Data Structures: Arrays and Structures

Wei-Mei Chen

Department of Electronic and Computer Engineering
National Taiwan University of Science and Technology

Introduction

Arrays

1. Array: a set of pairs, $\langle index, value \rangle$
2. Data structure

   For each index, there is a value associated with that index.
3. Representation
   - An array can be implemented by using consecutive memory.
   - We may call this a correspondence or a mapping.
4. When considering an ADT we are more concerned with the operations that can be performed on an array.

ADT Array

structure *Array*

objects: A set of pairs $\langle index, value \rangle$ where for each value of index there is a value from the set item. Let *index* be a finite ordered set of one or more dimensions, for example, $\{0, \dots, n-1\}$ 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 size of the ith dimension.
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 $\langle i, x \rangle$ has been insert **else return** error.

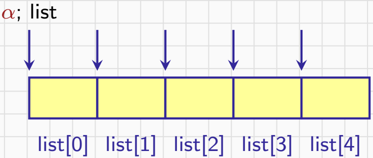
Arrays in C

`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 $= \alpha$
  - list[1]: $\alpha +$ sizeof(int)
  - list[i]: $\alpha +$ i*sizeof(int)



- list: a pointer to list[0]

  (list + i): a pointer to list[i] (i.e. &list[i])

  *(list + i): list[i]

One-Dimensional Array Addressing

☞ Consider

```
int one[] = {0, 1, 2, 3, 4};
```

Goal: print out address and value

```
print1(&one[0], 5);
```

```
void print1( int *ptr, int rows)
{
    int i;
    printf("Address Contents\n");
    for( i = 0; i < rows; i++ )
        printf("%8u%5d\n", ptr+i, *(ptr+i));
    printf("\n");
}
```

Address	Contents
12344868	0
12344872	1
12344876	2
12344880	3
12344884	4

Dynamically Allocation

How Large Should the Size of Your Array Be?

- A good solution is to defer this decision to run time and allocate the array.

```
int i, n, *list;
printf("Enter the number of numbers to generate:  ");
scanf("%d", &n);
if( n < 1 ){
    fprintf(stderr, "Improper value of n \n");
    exit(EXIT_FAILURE);
}
MALLOC(list, n*sizeof(int));
```

How Large Should the Size of Your Array Be?

- A good solution is to defer this decision to run time and allocate the array.

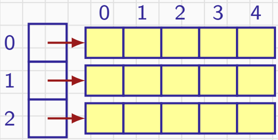
```
int i, n, *list;
printf("Enter the number of numbers to generate:  ");
scanf("%d", &n);
if( n < 1 ){
   fprintf(stderr, "Improper value of n \n");
   exit(EXIT_FAILURE);
}
MALLOC(list, n*sizeof(int));
```

✗ `list[MAX_SIZE]`

```
#define MALLOC(p,s)
  if(!((p)=malloc(s) ) ){
    fprintf(stderr,"Insufficient memory");
}
```

Two-Dimensional Arrays

- Array of arrays representation
- Example `int x[3][5];`



- x[i][j]
  1. accessing the pointer in x[i] ➠ the [0]th element of row i.
  2. adding j*sizeof(int) ➠ the [j]th element of row i

Multidimensional Arrays

- Dynamically create a 2-D array

```
int** make2dArray(int rows, int cols)
{   int **x, i;
    MALLOC( x, rows * sizeof(*x));
    for ( i = 0; i < rows; i++)
        MALLOC(x[i], cols * sizeof(**x));
    return x;
}
```

```
int **myArray;
myArray = make2dArray(5,10);
myArray[2][4] = 6;
```

- **calloc**: allocation and initialization
  realloc: grows or shrinks a block of memory
- A three-dimensional array is represented as a one-dimensional array, each whose elements is a two-dimensional array.

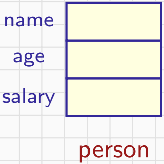
Structures and Unions

Structures

- Arrays are collections of data of the same type.
- A structure is a collection of data items, where each item is identified as to its type and name.
  - ☞ This mechanism is called the `struct`.

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


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



•: structure member operator

Structures

- Arrays are collections of data of the same type.
- A structure is a collection of data items, where each item is identified as to its type and name.
  - ☞ This mechanism is called the `struct`.

