제 2 장 배열과 구조

2.1 추상 데이터 타입으로서의 배열

□ 프로그래머의 입장에서

- 배열은 “일련의 연속적인 메모리 위치”

⇒ <index, value>의 쌍으로 구성된 집합

- 대응 (correspondence) 또는 사상(mapping)이라 한다
  - 정해진 인덱스 각각에 대해 연관된 하나의 값이 있다

⇒ 추상 데이터 타입(ADT) 관점에서의 배열

- “일련의 연속적인 메모리 위치” 보다는 더 일반적인 구조

▶ 페이지 52, 추상 데이터 타입 Array

structure *Array*

objects : *index*의 각 값에 대하여 집합 *item*에 속한 한 값이 존재하는
<*index*, *value*>쌍의 집합. *index*는 일차원 또는 다차원의 유한순서 집합이다.

functions :

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

Array Create(*j*, *list*)
::= return *j*차원의 배열. 여기서 *list*는 *i*번째 원소가 *i*번째 차원의 크기인
j-tuple이며 *item*들은 정의되지 않았음.

Item Retrieve(*A*, *i*)
::= if ($i \in index$) return 배열 *A*의 인덱스 *i*값과 관련된 항목
else return 오류

Array Store(*A*, *i*, *x*)
::= if ($i \in index$) return 새로운 쌍 <*i*, *x*>가 삽입된 배열 *A*
else return 오류

end *Array*

□ C 언어에서의 배열

- 배열의 선언

  int *list*[5]; /* 다섯 개의 정수 값을 갖는 배열 */

  **plist*[5]; /* 정수를 가리키는 다섯 개의 포인터형 배열 */

- 배열의 구현

  *list*의 기본주소(base address) = *list*[0]의 주소 = α

  list[1] = α + sizeof(int)　　*list*[2] = $\alpha + 2 \times$ sizeof(int)

  list[3] = $\alpha + 3 \times$ sizeof(int)　　*list*[4] = $\alpha + 4 \times$ sizeof(int)

- 배열과 포인터

  int **list1*; int *list2*[5];

  /* *list1*과 *list2* 모두 정수를 가리키는 포인터

  *list2*의 경우에는 다섯 개의 정수를 위한 메모리 위치가 예약되어 있는 점이 다름 */

▶ 페이지 54, 프로그램 2.1 : 배열 프로그램의 예

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;
void main(void)
{
   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;
 }
```

▶ 페이지 54, 프로그램 2.2 : 주소를 사용한 일차원 배열의 접근

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

2.2 구조 및 유니언

2.2.1 구 조

□ 구조(structure) → **struct**

- 타입이 다른 데이터를 그룹화하는 방법

▶ 구조 예 → 페이지 55 ~ 58

- 구조체 변수의 선언

```
struct {
    char name[10];      /* 문자 배열로 된 이름 */
    int age;            /* 사람의 나이를 나타내는 정수 값 */
    float salary;       /* 각 개인의 월급을 나타내는 float 값 */
} person;

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

- 구조체 타입의 정의 및 사용

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

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

```
human_being person1, person2;

if (strcmp(person1.name, person2.name))
    printf("두 사람의 이름은 다르다.\n");
else
    printf("두 사람의 이름은 같다.\n");

strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

- 구조체 내의 구조체의 사용

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

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

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

2.2.2 유니언

□ 유니언(union)

- 모든 필드들이 메모리 공간을 공유한다

✧ **유니언 예 → 페이지 58 ~ 59**

```
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;

person1.sex_info.sex = male;
person1.sex_info.u.beard = FALSE;

person2.sex_info.sex = female;
person2.sex_info.u.children = 4;
```

2.2.3 구조의 내부 구현

- 메모리 상에서 구조의 필드를 어떻게 저장하는지 정확히 알 필요는 없다
  - 두 개의 연속적인 구성 요소가 메모리 상에서 적절히 맞추기 위해 구조 내에 빈 공간을 두거나 채워넣기를 할 수도 있다
  - 페이지 59, 중간 예 참고
  - padding, alignment
    - 일반적으로 word의 단위로 …

2.2.4 자체 참조 구조

□ 자체 참조 구조(self-referential structure)

- 구성 요소 중에 자기 자신을 가리키는 포인터가 한 개 이상 존재하는 구조
- 명시적으로 메모리를 할당받고 반납하기 위해 동적 기억장소 관리 루틴(malloc과 free)을 필요로 한다

▶ 자체 참조 구조 예 → 페이지 60

