

## 제2장 리스트

배열, 연결리스트, 이중연결리스트, 원형리스트

# 리스트

---

- ▶ 리스트(List)는 일련의 동일한 타입의 항목(item)들
- ▶ 실생활의 예: 학생 명단, 시험 성적, 서점의 신간 서적, 상점의 판매 품목, 실시간 급상승 검색어, 버킷 리스트 등
- ▶ 리스트의 구현:
  - ▶ 1차원 배열
  - ▶ 단순연결리스트
  - ▶ 이중연결리스트
  - ▶ 환형연결리스트

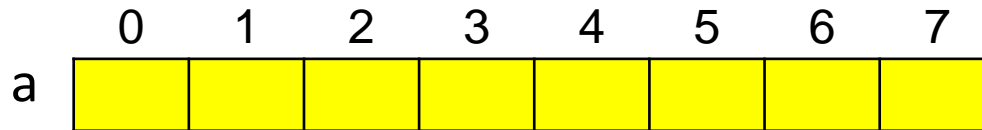
## 2.1 배열

---

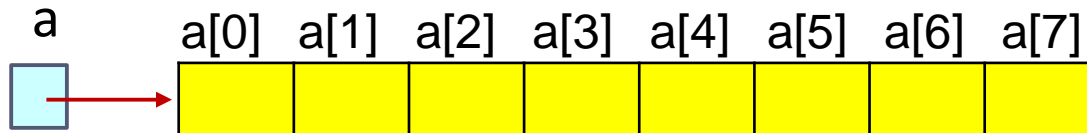
- ▶ 동일한 타입의 원소들이 연속적인 메모리 공간에 할당되어 각 항목이 하나의 원소에 저장되는 기본적인 자료구조
  - ▶ 특정 위치의 원소에 접근할 때에는 배열 인덱스를 이용하여  $O(1)$ 시간에 접근
  - ▶ 새 항목이 배열 중간에 삽입되거나 중간에 있는 항목을 삭제하면, 뒤 따르는 항목들을 한 칸씩 뒤로 또는 앞으로 이동시켜야 하므로 삽입이나 삭제 연산은 항상  $O(1)$  시간에 수행할 수 없는 단점
  - ▶ 원하는 원소의 위치를 찾는 것도  $O(1)$ 의 시간에 수행할 수 없음

# 배열의 메모리 구조

1차원 배열의 일반적인 표현



자바 언어의 특성을 반영한 표현



- ▶ a가 배열 이름인 동시에 배열의 첫번째 원소의 레퍼런스를 저장
- ▶ a[i]는 인덱스 i를 가지는 원소를 가리키는 레퍼런스
- ▶ 동적배열(Dynamic Array): 프로그램이 실행되는 동안에 할당된 배열

# 배열의 인덱스 주소

---

- ▶ 각 원소  $a[i]$ 는  $a$ 가 가지고 있는 레퍼런스에  
원소의 크기(바이트)  $\times i$ 를 더하여  $a[i]$ 의 레퍼런스를 계산
  - ▶  $a[i] = a + (\text{원소의 크기} \times i)$
  - ▶ char 배열 원소 크기 = 2바이트, int 배열 원소 크기 = 4바이트

# Overflow

---

- ▶ 배열은 미리 정해진 크기의 메모리 공간을 할당 받아 사용해야 하므로, 빈자리가 없어 새 항목을 삽입할 수 없는 상황(Overflow) 발생
- ▶ Overflow 가 발생하면 에러 처리를 하여 프로그램을 정지시키는 방법이 주로 사용. 하지만 프로그램의 안정성을 향상시키기 위해 다음과 같은 방법을 사용

## [핵심 아이디어]

배열에 overflow가 발생하면 배열 크기를 2배로 확장한다.  
또한 배열의 3/4이 비어 있다면 배열 크기를 1/2로 축소한다.

# ArrayList 클래스(1)

*Vector나 ArrayList를 쓰면 되는데,  
왜 ArrayList를 직접 구현하는 걸  
배우나요???*

## ▶ 리스트를 배열로 구현: ArrayList 클래스

```
1 import java.util.NoSuchElementException;
2 public class ArrayList <E> {
3     private E a[];    // 리스트의 항목들을 저장할 배열
4     private int size; // 리스트의 항목 수
5
6     public ArrayList() { // 생성자
7         a = (E[]) new Object[1]; // 최초로 1개의 원소를 가진 배열 생성
8         size = 0;                // 항목 수를 0으로 초기화
9     }
10    public boolean isEmpty() {return size == 0;} // 리스트가 empty이면 true 리턴
11 }
```

- ▶ Line 01: java.util 라이브러리에 선언되어 있는 NoSuchElementException 클래스를 이용하여 리스트가 empty인 상황에서 항목을 읽으려고 하면(즉, underflow가 발생하면) 프로그램을 정지시키는 예외처리
- ▶ Line 06-09: ArrayList 클래스의 생성자는 크기가 1인 generic 타입의 배열과 배열에 저장된 항목 수를 저장하는 size를 0으로 초기화
- ▶ Line 02: <E>로 generic type으로 어떤 타입도 저장할 수 있게
  - C++에서 template class와 같은 역할

