Chapter 2
Arrays and Structures

Yi-Fen Liu

Department of IECS, FCU

References:

- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures (2nd Edition)*
- Slides are credited from Prof. Chung, NTHU

Outline

- The array as an Abstract Data Type
- Structures and Unions
- The polynomial Abstract Data Type
- The Sparse Matrix Abstract Data Type
- The Representation of Multidimensional Arrays
- String Matching

THE ARRAY AS AN ADT

The array as an ADT (1)

- Arrays
  - A set of pairs, <*index*, *value*>
  - Data structure
    - For each index, there is a value associated with that index.
  - Representation (possible)
    - Implemented by using consecutive memory.
    - In mathematical terms, we call this a *correspondence* or a *mapping*.

The array as an ADT (2)

- When considering an ADT we are more concerned with the operations that can be performed on an array.
  - Aside from *creating* a new array, most languages provide only two standard operations for arrays, one that *retrieves* a value, and a second that *stores* a value.
  - The advantage of this ADT definition is that it clearly points out the fact that the array is a more general structure than “a consecutive set of memory locations.”

The array as an ADT (3)

structure *Array* is

objects: A set of pairs <*index*, *value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, $\{0, \cdots, n-1\}$ 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.

functions:

for all $A \in Array$, $i \in index$, $x \in item$, j, $size \in$ integer

Array Create(j, *list*)	::=	**return** an array of j dimensions where *list* is a j-tuple whose ith element is the the size of the ith dimension. *Items* are undefined.
Item Retrieve(A, i)	::=	**if** ($i \in index$) **return** the item associated with index value i in array A **else return** error
Array Store(A,i,x)	::=	**if** (i in *index*) **return** an array that is identical to array A except the new pair <i, x> has been inserted **else return** error.

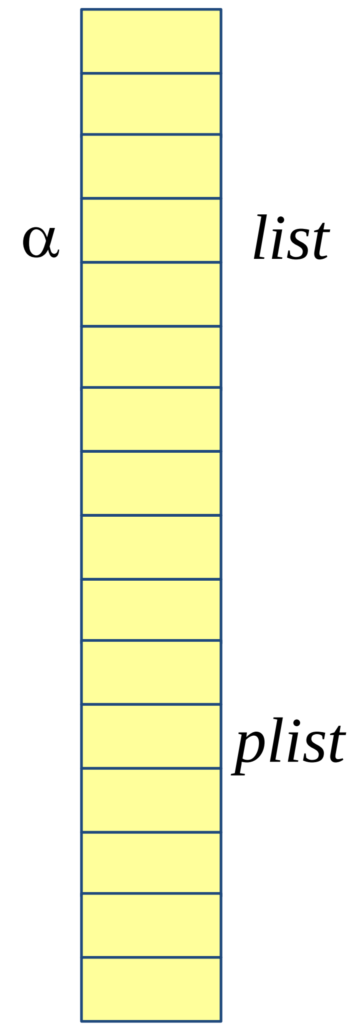
end *Array*

Abstract Data Type *Array*

The array as an ADT (4)

- Arrays in C
  - int list[5], *plist[5];
  - list[5]: (five integers)
    - list[0], list[1], list[2], list[3], list[4]
  - *plist[5]: (five pointers to integers)
    - plist[0], plist[1], plist[2], plist[3], plist[4]
  - implementation of 1-D array

list[0]	base address = α
list[1]	α + sizeof(int)
list[2]	α + 2*sizeof(int)
list[3]	α + 3*sizeof(int)
list[4]	α + 4*sizeof(int)



The array as an ADT (5)

– Compare int *list1 and int list2[5] in C.
  Same: list1 and list2 are pointers
  Difference: list2 reserves five locations

– Notations
  list2 — a pointer to list2[0]
  (list2 + *i*) — a pointer to list2[*i*] (&list2[*i*])
  *(list2 + *i*) — list2[*i*]

Example

Address	Contents
1228	0
1230	1
1232	2
1234	3
1236	4

- 1-dimension array addressing
  - int one[] = {0, 1, 2, 3, 4};
  - Goal: print out address and value

```
void print1(int *ptr, int rows) {
/* print out a one-dimensional array using a pointer */
  int i;
  printf("Address Contents\n");
  for (i=0; i < rows; i++)
     printf("%8u%5d\n", ptr+i,
     *(ptr+i) );
  printf("\n");
}
```

STRUCTURES AND UNIONS

Structures and Unions (1)

