

# <자료구조 및 실습>

## 3. 리스트

한국외국어대학교  
컴퓨터.전자시스템공학전공  
2016년 1학기  
고 석 훈

# 학습 목표

- 순차 자료구조의 의미와 특징을 알아본다.
- 선형 리스트의 구조와 연산을 알아본다.
- 선형 리스트의 C 프로그램 구현을 알아본다.
- 선형 리스트의 응용 방법을 알아본다.

# 리스트 [1/2]

## ● 리스트(list)

- 자료를 나열한 목록

동창 이름 리스트	좋아하는 음식 리스트	오늘의 할일 리스트
홍길동	김치찌개	운동
이순신	닭볶음탕	자료구조 수업
윤봉길	된장찌개	동아리 공연 연습
안중근	잡채	과제 제출
...	...	...

## ● 선형 리스트(linear list)

- 순서 리스트(ordered list)
- 자료들 간에 순서를 갖는 리스트

동창 이름 선형 리스트		좋아하는 음식 선형 리스트		오늘의 할일 선형 리스트	
1	홍길동	1	김치찌개	1	운동
2	이순신	2	닭볶음탕	2	자료구조 수업
3	윤봉길	3	된장찌개	3	동아리 공연 연습
4	안중근	4	잡채	4	과제 제출
...	...	...	...	...	...

# 리스트 [2/2]

- 리스트의 표현 형식

- 선형 리스트에서 원소를 나열한 순서는 원소들의 순서가 된다.

리스트 이름 = (원소1, 원소2, ..., 원소n)

- 예제) 동창이름 선형 리스트의 표현

동창 = (홍길동, 이순신, 윤봉길, 안중근)

- 공백 리스트

- 원소가 하나도 없는 리스트
- 빈 괄호를 사용하여 표현

리스트 이름 = ( )

# 순차 자료구조 [1/2]

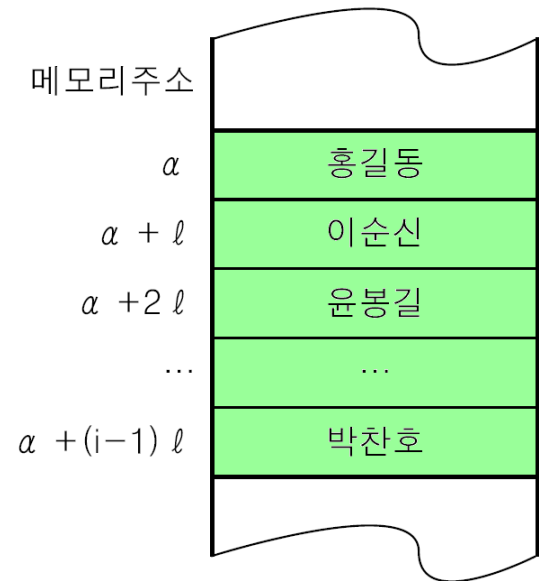
- 순차 자료구조(sequential data structure)
  - 원소들의 논리적 순서와 원소들이 저장된 물리적 순서가 동일
  - 예제) 동창이름 선형 리스트가 메모리에 저장된 물리적 구조



# 순차 자료구조 [2/2]

## ● 순차 자료구조의 원소 위치 계산

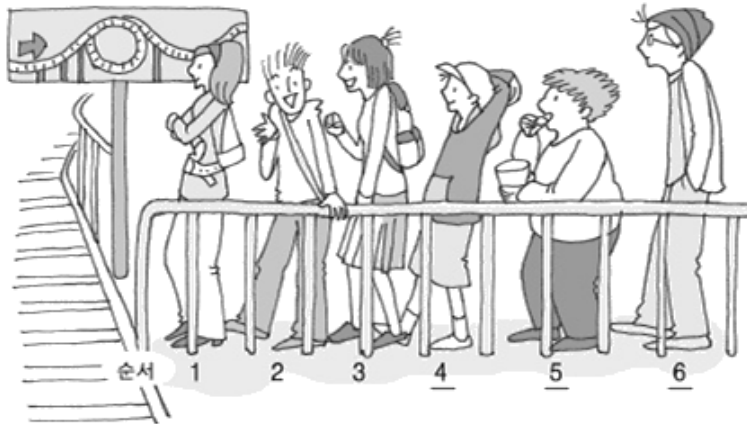
- 선형 리스트가 저장된 시작 위치:  $\alpha$ , 원소의 길이:  $\ell$
- $i$ 번째 원소의 위치 =