## ArrayList 클래스(2)

```
public E peek(int k) { // k번째 항목을 리턴, 단순히 읽기만 한다.
    if (isEmpty() || k < 0 || k >= size )
        throw new NoSuchElementException(); // underflow 등의 예외적인 경우
    return a[k];
}
```

- ▶ peek() 메소드: k번째 저장된 항목을 탐색,  $k = 0, 1, \dots$ .
  - ▶ Line 04: a[k]를 리턴,  $k < \text{size}$ 라고 가정

```
01 public void insertLast(E newItem) { // 가장 뒤에 새 항목 삽입
02     if (size == a.length)           // 배열에 빈 공간이 없으면
03         resize(2*a.length);         // 배열 크기 2배로 확장
04     a[size++] = newItem;             // 새 항목 삽입
05 }
```

- ▶ insertLast() 메소드: 새 항목(newItem)을 가장 뒤에 삽입
  - ▶ Line 03: overflow가 발생하면 resize() 메소드를 호출하여 배열 크기를 2배로 확장



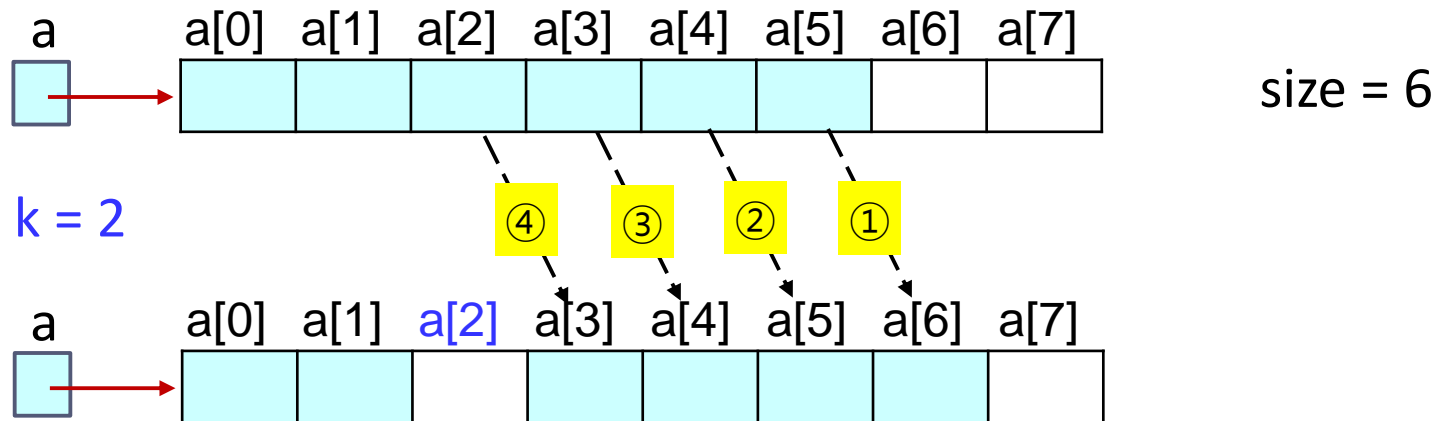
## ArrayList 클래스(3)

```
06 public void insert(E newItem, int k) { // 새 항목을 k-1번째 항목 다음에 삽입
07     if (size == a.length)             // 배열에 빈 공간이 없으면
08         resize(2*a.length);           // 배열 크기 2배로 확장
09     for (int i = size-1; i >= k; i--) a[i+1] = a[i]; // 한 칸씩 뒤로 이동
10     a[k] = newItem;
11     size++;
12 }
```

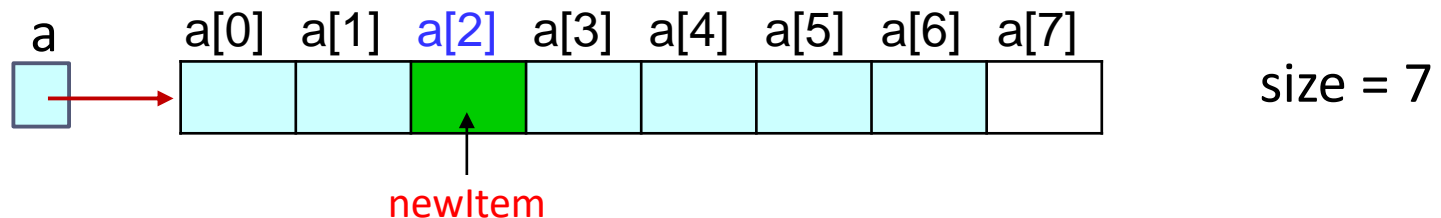
- ▶ insert() 메소드: 새 항목을 k-1 번째 항목 다음에 삽입
  - ▶ Line 08: overflow 발생시 resize() 메소드를 호출하여 배열 크기를 2배로 확장
  - ▶ Line 09: 새 항목을 위한 항목 이동

# ArrayList 클래스(4)

k번째 항목부터 마지막 항목까지 한 칸씩 이동



새 항목 `a[k]`에 삽입



# ArrayList 클래스(5)

```
01 private void resize(int newSize) {           // 배열 크기 조절
02     Object[] t = new Object[newSize];        // newSize 크기의 새로운 배열 t 생성
03     for (int i = 0; i < size; i++)
04         t[i] = a[i];                          // 배열 s를 배열 t로 복사
05     a = (E[]) t;                              // 배열 t를 배열 s로
06 }
```