- Structures (Records)
  - Arrays are collections of data of the same type
  - In C there is an alternate way of grouping data that permit the data to vary in type
    - This mechanism is called the *struct*, short for structure
  - A structure is a collection of data items, where each item is identified as to its type and name

```
struct {
        char name[10];
        int age;
        float salary;
        } person;
```

```
strcpy(person.name,"james");
person.age = 10;
person.salary = 35000;
```

Structures and Unions (2)

- Create structure data type
  - We can create our own structure data types by using the typedef statement as below

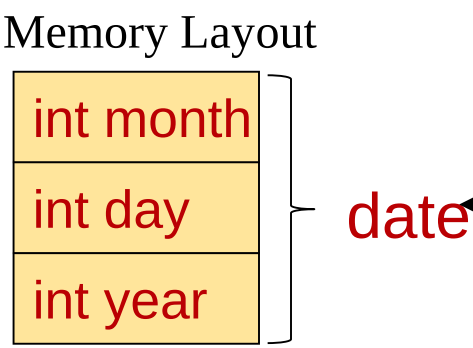
```
typedef struct human_being {    or    typedef struct {
        char name[10];                        char name[10];
        int age;                              int age;
        float salary;                         float salary;
        };                                    } human_being;
```

  - This says that human_being is the name of the type defined by the structure definition, and we may follow this definition with declarations of variables such as:

human_being person1, person2;

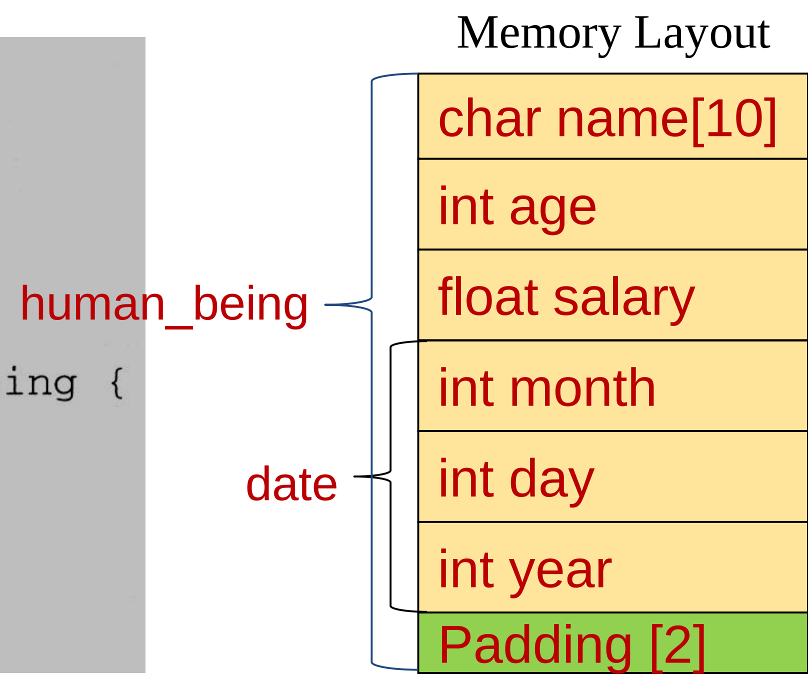
Structures and Unions (3)

- We can also embed a structure within a structure.




```
typedef struct {
        int month;
        int day;
        int year;
        } date;

typedef struct human_being {
        char name[10];
        int age;
        float salary;
        date dob;
        };
```



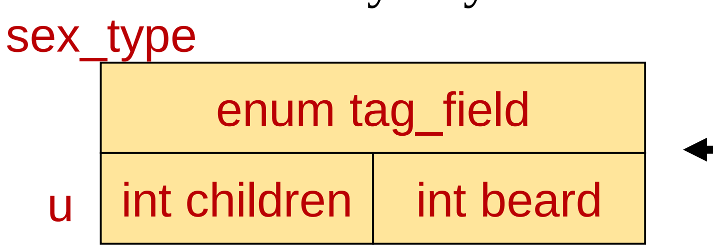
  - A person born on February 11, 1994, would have values for the date struct set as

```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

Structures and Unions (4)

- Unions
  - A union declaration is similar to a structure
  - The fields of a union *must share their memory space*
  - Only one field of the union is “active” at any given time

Memory Layout



person1.sex_info.sex = male;
person1.sex_info.u.beard = FALSE;
and
person2.sex_info.sex = female;
person2.sex_info.u.children = 4;