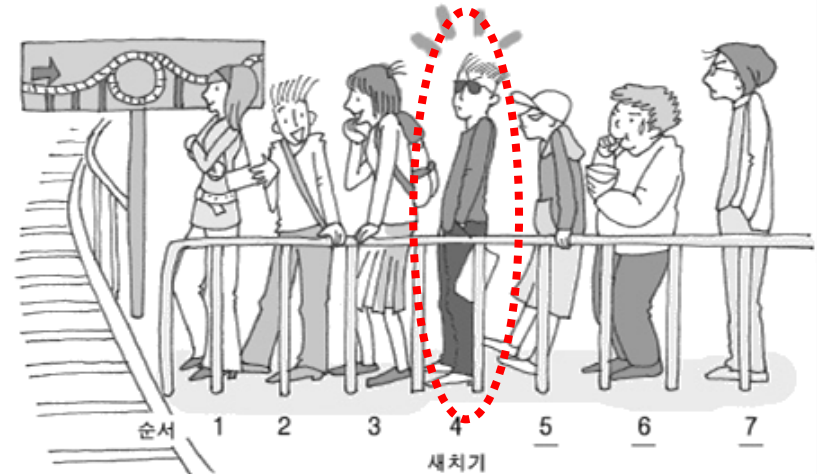
# 선형 리스트의 원소 삽입 [1/3]

## ● 선형 리스트의 원소 삽입

- 선형리스트 중간에 원소가 삽입되면,  
그 이후의 원소들은 한자리씩 자리를 뒤로 이동하여  
물리적 순서를 논리적 순서와 일치시킨다.



(a) 새치기 전

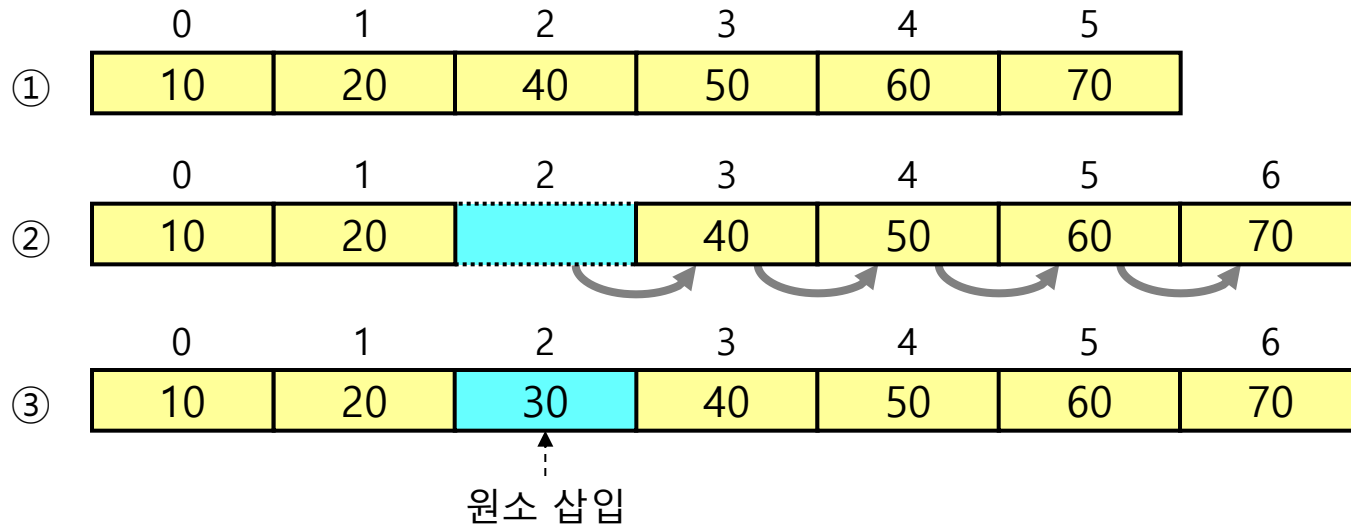


(b) 새치기 후

# 선형 리스트의 원소 삽입 [2/3]

## ● 원소 삽입 방법

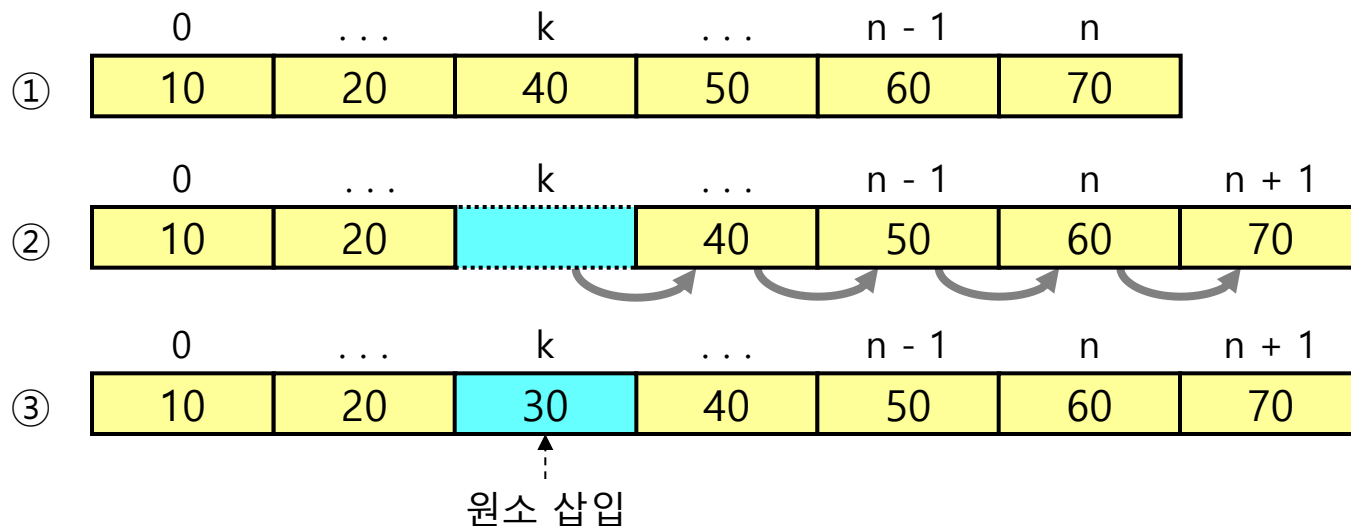
- 원소를 삽입할 빈 자리 만들기  
(삽입할 자리 이후의 원소들을 한자리씩 뒤로 이동 시키기)
- 준비한 빈 자리에 원소 삽입하기