- ▶ resize() 메소드: 배열의 크기를 확대 또는 축소
  - ▶ Line02: newSize 크기의 배열 t를 동적으로 생성
  - ▶ Line 03 ~ 04: 배열 a의 원소들을 배열 t로 복사
  - ▶ Line 05: a가 t를 참조
  - ▶ 기존의 배열 a는 가비지 컬렉션에 의해 처리

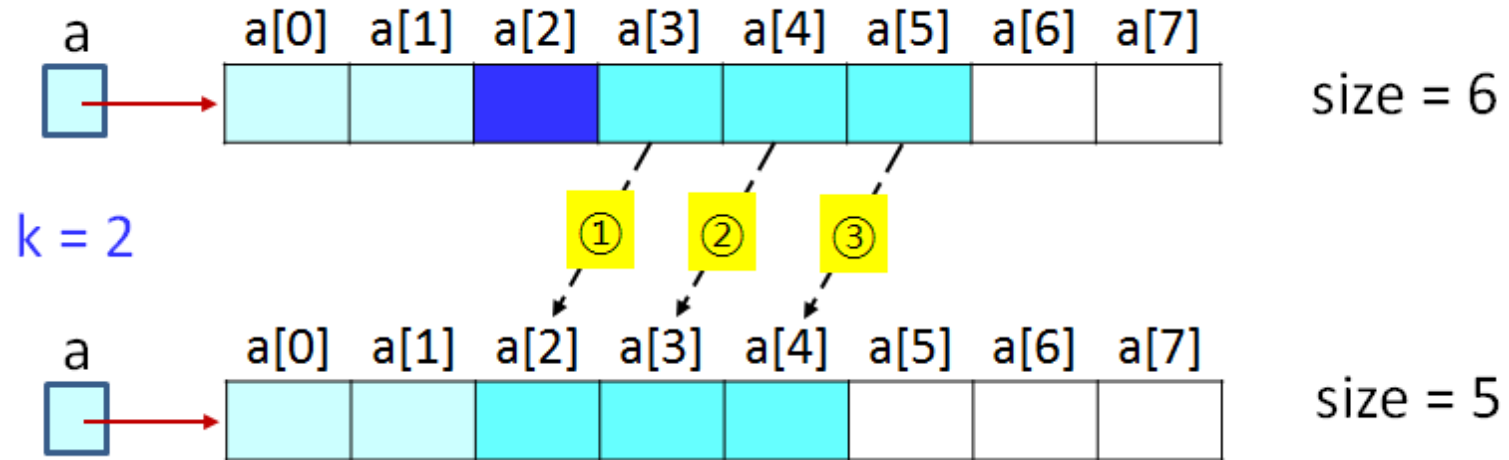
## ArrayList 클래스(6)

```
01 public E delete(int k) { // k번째 항목 삭제
02     if (isEmpty()) throw new NoSuchElementException(); // underflow 경우에 프로그램 정지
03     E item = a[k];
04     for (int i = k; i < size; i++) a[i] = a[i+1]; // 한 칸씩 앞으로 이동
05     size--;
06     if (size > 0 && size == a.length/4) // 배열에 항목들이 1/4만 차지한다면
07         resize(a.length/2);           // 배열을 1/2 크기로 축소
08     return item;
09 }
```

- ▶ delete(): k번째 항목을 삭제
  - ▶ Line 02: underflow를 검사
  - ▶ Line 03: 삭제되는 항목을 지역변수인 item에 저장
  - ▶ Line 04: a[k+1]부터 a[size-1]까지 한 칸씩 앞으로 이동하여 a[k]의 빈칸을 메움