```
typedef struct sex_type {
    enum tag_field {female, male} sex;
    union {
        int children;
        int beard ;
        } u;
    };
typedef struct human_being {
    char name[10];
    int age;
    float salary;
    date dob;
    sex_type sex_info;
    };
human_being person1, person2;
```

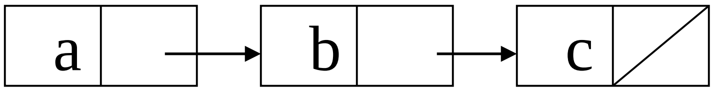
Structures and Unions (5)

- Internal implementation of structures
  - The fields of a structure in memory will be stored in the same way using increasing address locations in the order specified in the structure definition
  - Holes or padding may actually occur
    - Within a structure to permit two consecutive components to be properly aligned within memory
  - The size of an object of a struct or union type is the amount of storage necessary to represent the largest component, including any padding that may be required

Structures and Unions (6)

- Self-Referential Structures
  - One or more of its components is a pointer to itself
  - typedef struct list {
char data;
list *link;
}
  - list item1, item2, item3;
item1.data='a';
item2.data='b';
item3.data='c';
item1.link=&item2;
item2.link=&item3;
item3.link=NULL;

Construct a list with three nodes
item1.link=&item2;
item2.link=&item3;
malloc: obtain a node (memory)
free: release memory



THE POLYNOMIAL ADT

Ordered or Linear List

- Examples
  - Ordered (linear) list: ($item_1$, $item_2$, $item_3$, …, $item_n$)
    - (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
    - (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
    - (basement, lobby, mezzanine, first, second)
    - (1941, 1942, 1943, 1944, 1945)
    - (a_1, a_2, a_3, …, a_{n-1}, a_n)

Operations on Ordered List

- Operations
  - **Finding** the length, n , of the list.
  - **Reading** the items from left to right (or right to left).
  - **Retrieving** the i'th element.
  - **Storing** a new value into the i'th position.
  - **Inserting** a new element at the position i , causing elements numbered i, i+1, …, n to become numbered i+1, i+2, …, n+1
  - **Deleting** the element at position i , causing elements numbered i+1, …, n to become numbered i, i+1, …, n-1

The Polynomial

- Polynomial examples
  - Two example polynomials are
    - $A(x) = 3x^{20}+2x^5+4$
    - $B(x) = x^4+10x^3+3x^2+1$
  - Assume that we have two polynomials, $A(\mathrm{x}) = \Sigma a_i x^i$ and $B(x) = \Sigma b_i x^i$ where x is the variable, a_i is the coefficient, and i is the exponent, then
    - $A(x) + B(x) = \Sigma(a^i + b^i)x^i$
    - $A(x) \cdot B(x) = \Sigma(a^i x^i \cdot \Sigma(b^j x^j))$
    - Similarly, we can define subtraction and division on polynomials, as well as many other operations

- An ADT Definition of a Polynomial

structure *Polynomial* is

objects: $p(x) = a_1x^{e_1} + \cdots + a_nx^{e_n}$; 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*, *poly*1, *poly*2 $\in$ *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 Lead_Exp(*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 *poly*1 + *poly*2
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 $\cdot$ *poly*2

end *Polynomial*

Abstract data type *Polynomial*

Polynomial Addition (1)

- $A(x) = 3x^{20}+2x^5+4$ and $B(x) = x^4+10x^3+3x^2+1$
- $D(x) = A(x) + B(x)$

	x^{20}	x^6	x^5	x^4	x^3	x^2	x^1	x^0
	↓ Lead_Exp()							
$A(x)$	3		2					4
$B(x)$				1	10	3		1
				↑ Lead_Exp()				
$D(x)$								

Polynomial Addition (2)

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

advantage: easy implementation
disadvantage: waste space when sparse
  ```

The Polynomial ADT (1)

- There are two ways to create the type polynomial in C
  - Representation I
    - ```
      #define MAX_degree 101
      /*MAX degree of polynomial+1*/
      typedef struct{
           int   degree;
           float coef[MAX_degree];
      } polynomial;
      ```

Drawback: The first representation may waste space.

