# 행렬 곱셈

- a(m ×n)와 × b(n ×p)의 곱셈
- 차원은 m×p

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \qquad ,0 \le i < m, 0 \le j < p$$

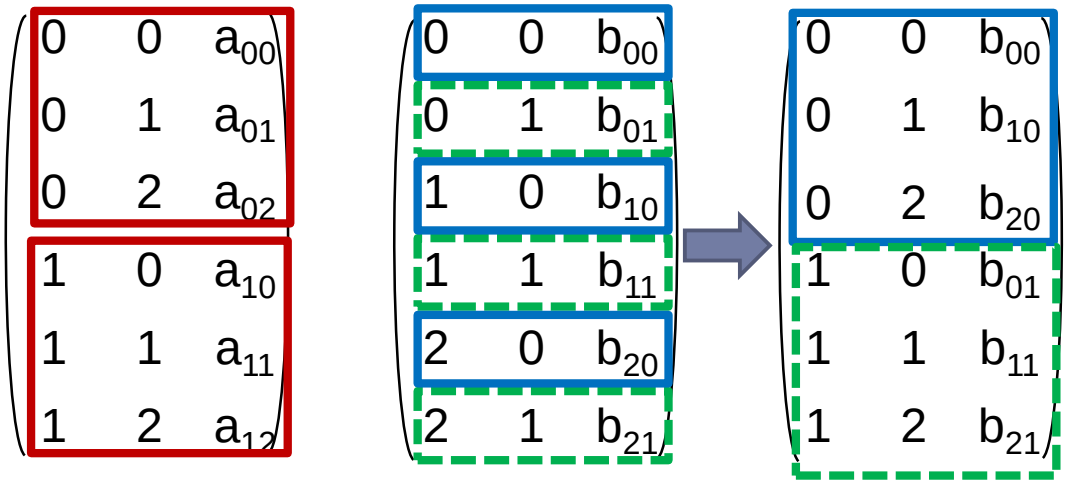
$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- 순서 리스트로 표현된 두 희소 행렬의 곱셈
  - 목표 - d의 원소를 행별로 계산
  - 조건 - 이전 계산 원소를 이동하지 않고 적절한 위치에 저장
  - 행렬 a의 한 행을 선택하고 j = 0,1,…,b.cols-1에 대해 b의 j열에 있는 모든 원소를 찾음
  - j열에 있는 원소 찾기를 효율화시키기 위해서 b를 전치
  - a의 i행과 b의 j열의 원소들이 정해지면 다항식 덧셈과 유사한 합병연산 수행

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \times \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \end{pmatrix}$$

$$= \begin{pmatrix} a_{00}xb_{00}+a_{01}xb_{10}+a_{02}xb_{20} & a_{00}xb_{01}+a_{01}xb_{11}+a_{02}xb_{21} \\ a_{10}xb_{00}+a_{11}xb_{10}+a_{12}xb_{20} & a_{10}xb_{01}+a_{11}xb_{11}+a_{12}xb_{21} \end{pmatrix}$$

a행렬은 행 중심으로, b행렬은 열 중심으로 접근하는 것에 착안
a행렬은 희소행렬 표현을 그대로 사용하고, b행렬은 전치행렬을 사용

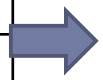


$$\begin{pmatrix} 1 & 0 & 2 \\ 0 & 3 & 0 \\ 4 & 0 & 0 \\ 0 & 0 & 5 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1*1 & & 1*1+2*1 \\ 3*1 & 3*1 & 3*1 \\ 4*1 & & 4*1 \\ & & 5*1 \end{pmatrix}$$

| | | |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 2 | 2 |
| 1 | 1 | 3 |
| 2 | 0 | 4 |
| 3 | 2 | 5 |

*

| | | |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 2 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |
| 1 | 2 | 1 |
| 2 | 2 | 1 |



| | | |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |
| 2 | 0 | 1 |
| 2 | 1 | 1 |
| 2 | 2 | 1 |