# ArrayList 클래스(7)

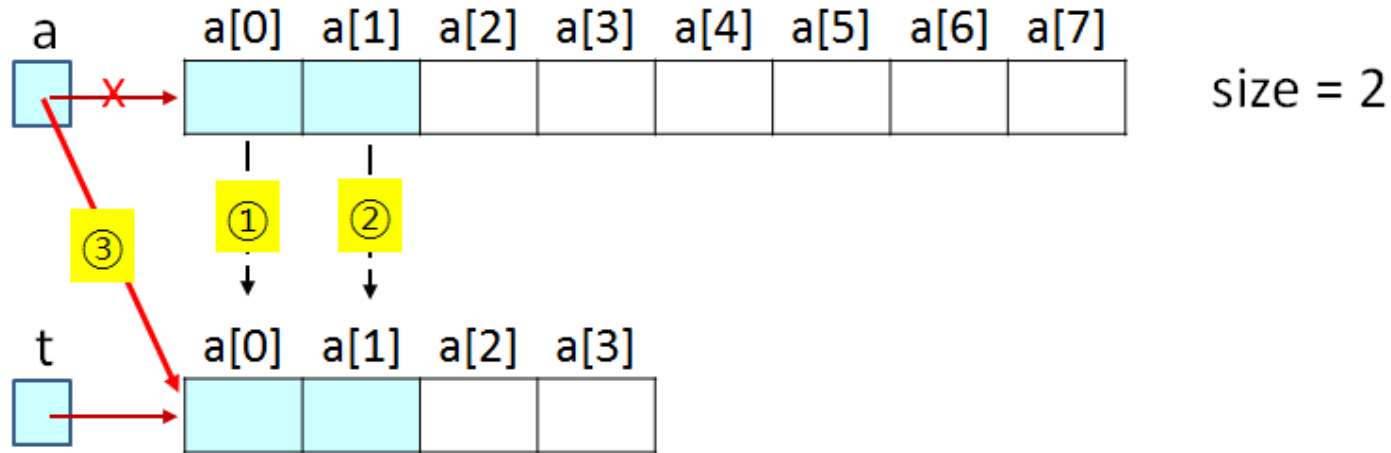
## ▶ 삭제된 원소의 공간 메우기



- Line 05: `size`를 1 감소시키고,
- Line 06~07: 항목 수가 배열 크기의 1/4이 되면 배열 크기를 1/2로 축소
- 축소된 배열에서 앞쪽의 1/2은 항목들로 차있고 뒤쪽 1/2은 비어있음

# ArrayList 클래스(8)

## ▶ 배열의 크기 축소



- ▶ 배열 크기가 8이고, 실제 두 개의 항목만 있는 경우, 배열 크기를 4로 축소
- ▶ 항목 수가 배열 크기의 1/4일 때 배열 크기를 1/2로 축소시키는 것은 축소된 배열에 1/2은 항목들로 차있고, 나머지 1/2은 비어있는 상태로 만들기 위해

# ArrayList 클래스(9)

## ▶ 실행 예제

```
1 public class main {  
2     public static void main(String[] args) {  
3         ArrayList<String> s = new ArrayList<String>();  
4         s.insert("apple");    s.print();    s.insert("orange");    s.print();  
5         s.insert("cherry");  s.print();    s.insert("pear");    s.print();  
6         s.insert("grape",1); s.print();    s.insert("lemon",4); s.print();  
7         s.insert("kiwi");    s.print();  
8         s.delete(4);         s.print();    s.delete(0);         s.print();  
9         s.delete(0);         s.print();    s.delete(3);         s.print();  
10        s.delete(0);         s.print();  
11  
12        System.out.println("1번째 항목은 "+s.peek(1)+"이다."); System.out.println();  
13    }  
14 }
```

- ▶ 7개의 항목을 insertLast()와 insert()로 항목들을 삽입하여 배열의 크기가 2배로 확장
- ▶ 항목을 연속적으로 삭제하며 배열 축소
- ▶ 마지막으로 첫번째 항목을 탐색.

# ArrayList 클래스(10)

## 프로그램 실행 결과

```
Problems @ Javadoc Console Console x
<terminated> main (49) [Java Application] C:\Program Files\Java\jdk1.8.0_40\bin\javaw.exe

apple
apple orange
apple orange cherry null
apple orange cherry pear
apple grape orange cherry pear null null null
apple grape orange cherry lemon pear null null
apple grape orange cherry lemon pear kiwi null
apple grape orange cherry pear kiwi null null null
grape orange cherry pear kiwi null null null null
orange cherry pear kiwi null null null null
orange cherry pear null null null null
cherry pear null null

1번째 항목은 pear이다.
```

2배로 확장

2배로 확장

2배로 확장

삭제

1/2로 축소



# ArrList 클래스(11) - C++

```
template <class T>
class ArrList
{
public:
    ArrList(int n = 10) ;
    void peek(int k) ;
    void insertLast(T newItem) ;
    void insertElement(T newItem, int k) ;
    T deleteElement(int k) ;
private:
    T *array ;
    int size ;
    int capacity ;
    void resize(int newSize) ;
}

ArrList::ArrList(int n = 8)
{
    if (n < 1) throw "Size must be > 0" ;
    capacity = n ;
    size = 0 ;
    array = new int[n] ;
}
```

```
void ArrList::peek(int k)
{
    if (size == 0)
        throw "Array has no elements" ;
    if (k > size)
        throw "Index is larger than the size of array" ;
    return array[k] ;
}

template <class T>
void ArrList::insertLast(T newItem)
{
    if (size == capacity)
        resize(2*capacity) ;
    a[size++] = newItem ;
}
```

# ArrList 클래스(12) - C++

```
template <class T>
void ArrList::insertElement(T newItem, int k)
{
    if (size == capacity)
        resize(2*capacity)
    for (int i = size - 1 ; i >= k ; i --)
        array[i+1] = array[i] ;
    array[k] = newItem ;
    size ++ ;
}
```

```
template <class T>
void ArrList::resize(int newSize)
{
    if (newSize < 0 ) throw "New Size should be >= 0"
    T *newArray = new T[newSize] ;
    for ( i = 0 ; i < size ; i ++ )
        newArray[i] = array[i]
    delete [] array ;
    array = newArray ;
}
```

```
template <class T>
T ArrList::deleteElement(int k)
{
    if (size == 0) throw "ArrList has no element" ;
    T item = a[k] ;
    for (int i = k ; i < size ; i++) a[i] = a[i+1] ;
    size -- ;
    if (size > 0 && size == capacity/4)
        resize(capacity/2)
    return item ;
}
```