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


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

name	james
age	10
salary	35000

person

•: structure member operator

Create Your Structure Data Type

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

```
humanBeing person1, person2;


int humansEqual (humanBeing person1, humanBeing person2)
{
    if (strcmp(person1.name, person2.name))
        return FALSE;
    if (person1.age != person2.age)
        return FALSE;
    if (person1.salary != person2.salary)
        return FALSE;
    return TRUE;
}
```

Embed a Structure within a Structure

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

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

humanBeing person1;
person1.dob.month=5;
person1.dob.day=11;
person1.dob.year=1944;
```

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

```
typedef struct sexType {
    enum tagField {female, male} sex;
    union {
        int children;
        int beard;
    }u;
} sexType;
typedef struct humanBeing {
    char name[10];
    int age;
    float salary;
    date dob;
    sexType sexInfo;
} humanBeing;
```

```
humanBeing person1, person2;
person1.sexInfo.sex = male;
person1.sexInfo.u.beard = FALSE;
person2.sexInfo.sex = female;
person2.sexInfo.u.children = 4;
```

Internal Implementation of Structures

- ```
  struct {int i, j; float a, b;}
  struct {int i; int j; float a; float b;}
  ```
  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 must begin and end on the same type of memory boundary (e.g. a multiple of 4, 8, or 16).
  ```

Example

```c
#include "stdio.h"
union A {
  int i;          // 4 bytes
  char c[6];      // 6 bytes
};

struct B {
  int n;          // 4 bytes
  double m;       // 8 bytes
};

int main(void) {
  union A a;
  struct B b;
  printf("Size of A is %lu\n", sizeof(a));
  printf("Size of B is %lu\n", sizeof(b));
  return 0;
}
```

```
Size of A is 8
Size of B is 16
```

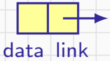
Self-Referential Structures

One or more of its components is a pointer to itself.

```
typedef struct list{
   char data;
   struct list *link;
} list;

list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;

item1.link = &item2;
item2.link = &item3;
```



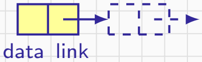
Self-Referential Structures

One or more of its components is a pointer to itself.

```
typedef struct list{
    char data;
    struct list *link;
} list;

list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;

item1.link = &item2;
item2.link = &item3;
```



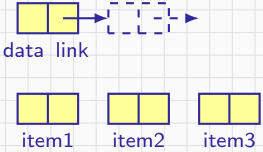
Self-Referential Structures

One or more of its components is a pointer to itself.

```
typedef struct list{
    char data;
    struct list *link;
} list;

list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;

item1.link = &item2;
item2.link = &item3;
```



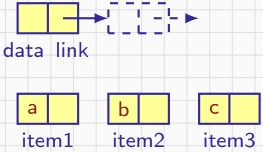
Self-Referential Structures

One or more of its components is a pointer to itself.

```
typedef struct list{
    char data;
    struct list *link;
} list;

list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;

item1.link = &item2;
item2.link = &item3;
```



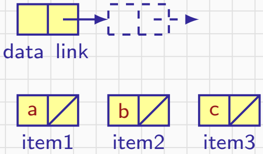
Self-Referential Structures

One or more of its components is a pointer to itself.

```
typedef struct list{
    char data;
    struct list *link;
} list;

list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;

item1.link = &item2;
item2.link = &item3;
```



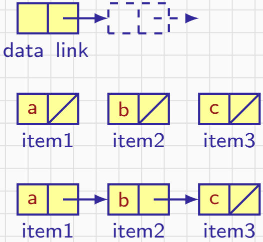
Self-Referential Structures

One or more of its components is a pointer to itself.

```
typedef struct list{
   char data;
   struct list *link;
} list;

list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;

item1.link = &item2;
item2.link = &item3;
```



Polynomials

Ordered Lists

Ordered (linear) list: (item$_1$, item$_2$, item$_3$,$\cdots$, 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)

Operations on Ordered Lists

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

➠ **Implementation:**
sequential mapping or non-sequential mapping.

Polynomials

- Two polynomials are:
  $A(x) = 3x^{20} + 2x^3$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$
- Assume that we have two polynomials,

$$A(x) = \sum a_i x^i \text{ and } B(x) = \sum b_i x^i,$$

  then:

$$A(x) + B(x) = \sum \Big(a_i + b_i\Big) x^i$$

$$A(x) \cdot B(x) = \sum \Big(a_i x^i \cdot \sum \Big(b_j x^j\Big)\Big)$$

- Similarly, we can define subtraction and division on polynomials, as well as many other operations.