# 선형 리스트의 원소 삽입 [3/3]

- $(n + 1)$ 개의 원소로 이루어진 선형 리스트에서  $k$ 번 자리에 원소를 삽입하는 경우의 이동회수?

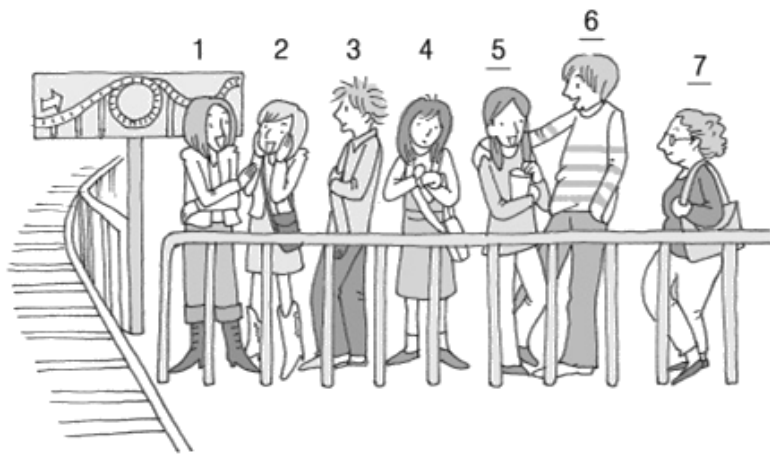


# 선형 리스트의 원소 삭제 [1/3]

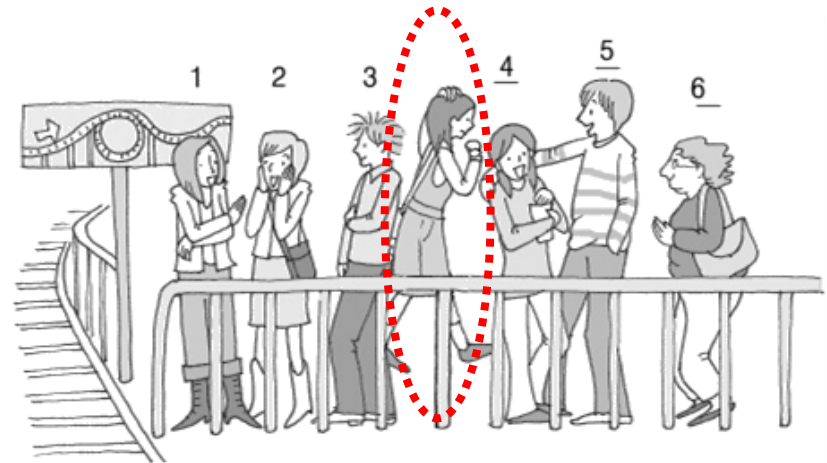
## ● 선형 리스트에서의 원소 삭제

### ■ 선형리스트 중간에서 원소가 삭제되면,

그 이후의 원소들은 한자리씩 자리를 앞으로 이동하여  
물리적 순서를 논리적 순서와 일치시킨다.