```
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;
```

2.3 다항식 추상 데이터 타입

2.3.1 순서 리스트

□ 배열은 다른 추상 데이터 타입의 구현에도 사용할 수 있다.

□ 순서 리스트 (ordered list) = 선형 리스트 (linear list)

- ✓ (일요일, 월요일, 화요일, 수요일, 목요일, 금요일, 토요일)
- ✓ (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
- ✓ () : empty list

⇒ 순서리스트의 추상적 표현

- ◆ $(item_0, item_1, ..., item_{n-1})$, $item_i$: 집합 S에 있는 원소

□ 순서 리스트 기본 연산들

- 리스트 길이 n의 계산
- 리스트 항목을 왼쪽에서 오른쪽 (오른쪽에서 왼쪽)으로 읽기
- 리스트로부터 i번째 항목 검색, $0 \le i < n$
- 리스트의 i번째 항목 대체, $0 \le i < n$
- 리스트의 i번째 위치에 새로운 항목을 삽입, $0 \le i < n$
  - $i, i+1, ..., n-1$ 항목 번호를 $i+1, i+2, ..., n$으로 만듦
- 리스트의 i번째 항목을 제거, $0 \le i < n$
  - $i+1, ..., n-1$ 항목 번호를 $i, i+1, ..., n-2$으로 만듦

⇒ 이러한 연산들을 효과적으로 수행할 수 있도록 순서 리스트를 표현

□ 배열

- 순서리스트를 표현하는 가장 보편적인 방법
- 순차적 사상(sequential mapping)
  - 배열 인덱스 i와 리스트 원소 $item_i$를 대응시키는 배열로 표현

→ 기억장소의 순차주소를 이용한 리스트에 대해 삽입과 삭제를 할 경우 순차적 사상을 유지하기 위해서는 데이타의 나머지 부분이 앞 혹은 뒤로 적절히 이동되어야 한다

→ overhead

→ 연결 리스트(linked list)

2.3.2 다항식

□ 기호로 표현된 다항식을 조작할 수 있도록 하는 부프로그램을 만든다

- 다항식을 컴퓨터 구조에 맞게 정의해야

□ 다항식

- 각 항이 ax^e의 형태로 이루어진 항들의 합
- 차수 (degree)
  - 다항식의 가장 큰 지수

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

□ 다항식 연산

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

- 합 : $A(x) + B(x) = \sum (a_i + b_i) x^i$
- 곱 : $A(x) \cdot B(x) = \sum (a_i x^i \cdot \sum b_j x^j)$

▶ 페이지 63 ~ 64, 구조 2.2 : *Polynomial* 추상 데이터 타입

struct *Polynomial*

objects : $P(x) = a_1x^{e1} + \ldots + a_nx^{en}$: $<e_i , a_i>$의 순서쌍으로 된 집합이다. 여기서, a_i는 *Coefficient*이고, e_i는 *Exponents*이다. e_i는 0 또는 0보다 큰 정수이다.

functions :

모든 *poly*, *poly*1, *poly*2 ∈ *polynomial*, *coef* ∈ *Coefficient*, *expon* ∈ *Exponents*에 대해

Polynomical Zero()	::= return 다항식, $p(x) = 0$
Boolean IsZero()	::= if (*poly*) return *FALSE* else return *TRUE*
Coefficient Coef(*poly*, *expon*)	::= if (*expon* ∈ *poly*) return 계수 else return 0
Exponent Leap_Exp(*poly*)	::= return *poly*에서 제일 큰 지수
Polynomial Attach(*poly*, *coef*, *expon*)	::= if (*expon* ∈ *poly*) return 오류 else return <*coef*, *expon*> 항이 삽입된 다항식 *poly*
Polynomial Remove(*poly*, *expon*)	::= if (*expon* ∈ *poly*) return 지수가 *expon*인 항이 삭제된 다항식 *poly* else return 오류
Polynomial SingleMult(*poly*, *coef*, *expon*)	::= return 다항식 $poly \cdot coef \cdot x_{expon}$
Polynomial Add(*poly*1, *poly*2)	::= return 다항식 *poly*1 + *poly*2
Polynomial Mult(*poly*1, *poly*2)	::= return 다항식 *poly*1 · *poly*2

end *Polynomial*

□ 다항식의 표현

- 표현상의 규칙
  - 지수는 유일하고 내림차순 이어야 한다
    : IsZero, Coef, Remove 등의 연산이 간편해진다

▶ 페이지 64, 프로그램 2.4 : 함수 *padd*의 초기 버전