ADT Polynomial

structure *Polynomial*

objects: $p(x) = a_1x^{e_1} + a_2x^{e_2} + \cdots + a_nx^{e_n}$; a set of ordered pairs of $\langle e_i, a_i \rangle$ 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 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 $\langle coef, expon \rangle$ 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($poly1$, $poly2$)	::=	**return** the polynomial $poly1 + poly2$
Polynomial Mult($poly1$, $poly2$)	::=	**return** the polynomial $poly1 \cdot poly2$

Initial Version of Function padd

$D(x) = A(x) + B(x)$

```
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));
    }
}
```

To simply operations ➠ "expon": in decreasing order

Polynomial Representation

- Representation I

```
#define MAX_DEGREE 101
typedef struct polynomial{
    int degree;
    float coef[MAX_DEGREE];
} polynomial;
```

If a is of type **polynomial** and

$A(x) = \sum_{i=0}^{n} a_i x^i$

a.degree $= n$ and

a.coef[i] $= a_{n-i}$ or $= 0$

☞ This representation is very simple, but wastes space.

a.degree $<<$ **MAX_DEGREE** (much less than)

- Representation II

```
#define MAX_TERMS 100
typedef struct polynomial{
    float coef;
    int expon;
} polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

The total terms must no more than **MAX_DEGREE**.

An Example of Polynomials

- Use one global array to store all polynomials
  These polynomials are stored in the array *terms*
- $A(x) = 2x^{1000} + 1$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$

	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

Poly	$< start, finish >$
A	$< 0, 1 >$
B	$< 2, 5 >$

- storage requirements: *start*, *finish*, $2 * (finish - start + 1)$
  - ➠ non-sparse: twice as much as Representation I
    when all the items are nonzero

```
void padd(int startA,int finishA,int startB,int finishB,int *startD,int *finishD)
{  float coefficient;
   *startD = avail;
   while (startA <= finishA && startB <= finishB)
      switch(COMPARE(terms[startA].expon, terms[startB].expon)) {
         case -1:  attach(terms[startB].coef, terms[startB].expon);
             startB++;
             break;
         case 0:  coefficient = terms[startA].coef + terms[startB].coef;
             if (coefficient)
                attach(coefficient, terms[startA].expon);
             startA++;
             startB++;
             break;
         case 1:  attach(terms[startA].coef, terms[startA].expon);
             startA++;
             break;
   }
   for( ; startA <= finishA; startA++)
      attach(terms[startA].coef,terms[startA].expon);
   for( ; startB <= finishB; startB++)
      attach(terms[startB].coef,terms[startB].expon);
   *finishD = avail - 1;
}
```

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

Function to Add a New Term

```
void attach(float coefficient, int exponent)
{
    if (avail >= MAX_TERMS) {
        fprintf(stderr, "Too many terms in the polynomial \n");
        exit(1);
    }
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;
}
```

Problem: Compaction is required when polynomials that are no longer needed.

Sparse Matrices

Matrices

- A matrix contains m rows and n columns of elements, we write $m \times n$ to designate a matrix with m rows and n columns.

$$
\begin{array}{c}
\begin{array}{cccc} & c_0 & c_1 & c_2 \end{array} \\
\begin{array}{c} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{array}
\begin{bmatrix}
-27 & 3 & 4 \\
6 & 82 & -2 \\
109 & -64 & 11 \\
12 & 8 & 9 \\
48 & 27 & 47
\end{bmatrix}_{5 \times 3}
\end{array}
\qquad
\begin{array}{c}
\begin{array}{ccccccc} & c_0 & c_1 & c_2 & c_3 & c_4 & c_5 \end{array} \\
\begin{array}{c} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{array}
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}_{6 \times 6}
\end{array}
$$

- 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

- 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 $\langle row, col, value \rangle$.