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

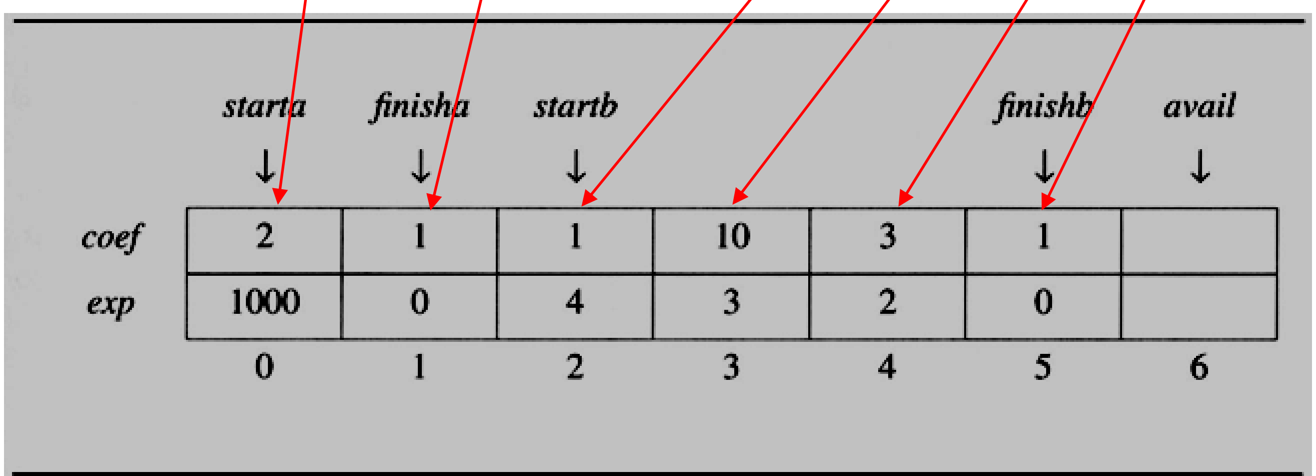
Use one global array to store all polynomials

The Polynomial ADT (2)

- Use one global array to store all polynomials
  - The figure shows how these polynomials are stored in the array terms.

specification poly	representation <start, finish>
A	<0,1>
B	<2,5>

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



	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

Array representation of two polynomials

The Polynomial ADT (3)

- A C function that adds two polynomials,
  A and B, represented as above to obtain
  D = A + B.
  - To produce D(x), padd adds A(x) and B(x) term by term.

Analysis: $O(n+m)$
where n (m) is the number of nonzeros in A (B).

```
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;
}
```

Function to add two polynomials

```
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;
}
```

Function to add two polynomials

$A(x) = 2x^{1000}+1$

$B(x) = x^4+10x^3+3x^2+1$

starta = 2 finisha = 1

startb = 6 finishb = 5

starta <= finisha = FALSE

startb <= finishb = FALSE

Term

$2x^{1000}$
1
x^4
$10x^3$
$3x^2$
1
$2x^{1000}$
x^4
$10x^3$
$3x^2$
2

The Polynomial ADT (5)

Problem: Compaction is required when polynomials that are no longer needed. (data movement takes time)

```
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(1);
   }
   terms[avail].coef = coefficient;
   terms[avail++].expon = exponent;
}
```

Function to add a new term

THE SPARSE MATRIX ADT

The Sparse Matrix (1)

- In mathematics, a matrix contains *m* rows and *n* columns of elements, we write *m*×*n* to designate a matrix with *m* rows and *n* columns.

5*3
15/15

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

(a)

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

(b)

Two matrices

6*6
8/36

Sparse Matrix

The Sparse Matrix (2)

- The standard representation of a matrix is a two dimensional array defined as

$$a[MAX\_ROWS][MAX\_COLS]$$

  - We can locate quickly any element by writing $a[i][j]$
- **Sparse matrix wastes space**
  - We must consider alternate forms of representation
  - Our representation of sparse matrices should store only nonzero elements
  - Each element is characterized by <*row*, *col*, *value*>

The Sparse Matrix ADT (1)

- A minimal set of operations
  - Matrix creation
  - Addition
  - Multiplication
  - Transpose

structure *Sparse_Matrix* 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 $a, b \in$ *Sparse_Matrix*, $x \in$ *item*, i, j, *max_col*, *max_row* $\in$ *index*

Sparse_Matrix Create(*max_row*, *max_col*) ::=
  return a *Sparse_Matrix* that can hold up to *max_items* = *max_row* $\times$ *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col*.

Sparse_Matrix Transpose(a) ::=
  return the matrix produced by interchanging the row and column value of every triple.

Sparse_Matrix 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

Sparse_Matrix 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.