```
/* d = a + b, 여기서 a, b, d는 다항식이다 */
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));
    }
}
```

□ 첫번째 표현 방법

- $A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

→ $A = (n, a_n, a_{n-1}, \ldots, a_1, a_0)$

✧ **페이지 63 ~ 64, 첫번째 표현 방법**

$$A(x) = \sum a_i x^i \quad \Longrightarrow \quad \begin{array}{l} \text{a.degree} = n \\ \text{a.coef}[i] = a_{n-i},\ 0 \le i \le n \end{array}$$

```
#define MAX_DEGREE  101  /* 다항식의 최대 차수 + 1 */
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
} Polynomial;
```

- 문제점
  - 많은 컴퓨터 기억 공간의 낭비를 초래
    : *a.degree* << MAX_DEGREE이거나 희소 다항식의 경우
  - ✓ $A(x) = 2x^{1000} + 1$

□ 두번째 표현 방법

- $A = (m, e_{m-1}, b_{m-1}, e_{m-2}, b_{m-2}, ..., e_0, b_0), e_{m-1} > e_{m-2} > ... > e_0 \geq 0$, m : 0이 아닌 항수

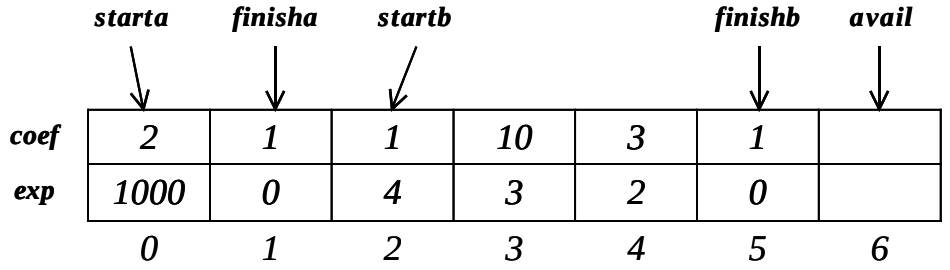
✧ **페이지 65, 두번째 표현 방법**

```
#define MAX_TERMS  100  /* 항 배열의 크기 */
typedef struct {
        float coef;
        int expon;
} Polynomial;
Polynomial terms[MAX_TERMS];
int avail = 0;
```

✧ **페이지 65, 그림 2.2 : 두 다항식의 배열 표현**

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

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



✧ 페이지 66 ~ 67, 프로그램 2.5 : 두 다항식을 더하는 함수