(a) 나가기 전



(b) 나간 후

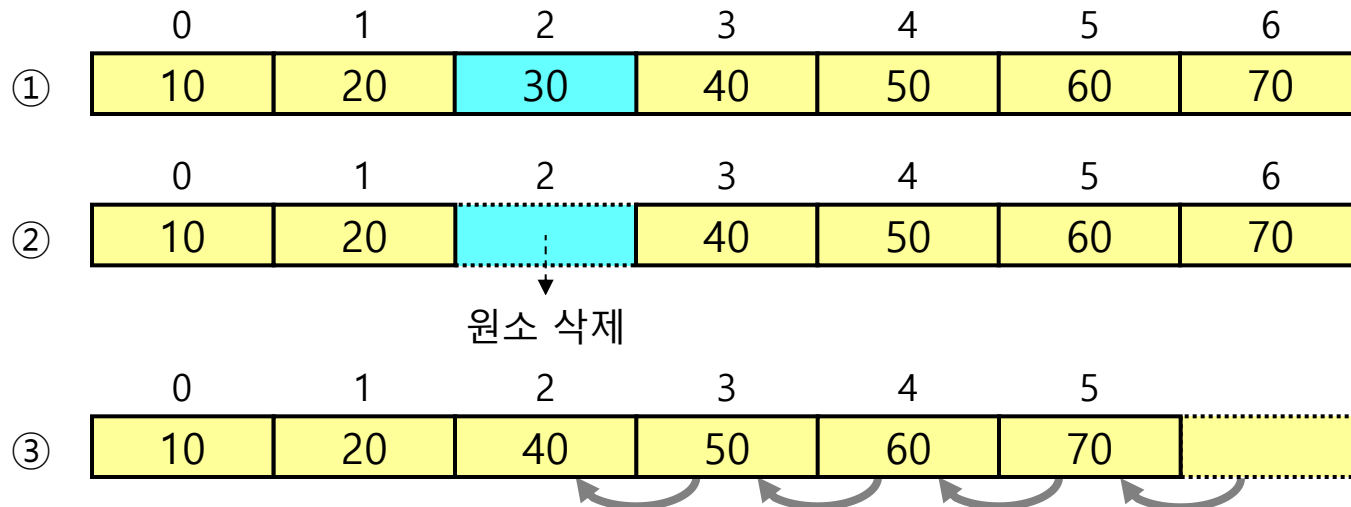
# 선형 리스트의 원소 삭제 [2/3]

## ● 원소 삭제 방법

### ■ 원소 삭제하기

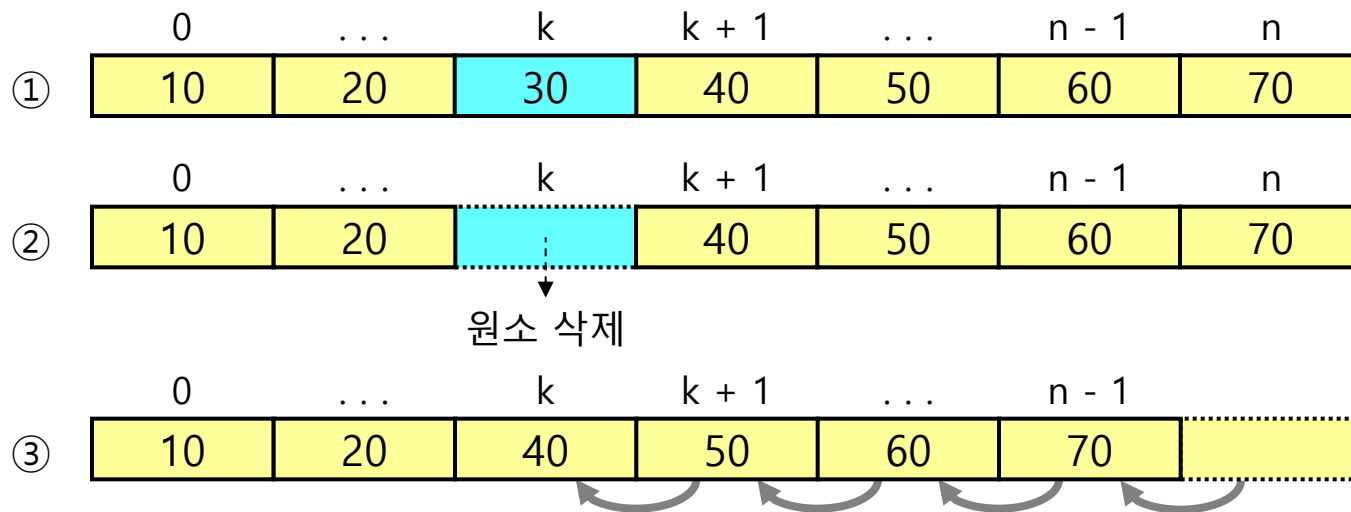
### ■ 삭제한 빈 자리 채우기

(삭제한 자리 이후의 원소들을 한자리씩 앞으로 이동 시키기)



# 선형 리스트의 원소 삭제 [3/3]

- $(n + 1)$ 개의 원소로 이루어진 선형 리스트에서  $k$ 번 자리의 원소를 삭제한 경우의 이동회수?