Abstract data type *Sparse_Matrix*

The Sparse Matrix ADT (2)

- The Create operation

Sparse_Matrix Create(*max_row*, *max_col*) ::=

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

The Sparse Matrix ADT (3)

- Represented by a two-dimensional array
- Each element is characterized by <*row*, *col*, *value*>
- row, column in ascending order

# of rows (columns) # of nonzero terms

	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)

transpose

	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)

Sparse matrix and its transpose stored as triples

Transpose a Matrix

- For each row *i*
  - take element <*i*, *j*, *value*> and store it in element <*j*, *i*, *value*> of the transpose
  - **difficulty**: where to put <j, i, value>
(0, 0, 15) ====> (0, 0, 15)
(0, 3, 22) ====> (3, 0, 22)
(0, 5, -15) ====> (5, 0, -15)
(1, 1, 11) ====> (1, 1, 11)
Move elements down very often
- *For all elements in column j,*
  - place element <*i*, *j*, *value*> in element <*j*, *i*, *value*>

Transpose 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

For all column i in a

For all element j in a

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

```
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++;
            }
   }
}
```

==> O(columns*elements)

Transpose of a sparse matrix

Discussion

- Compared with 2-D array representation
  - O(*columns*elements*) *v.s.* O(*columns*rows*)
  - elements --> columns * rows when non-sparse, O($columns^2$*rows*)
- Problem: Scan the array “columns” times
  - In fact, we can transpose a matrix represented as a sequence of triples in O(*columns + elements*) time
- Solution
  - First, determine the number of elements in each column of the original matrix
  - Second, determine the starting positions of each row in the transpose matrix

Fast Transpose of a Sparse Matrix (1)

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

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

- How to predict the location?
  - First, determine the number of elements in each column of the original matrix
  - Second, determine the starting positions of each row in the transpose matrix

Fast Transpose of a Sparse Matrix (2)

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

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms	2	1	2	2	0	1
starting_pos	1	3	4	6	8	8

Determine the number of elements in each column

```
for (i = 0; i < num_cols; i++)
   row_terms[i] = 0;
for (i = 1; i <= num_terms; i++)
   row_terms[a[i].col]++;
```

Determine the starting positions of each row

```
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
   starting_pos[i] =
               starting_pos[i-1] + row_terms[i-1];
```

Fast Transpose of a Sparse Matrix (3)

	[0]	[1]	[2]	[3]	[4]	[5]
starting_pos	3	4	6	8	8	9

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

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

```
for (i = 1; i <= num_terms; i++) {
   j = starting_pos[a[i].col]++;
   b[j].row = a[i].col;   b[j].col = a[i].row;
   b[j].value = a[i].value;
}
```

Fast Transpose of a Sparse Matrix (4)

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

Fast transpose of a sparse matrix

Matrix Multiplication

- Definition
  - Given A and B where A is $m \times n$ and B is $n \times p$, the product matrix D has dimension $m \times p$. Its <*i*, *j*> element is

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

for $0 \leq i < m$ and $0 \leq j < p$.

- Example

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

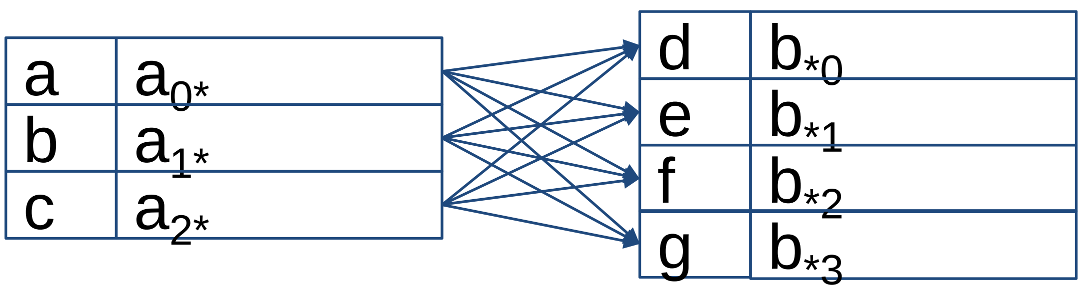
Multiplication of two sparse matrices

Sparse Matrix Multiplication (1)

- Definition
  - $[D]_{m \times p} = [A]_{m \times n} \times [B]_{n \times p}$
- Procedure
  - Fix a row of A and find all elements in column j of B for j = 0, 1, …, p-1.
- Alternative 1
  - Scan all of B to find all elements in j
- Alternative 2
  - Compute the transpose of B
    (Put all column elements consecutively)
  - Once we have located the elements of row i of A and column j of B we just do a merge operation similar to that used in the polynomial addition

