Chap 2. Arrays and Structures (2)

Contents

2.4 Polynomials

2.4.1 The Abstract Data Type

- ***Ordered list* or *linear list***
  - – an ordered set of data items

    ex) Days-of-week

(Sun,	Mon,	Tue,	Wed,	Thu,	Fri,	Sat)	: *list*
1st	2nd	3rd	4th	5th	6th	7th	: *order*

    - denote as $(item_0, item_1, \ldots, item_{n-1})$
    - empty list : ( )
  - – operations on ordered list

i. find the length *n*
ii. read the items in a list from right to left (or left to right)
iii. retrieve *i*th item, $0 \leq i < n$
iv. replace *i*th item's value, $0 \leq i < n$
v. insert *i*th position, $0 \leq i < n$: i, i+1, ..., n-1 → i+1, i+2, ..., n
vi. delete *i*th item, $0 \leq i < n$: i+1, ..., n-1 → i, i+1, ..., n-2

Implementation of Ordered List

- Array
  - associate the list element, $item_i$, with the array index i
  - *sequential mapping*
  - retrieve, replace an item, or find the length of a list, in constant time
  - problems in insertion and deletion
    - sequential mapping forces us to move items

- Linked List
  - *Non-sequential mapping*
  - Chapter 4

A Problem Requiring Ordered Lists

- Manipulation of symbolic polynomials

$$A(x) = 3x^{20} + 2x^5 + 4\ ,\quad B(x) = x^4 + 10x^3 + 3x^2 + 1$$

  – degree : the largest exponent of a polynomial

$$\boxed{\begin{array}{l}\text{When } A(x) = \Sigma\, a_i x^i \text{ and } B(x) = \Sigma\, b_i x^i, \\ \quad A(x) + B(x) = \Sigma\,(a_i + b_i)x^i \\ \quad A(x)\, B(x) = \Sigma\,(a_i\, x^i\, \Sigma\,(b_j\, x^j))\end{array}}$$

  – assumption: unique exponents arranged in decreasing order

ADT *Polynomial* is

objects: $p(x) = a_1 x^{e_1} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers >= 0

functions:

for all $poly, poly1, poly2 \in Polynomial$, $coef \in Coefficients$, $expon \in Exponents$

Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** ($expon \in poly$) **return** its coefficient **else return** zero
Exponent LeadExp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** ($expon \in poly$) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** ($expon \in poly$) **return** the polynomial *poly* with the term whose exponent is *expon* deleted **else return** error
Polynomial SingleMult(*poly*, *coef*, *expon*)	::=	**return** the polynomial $poly \cdot coef \cdot x^{expon}$
Polynomial Add(*poly*1, *poly*2)	::=	**return** the polynomial $poly1 + poly2$
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial $poly1 \cdot poly2$

end *Polynomial*

ADT 2.2: Abstract data type *Polynomial*

2.4.2 Polynomial Representation

- polynomial addition

$D(x) = 0$
$A(x) = 2x^{1000} + 2x^{3}$
$B(x) = x^{4} + 10\,x^{3} + 3\,x^{2} + 1$

(step1)
$D(x) = 2x^{1000}$
$A(x) = 2x^{3}$
$B(x) = x^{4} + 10\,x^{3} + 3\,x^{2} + 1$

(step2)
$D(x) = 2x^{1000} + x^{4}$
$A(x) = 2x^{3}$
$B(x) = 10\,x^{3} + 3\,x^{2} + 1$

(step3)
$D(x) = 2x^{1000} + x^{4} + 12x^{3}$
$A(x) = 0$
$B(x) = 3\,x^{2} + 1$

(step4)
$D(x) = 2x^{1000} + x^{4} + 12x^{3} + 3\,x^{2} + 1$
$A(x) = 0$
$B(x) = 0$

```
#define COMPARE(x, y) ( ((x) < (y)) ? -1 : ((x) == (y)) ? 0: 1 )
```

```
/* d = a + b, where a, b, and d are polynomials */
d = Zero();
while (! IsZero(a) && ! IsZero(b)) do {
   switch COMPARE(LeadExp(a), LeadExp(b)) {
      case -1: d =
         Attach(d,Coef(b,LeadExp(b)),LeadExp(b));
         b = Remove(b,LeadExp(b));
         break;
      case 0: sum = Coef( a, LeadExp(a))
                    + Coef(b, LeadExp(b));
         if (sum) {
            Attach(d,sum,LeadExp(a));
         }
         a = Remove(a,LeadExp(a));
         b = Remove(b,LeadExp(b));
         break;
      case 1: d =
         Attach(d,Coef(a,LeadExp(a)),LeadExp(a));
         a = Remove(a,LeadExp(a));
   }
}
insert any remaining terms of a or b into d
```

