제2장 Arrays and structures

경북대학교 임경식 교수

2.1 Arrays

Intuitively an array is a set of pairs, <*index, value*>, such that each index that is defined has a value associated with it.

ADT *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*

ADT 2.1: Abstract Data Type *Array*

Arrays in C

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
void main(void)
{
   int i;
   for (i = 0; i < MAX_SIZE; i++)
      input[i] = i;
   answer = sum(input, MAX_SIZE);
   printf("The sum is: %f\n", answer);
}
float sum(float list[], int n)
{
   int i;
   float tempsum = 0;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

Program 2.1: Example array program

2.2 Dynamically allocated arrays

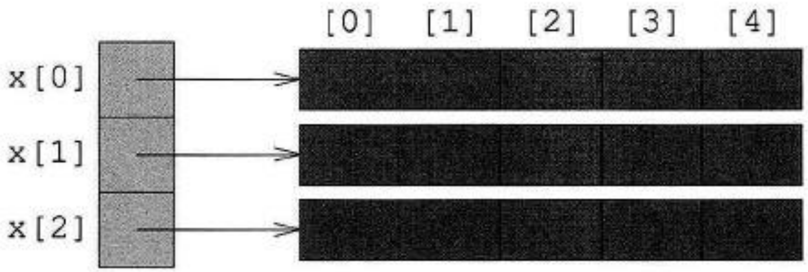
- One-dimensional arrays

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

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));
```

2.2 Dynamically allocated arrays

- Two-dimensional arrays (e.g. int x [3] [5] ;)




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

```
int** make2dArray(int rows, int cols)
{/* create a two dimensional rows × cols array */
   int **x, i;

   /* get memory for row pointers */
   MALLOC(x, rows * sizeof (*x));;

   /* get memory for each row */
   for (i = 0; i < rows; i++)
     MALLOC(x[i], cols * sizeof(**x));
   return x;
}
```

2.3 Structures and unions

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

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

2.3 Structures and unions

```
typedef struct humanBeing {    or    typedef struct {
        char name[10];                       char name[10];
        int age;                             int age;
        float salary;                        float salary;
        };                                   } humanBeing;
humanBeing person1, person2;
```

```
if (humansEqual(person1,person2))
   printf("The two human beings are the same\n");
else
   printf("The two human beings are not the same\n");
```

```
int humansEqual(humanBeing person1,
                             humanBeing person2)
{/* return TRUE if person1 and person2 are the same human
    being otherwise return FALSE */
   if (strcmp(person1.name, person2.name))
      return FALSE;
   if (person1.age != person2.age)
      return FALSE;
   if (person1.salary != person2.salary)
      return FALSE;
   return TRUE;
}
```

2.3 Structures and unions

- Nested structures

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

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

humanBeing person1, person2;

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

2.3 Structures and unions

- Self-referential structures

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

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;
```

2.4 Polynomials

❖ Ordered or linear list

- Days of the week: (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
- Values in a deck of cards: (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)

❖ Operations

- Finding the length, n, of a list.
- Reading the items in a list from left to right (or right to left).
- Retrieving the ith item from a list, $0 \le i < n$.
- Replacing the item in the ith position of a list, $0 \le i < n$.
- Inserting a new item in the ith position of a list, $0 \le i \le n$. The items previously numbered $i, i+1, \cdots, n-1$ become items numbered $i+1, i+2, \cdots, n$.
- Deleting an item from the ith position of a list, $0 \le i < n$. The items numbered $i+1, \cdots, n-1$ become items numbered $i, i+1, \cdots, n-2$.

❖ Implementation – sequential mapping using arrays

2.4 Polynomials

- A polynomial is a sum of terms, where each term has a form *ax^e,* where *x* is the variable, *a* is the coefficient, and *e* is the exponent.

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*, *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 LeadExp(*poly*)	::=	**return** the largest exponent in *poly*

2.4 Polynomials

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 Polynomials

```
/* 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

2.4 Polynomials

❖ Polynomial representation

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

Now if *a* is of type *polynomial* and $n <$ *MAX_DEGREE*, the polynomial $A(x) = \sum_{i=0}^{n} a_i x^i$ would be represented as:

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

➢ If a.degree << MAX_DEGEREE, waste a lot of spaces

➢ If polynomial is sparse, waste a lot of spaces

2.4 Polynomials

❖ Polynomial representation

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

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

Our specification used *poly* to refer to a polynomial, and our representation translated *poly* into a *<start, finish* > pair. Therefore, to use *A (x)* we must pass in *startA and finishA.*

Polynomial addition

```
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++;
  }
```

Polynomial addition

```
   /* 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;
}

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;
}
```

Time complexity O(n+m) and
Space complexity O(n+m),
where n & m are the # of nonzero terms in A and B, respectively.

2.5 Sparse matrices

As computer scientists, our interest centers not only on the specification of an appropriate ADT, but also on finding representations that let us efficiently perform the operations described in the specification.

$$
\begin{array}{cc}
 & \begin{array}{ccc} \text{col } 0 & \text{col } 1 & \text{col } 2 \end{array} \\
\begin{array}{c} \text{row } 0 \\ \text{row } 1 \\ \text{row } 2 \\ \text{row } 3 \\ \text{row } 4 \end{array} &
\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}
\end{array}
\qquad
\begin{array}{cc}
 & \begin{array}{cccccc} \text{col } 0 & \text{col } 1 & \text{col } 2 & \text{col } 3 & \text{col } 4 & \text{col } 5 \end{array} \\
\begin{array}{c} \text{row } 0 \\ \text{row } 1 \\ \text{row } 2 \\ \text{row } 3 \\ \text{row } 4 \\ \text{row } 5 \end{array} &
\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}
\end{array}
$$

2.5 Sparse matrices

❖ Specification of a sparse matrix ADT

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.

2.5 Sparse matrices

❖ Sparse matrix represenation

	col 0	col 1	col 2	col 3	col 4	col 5
row 0	15	0	0	22	0	−15
row 1	0	11	3	0	0	0
row 2	0	0	0	−6	0	0
row 3	0	0	0	0	0	0
row 4	91	0	0	0	0	0
row 5	0	0	28	0	0	0

=> <row, col, value>

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

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

Space complexity   O(row * col)

	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

Space complexity   O(elements)

2.5 Sparse matrices

❖ Transposing a sparse matrix

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

	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

Original sparse matrix

	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

Transposed sparse matrix

2.5 Sparse matrices

❖ Transposing a sparse matrix

```
for each row i
   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

(0, 0, 15), which becomes (0, 0, 15)
(0, 3, 22), which becomes (3, 0, 22)
(0, 5, −15), which becomes (5, 0, −15)
(1, 1, 11), which becomes (1, 1, 11)
(1, 2, 3), which becomes (2, 1, 3)

- Cannot maintain the correct order, resulting in data movement
- Time complexity O(elements^2)

2.5 Sparse matrices

❖ Transposing a sparse matrix

```
for all elements in column j
   place element <i, j, value> in
   element <j, i, value>
```

	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

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

➢ Time complexity   O(col * elements)

Remember that transposing a two-dimensional array takes O(row * col).

2.5 Sparse matrices

❖ Fast transposing a sparse matrix

➢ Time complexity O(col + elements)

	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

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

	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 (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];
```

2.5 Sparse matrices

```
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;
      }
   }
}
```

Thank You !

▌ 노력 없이 이룰 수 있는 것 아무것도 없다.