# 선형 리스트의 구현: 1차원 배열 [1/3]

- 분기별 판매량을 1차원 배열을 이용한 선형 리스트로 구현

분기	1/4분기	2/4분기	3/4분기	4/4분기
판매량	157	209	251	312

## ■ 1차원 배열을 이용한 구현

```
int sale[4] = { 157, 209, 251, 312 };
```

## ■ 논리적 구조

	[0]	[1]	[2]	[3]
sale	157	209	251	312

## ■ 물리적 구조



# 선형 리스트의 구현: 1차원 배열 [2/3]

## ● C 프로그램 구현

```
#include <stdio.h>

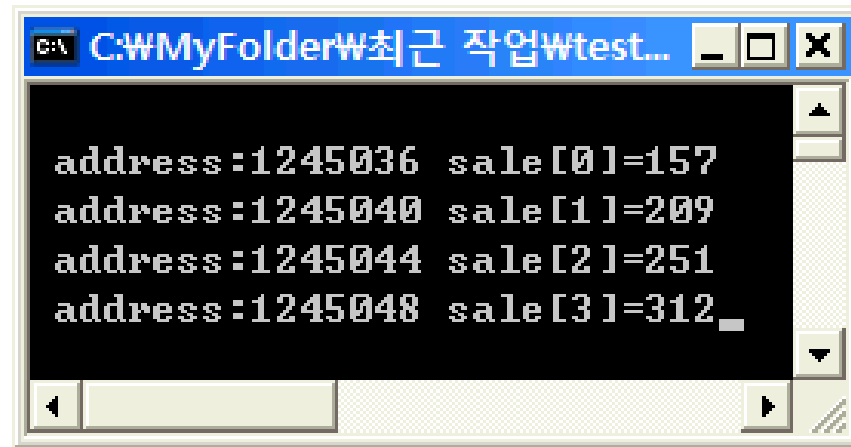
void main( )
{
    int  i, sale[4] = { 157, 209, 251, 312 };

    for (i = 0; i < 4; i++) {
        printf("\n address:%u sale[%d]=%d",
               &sale[i], i, sale[i]);
    }

    getchar( );
}
```

# 선형 리스트의 구현: 1차원 배열 [3/3]

## ● 실행 결과

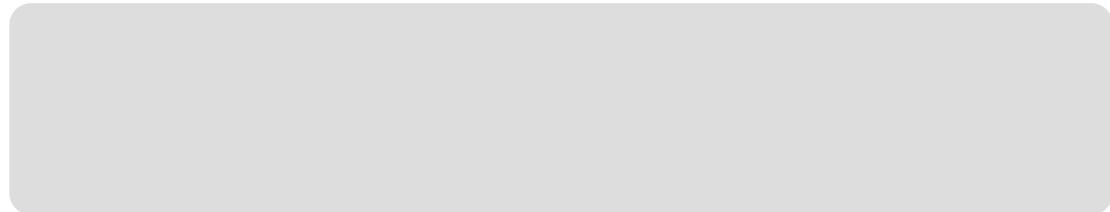


```
C:\WMyFolderW최근 작업wtest...  
address:1245036 sale[0]=157  
address:1245040 sale[1]=209  
address:1245044 sale[2]=251  
address:1245048 sale[3]=312
```

## ● sale[2]의 주소?

- 배열 sale의 시작주소 = 1245036, 인덱스 = 2

- sale[2]의 주소



- 논리 순서대로 메모리에 연속하여 저장된 순차구조임을 확인!

# 선형 리스트의 구현: 2차원 배열 [1/4]

- 연/분기별 판매량을 2차원 배열을 이용한 선형 리스트로 구현

년	분기	1/4분기	2/4분기	3/4분기	4/4분기
2013년		63	84	140	130
2014년		157	209	251	312

- 2차원 배열을 이용한 구현

```
int sale[2][4] = { { 63, 84, 140, 130 },  
                  { 157, 209, 251, 312 } };
```

- 논리적 구조

	[0]	[1]	[2]	[3]
sale [0]	63	84	140	130
[1]	157	209	251	312



# 선형 리스트의 구현: 2차원 배열 [2/4]

## ● 2차원 배열의 물리적 저장 방법

- 2차원의 논리적 순서를  
1차원의 물리적 순서로 변환
- 행 우선 순서 방법(row major order)  
2차원 배열의 첫 번째 인덱스인  
행(row) 번호를 기준으로 사용
- 원소의 위치 계산 방법  
행의 개수가  $n_i$ , 열의 개수가  $n_j$ 인  
2차원 배열  $A[n_i][n_j]$ 의 시작주소가  $\alpha$ ,  
원소의 크기가  $\ell$  일 때,  $i$ 행  $j$ 열 원소  
즉,  $A[i][j]$ 의 위치는?

	[0]	[1]	[2]	[3]
sale [0]	63	84	140	130
[1]	157	209	251	312



행 우선 순서 방법

# 선형 리스트의 구현: 2차원 배열 [3/4]

## ● C 프로그램 구현

```
#include <stdio.h>

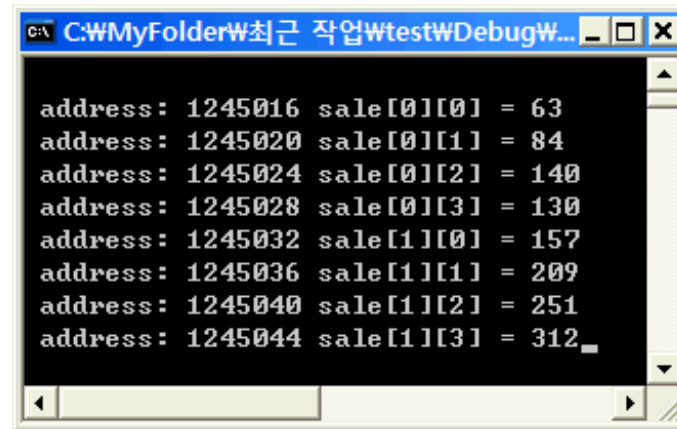
void main( )
{
    int  i, j;
    int  sale[2][4] = { { 63, 84, 140, 130 },
                        { 157, 209, 251, 312 } };

    for (i = 0; i < 2; i++)
        for (j = 0; j < 4; j++)
            printf("\n address: %u sale[%d][%d] = %d",
                    &sale[i][j], i, j, sale[i][j]);

    getchar( );
}
```