Program 2.5: Initial version of *padd* function

Representation of polynomials in C

(1)
```
#define MAX_DEGREE 101 /*Max degree of polynomial+1*/
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

```
polynomial a;
```

$A(x) = \Sigma_{i=0}^{n} a_i x^i$ would be represented as :

$$\texttt{a.degree} = n$$
$$\texttt{a.coef[i]} = a_{n-i},\ 0 \le i \le n \quad , n < MAX\_DEGREE$$

a	*degree*	n											
	coef	a_n	a_{n-1}		…	a_{n-i}	…	a_1	a_0				
		[0]	[1]			[i]		[n-1]	[n]				[100]

a.coef[i] is the coefficient of x^{n-i}

Representation of polynomials in C(cont')

(2)

```
#define MAX_TERMS 100 /*size of terms array*/
typedef struct {
        float coef;
        int expon;
        }term;
term terms[MAX_TERMS];
int avail = 0;
```

$$A(x) = 2x^{1000} + 1 \text{ and } B(x) = x^4 + 10\,x^3 + 3\,x^2 + 1$$

	startA	*finishA*	*startB*			*finishB*	*avail*
	↓	↓	↓			↓	↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

	sA	*fA*	*sB*			*fB*	*avail*					
coef	**2**	**1**	**1**	**10**	**3**	**1**						
exp	1000	0	4	3	2	0						

		sA *fA*	*sB*			*fB*	*sD*	*avail*				
coef	**2**	**1**	**1**	**10**	**3**	**1**	**2**					
exp	1000	0	4	3	2	0	1000					

		sA *fA*		*sB*		*fB*	*sD*		*avail*			
coef	**2**	**1**	**1**	**10**	**3**	**1**	**2**	**1**				
exp	1000	0	4	3	2	0	1000	4				

		sA *fA*			*sB*	*fB*	*sD*			*avail*		
coef	**2**	**1**	**1**	**10**	**3**	**1**	**2**	**1**	**10**			
exp	1000	0	4	3	2	0	1000	4	3			

		sA *fA*				*sB* *fB*	*sD*				*avail*	
coef	**2**	**1**	**1**	**10**	**3**	**1**	**2**	**1**	**10**	**3**		
exp	1000	0	4	3	2	0	1000	4	3	2		

		fA	*sA*			*fB*	*sB* *sD*				*fD*	*avail*
coef	**2**	**1**	**1**	**10**	**3**	**1**	**2**	**1**	**10**	**3**	**2**	
exp	1000	0	4	3	2	0	1000	4	3	2	0	

```
void padd(int startA,int finishA,int startB, int finishB,
                                         int *startD,int *finishD)
{/* add A(x) and B(x) to obtain D(x) */
  float coefficient;
  *startD = avail;
  while (startA <= finishA && startB <= finishB)
    switch(COMPARE(terms[startA].expon,
                   terms[startB].expon)) {
      case -1: /* a expon < b expon */
            attach(terms[startB].coef,terms[startB].expon);
            startB++;
            break;
      case 0: /* equal exponents */
            coefficient = terms[startA].coef +
                          terms[startB].coef;
            if (coefficient)
              attach(coefficient,terms[startA].expon);
            startA++;
            startB++;
            break;
      case 1: /* a expon > b expon */
            attach(terms[startA].coef,terms[startA].expon);
            startA++;
  }
  /* add in remaining terms of A(x) */
  for(; startA <= finishA; startA++)
    attach(terms[startA].coef,terms[startA].expon);
  /* add in remaining terms of B(x) */
  for( ; startB <= finishB; startB++)
    attach(terms[startB].coef, terms[startB].expon);
  *finishD = avail-1;
}
```

iterations $\leq m+n-1$ (beside the while loop)

$\leq m$ (beside the loop for remaining terms of A(x))

$\leq n$ (beside the loop for remaining terms of B(x))

Program 2.6: Function to add two polynomials ※ using (2)