```
#define MAX_TERMS 101
typedef struct term {
    int col;
    int row;
    int value;
} term;
term a[MAX_TERMS];
```

ADT Sparse_Matrix

A minimal set of operations includes matrix creation, addition, multiplication, and transpose.

structure *Sparse_Matrix*

objects: a set of triples, $\langle row, column, value \rangle$, 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_Marix 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 and b according to the formula $d[i][j] = \sum(a[i][k] \cdot b[k][j])$ **else return** error

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

- for each **row** i

  take element $\langle i, j, value \rangle$ and store it

  as element $\langle j, i, value \rangle$ of the transpose.
- Difficulty: where to place $\langle j, i, value \rangle$?
- for all elements in **column** j,

  place element $\langle i, j, value \rangle$ in element $\langle j, i, value \rangle$

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

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

- for each **row** i
  take element $\langle i, j, value\rangle$ and store it
  as element $\langle j, i, value\rangle$ of the transpose.
- Difficulty: where to place $\langle j, i, value\rangle$?
- for all elements in **column** j,
  place element $\langle i, j, value\rangle$ in element $\langle j, i, value\rangle$

Transpose a Matrix

```
void transpose (term a[], term b[])
{
    int n,i,j,currentb;
    n = a[0].value;
    b[0].row = a[0].col;
    b[0].col = a[0].row;
    b[0].value = n;
    if (n > 0)
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
            for (j = 1; j<= n; j++)
                if (a[j].col == i) {
                    b[currentb].row = a[j].col;
                    b[currentb].col = a[j].row;
                    b[currentb].value = a[j].value;
                    currentb++;
                }
    }
}
```

Transpose a Matrix

```
void transpose (term a[], term b[])
{
    int n,i,j,currentb;
    n = a[0].value;
    b[0].row = a[0].col;
    b[0].col = a[0].row;
    b[0].value = n;
    if (n > 0)
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
            for (j = 1; j<= n; j++)
                if (a[j].col == i) {
                    b[currentb].row = a[j].col;
                    b[currentb].col = a[j].row;
                    b[currentb].value = a[j].value;
                    currentb++;
                }
    }
}
```

Scan the array "columns" times. The array has "elements" elements.

➠ $O(columns * elements)$

$O(columns * elements)$ vs. $O(columns * rows)$

- The complexity $O(columns * elements)$
  If the matrix is **not** sparse, $elements \rightarrow columns * rows$
  Then $O(columns * elements) = O(columns * columns * rows)$.
  It spends too much time to reduce the storage .
- Consider the 2-D array representation

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

  ➠ $O(columns * rows)$
- Problem of **transpose**: Scan the array "columns" times.
  ➠ Determine the number of elements in each column of the original matrix.
  ➠ the starting positions of each row in the transpose matrix

```
void fastTranspose(term a[], term 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){
      for (i = 0; i < numCols; i++)
         rowTerms[i] = 0;
      for (i = 1; i <= numTerms; i++)
         rowTerm[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;
      }
   }
}
```

$O(columns * rows)$ vs. $O(columns + elements)$

	[0]	[1]	[2]	[3]	[4]	[5]
$rowTerms =$	$\underline{2}$	1	2	2	0	1
$startingPos =$	$\underline{1}$	$\underline{3}$	4	6	8	8

	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

- Compared with 2-D array representation $O(columns * rows)$
- $O(columns + elements) = O(columns + columns * rows)$
  $= O(columns * rows)$
- Cost: Additional *rowTerms* and *startingPos* arrays are required.
  ➠ Share!

Matrix Multiplication

- Definition: Given A and B where $A_{m \times n}$ and $B_{n \times p}$, the product matrix D has dimension $m \times p$. Its $\langle i, j \rangle$ element is

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \text{ for } 0 \leq i < m \text{ and } 0 \leq j < p.$$

- Example

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

Sparse Matrix Multiplication

- If the triples of matrices are ordered by row and within rows by columns

  Step 1: Pick row i of A

  Step 2: Find all elements in column j of B (by scanning)

- Avoiding to sacn all of B for all elements in j
  - ☞ Compute B^T OR
  - ☞ Put all column elements in consecutive order.

$$A = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 4 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 0 & 2 \\ -1 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix} \quad B^T = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 5 \end{bmatrix}$$

Sparse Matrix Multiplication

- If the triples of matrices are ordered by row and within rows by columns

  Step 1: Pick row i of A

  Step 2: Find all elements in column j of B (by scanning)

- Avoiding to sacn all of B for all elements in j

  ☞ Compute B^T OR

  ☞ Put all column elements in consecutive order.

$$A = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 4 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 0 & 2 \\ -1 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix} \quad B^T = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 5 \end{bmatrix}$$

B

b[0]	3	3	4
b[1]	0	0	3
b[2]	0	2	2
b[3]	1	0	−1
b[4]	2	2	5

B^T

b[0]	3	3	4
b'[1]	0	0	3
b'[2]	0	1	−1
b'[3]	2	0	2
b'[4]	2	2	5