```
void padd(int starta, int finisha, int startb, int finishb,
          int *startd, int *finishd)
{   /* A(x)와 B(x)를 더하여 D(x)를 생성한다 */
    float coefficient;  *startd = avail;
    while (starta <= finisha && startb <= finishb)
        switch (COMPARE(terms[starta].expon, terms[startb].expon)) {
            case –1 :   /* a의 expon이 b의 expon보다 작은 경우 */
                        attach(terms[startb].coef, terms[startb].expon);
                        startb++; break;
            case  0 :   /* 지수가 같은 경우 */
                        coefficient = terms[starta].coef + terms[startb].coef;
                        if (coefficient) attach(coefficient, terms[starta].expon);
                        starta++; startb++; break;
            case  1 :   /* a의 expon이 b의 expon보다 큰 경우 */
                        attach(terms[starta].coef, terms[starta].expon);
                        starta++; break;
        }
    /* A(x)의 나머지 항들을 첨가한다 */
    for (; starta <= finisha; starta++)
        attach(terms[starta].coef, terms[starta].expon);
    /* B(x)의 나머지 항들을 첨가한다 */
    for (; startb <= finishb; startb++)
        attach(terms[startb].coef, terms[startb].expon);
    *finishd = avail – 1;
}
```

✍ 분석

: m : # of nonzero terms in A

: n : # of nonzero terms in B

: $O(n + m)$

◆ 문제점

– avail은 MAX_TERMS가 될 때까지 증가

: MAX_TERMS를 넘어가는 경우에는 불필요한 다항식이 존재하지 않는 한 연산 종료

– 불필요한 다항식이 있는 경우에 배열의 끝에 연속적인 가용 공간을 생성하는 압축 함수 구성해야

2.4 희소 행렬

2.4.1 개 요

□ 희소 행렬(sparse matrix)

◆ 많은 항들이 0인 행렬

✧ **페이지 59, 그림 2.3 : 두 행렬**

$$\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -0.3 \\ 109 & -64 & 4 \\ .12 & 8 & 9 \\ 3.4 & 36 & 27 \end{bmatrix} \qquad \begin{bmatrix} 15 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 28 \\ 22 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -15 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

– 오른쪽 행렬의 경우, 36개 항 중에서 8만이 0이 아니므로 희소 행렬이라 할 수 있다

□ 행렬의 표현 : 이차원 배열, *a*[MAX_ROWS][MAX_COLS]에 저장

- 어떤 원소도 *a[i][j]*로 표현하고 빠르게 찾을 수 있다

◆ 희소 행렬의 경우, 필요없는 기억 장소를 많이 차지하므로 낭비

- 특히, 매우 큰 행렬의 경우에는 새로운 기법에 따른 저장이 요구됨

→ 행렬의 0이 아닌 원소만을 기억시키는 표현법이 필요

⇒ 다른 방식으로 표현

◆ 3원소쌍 <row, col, value>의 사용

- 행번호가 증가하는 순서로 / 행이 같으면 열 번호가 증가하는 순서

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow$$

< 0, 0, 15 >
< 0, 3, 2 >
< 0, 5, -15 >
< 1, 1, 11 >
< 1, 2, 3 >
< 2, 3, -6>
< 4, 0, 91 >
< 5, 2, 28>

▶ 페이지 70, 구조 2.3 : *Sparse_Matrix* 추상 데이터 타입

struct *Sparse_Matrix*
  objects : 3원소쌍 <행, 열, 값>의 집합이다. 여기서, 행과 열은 정수이고 이 조합은 유일하며,
    값은 *item* 집합의 한 원소이다.
  functions :
    모든 $a, b \in$ *Spart_Matrix*, $x \in$ *item*, *i*, *j*, *max_col*, *max_row* $\in$ *index*에 대해
    Sparse_Matrix Create(*max_row*, *max_col*)
      ::= return *max_items*까지 저장할 수 있는 *Sparse_Matrix*,
        여기서 최대 행의 수는 *max_row*이고 최대 열의 수는 *max_col*이라
        할 때 *max_items* = *max_row* × *max_col*이다.
    Sparse_Matrix Transpose(*a*)
      ::= return 모든 3원소쌍의 행과 열의 값을 교환하여 얻은 행렬
    Sparse_Matrix Add(*a*, *b*)
      ::= if *a*와 *b*의 차원이 같으면 return 대응항들,
          즉 똑같은 행과 열의 값을 가진 항들을 더해서 만들어진 행렬
        else return 오류
    Sparse_Matrix Multiply(*a*, *b*)
      ::= if *a*의 열의 수와 *b*의 행의 수가 같으면
          return 다음 공식에 따라 *a*와 *b*를 곱해서 생성된 행렬 *d* :
          $d[i][j] = \Sigma(a[i][k] \cdot b[k][j])$, 여기서 *d*[*i*][*j*]는 (*i*, *j*)번째 요소이다
        else return 오류

→ 바뀐 표현 방법에 의해 더 많은 시간이 필요하다면 의미가 없다

▶ 페이지 70 ~ 71, 3원소 방식에 의한 표현 예

- 행번호가 증가하는 순서로 / 행이 같으면 열 번호가 증가하는 순

```
#define MAX_TERMS  101  /* 최대 항의 수 + 1 */
typedef struct {
        int col;
        int row;
        int value;
} Term;
Term a[MAX_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

2.4.2 행렬의 전치

□ 행렬의 전치?