Sparse Matrix Multiplication (2)

- General case
  - $d_{ij}=a_{i0}*b_{0j}+a_{i1}*b_{1j}+\ldots+a_{i(n-1)}*b_{(n-1)j}$
  - Array A is grouped by *i*, and after transpose, array B is also grouped by *j*



The multiply operation generate entries:
a*d , a*e , a*f , a*g , b*d , b*e , b*f , b*g , c*d , c*e , c*f , c*g

Sparse Matrix Multiplication (3)

$$A = \begin{pmatrix} 1 & 0 & 2 \\ -1 & 4 & 6 \end{pmatrix} \quad B^T = \begin{pmatrix} 3 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 5 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 0 & 2 \\ -1 & 0 & 0 \\ 0 & 0 & 5 \end{pmatrix} \quad A \times B = \begin{pmatrix} 3 & 0 & 12 \\ -7 & 0 & 28 \end{pmatrix}$$

	row	col	value
A[0]	2	3	5
A[1]	0	0	1
A[2]	0	2	2
A[3]	1	0	-1
A[4]	1	1	4
A[5]	1	2	6

	row	col	value
$B^T[0]$	3	3	4
$B^T[1]$	0	0	3
$B^T[2]$	0	1	-1
$B^T[3]$	2	0	2
$B^T[4]$	2	2	5

	row	col	value
D[0]	2	3	?
D[1]	0	0	3
D[2]	0	2	12
D[3]	1	0	-7
D[4]	1	2	28

Sparse Matrix Multiplication (4)

$$A = \begin{pmatrix} 1 & 0 & 2 \\ -1 & 4 & 6 \end{pmatrix} \quad B^T = \begin{pmatrix} 3 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 5 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 0 & 2 \\ -1 & 0 & 0 \\ 0 & 0 & 5 \end{pmatrix} \quad A \times B = \begin{pmatrix} 3 & 0 & 12 \\ -7 & 0 & 28 \end{pmatrix}$$

row* = A[1].*row

column* = B^T[1].*row

***sum* = 0**

		row	col	value
	A[0]	2	3	5
i	A[1]	0	0	1
i	A[2]	0	2	2
	A[3]	1	0	-1
	A[4]	1	1	4
	A[5]	1	2	6

		row	col	value
	B^T[0]	3	3	4
j	B^T[1]	0	0	3
j	B^T[2]	0	1	-1
j	B^T[3]	2	0	2
	B^T[4]	2	2	5

		row	col	value
totald	D[0]	?	?	?
	D[1]	0	0	3

if **(A[*i*].col = B^T[*j*].col)**
(sum += A[*i*++].value * B^T[*j*++].value);
else if **(A[*i*].col < B^T[*j*].col)**
i **++;**
else
***j*++;**

***i* = 1, *j* = 1, *sum* += 1 × 3**
***i* = 2, *j* = 2, *j*++;**
***i* = 2, *j* = 3, B^T[*j*].row != column**
storesum()
Reset sum, i and column

Sparse Matrix Multiplication (5)

$A = \begin{pmatrix} 1 & 0 & 2 \\ -1 & 4 & 6 \end{pmatrix}$ $B^T = \begin{pmatrix} 3 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 5 \end{pmatrix}$ $B = \begin{pmatrix} 3 & 0 & 2 \\ -1 & 0 & 0 \\ 0 & 0 & 5 \end{pmatrix}$ $A \times B = \begin{pmatrix} 3 & 0 & 12 \\ -7 & 0 & 28 \end{pmatrix}$

row* = A[*i*].*row

column* = B^T[*j*].*row

***sum* = 12**

		row	col	value
	A[0]	2	3	5
i	A[1]	0	0	1
i	A[2]	0	2	2
i	A[3]	1	0	-1
	A[4]	1	1	4
	A[5]	1	2	6

		row	col	value
	B^T[0]	3	3	4
	B^T[1]	0	0	3
	B^T[2]	0	1	-1
j	B^T[3]	2	0	2
j	B^T[4]	2	2	5
j	B^T[5]	??		