```c
void attach(float coefficient, int exponent)
{/* add a new term to the polynomial */
   if (avail >= MAX_TERMS) {
      fprintf(stderr,"Too many terms in the polynomial\n");
      exit(EXIT_FAILURE);
   }
   terms[avail].coef = coefficient;
   terms[avail++].expon = exponent;
}
```

Program 2.7: Function to add a new term

- **Analysis of *padd***
  - Let m and n be the number of nonzero terms in A and B, respectively.
  - ① If m>0 and n>0, **while loop**
    - each iteration : O(1)
    - The iteration terminates when either *startA* or *startB* exceeds *finishA* or *finishB*, respectively
    - The number of iterations is bounded by $m+n-1$
      - the worst case : ex) $a(x) = x^6+x^4+x^2+x^0$, $b(x) = x^7+x^5+x^3+x^1$
  - ② The remaining **two for loops** → O(m+n)
  - ①&② → **O(m+n)**

2.5 Sparse Matrix

2.5.1 The Abstract Data Type

- Standard representation of a matrix
  - A[MAX_ROWS][MAX_COLS]

$$\begin{array}{cc} & \begin{array}{ccc} \text{col 0} & \text{col 1} & \text{col 2} \end{array} \\ \begin{array}{c} \text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \end{array} & \begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix} \end{array}$$

(a)

$$\begin{array}{cc} & \begin{array}{cccccc} \text{col 0} & \text{col 1} & \text{col 2} & \text{col 3} & \text{col 4} & \text{col 5} \end{array} \\ \begin{array}{c} \text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \\ \text{row 5} \end{array} & \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

(b)

Figure 2.4: Two matrices

- Sparse matrix
  - $m \times n$ matrix A $s.t.$ $\dfrac{\text{no. of nonzero elements}}{\text{m} \times \text{n}} << 1$

ADT *SparseMatrix* is

objects: a set of triples, <*row*, *column*, *value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

for all $a, b \in SparseMatrix, x \in item, i, j, maxCol, maxRow \in index$

SparseMatrix Create(*maxRow*, *maxCol*) ::=

return a *SparseMatrix* that can hold up to $maxItems = maxRow \times maxCol$ and whose maximum row size is *maxRow* and whose maximum column size is *maxCol*.

SparseMatrix Transpose(*a*) ::=

return the matrix produced by interchanging the row and column value of every triple.

SparseMatrix Add(*a*, *b*) ::=

if the dimensions of *a* and *b* are the same

return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.

else return error

SparseMatrix Multiply(*a*, *b*) ::=

if number of columns in *a* equals number of rows in *b*

return the matrix *d* produced by multiplying *a* by *b* according to the formula: $d[i][j] = \sum(a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i, j)th element

else return error.

ADT 2.3: Abstract data type *SparseMatrix*

2.5.2 Sparse Matrix Representation

- *An array of triples*
  – <row, column, value> : 3-tuples (triples)

SparseMatrix Create(*maxRow*, *maxCol*) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

$$\begin{array}{c} \\ \text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \\ \text{row 5} \end{array}
\begin{array}{c} \begin{array}{cccccc} \text{col 0} & \text{col 1} & \text{col 2} & \text{col 3} & \text{col 4} & \text{col 5} \end{array} \\
\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	–15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	–6
[7]	4	0	91
[8]	5	2	28

Use *an array of triples* (cont')

- a[0].row : the number of rows

  a[0].col : the number of columns

  a[0].value : the total number of nonzero entries

- The triples are ordered by row and within rows by columns.

  (*row major ordering*)

2.5.3 Transposing a Matrix

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

Figure 2.5: Sparse matrix and its transpose stored as triples

- Is this a good algorithm for transposing a matrix?