=

| | | |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 2 | 1+2 |
| 1 | 0 | 3 |
| 1 | 1 | 3 |
| 1 | 2 | 3 |
| 2 | 0 | 4 |
| 2 | 2 | 4 |
| 3 | 2 | 5 |

# 희소 행렬의 곱셈

- **Multiply의 분석**
  - a의 행 r이 곱해지는 동안의 소요시간 : O(r(b.cols • $t_r$+b.terms))
    - $t_r$은 a의 행 r에 있는 항의 총 개수
  - a의 행 currRowA에 대해 한번 반환하는데 걸리는 시간
  - : O(b.cols • a.terms +a.rows • b.terms)
  - 전체소요시간 :

$$O(\sum_r (b.cols \cdot t_r + b.terms)) = O(b.cols \cdot a.term + a.rows \cdot b.terms)$$

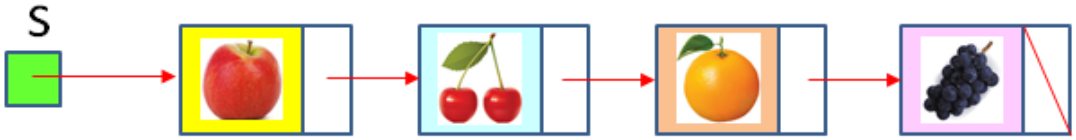
- **통상적인 곱셈 알고리즘**

```
1 SparseMatrix SparseMatrix::Multiply(SparseMatrix b)
2 {// Return the product of the sparse matrices *this and b.
3       if (cols != b.rows) throw "Incompatible matrices";
4
5       SparseMatrix bXpose = b.FastTranspose();
6       SparseMatrix d(rows,b.cols,0);
7       int currRowIndex = 0,  currRowBegin = 0, currRowA = smArray[0].row;
8
9       // 프로그램 작성을 용이하게 하기 위해, 가짜 원소를 하나 만들고 “틀린 원소 "  하나 추가
10      if (terms == capacity) ChangeSize1D(terms + 1);
11     bXpose.ChangeSize1D(bXpose.terms + 1);
12      smArray[terms].row = rows;
13      bXpose.smArray[b.terms].row = b.cols;
14      bXpose.smArray[b.terms].col = -1 ;

15      int sum = 0;   // 한 원소의 결과를 저장하기 위한 변수
16      while (currRowIndex < terms)
17      {// 곱셈 결과를 저장하는 d에 currentRowA부터 값을 채워넣기
18          Int currColB = bXpose.smArray[0].row;
19          Int currColIndex = 0;
20          while (currColIndex <= b.terms)
21          {// 내 행렬의 currRowA의 행과 b의 currColB의 값을 곱해 나가기
22              If (smArray[currRowIndex].row != currRowA)
23              {// currRowA의 원소가 없거나 다 처리한 경우
24                  d.StoreSum(sum, currRowA, currColB);  // 지금까지 구한 sum을 새 행렬에 저장
25                  Sum = 0 ;   // reset sum
26                  currRowIndex = currRowBegin;
27                  // 다음 열로 진행
28                 while (bXpose.smArray[currColIndex].row == currColB)
29                     currColIndex++;
30                 currColB = bXpose.smArray[currColIndex].row;
```

```
31              }
32              else if (bXpose.smArray[currColIndex].row != currColB)
33              {// b행렬에 currColB를 모두 다 처리한 경우
34                  d.StoreSum(sum, currRowA, currColB);
35                  sum = 0 ;   // reset sum
36                  // 다음 열부터 진행
37                 currRowIndex = currRowBegin;
38                 currColB = bXpose.smArray[currColIndex].row;
39             }
40
41             else if (smArray[currRowIndex].col < bXpose.smArray[currColIndex].col)
42                    currRowIndex++ ;   // b배열의 다음 행에 대한 계산 계속
43
44             else if (smArray[currRowIndex].col == bXpose.smArray[currColIndex].col)
45             {// 내 행렬의 열과 전치행렬의 열(b행렬에서는 행)의 값이 같으면 곱해야 할 값
46                    sum += smArray[currRowIndex].value * bXpose.smArray[currColIndex].value;
47                    currRowIndex++; currColIndex++;
48              }
49
50              else currColIndex++;  // next term in currColB
51         } // end of while (currColIndex <= b.terms)
52
53         while (smArray[currRowIndex].row == currRowA) // advance to next row
54            currRowIndex++;
55          currRowBegin = currRowIndex;
56          currRowA = smArray[currRowIndex].row;
57     } // end of while (currRowIndex < terms)
58     return d;
59 }
```