# 선형 리스트의 구현: 2차원 배열 [4/4]

## ● 실행 결과



```
C:\WMyFolder\최근 작업\wtest\WDebugW...  
address: 1245016 sale[0][0] = 63  
address: 1245020 sale[0][1] = 84  
address: 1245024 sale[0][2] = 140  
address: 1245028 sale[0][3] = 130  
address: 1245032 sale[1][0] = 157  
address: 1245036 sale[1][1] = 209  
address: 1245040 sale[1][2] = 251  
address: 1245044 sale[1][3] = 312
```

## ● sale[1][2]의 주소?

■ 시작주소  $\alpha = 1245016$ ,  $n_i = 2$ ,  $n_j = 4$ ,  $i = 1$ ,  $j = 2$ ,  $\ell = 4$

■ sale[1][2]의 주소 =

=  
=  
=  
=

■ 행 우선 순서 방법으로 2차원 배열이 저장됨을 확인

# 다항식 추상 자료형(ADT) [1/2]

## ● 다항식

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0 x^0$$

- $ax^e$  형식의 항들의 합으로 구성된 식

$a$  : 계수(coefficient),  $x$  : 변수(variable),  $e$  : 지수(exponent)

- ◆ 지수에 따라 내림차순으로 항을 나열
- ◆ 다항식의 차수 : 가장 큰 지수
- ◆ 다항식 항의 최대 개수 = (차수 + 1) 개

- 예)  $A(x) = 2x^4 - 9x^3 + 5x^2 + 1$

$$B(x) = 3x^3 + 5x + 7$$

# 다항식 추상 자료형(ADT) [2/2]

이름 : 다항식 (Polynomial)

데이터 : 지수  $e_i$  와 계수  $a_i$  의 순서쌍  $\langle e_i, a_i \rangle$ 의 집합으로 표현된

다항식  $P(x) = a_n x^{e_n} + a_{n-1} x^{e_{n-1}} + \dots + a_i x^{e_i} + \dots + a_1 x^1 + a_0 x^0$  ( $e_i$ 는 음이 아닌 정수)

연산 :

zeroPoly(p)	// 다항식 p를 0으로 초기화
isZeroPoly(p)	// 다항식 p가 0이면 true, 0이 아니면 false
getCoef(p, e)	// 지수 e인 계수 a를 반환, 지수 e가 없는 경우 0을 반환
getMaxExp(p)	// 다항식 p에서 가장 큰 지수를 반환, p가 0인 경우 -1 반환
addTerm(p, a, e)	// 지수 e항이 있는 경우 계수에 a를 더하고, 없는 경우 새로운 항 $\langle e, a \rangle$ 추가
delTerm(p, e)	// 지수가 e항을 삭제
addPoly(p1, p2)	// 두 다항식 p1과 p2의 합을 반환
multPoly(p1, p2)	// 두 다항식 p1과 p2의 곱을 반환

# 다항식의 덧셈 알고리즘

// 주어진 두 다항식 A와 B를 더하여 결과 다항식 C를 반환하는 알고리즘

**AddPoly**(A, B)

C  $\leftarrow$  A;

**while** (**not** isZeroPoly(B)) **do** {

$e_{\max} \leftarrow \text{getMaxExp}(B);$

    C  $\leftarrow \text{addTerm}(C, \text{coef}(B, e_{\max}), e_{\max});$

    B  $\leftarrow \text{delTerm}(B, e_{\max});$

}

**return** C;

**End** AddPoly( )

// B의 최대 지수항을 C에 추가

// B의 최대 지수항을 B에서 제거

# 다항식의 표현

## ● 다항식의 논리적 표현

- 각 항의 지수와 계수의 쌍에 대한 선형 리스트

$$A(x) = 2x^4 - 9x^3 + 5x^2 + 1$$

$$\rightarrow A = (<4, 2>, <3, -9>, <2, 5>, <0, 2>)$$

$$B(x) = 3x^3 + 5x + 7$$

$$\rightarrow B = (<3, 3>, <1, 5>, <0, 7>)$$

# 다항식의 표현: 1차원 배열 [1/2]

- 1차원 배열에서 하나의 원소로 하나의 항 표현
  - 배열의 인덱스 → 각 항의 지수(exponent)를 의미
  - 배열의 원소 → 각 항의 계수(coefficient)를 의미
  - 크기 100개의 배열 선언시 0차~99차 항까지 허용

$$A(x) = 3x^{15} - 2x^3 + 5x^2 - 1$$



A[ ]	0	0	0	0	0	...	0	...	0
	[0]	[1]	[2]	[3]	[4]	...	[15]	...	[99]



# 다항식의 표현: 1차원 배열 [2/2]

- 희소 다항식에 대한 1차원 배열 저장

- 예)  $C(x) = 3x^{1000} + x + 4$

	[0]	[1]	[2]	[3]	...	[997]	[998]	[999]	[1000]
C	4	1	0	0	...	0	0	0	3

998개 항이 모두 0

- 차수가 1000이므로 크기가 1001인 배열을 사용하는데, 항이 3개 뿐이므로 배열의 원소 중에서 3개만 사용  
➔ 998개의 배열 원소에 대한 메모리 공간 낭비!

