### T1 = A + B

### T2 = C + D

### SUM = T1 + T2

#### T1, T2가 사용한 공간은 반환되어야 함

### demand a complex compacting routine to make a large, contiguous free space

### create dynamically the array by using new

## 메모리 관리 문제

### linked list에서 중요한 문제임

### the array is created dynamically by using new

# 2.4 SPARSE MATRICES

# 2.4.0 The class Matrix

###  Template<class T>

### class Matrix {

###  public:

###  Matrix(int r=0, int c=0);

###  Matrix(const matrix<T>& m);

###  ~Matrix() {delete [] element;}

###  Int Rows() const {return rows;}

###  Int Columns() const {return cols;}

###  T& operator() (int, int j) const;

###  . . .

### private:

###  int rows, cols;

###  T \*element;

###  };

# 2.4.1 Introduction

## sparse matrices

### have many zero entries



#### consider an alternate form of representation

##### should explicitly store only the nonzero elements



# 2.4.2 Sparse Matrix Representation

## a set of triples 표현

## represent a matrix by using the triple <row, col, value>

### use an array of triples to represent a sparse matrix

### row-major ordering

### the column indices are in ascending order

###  class SparseMatrix

###  //triples <row, column, value> 표현

###  class MatrixTerm {

###  friend class SparseMatrix

###  private:

###  int row, col, value;

###  };

###  class SparseMatrix {

###  private:

###  int Rows, Cols, Terms;

###  // set 표현

###  MatrixTerm smArray[MaxTerms];

###  };

###  // Rows: # of rows, Cols : # of columns, Terms : # of nonzero terms



# 2.4.3 Transposing a Matrix

## transpose a matrix

### must interchange the rows and columns

###  for (each row i)

###  take element (i, j, value) and

###  store it in (j, i, value) of the transpose

## the difficulty of transposing a matrix

### not know where to put the element (j, i, value) until all other elements that precede it have been processed

####  (0, 0, 15) becomes (0, 0, 15) // no problem

####  (0, 3, 22) becomes (3, 0, 22) // where to put ??

###  for (all elements in column j)

###  place element (i, j, value) in position (j, i, value)

#### "find all elements in column 0 and store them in row 0"



#### the variable CurrentB : the position in b where the next term in the transpose is to be inserted

# Analysis of Transpose

### O(terms \* columns) // terms = rows \* columns

#### becomes O(rows \* columns2) where terms = columns \* rows

## the computing time of the transpose of a rows x columns matrix in time Ο(rows \* columns)

### for (int j = 0; j < cols, …)

### for (int i = 0; I < rows, …)

### B[j][I] = A[I][j]

## transpose a matrix represented as a sequence of triples in time Ο(terms + columns)

### called FastTranspose



#### proceeds by first determining the number of elements in each column of A

#### obtain easily the starting point in b of each of its rows

### the starting point, RowStart[i] of row i, of B

###  = RowStart[i-1] + RowSize[i-1]

###  // RowSize[i-1] = # of elements in row i-1 of B

### after execution of the third for loop(line 15), the values of RowSize and RowStart:

####  [0] [1] [2] [3] [4] [5]

####  RowSize = 2 1 2 2 0 1

####  RowStart= 0 2 3 5 7 7

## the computing time of FastTranspose

### O(columns + terms)

### four loops in fastTranspose, which are executed columns, terms, columns-1, and terms time respectively

# 2.4.4 Matrix Multiplication

## Def) given A[m x n] and B[n x p], the product matrix Result[m x p] = A[m x n] \* B[n x p]

### result ij = aik bkj



## multiply two sparse matrices A and B represented as ordered lists

### pick a row of A and find all elements in column j of B

### have to scan all of B to find all the elements in column j

### avoid this by first computing the transpose of B

#### put all column elements in consecutive order

### do a merge operation





Program 2.13: Multiplying sparse matrix



#### currRowA : the row of A that is currently being multiplied with the columns of B

#### currRowBegin: the position in a of the first element of row currRowA

#### currColB : the column of B that is currently being multiplied with the row currRowA of A

#### LastInResult: the position occupied by the last element of result

# Analysis of Multiply

## lines 4 to 18: require only O(B.Cols + B.Terms)

## lines 19 to 61: executed at most A.rows

### lines 22 to 57: executed at most B.Terms

### tr : # of terms in row r of A

### the value of currRowIndex can increase at most tr times before currRowIndex moves to the next row of A

#### reset currRowIndex to currRowBegin in line 30

### resetting can occur at most B.cols times

#### the total maximum increment in currRowIndex : B.cols \* tr

### the maximum number of iterations of lines 22 to 57: B.cols + B.Terms + B.cols \* tr

### the iteration with row currRowA of A : O(B.cols \* tr + B.Terms)

### the overall time : O((B.cols \* tr + B.Terms)) = O(B.Cols\* A.Terms + A.Rows\*B.Terms)

## the classical multiplication algorithm

###  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][k] \* b[k][j];

###  c[i][j] = sum;

###  }

### O(A.Rows \* A.Cols \* B.Cols)

## in the worst case, A.Terms = A.Rows \* A.Cols or B.Terms = A.Cols \* B.Rows

### Multiply becomes slower by a constant factor

# 2.5 Representation of Arrays

## Develop a representation in which mapping A[i][j]..[l] onto a position in a one-dimensional C++ array

### Can retrieve multidimensional arrays efficiently

## row major ordering or column major ordering

### A[4..5][2..4][1..2][3..4]

#### using row major order

##### A[4,2,1,3], A[4,2,1,4], A[4,2,2,3], ..

##### lexicographic order

#### A[4,2,1,3] is stored at position 0

#### A[4,2,1,4] is stored at position 1

### C++ 언어가 row major order를 지원, 알고리즘은 column major order가 필요할 경우

## sequential representation of A[u1]

### A[0] = A1, A[i] = A1 + i



## sequential representation of A[u1][u2]

### A[0][0] = A1, A[i][0] = A1 + i \* u2, A[i][j] = A1 + i\*u2 + j



## sequential representation of A[u1][u2][u3]

### A[0][0][0] = A1, A[i][0][0] = A1 + i\*u2\*u3, A[i][j][0] = A1 + i\*u2\*u3 + j\*u3

### A[i][j][k] = A1 + i\*u2\*u3 + j\*u3 + k

### Fig 2

# 2.6 Representation of Special forms of Square Matrices

## Diagonal

### M(i,j) = 0 for i != j

## Tridiagonal

### M(i,j) = 0 for |i-j| > 1

## Lower triangular

### M(i,j) = 0 for i < j

## Upper triangular

### M(i,j) = 0 for i > j

## Symmetric

### M(i,j) = M(j,i) for all i and j

# 2.6.1 Diagonal Matrix

## One way to represent an n x n diagonal matrix D with values of type T

### T d[n][n]

### T d[n]

### Template<class T>

### Class DiagonalMatrix {

###  Public:

###  DiagonalMatrix(int size =0)

###  {n = size; d = new T [n];}

###  ~ DiagonalMatrix() {delete [] d;}

###  DiagonalMatrix<T>&

###  Store(const T& x, int I, int j):

###  T Retrieve(int I, int j) const;

###  Private:

###  Int n; //matrix dimension

###  T \*d; //1D array for diagonal elements

###  };

본 강의 자료의 그림 및 알고리즘 발췌

저자 : HOROWITZ

타이틀 : FUNDAMENTALS OF DATA STRUCTURES IN C++ 2nd Edition (2006)

공저 : SAHNI, MEHTA

 출판사 : Silicon Press