- *a[i][j]* ➔ *b[i][j]*가 되는 것 !!

□ 삼원소 표현을 사용하지 않았을 때의 행렬의 전치

각 행 *i*에 대해서
원소 < *i*, *j* > 을 가져와서
전치 행렬의 원소 < *j*, *i*, 값 >으로 저장

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \begin{bmatrix} 15 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 28 \\ 22 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -15 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

▶ **페이지 71, 그림 2.4 : 원소쌍으로 저장된 희소 행렬과 전치 행렬**

	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

▶ 페이지 73, 프로그램 2.7 : 희소 행렬의 전치

```
void transpose ( term a[], term b[])
/* a를 전치시켜 b를 생성 */
{
    int n, i, j, currentb;
    n = a[0].value; /* 총 원소수 */
    b[0].row = a[0].col; b[0].col = a[0].row; b[0].value = n;
    if (n > 0) {      /* 0이 아닌 행렬 */
        currentb = 1;
        for (i = 0; i < a[0].col; i++)      /* a에서의 열별로 전치 */
            for (j = 1; j <= n; j++)        /* 현재의 열로부터 원소를 찾는다 */
                if (a[j].col == i) {        /* 현재의 열에 있는 원소를 b에 첨가한다 */
                    b[currentb].row = a[j].col; b[currentb].col = a[j].row;
                    b[currentb].value = a[j].value; currentb++;
                }
    }
}
```

✍ 분석

- 이차원 배열의 전치
  - : O(*rows* × *columns*)
- transpose 알고리즘
  - : O(*elements* × *columns*)
  - : O(*rows* × *columns*^{2})

→ 공간 절약으로 인해 시간 복잡도는 오히려 증가

→ better algorithm

- fast_transpose

▶ 페이지 74, 프로그램 2.8 : 희소 행렬의 빠른 전치

```
void fast_transpose ( term a[], term 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) {    /* 0이 아닌 행렬 */
        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;
        }
    }
}
```

✍ 분석

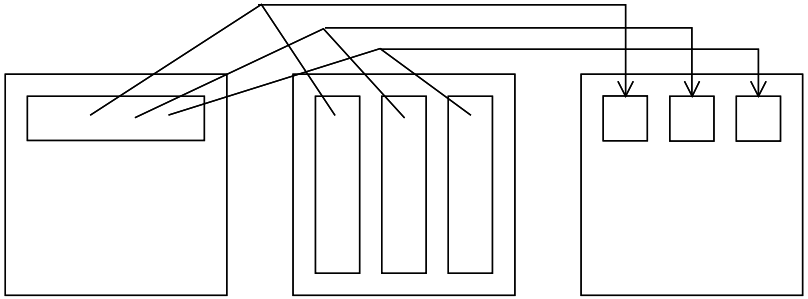
– O(*elements* + *columns*)

→ O(*rows* × *columns*)

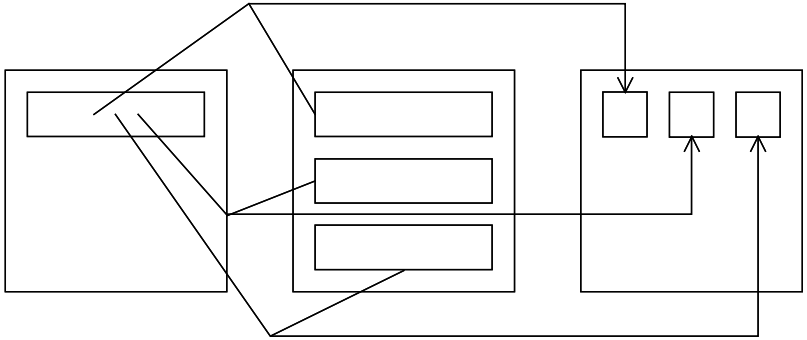
2.4.3 행렬 곱셈

▶ 일반적인 행렬 곱셈

- $C = A \times B$의 경우, A의 각 행과 B의 열을 곱하여 C의 원소로



⇒ 전치 행렬의 사용 : A와 B의 전치 행렬 B^T를 사용하면



- A의 각 행과 B의 행을 곱하여 C의 원소로

▶ 희소 행렬의 곱셈

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 225 & 0 & -420 & 330 & 0 & -225 \\ 0 & 121 & 33 & -18 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1365 & 0 & 0 & 2002 & 0 & -1365 \\ 0 & 0 & 0 & -168 & 0 & 0 \end{bmatrix}$$