# 다항식의 표현: 2차원 배열

## ● 2차원 배열을 이용한 순차 자료구조 표현

■ 다항식의 각 항에 대한 <지수, 계수>의 쌍을 2차원 배열에 저장




◆ 2차원 배열의 행의 개수 = 다항식의 항의 개수

◆ 2차원 배열의 열의 개수 = 2개 (지수, 계수)

◆ 1차원 배열을 사용하는 방법보다 메모리 사용량 감소

➔ 공간 복잡도 감소 ➔ 프로그램 성능 향상!

$$C(x) = 3x^{1000} + x + 4$$

	[0]	[1]	
[0]	1000	3	 $3x^{1000}$
[1]	1	1	 $x$
[2]	0	4	 $4$

# 행렬의 표현 [1/3]

- 행렬의 순차 자료구조 표현
  - 2차원 배열 사용
  - $m \times n$  행렬을  $m$ 행  $n$ 열의 2차원 배열로 표현

$A =$	$\begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{vmatrix}$	$\rightarrow$	$A[3][4]$				
				[0]	[1]	[2]	[3]
			[0]	1	2	3	4
			[1]	5	6	7	8
			[2]	9	10	11	12

# 행렬의 표현 [2/3]

## ● 희소 행렬의 경우

- 사용하지 않는 원소가 많아 메모리 낭비가 큼

B =	0	0	2	0	0	0	12
	0	0	0	0	7	0	0
	23	0	0	0	0	0	0
	0	0	0	31	0	0	0
	0	14	0	0	0	25	0
	0	0	0	0	0	0	6
	52	0	0	0	0	0	0
	0	0	0	0	11	0	0

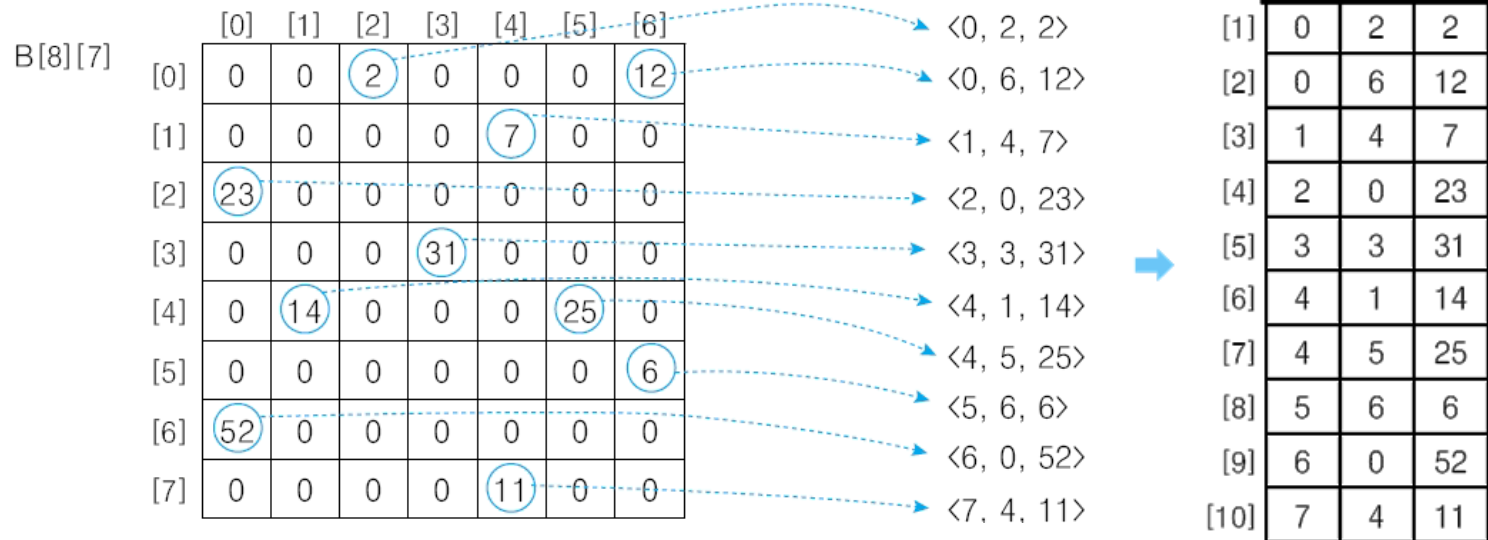
→

		[0]	[1]	[2]	[3]	[4]	[5]	[6]
B[8][7]	[0]	0	0	2	0	0	0	12
	[1]	0	0	0	0	7	0	0
	[2]	23	0	0	0	0	0	0
	[3]	0	0	0	31	0	0	0
	[4]	0	14	0	0	0	25	0
	[5]	0	0	0	0	0	0	6
	[6]	52	0	0	0	0	0	0
	[7]	0	0	0	0	11	0	0