# 수행시간

---

## ▶ peek() 메소드

- ▶ 인덱스를 이용하여 배열 원소를 직접 접근하므로  $O(1)$

## ▶ 삽입이나 삭제

- ▶ 새 항목을 중간에 삽입하거나 중간에 있는 항목을 삭제한 후에 뒤 따르는 항목들을 한 칸씩 앞이나 뒤로 이동해야
  - ▶ 각각 최악의 경우는  $O(N)$  시간 소요
- ▶ 새 항목을 가장 뒤에 삽입하는 경우는  $O(1)$  시간
- ▶ 배열의 크기를 확대 또는 축소시키는 것도 최악경우는  $O(N)$  시간
- ▶ 상각분석에 따르면 삽입이나 삭제의 평균 수행시간은  $O(1)$

# 배열 응용 문제 - 다항식 (polynomial)

▶  $a(x)=3x^2+2x-4$ ,  $b(x)=x^8-10x^5-3x^3+1$

▶ 계수(coefficient) : 3, 2, -4

▶ 지수(exponent) : 2, 1, 0

▶ 변수(variable) : x

▶ 차수(degree): 0이 아닌 제일 큰 지수

▶ 다항식의 합과 곱

▶  $a(x) + b(x) = \sum (a^i + b^i)x^i$

▶  $a(x) \cdot b(x) = \sum (a^i x^i \cdot \sum (b^j x^j))$

$A(x) = \sum a_i x^{e_i}$  : 항의 합

$a_i$  : 계수(coefficient),  $a_i \neq 0$

$e^i$  : 지수(exponent), unique,  $\geq 0$

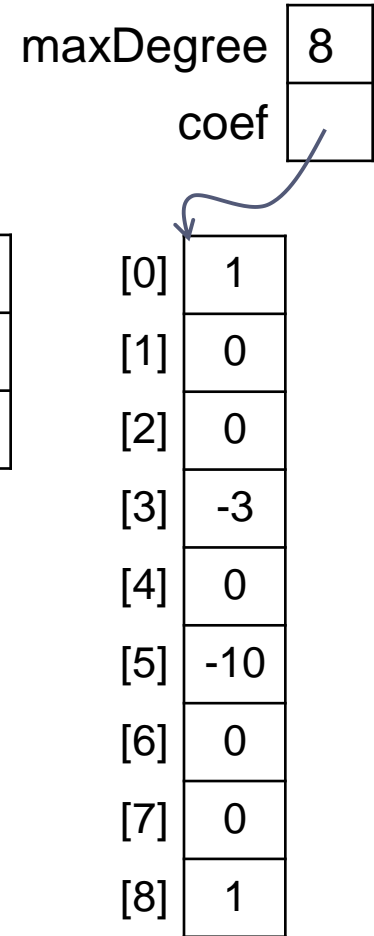
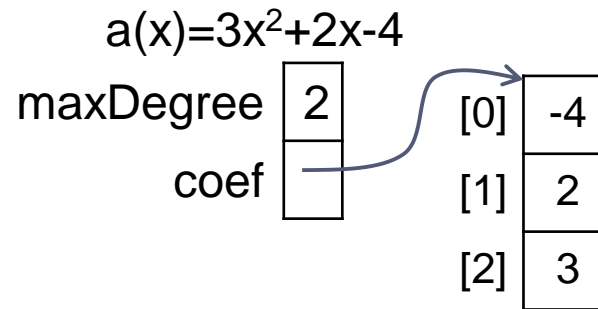
x : 변수(variable)

# 다항식 표현

$$b(x)=x^8-10x^5-3x^3+1$$

## ▶ [표현 1] Polynomial의 전용 데이터 멤버 선언

```
class Polynomial
{
    int maxDegree;
    float coef[] ;
}
```



- ▶ 장점 : 대부분 다항식의 연산(덧셈, 뺄셈, 계산, 곱셈 등)을 위한 알고리즘을 간단하게 구성할 수 있음
- ▶ 단점 : 희소 다항식에서 기억 공간 낭비
  - ▶ (예) 다항식  $x^{1000}+1$  → coef에서 999개의 엔트리는 0
  - 표현 2방법(클래스 term) 이용