		row	col	value
	D[0]	?	?	?
totald	D[1]	0	0	3
	D[2]	0	2	12

if **(A[*i*].col = B^T[*j*].col)**
(sum += A[*i*++].value * B^T[*j*++].value);
else if **(A[*i*].col < B^T[*j*].col)**
i **++;**
else
j**++;**

i **= 1,** ***j*** **= 3,** ***sum*** **+= 1 × 2**
i **= 2,** ***j*** **= 4,** ***sum*** **+= 2 × 5**
i **= 3,** ***j*** **= 5, A[*i*].row != row**
storesum()
Reset sum, i and column

Sparse Matrix Multiplication (6)

```
void mmult(term a[], term b[], term d[])
/* multiply two sparse matrices */
{
  int i, j, column, totalb = b[0].value, totald = 0;
  int rows_a = a[0].row, cols_a = a[0].col,
  totala = a[0].value; int  cols_b = b[0].col,
  int row_begin = 1, row = a[1].row, sum = 0;
  int new_b[MAX_TERMS][3];
  if (cols_a != b[0].row) {
    fprintf(stderr,"Incompatible matrices\n");
    exit(1);
  }
  fast_transpose(b,new_b);
  /* set boundary condition */
  a[totala+1].row = rows_a;
  new_b[totalb+1].row = cols_b;
  new_b[totalb+1].col = 0;
  for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1;) {
      /* multiply row of a by column of b */
        if (a[i].row != row) {
          storesum(d,&totald,row,column,&sum);
          i = row_begin;
          for (; new_b[j].row == column; j++)
            ;
          column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
          storesum(d, &totald, row, column, &sum);
          i = row_begin;
          column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[j].col)) {
          case -1: /* go to next term in a */
                i++;  break;
          case 0: /* add terms, go to next term in a and b*/
                sum += ( a[i++].value * new_b[j++].value);
                break;
          case 1 : /* advance to next term in b */
                j++;
        }
    }  /* end of for j <= totalb+1 */
    for (; a[i].row == row; i++)
      ;
    row_begin = i; row = a[i].row;
  } /* end of for i<=totala */
  d[0].row = rows_a;
  d[0].col = cols_b; d[0].value =  totald;
}
```

Sparse Matrix Multiplication (7)

```
void storesum(term d[], int *totald, int row, int column,
                                          int *sum)
{
/* if *sum != 0, then it along with its row and column
position is stored as the *totald+1 entry in d */
   if (*sum)
      if (*totald < MAX_TERMS) {
         d[++*totald].row = row;
         d[*totald].col = column;
         d[*totald].value = *sum;
         *sum = 0;
      }
      else {
         fprintf(stderr,"Numbers of terms in product
                                    exceeds %d\n",MAX_TERMS);
         exit(1);
      }
}
```

Analyzing The Algorithm (1)

- cols_b * termsrow1 + totalb +
  cols_b * termsrow2 + totalb +
  … +
  cols_b * termsrowp + totalb
  = cols_b * (termsrow1 + termsrow2 + … + termsrowp)
  + rows_a * totalb
  = cols_b * totala + row_a * totalb

  O(cols_b * totala + rows_a * totalb)

Analyzing The Algorithm (2)