# 행렬의 표현 [3/3]

## ● 희소 행렬에 대한 2차원 배열 표현

- 0이 아닌 원소만 추출하여 <행번호, 열번호, 원소>쌍을 저장
- 원래의 행렬에 대한 정보를 순서쌍으로 작성하여 0번 행에 저장  
<행의 개수, 열의 개수, 0이 아닌 원소의 개수>



# 희소 행렬의 추상 자료형(ADT)

이름 : Sparse\_Matrix

데이터 : 3원소쌍 <행 인덱스, 열 인덱스, 원소 값>의 집합

연산 :  $a, b, c \in \text{Sparse\_Matrix}$ ;  $v \in \text{value}$ ;  $i \in \text{Row}$ ;  $j \in \text{Column}$ ;

// a, b는 희소행렬, c는 행렬, u, v는 행렬의 원소값을 나타내며,  
// i와 j는 행 인덱스와 열 인덱스를 나타낸다.

smCreate(m, n) ::= **return** an empty sparse matrix with  $m \times n$ ;  
//  $m \times n$ 의 공백 희소행렬을 만드는 연산

smTranspose(a) ::= **return** b where  $b[j, i] \leftarrow v$  when  $a[i, j] = v$ ;  
// 희소행렬  $a[i, j]=v$ 를  $b[j, i]=v$ 로 전치시킨 전치행렬 b를 구하는 연산

smAdd(a, b) ::= **if** (a.dimension = b.dimension)  
**then return** c where  $c[i, j] \leftarrow a[i, j] + b[i, j]$ ;  
**else return** error;  
// 차수가 같은 희소행렬 a와 b를 합한 행렬 c를 구하는 연산

smMulti(a, b) ::= **if** (a.n = b.m) **then return** c where  $c[i, j] \leftarrow a[i, k] \times b[k, j]$ ;  
**else return** error;  
// 희소행렬 a의 열의 개수(n)와 희소행렬 b의 행의 개수(m)가  
// 같은 경우에 두 행렬의 곱을 구하는 연산

# 전치 연산 알고리즘

**smTranspose(a[])**

```
m ← b[0, 1] ← a[0, 0]; // 희소 행렬 a의 행 수, 전치 행렬 b의 열 수 지정
n ← b[0, 0] ← a[0, 1]; // 희소 행렬 a의 열 수, 전치 행렬 b의 행 수 지정
v ← b[0, 2] ← a[0, 2]; // 희소 행렬 a에서 0이 아닌 원소 수, 전치 행렬 b의 원소 수 지정
if (v > 0) then { // 0이 아닌 원소가 있는 경우에만 전치 연산 수행
    p ← 1;
    for (i ← 0; i < n; i ← i + 1) do { // 희소 행렬 a의 열 순서대로 (b의 행 순서대로)
        for (j ← 1; j ≤ v; j ← j + 1) do { // 희소 행렬 a의 전체 원소에서
            if (a[j, 1] = i) then { // 해당 열(i)에 속하는 원소가 있으면 b에 삽입
                b[p, 0] ← a[j, 1];
                b[p, 1] ← a[j, 0];
                b[p, 2] ← a[j, 2];
                p ← p + 1;
            }
        }
    }
}
return b[];
end smTranspose( )
```

# 전치 연산 프로그램

```
typedef struct {
    int row;
    int col;
    int value;
} term;

void smTranspose(term a[], term b[]) {
    int m, n, v, i, j, p;
    m = b[0].col = a[0].row;           // 전치 행렬 b의 열 수 <- 희소 행렬 a의 행 수
    n = b[0].row = a[0].col;           // 전치 행렬 b의 행 수 <- 희소 행렬 a의 열 수
    v = b[0].value = a[0].value;       // 전치 행렬 b의 원소 수 <- 희소 행렬 a의 원소 수
    if (v > 0) {                       // 0이 아닌 원소가 있는 경우에만 전치 연산 수행
        p = 1;
        for (i = 0; i < n; i++)        // 희소 행렬 a의 열 순서대로 (b의 행 순서대로)
            for (j = 1; j <= v; j++)    // 희소 행렬 a의 전체 원소에서
                if (a[j].col == i) {    // 현재의 열에 속하는 원소가 있으면 b[]에 삽입
                    b[p].row = a[j].col;
                    b[p].col = a[j].row;
                    b[p].value = a[j].value;
                    p++;
                }
    }
}
```



# 요약

## ● 리스트

- 리스트(list) : 자료를 나열한 목록
- 선형 리스트(linear list) : 자료들 간에 순서를 갖는 리스트

## ● 순차 자료구조(sequential data structure)

- 원소들의 논리적 순서와 원소들이 저장된 물리적 순서가 동일
- 삽입, 삭제 연산 후에 연속적인 물리 주소를 유지하기 위해 원소들을 이동시키는 오버헤드 발생
- 순차 자료구조가 사용하는 배열의 메모리 비효율성 문제 발생