# 다항식 표현

## ▶ [표현 2] 모든 다항식은 배열 termArray를 이용해 표현

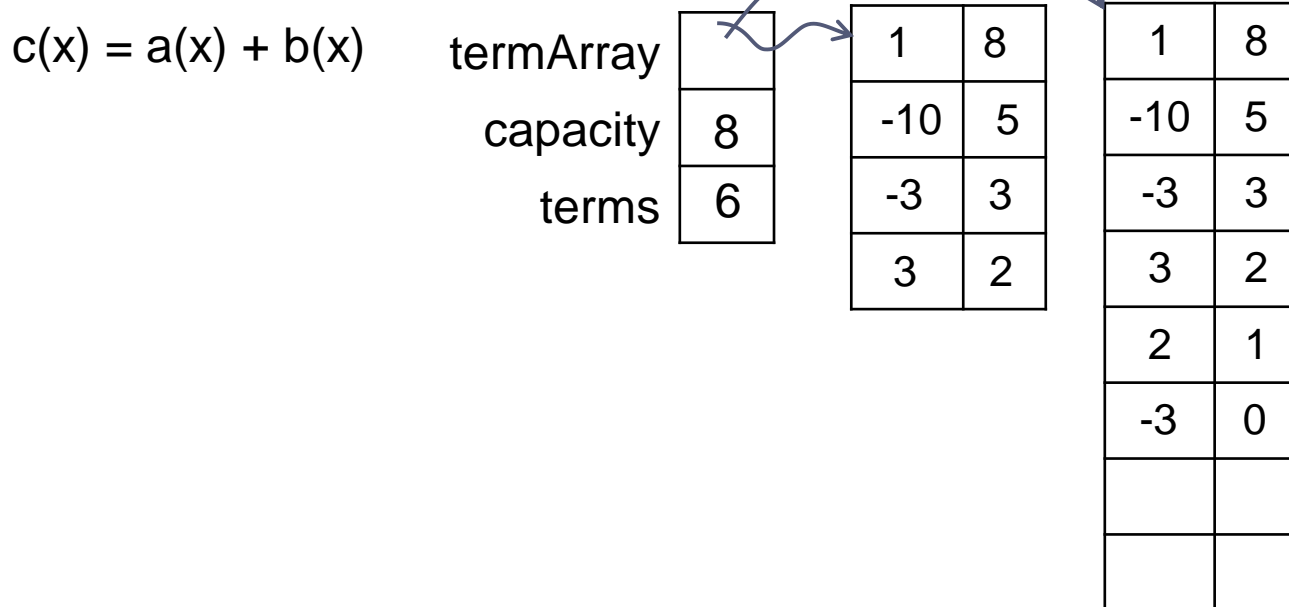
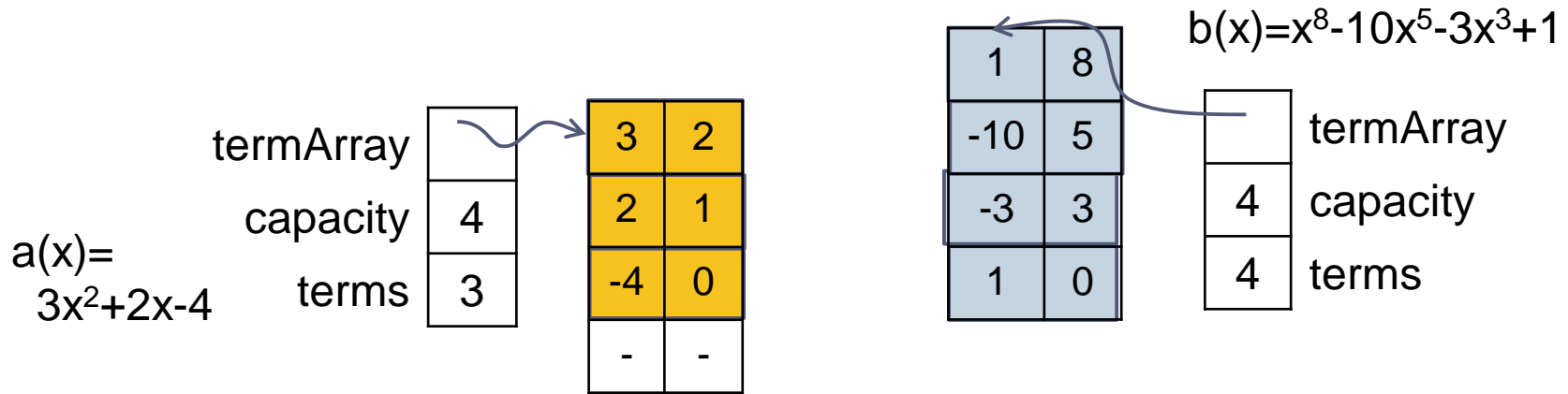
- ▶ 계수가 0이 아닌 항만 저장.
- ▶ termArray의 각 원소는 term 타입
- ▶ Polynomial의 정적 클래스 데이터 멤버

```
class Term
{
    float coef; // 계수
    int exp;    // 지수
};
```

## ▶ Polynomial의 전용 데이터 멤버 선언

```
class Polynomial
{
    Term *termArray ; // 0이 아닌 항의 배열
    int capacity;      // termArray의 크기
    int terms;         // 0이 아닌 항의 수
};
```

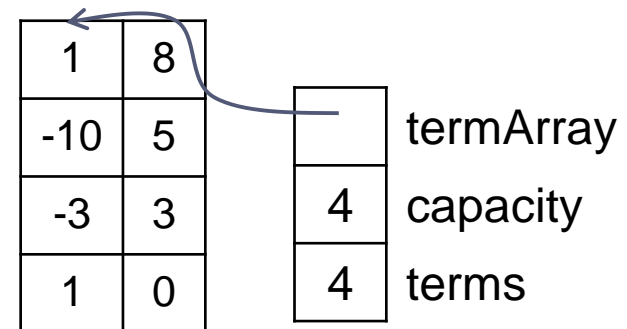
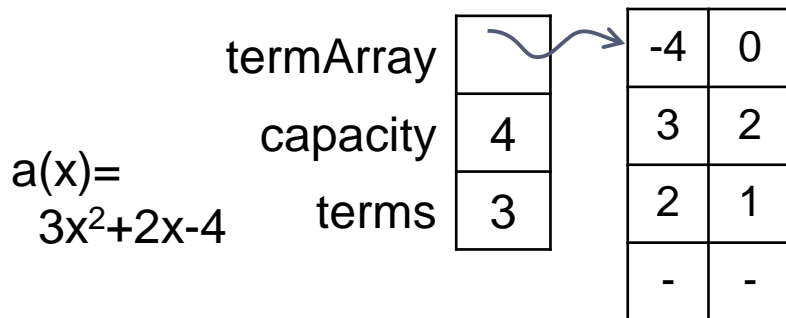
# 다항식 표현과 덧셈 Add()