```
void mmult(term a[], term b[], term d[])
.........
    fastTranspose(b,newB);
    a[totalA+1].row = rowsA;
    newB[totalB+1].row = colsB;
    newB[totalB+1].col = 0;
    for (i = 1; i <= totalA; ) {
        column = newB[1].row;
        for (j = 1; j <= totalB+1; ) {
            if (a[i].row != row) {
                storeSum(d,&totalD,row,column,&sum);
                i = rowBegin;
                for (;newB[j].row == column; j++)
                    ;
                column = newB[j].row;
            };
            else if (newB[j].row != column) {
                storeSum(d,&totalD,row,column,&sum);
                i = rowBegin;
                column = newB[j].row;
            }
            else switch (COMPARE(a[i].col, newB[j].col)) {
                case -1:
                    i++; break;
                case 0:
                    sum += ( a[i++].value * newB[j++].value);
                    break;
                case 1:
                    j++;
                }
            }
            .........
    }
```

```
void mmult(term a[], term b[], term d[])
.........
    fastTranspose(b,newB);
    a[totalA+1].row = rowsA;
    newB[totalB+1].row = colsB;
    newB[totalB+1].col = 0;
    for (i = 1; i <= totalA; ) {
        column = newB[1].row;
        for (j = 1; j <= totalB+1; ) {
            if (a[i].row != row) {
                storeSum(d,&totalD,row,column,&sum);
                i = rowBegin;
                for (;newB[j].row == column; j++)
                    ;
                column = newB[j].row;
            };
            else if (newB[j].row != column) {
                storeSum(d,&totalD,row,column,&sum);
                i = rowBegin;
                column = newB[j].row;
            }
            else switch (COMPARE(a[i].col, newB[j].col)) {
                case -1:
                    i++; break;
                case 0:
                    sum += ( a[i++].value * newB[j++].value);
                    break;
                case 1:
                    j++;
                }
            }
            .........
    }
```

$$O(\sum_{rows}(colsB \cdot termsRow + totalB))$$
$$= O(colsB \cdot totalA + rowsA \cdot totalB)$$

Matrix Multiplication Using Arrays