	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	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*
c[0]	6	6	11
[1]	0	0	225
[2]	0	2	−420
[3]	0	3	330
[4]	0	5	−225
[5]	1	1	121
[6]	1	2	33
[7]	1	3	−18
[8]	4	0	1365
[9]	4	3	2002
[10]	4	5	−1365
[11]	5	3	−168

▶ 페이지 76 ~ 77, 프로그램 2.9 : 희소 행렬의 곱셈

```
void mmult ( term a[], term b[], term d[]) /* 두 희소 행렬을 곱한다 */
{
    int i, j, column;
    int rows_a = a[0].row, cols_a = a[0].col, cols_b = b[0].col;
    int totala = a[0].value, totalb = b[0].value, totald = 0;
    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);
    /* 경계 조건 설정 */
    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; ) {
            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 :   i++; break;
                case  0 :   sum += (a[i++].value * new_b[j++].value); break;
                case  1 :   j++;
            }
        }
        for (; a[i].row == row; i++) ; row_begin = i; row = a[i].row;
    }
    d[0].row = rows_a; d[0].col = cols_b; d[0].value = totald;
}
```

2.5 다차원 배열의 표현

- 메모리를 1부터 m까지 번호가 붙은 워드들로 구성된 1차원으로 가정하고 n차원 배열을 표현

- 임의의 배열원소의 메모리위치가 효율적으로 결정될 수 있는 표현 방법을 택해야 한다
  - 배열을 사용하는 프로그램은 특정 순서없이 배열 원소들을 사용하므로 이런 표현법이 중요

- 배열 원소들을 쉽게 검색할 수 있을 뿐 아니라 특정한 배열에 할당될 기억장소의 양도 정할 수 있는 방법이 필요
  - 각 배열의 원소가 메모리의 한 워드를 필요로 한다고 가정하면 필요한 워드수는 배열의 원소수와 같다
  - $A(l_1 : u_1, l_2 : u_2, ..., l_n : u_n) \rightarrow \prod_{i=1}^{n} (u_i - l_i + 1)$

- 행우선 순위
  - A(4:5, 2:4, 1:2, 3:4) → 2 × 2 × 2 × 2 = 24개의 원소
  - 저장되는 순서
    - A(4,2,1,3), A(4,2,1,4), A(4,2,2,3), A(4,2,2,4), ...,
      A(4,3,1,3), A(4,3,1,4), A(4,3,2,3), A(4,3,2,3), ...,
      A(5,4,1,3), A(5,4,1,4), A(5,4,2,3), A(5,4,2,4)
  - → 열우선 순위

□ 변수의 이름을 메모리의 올바른 위치로 변환시킨다

- 1차원 배열 $A(1 : u_1)$
  - $A(1)$의 주소가 α
  - $A(i)$의 주소는 $\alpha + (i - 1)$
- 3차원 배열 $A(1 : u_1, 1 : u_2, 1 : u_3)$
  - $A(1, 1, 1)$의 주소가 α
  - $A(i, 1, 1)$의 주소는 $\alpha + (i - 1)u_2u_3$
  - $A(i, j, k)$의 주소는 $\alpha + (i - 1)u_2u_3 + (j - 1)u_3 + (k - 1)$
- n차원 배열 $A(1 : u_1, 1 : u_2, ..., 1 : u_n)$
  - $A(1, 1, ..., 1)$의 주소가 α
  - $A(i_1, 1, ..., 1)$의 주소는 $\alpha + (i_1 - 1)u_2u_3 ... u_n$
  - $A(i_1, i_2, ..., i_n)$의 주소는

$$
\begin{aligned}
&\alpha \quad + (i_1 - 1)u_2u_3 ... u_n \\
&\quad\quad + (i_2 - 1)u_3u_4 ... u_n \\
&\quad\quad + (i_3 - 1)u_4u_5 ... u_n \\
&\quad\quad + \ldots + (i_{n-1} - 1)u_n + (i_n - 1) \\
&= \alpha \quad + \Sigma\,(i_j - 1)a_j,\ a_j = \Pi u_k,\ 1 \le j < n,\ a_n = 1
\end{aligned}
$$