```
for each row i of original matrix
   take element <i, j, value> and store it
   as element <j, i, value> of the transpose;
```

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

a		***b***
(0, 0, 15)	→	(0, 0, 15)
(0, 3, 22)	→	(3,0, 22)
(0, 5, -15)	→	(5, 0, -15)
(1, 1, 11)	→	(1, 1, 11)
		data movement
(1, 2, 3)	→	(2, 1, 3)
		data movement

...

We must move elements to maintain the correct order!

- Using column indices

```
for all elements in column j
```
of original matrix
```
   place element <i, j, value> in
   element <j, i, value>
```
of the transpose

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

a		***b***
(0, 0, 15)	→	(0, 0, 15)
(4, 0, 91)	→	(0, 4, 91)
(1, 1, 11)	→	(1, 1, 11)
(1, 2, 3)	→	(2, 1, 3)
(5, 2, 28)	→	(2, 5, 28)
	…	

We can avoid data movement!

```
void transpose(term a[], term b[])
{/* b is set to the transpose of a */
   int n,i,j, currentb;
   n = a[0].value;          /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
   if (n > 0 )  { /* non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
         for (j = 1; j <= n; j++)
         /* find elements from the current column */
            if (a[j].col == i) {
            /* element is in current column, add it to b */
               b[currentb].row = a[j].col;
               b[currentb].col = a[j].row;
               b[currentb].value = a[j].value;
               currentb++;
            }
   }
}
```

Program 2.8: Transpose of a sparse matrix $O(\mathit{columns} \cdot \mathit{elements})$

- Analysis of *transpose*
  - Nested for loops are the decisive factor.
  - The remaining part requires only constant time.
  - Time complexity : **O(*columns* · *elements*)**
  - If *elements* = *rows* · *columns*, O(*columns*^{2} · *rows*)
    - To conserve space, we have traded away too much time.

cf) If the matrices are represented as 2D arrays,

```
for ( j = 0; j < columns; j++)
    for ( i = 0; i < rows; i++)
        b[j][i] = a[i][j];
```

- O(*columns* · *rows*)

- Fast transpose of a sparse matrix

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

	row	col	value
b[0]	6	6	8
→ [1]			
[2]			
→ [3]			
→ [4]			
[5]			
→ [6]			
[7]			
→ [8]			

① **calculation of rowTerms**

	[0]	[1]	[2]	[3]	[4]	[5]
rowTerms =	2	1	2	2	0	1
startingPos =	1	3	4	6	8	8

② **calculation of startingPos**

- Fast transpose of a sparse matrix(cont')

③ b(j,i)← a(i,j)

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

	[0]	[1]	[2]	[3]	[4]	[5]
rowTerms =	2	1	2	2	0	1
startingPos =	3	4	6	8	8	9

```
void fastTranspose(term a[], term b[])
{/* the transpose of a is placed in b */
   int rowTerms[MAX_COL], startingPos[MAX_COL];
   int i,j, numCols = a[0].col, numTerms = a[0].value;
   b[0].row = numCols;  b[0].col = a[0].row;
   b[0].value = numTerms;
   if (numTerms > 0) { /* nonzero matrix */
     for (i = 0; i < numCols; i++)
        rowTerms[i] = 0;
     for (i = 1; i <= numTerms; i++)
        rowTerms[a[i].col]++;
     startingPos[0] = 1;
     for (i = 1; i < numCols; i++)
        startingPos[i] =
                     startingPos[i-1] + rowTerms[i-1];
     for (i = 1; i <= numTerms; i++) {
        j = startingPos[a[i].col]++;
        b[j].row = a[i].col;   b[j].col = a[i].row;
        b[j].value = a[i].value;
     }
   }
}
```

calculation of rowTerms

calculation of startingPos

b(j,i) ← a(i,j)

Program 2.9: Fast transpose of a sparse matrix O(*columns* + *elements*)

- Analysis of *fastTranspose*
  - The number of iterations of the four loops
    - *numCols*, *numTerms*, *numCols*-1, *numTerms*, respectively
  - The statements within the loops require constant time.
  - Time complexity : **O(*columns*+ *elements*)**

  - If *elements* = *columns* · *rows*, O(*columns* · *rows*)
    - equals that of the 2D array representation

  - However, if *elements* << *columns* · *rows*,
    - much faster than 2D array representation

  - Thus, in this representation *we save both time and space*.