```
for (i =0; i < rowsA; i++){
    for( j=0; j < colsB; j++ ) {
        sum =0;
        for( k=0; k < colsA; k++ ){
            sum += (a[i][k] *b[k][j]);
            d[i][j] =sum;
        }
    }
}
```

➠ $O(rowsA \cdot colsA \cdot colsB)$

- If $totalA = colsA \cdot rowsA$ and $totalB = colsB \cdot rowsB$
  $\Rightarrow$ **mmult** is slower
- If $totalA << colsA \cdot rowsA$ and $totalB << colsB \cdot rowsB$
  $\Rightarrow$ **mmult** is faster

Representation of Two-Dimensional Arrays

- Represent multidimensional arrays:

  row major order and column major order

  ☞ Row major order stores multidimensional arrays by rows.

- $A[u_0][u_1]$ as u_0 rows

  ➠ $row_0, row_1, \cdots, row_{u_0-1}$ and

  each row containing u_1 elements.

  Assume that α is the address of $A[0][0]$, denoted by $@(A[0][0])$

  $\Rightarrow$ $@(A[i][0]) = \alpha + i \cdot u_1$ and $@(A[i][j]) = \alpha + i \cdot u_1 + j$

$[0][0]$	$\cdots$	$\cdots$	$[0][u_1-1]$
$[1][0]$	$\cdots$	$\cdots$	$[1][u_1-1]$
$\cdots$	$[i][j]$		$\cdots$
$\cdots$			$\cdots$
$[u_0-1][0]$	$\cdots$	$\cdots$	$[u_0-1][u_1-1]$

Representation of Two-Dimensional Arrays

- Represent multidimensional arrays:

  row major order and column major order

  ☞ Row major order stores multidimensional arrays by rows.

- $A[u_0][u_1]$ as u_0 rows

  ➠ $row_0, row_1, \cdots, row_{u_0-1}$ and

  each row containing u_1 elements.

  Assume that α is the address of $A[0][0]$, denoted by $@(A[0][0])$

  $\Rightarrow @(A[i][0]) = \alpha + i \cdot u_1$ and $@(A[i][j]) = \alpha + i \cdot u_1 + j$

$[0][0]$	$\cdots$	$\cdots$	$[0][u_1-1]$
$[1][0]$	$\cdots$	$\cdots$	$[1][u_1-1]$
$\cdots$		$[i][j]$	$\cdots$
$\cdots$			$\cdots$
$[u_0-1][0]$	$\cdots$	$\cdots$	$[u_0-1][u_1-1]$

$\boxed{row_0}$

Representation of Two-Dimensional Arrays

- Represent multidimensional arrays:

  row major order and column major order

  ☞ Row major order stores multidimensional arrays by rows.

- $A[u_0][u_1]$ as u_0 rows

  ➠ $row_0, row_1, \cdots, row_{u_0-1}$ and

  each row containing u_1 elements.

  Assume that α is the address of $A[0][0]$, denoted by $@(A[0][0])$

  $\Rightarrow @(A[i][0]) = \alpha + i \cdot u_1$ and $@(A[i][j]) = \alpha + i \cdot u_1 + j$

$[0][0]$	$\cdots$	$\cdots$	$[0][u_1-1]$
$[1][0]$	$\cdots$	$\cdots$	$[1][u_1-1]$
$\cdots$	$[i][j]$		$\cdots$
$\cdots$			$\cdots$
$[u_0-1][0]$	$\cdots$	$\cdots$	$[u_0-1][u_1-1]$

row_0	row_1

Representation of Two-Dimensional Arrays

- Represent multidimensional arrays:

  row major order and column major order

  ☞ Row major order stores multidimensional arrays by rows.

- $A[u_0][u_1]$ as u_0 rows

  ➠ $row_0, row_1, \cdots, row_{u_0-1}$ and

  each row containing u_1 elements.

  Assume that α is the address of $A[0][0]$, denoted by $@(A[0][0])$

  $\Rightarrow @(A[i][0]) = \alpha + i \cdot u_1$ and $@(A[i][j]) = \alpha + i \cdot u_1 + j$

$[0][0]$	$\cdots$	$\cdots$	$[0][u_1-1]$
$[1][0]$	$\cdots$	$\cdots$	$[1][u_1-1]$
$\cdots$	$[i][j]$		$\cdots$
$\cdots$			$\cdots$
$[u_0-1][0]$	$\cdots$	$\cdots$	$[u_0-1][u_1-1]$

row_0	row_1	row_2	$\cdots$	row_{u_0-1}

Representation of Multidimensional Arrays

α: addr of the first element

- $A[u_0][u_1][u_2]$ ➠ u_0 $(u_1 \times u_2)$ arrays.
  $\Rightarrow$ $@(A[i][0][0]) = \alpha + i \cdot u_1 \cdot u_2$ and

$$
\begin{aligned}
@(A[i][j][k]) = \alpha \quad & + \quad i \cdot u_1 \cdot u_2 \\
& + \quad j \cdot u_2 \\
& + \quad k
\end{aligned}
$$

Representation of Multidimensional Arrays

α: addr of the first element

- $A[u_0][u_1][u_2]$ ➠ u_0 $(u_1 \times u_2)$ arrays.
  $\Rightarrow$ $@(A[i][0][0]) = \alpha + i \cdot u_1 \cdot u_2$ and

$$\begin{aligned} @(A[i][j][k]) = \alpha \quad & + \quad i \cdot u_1 \cdot u_2 \\ & + \quad j \cdot u_2 \\ & + \quad k \end{aligned}$$

- $A[u_0][u_1] \cdots [u_{n-1}]$
  $\Rightarrow$ $@(A[i_0][0] \cdots [0]) = \alpha + i_0 \cdot u_1 \cdots u_{n-1}$ and

$$\begin{aligned} @(A[i_0][i_1] \cdots [i_{n-1}]) = \alpha \quad & + \quad i_0 \cdot u_1 \cdots u_{n-1} \\ & + \quad i_1 \cdot u_2 \cdots u_{n-1} \\ & + \quad \cdots\cdots \\ & + \quad i_{n-2} \cdot u_{n-1} \\ & + \quad i_{n-1} \end{aligned}$$