2.6 문자열

2.6.1 개 요

▶ **페이지 83, 구조 2.4 : *String* 추상 데이터 타입**

struct *String*
  objects : 0개 이상의 문자들의 유한 집합
  functions : 모든 $s, t \in$ *String*, $i, j, m \in$ 음이 아닌 정수
    String Null(m) ::= return 최대 길이가 m인 문자열
      초기의 *NULL*로 설정되며, *NULL*은 “”로 표현된다
    Integer Compare(s, t) ::= if (s와 t가 같으면) return 0
      else if (s가 t에 선행하면) return –1
      else return + 1
    Boolean IsNull(s) ::= if (Compare(s, *NULL*)) return *FALSE*
      else return *TRUE*
    Integer Length(s) ::= if (Compare(s, *NULL*)) return s의 문자수
      else return *TRUE*
    String Concat(s, t) ::= if (Compare(s, *NULL*))
      return s 뒤에 t를 붙인 문자열
      else return s
    String Substr(s, i, j) ::= if (($j > 0$) && ($i + j - 1$) < Length(s))
      return s에서 $i, i + 1, \ldots, i + j - 1$의 위치에 있는 문자열
      else return *NULL*

□ 문자열이란?

- 여러 개의 문자들로 구성된 일종의 배열 ??

□ C 언어에서의 문자열

- \0으로 끝나는 문자 배열

✧ **C 언어에서의 문자열, 페이지 85**

```
#define MAX_SIZE 100
char s[MAX_SIZE] = {"dog"};
char t[MAX_SIZE] = {"house"};
```

⟺

```
char s[] = {"dog"};
char t[] = {"house"};
```

s[0]	*s*[1]	*s*[2]	*s*[3]
d	o	g	\0

t[0]	*t*[1]	*t*[2]	*t*[3]	*t*[4]	*t*[5]
h	o	u	s	e	\0

✓ 페이지 84, 그림 2.7 : C 언어의 문자열 함수 참고

▶ **페이지 87, 프로그램 2.11 : 문자열 삽입 함수**

```
void strnins ( char *s, char *t, int i )
{
    /* 문자열 s의 i번째 위치에 문자열 t를 삽입 */
    char string[MAX_SIZE], *temp = string;
    if (i < 0 && i > strlen(s)) {
        fprintf(stderr, "Position is out of bounds\n"); exit(1);
    }
    if (!strlen(s)) strcpy(s, t);
    else if (strlen(t)) {
        strncpy(temp, s, i); strcat(temp, t); strcat(temp, (s + i));
        strcpy(s, temp);
    }
}
```

- 페이지 86, 그림 2.9 : 문자 삽입의 예

□ 문자열은 자료 구조의 가장 기본 !!

2.6.2 패턴 매칭

☐ 문자열 패턴 매칭(pattern matching)

- 문자열에서 주어진 패턴을 탐색

☐ 가장 간단한 패턴 매칭 알고리즘은?

- 문자열의 첫번째 문자와 패턴의 첫번째 문자부터 차례로 비교
- 다른 경우에는 문자열의 두번째 문자와 패턴의 첫번째 문자부터 …

▶ **페이지 87, C 언어에서의 *strstr* 사용 예**

```
if (t = strstr(string, pat))
    printf("The string from strstr is : %s\n", t);
else
    printf("The pattern was not found with strstr\n");
```

☐ strstr 대신 별도의 패턴 매칭의 구현이 필요한 이유

- 보다 좋은 알고리즘을 제시하기 위해 !!