- Compared with matrix multiplication using array
  - ```
    for (i =0; i < rows_a; i++)
        for (j=0; j < cols_b; j++) {
            sum =0;
            for (k=0; k < cols_a; k++)
                sum += (a[i][k] *b[k][j]);
            d[i][j] =sum;
        }
    ```
  - O(rows_a * cols_a * cols_b) *v.s.* O(cols_b * total_a + rows_a * total_b)
    ```

Analyzing The Algorithm (3)

- Optimal case
  total_a < rows_a * cols_a
  total_b < cols_a * cols_b
- Worse case
  total_a --> rows_a * cols_a, or
  total_b --> cols_a * cols_b

THE REPRESENTATION OF MULTIDIMENSIONAL ARRAYS

Multidimensional Arrays (1)

- If an array is declared
  $a[upper_0][upper_1]\ldots[upper_{n\text{-}1}]$,
  then it is easy to see that the number of elements in the array is

$$\prod_{i=0}^{n-1} upper_i$$

  - Where Π is the product of the $upper_i$'s.
  - Example
    - If we declare *a* as *a*[10][10][10], then we require 10*10*10 = 1000 units of storage to hold the array

Multidimensional Arrays (2)

- Represent multidimensional arrays
  - *row major order* and *column major order*
- Row major order stores multidimensional arrays by rows
  - $A[upper_0][upper_1]$ as
    $upper_0$ rows: row_0, row_1, …, $row_{upper_0\text{-}1}$,
    each row containing $upper_1$ elements

Multidimensional Arrays (3)

- Row major order: $A[i][j] : \alpha + i \times upper_1 + j$
- Column major order: $A[i][j] : \alpha + j \times upper_0 + i$

	col_0	col_1		$col_{u_1\text{-}1}$
row_0	A[0][0] α	A[0][1] $\alpha+1$		A[0][u_1-1] $\alpha+(u_1\text{-}1)$
row_1	A[1][0] $\alpha+u_1$	A[1][1] $\alpha+u_1+1$		A[1][u_1-1] $\alpha+2u_1\text{-}1$
$row_{u_0\text{-}1}$	A[u_0-1][0] $\alpha+(u_0\text{-}1)u_1$	A[u_0-1][1] $\alpha+(u_0\text{-}1)u_1+1$		A[u_0-1][u_1-1] $\alpha+u_0u_1\text{-}1$

Multidimensional Arrays (4)

- To represent a three-dimensional array, $A[upper_0][upper_1][upper_2]$, we interpret the array as $upper_0$ two-dimensional arrays of dimension $upper_1 \times upper_2$
  - To locate $a[i][j][k]$, we first obtain $\alpha + i \times upper_1 \times upper_2$ as the address of $a[i][0][0]$ because there are i two dimensional arrays of size $upper_1 \times upper_2$ preceding this element

    $$\alpha + i \times upper_1 \times upper_2 + j \times upper_2 + k$$

    as the address of $a[i][j][k]$

Multidimensional Arrays (5)

- Generalizing on the preceding discussion, we can obtain the addressing formula for any element $A[i_0][i_1]\ldots[i_{n-1}]$ in an n-dimensional array declared as

  $A[upper_0][upper_1]\ldots[upper_{n-1}]$

  – The address for $A[i_0][i_1]\ldots[i_{n-1}]$ is

$$\begin{aligned}
\alpha &+ i_0 upper_1 upper_2 \ldots upper_{n-1} \\
&+ i_1 upper_2 upper_3 \ldots upper_{n-1} \\
&+ i_2 upper_3 upper_4 \ldots upper_{n-1} \\
&\;\;\vdots \\
&+ i_{n-2} upper_{n-1} \qquad = \alpha + \sum_{j=0}^{n-1} i_j a_j \\
&+ i_{n-1}
\end{aligned}$$

$$\text{where: } \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ a_{n-1} = 1 \end{cases}$$

STRING MATCHING

Knuth, Morris, Pratt Pattern Matching Algorithm (1)

string   ababababbababaabb

pattern   ababbababaa

- Failure function
  - If $p = p_0 p_1 \ldots p_{n-1}$ is a pattern, then its failure function, f, is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 p_1 \ldots p_i = p_{j-i} p_{j-i+1} \ldots p_j \text{ if such an } i \geq 0 \text{ exists} \\ -1 \quad otherwise \end{cases}$$

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

Knuth, Morris, Pratt Pattern Matching Algorithm (2)

i	0	1	2	3	4	5	6	7	8	9	10
	a	b	a	b	b	a	b	a	b	a	a
f	-1	-1	0	1	-1	0	1	2	3	2	

j

```
void fail(char *pat)
{
    int n = strlen(pat);
    failure[0] = -1;
    for (j = 1; j < n; j++) {
        i = failure[j-1];
        while ((pat[j] != pat[i+1] && (i >= 0))
            i = failure[i];
        if (pat[j] == pat[i+1])
            failure[j] = i+1;
        else failure[j] = -1;
    }
}
```

j = 9
i = failure[9-1] = 3
pat[j] != pat[i+1]
i = failure[3] = 1
pat[j] = pat[i+1]
failure[j] = i+1=2

Knuth, Morris, Pratt Pattern Matching Algorithm (3)

i	0	1	2	3	4	5	6	7	8	9	10
	a	b	a	b	b	a	b	a	b	a	a
f	-1	-1	0	1	-1	0	1	2	3		

string abababbabababaabb

pattern ababbababaa

j = failure[j-1]+1=1+1=2

Knuth, Morris, Pratt Pattern Matching Algorithm (4)

```
int pmatch(char *string, char *pat)
{
    int i = 0, j = 0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while ( i < lens && j < lenp ) {
        if (string[i] == pat[j]) {
            i++;
            j++;
        } else if (j == 0)
            i++;
        else
            j = failure[j-1]+1;
        }
    return ( (j == lenp) ? (i-lenp) : -1);
}
```