# 다항식 덧셈

## ▶ Add 의 분석 :

- ▶ a와 b에서의 0이 아닌 항의 수(m, n)로 분석
- ▶ 전체 실행 시간:  $O(m+n)$ 
  - ▶ 배열을 두배 늘리는 것은 Add의 전체 실행 시간에 대해 기껏해야 상수 승의 영향을 줌. 즉 배열 두배 확장은 Add 전체 실행 시간의 아주 작은 부분이 됨
- ▶ v.s. 배열 표현시 실행시간은  $O(\max(m,n))$
- ▶ 만일 지수의 내림차순으로 정렬되어 있지 않다면  $O(m*n)$





# 배열 응용 문제 - 희소 행렬(Sparse matrices)

---

## ▶ $a[m][n]$

### ▶ $m \times n$ 행렬 $a$

- ▶  $m$  : 행의 수
- ▶  $n$  : 열의 수
- ▶  $m \times n$  : 원소의 수

### ▶ 희소 행렬(sparse matrix)

- ▶ 0이 아닌 원소수 / 전체 원소수  $\ll$  small  
→ 0이 아닌 원소만 저장할 필요 있으므로 시간/공간 절약

### ▶ 행렬에 대한 연산

- ▶ Creation(생성)
- ▶ Transpose(전치)
- ▶ Addition(덧셈)
- ▶ Multiplication(곱셈)

# 일반적인 행렬의 표현 및 연산

## ▶ 2차원 배열으로 표현

	0	1	2
0	-27	3	4
1	6	82	-2
2	109	-64	11
3	12	8	9
4	48	27	47

	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	22	-15
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

- ▶ 우측 행렬과 같은 형태를 희소행렬(sparse matrix)라고 함
- ▶ 곱셈 알고리즘

```
for (int i = 0; i < a.rows; i++)
    for (int j = 0; j < b.cols; j++)
    {
        sum = 0;
        for (int k = 0; k < A.Cols; k++)
            sum += (a[i][k] * b[k][j]);
        c[i][j] = sum;
    }
```

# 효율적인 희소 행렬 표현

## ▶ 표현 방법

- ▶ <행, 열, 값> 3원소 쌍(triple)으로 유일하게 식별 가능
- ▶ 행 우선 표기법을 선택
  - ▶ 첫번째 행의 3원소 쌍들은 행 순서로 저장한 뒤 두번째 행의 3원소 쌍을 순서대로 저장하는 방식
  - ▶ 한행에서 모든 3원소 쌍들은 열 인덱스가 오름차순
- ▶ 연산 종료를 보장하기 위해 행렬의 행과 열의 수와 0이 아닌 항의 수를 알아야 함.

```
class MatrixTerm
{
    int row ;
    int col ;
    int value;
};
```

# 효율적인 희소 행렬 표현

## ▶ 클래스 SparseMatrix 내부 정의

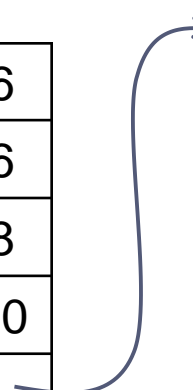
- ▶ rows : 행의 수
- ▶ cols : 열의 수
- ▶ terms: 0이 아닌 항의 총 수
- ▶ capacity: smArray의 크기

```
class SparseMatrix
{
    int rowSize ;
    int colSize ;
    int termSize ;
    int capacity;
    MatrixTerm smArray[] ;
}
```

- ▶ 연산의 구현을 위해서는  
일관성 있는 smArray의 표현이 필요

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

rowSize	6
colSize	6
termSize	8
capacity	10
smArray	



0	0	15
0	3	22
0	5	-15
1	1	11
1	2	3
2	3	-6
4	0	91
5	2	28

# 원소 쌍으로 저장된 희소행렬과 전치행렬

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

행      렬      값

smArray	[0]	0	0	15
	[1]	0	3	22
	[2]	0	5	-15
	[3]	1	1	11
	[4]	1	2	3
	[5]	2	3	-6
	[6]	4	0	91
	[7]	5	2	28

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

전치행렬      행      렬      값

smArray	[0]	0	0	15
	[1]	0	4	91
	[2]	1	1	11
	[3]	2	1	3
	[4]	2	5	28
	[5]	3	0	22
	[6]	3	2	-6
	[7]	5	0	-15

# 행렬의 전치

- ▶ 원래의 행렬 각 행  $i$ 에 대해서 원소  $(i, j, 값)$  을 가져와서 전치행렬의 원소  $(j, i, 값)$  으로 저장

- ▶ (예)  $(0, 0, 15) \rightarrow (0, 0, 15)$        $(0, 3, 22) \rightarrow (3, 0, 22)$   
 $(0, 5, -15) \rightarrow (5, 0, -15)$        $(1, 1, 11) \rightarrow (1, 1, 11)$
- ▶ 올바른 순서 유지 위해 기존원소 이동시켜야 하는 경우 발생