□ 패턴의 마지막 문자를 먼저 검사하는 패턴 매칭

✧ **페이지 88, 프로그램 2.12 : 패턴의 마지막 문자를 먼저 검사하는 패턴 매칭**

```
int nfind(char *string, char *pat)
{
    /* 먼저 패턴의 마지막 문자를 매치시켜 본 뒤에, 처음부터 매치시킨다 */
    int i, j, start = 0;
    int lasts = strlen(string) – 1;
    int lastp = strlen(pat) – 1;
    int endmatch = lastp;
    for (i = 0; endmatch <= lasts; endmatch++, start++) {
        if (string[endmatch] == pat[lastp])
            for (j = 0, i = start; j < lastp && start[i] == pat[j]; i++, j++) ;
        if (j == lastp)
            return start;
    }
    return –1;
}
```

◆ 페이지 89, 그림 2.10 : *nfind*의 시뮬레이션

– *pat* = “aab”
string = “ababbaabaa”

- nfind의 분석
  - O($n \cdot m$)
  - → 이상적인 알고리즘은
    - : O(strlen(*string*) + strlen(*pat*))
    - : 최악의 경우, 패턴과 문자열의 모든 문자들을 적어도 한번씩 검색해야 하므로
- → 새로운 알고리즘
  - Knuth-Morris-Pratt algorithm

□ Knuth-Morris-Pratt 패턴 매칭 알고리즘

- idea
  - 패턴에 대한 문자열의 탐색이 뒤로 진행하는 일이 없도록 하자
  - 매치되지 않는 경우, 패턴 내의 문자와 매치되지 않은 패턴 내의 위치 정보를 이용하여 어디에서 탐색을 계속할 것인지 결정

✧ 페이지 90, Knuth-Morris-Pratt의 설계

- $S_i \neq$ a이면 S_{i+1}을 비교

s =	'-	a	b	?	?	?	.	.	.	. ?'
pat =		a	b	c	a	b	c	a	c	a b

- $S_i\ S_{i+1}$ = ab이고 $S_{i+2} \neq$ c이면 S_{i+2}를 *pat*의 첫 문자와 비교

  : S_{i+1} = b이면 $S_{i+1} \neq$ a임을 알 수 있으므로

s =	'-	a	b	c	a	?	.	.	.	. ?'
pat =		a	b	c	a	b	c	a	c	a b

- $S_i\ S_{i+1}\ S_{i+2}\ S_{i+3}$ = abca이고 $S_{i+4} \neq$ b이면 S_{i+4}를 *pat*의 두번째 문자와 비교

- 패턴 내의 문자들을 알고 *s*내의 문자와 매치되지 않는 패턴 내의 위치를 알아냄으로써 *s*안에서 후진하지 않고 *pat* 내의 어느 위치로부터 탐색을 계속할 것인지를 결정

  → 실패 함수의 사용

✧ **페이지 91, 실패 함수와 패턴**

$f(j)$ = 제일 큰 $i < j$ such that $p_0p_1 \ldots p_i = p_{j-i}p_{j-i+1} \ldots p_j, i \geq 0$인 i가 존재하는 경우
-1, 그외의 경우

– $pat = abcabcacab$

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

– 실패 함수 값이 의미는?

: 해당 문자까지 매치가 성공했으나 바로 다음 문자에서 실패한 경우, 문자열 s의 매치되지 않은 문자와 매치를 시도할 패턴 내의 문자의 위치에 대한 정보 !!

: 해당 문자까지는 매치된 것으로 생각할 수 있음의 의미

– 참고로, 모든 문자가 다른 패턴의 실패함수는?

: 모두 –1

✧ **페이지 91 ~ 92, 프로그램 2.13 : Knuth-Morris-Pratt의 패턴 매칭 알고리즘**

```
int pmatch(char *string, char *pat)
{
    int i = 0, j = 0;
    int lens = strlen(string), 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 );
}
```

– 예

: *pat* = abcabcacab
string = abcabcabcabcacababcabc

✧ **페이지 92, 실패 함수의 개선**

페이지 93, 프로그램 2.14 : 실패 함수의 계산

$$f(j) = \begin{cases} -1, & j = 0 \\ f^m(j-1)+1, & m\ P_{f^k(j-1)+1} = P_j \quad k \\ -1, & \text{윗식을 만족하는 k가 없을 경우} \end{cases}$$

```
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 = failue[i];
        if (pat[j] = pat[i + 1])
            failure[j] = i + 1;
        else failure[j] = -1;
    }
}
```

✍ 분석

- pmatch : O(*m*) = O(strlen(*string*))
- fail : O(*n*) = O(strlen(*pat*))

→ O(*m* + *n*) = O(strlen(*string*) + strlen(*pat*))