		행	렬	값	전치행렬		행	렬	값
smArray	[0]	0	0	15	smArray	[0]	0	0	15
	[1]	0	3	22		[1]	3	0	22
	[2]	0	5	-15		[2]	5	0	-15
	[3]	1	1	11		[3]	1	1	11
	[4]	1	2	3		[4]	2	1	3
	[5]	2	3	-6		[5]	3	2	-6
	[6]	4	0	91		[6]	0	4	91
	[7]	5	2	28		[7]	2	5	28

# 행렬의 전치 - 열우선 검색 방법

## ▶ 알고리즘

- ▶ for (열 j에 있는 모든 원소에 대해)
  - ▶ 원소(i, j, 값)을 원소(j, i, 값)으로 저장

```
3 SparseMatrix trasMatrix(colSize, rowSize, termSize);
4 if (termSize > 0)
5     { // nonzero 행렬에 대해서
6         int currentB = 0 ;
7         for (int c = 0; c < colSize ; c++) // 각 열에 대해서 처리
8             for (int i = 0 ; i < termSize ; i++)
9                 // 작은 열부터의 원소를 전치된 행렬에 하나씩 복사
10                    if (smArray[i].col == c)
11                        {
12                            transMatrix.smArray[currentB].row = c;
13                            transMatrix.smArray[currentB].col = smArray[i].row;
14                            transMatrix.smArray[currentB++].value = smArray[i].value;
15                        }
16    }
17 return transMatrix;
```

# 행렬의 전치 - 열우선 검색 방법

## ▶ Transpose의 분석

- ▶ 총 실행시간:  $O(\text{terms} \cdot \text{cols})$
- ▶ \*this와 b가 필요로 하는 공간 외에 이 함수는 변수 c, i, currentB를 위한 고정된 공간만을 추가로 필요로 한다.
- ▶ 최악의 경우 실행시간은  $O(\text{rows} \cdot \text{cols} \cdot \text{cols})$ 
  - ▶ 공간 절약을 위해 시간을 희생한 결과

## ▶ 단순 2차원 배열 표현되는 경우

- ▶  $O(\text{rows} \cdot \text{cols})$ 시간내의  $\text{rows} \cdot \text{cols}$ 크기 행렬의 전치를 얻을 수 있음.
- ▶ 가장 단순한 형태의 알고리즘

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



# 행렬의 전치 - FastTranspose

## ▶ 메모리를 조금 더 사용한 개선 알고리즘 : FastTranspose

- ▶ 먼저 행렬 \*this의 각 열에 대한 원소 수를 구함

- ▶ 전치 행렬 b의 각 행의 원소 수를 결정

```
for (i = 0 ; i < terms ; i++)  
    rowSize[smArray[i].col]++ ;
```

- ▶ 이 정보에서 전치행렬 b의 각행의 시작위치 구함

- ▶ 원래 행렬 a에 있는 원소를 하나씩 전치 행렬 b의 올바른 위치로 옮김

	ROW_SIZE	ROW_START
[0]	2	0
[1]	1	2
[2]	2	3
[3]	2	5
[4]	0	7
[5]	1	7

↑                      ↑  
# of terms          starting position  
in b's row (a's col)      of b's row

	행	렬	값		행	렬	값
[0]	0	0	15	[0]	0	0	15
[1]	0	3	22	[1]	0	4	91
[2]	0	5	-15	[2]	1	1	11
[3]	1	1	11	[3]	2	1	3
[4]	1	2	3	[4]	2	5	28
[5]	2	3	-6	[5]	3	0	22
[6]	4	0	91	[6]	3	2	-6
[7]	5	2	28	[7]	5	0	-15

- ▶ 실행시간:  $O(\text{columns} + \text{terms})$

```

1 SparseMatrix SparseMatrix::FastTranspose()
2 {
3     SparseMatrix b(cols,rows,terms);
4     if (terms > 0)
5     { // nonzero 행렬에 대해서
6         int *rowSize = new int[cols];
7         int *rowStart = new int[cols];
8         // rowSize에 원본 행렬의 각 열의 원소 또는 전치행렬의 각 행의 원소 개수 계산하여 저장
9         fill(rowSize, rowSize+cols,0); // initialize
10        for (i = 0 ; i < terms ; i++) rowSize[smArray[i].col]++;
11        // rowStart에 전치행렬의 각 열의 시작 위치 계산하여 저장
12        rowStart[0] = 0 ;
13        for (i = 1 ; i < cols ; i++) rowStart[i] = rowStart[i-1] + rowSize[i-1] ;

14        for (i = 0 ; i < terms ; i++)
15        { // 원본 행렬에서 결과 전치 행렬 b에 각 원소를 저장
16            int j = rowStart[smArray[i].col];
17            b.smArray[j].row = smArray[i].col;
18            b.smArray[j].col = smArray[i].row ;
19            b.smArray[j].value = smArray[i].value ;
20            rowStart[smArray[i].col]++ ;
21        }
22        delete [] rowSize ;
23        delete [] rowStart ;
24    }
25    return b ;
26 }

```