# 2.2 단순연결리스트

- 단순연결리스트(Singly Linked List)는 동적 메모리 할당을 이용해 리스트를 구현하는 가장 간단한 형태의 자료구조
- 동적 메모리 할당을 받아 노드(node)를 저장하고, 노드는 레퍼런스를 이용하여 다음 노드를 가리키도록 만들어 노드들을 한 줄로 연결시킴



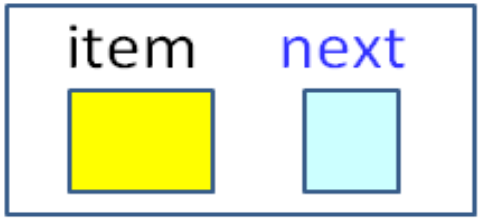
# 단순연결리스트

- 연결리스트에서는 삽입이나 삭제 시 항목들의 이동이 필요 없음
- 배열의 경우 최초에 배열의 크기를 예측하여 결정해야 하므로
  대부분의 경우 배열에 빈 공간을 가지고 있으나,
  연결리스트는 빈 공간이 존재하지 않음
- 연결리스트에서는 항목을 탐색하려면 항상 첫 노드부터 원하는 노드를 찾을 때까지 차례로 방문하는 순차탐색(Sequential Search)을 해야

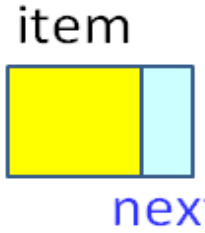
# 단순연결리스트의 노드를 위한 Node 클래스

```
01 public class Node <E> {
02     private E       item;
03     private Node<E> next;
04     public Node(E newItem, Node<E> node){  // 생성자
05         item = newItem;
06         next = node;
07     }
08     // get과 set 메소드들
09     public E       getItem() { return item; }
10     public Node<E> getNext() { return next; }
11     public void    setItem(E newItem)      { item = newItem; }
12     public void    setNext(Node<E> newNext){ next = newNext; }
13 }
```

- Node 객체는 항목을 저장할 item과 Node 레퍼런스를 저장하는 next를 가짐
- Line 04~07: Node 생성자
- Line 09~1: item과 next를 위한 get 과 set 메소드들




Node 객체의 표현




Node 객체의 간략한 표현

# 리스트를 단순연결리스트로 구현한 SList 클래스

```
01 import java.util.NoSuchElementException;
02 public class SList <E> {
03     protected Node head;   // 연결 리스트의 첫 노드 가리킴
04     private   int  size;
05     public SList(){        // 연결 리스트 생성자
06         head = null;
07         size = 0;
08     }
   // 탐색, 삽입, 삭제 연산을 위한 메소드 선언

   }
```

- Line 01: NoSuchElementException은 java.util 라이브러리에 선언된 클래스로서 underflow 발생 시 프로그램을 정지시키기 위한 import문
- Line 05~08: Slist 생성자는 연결리스트의 첫 노드를 가리키는 head를 null로 초기화하고 연결리스트의 노드 수를 저장하는 size를 0으로 초기화

# C++에서의 연결 리스트

```
template <class T> class Chain;  // 전방 선언

template <class T>
class ChainNode {
friend class Chain<T>;
private:
        T data;
        ChainNode<T> *link;
};

template <class T>
class Chain {
public:
        Chain() {first = 0;};  // first를 0으로 초기화
        // 체인 조작 연산들
        .
        .
private:
       ChainNode<T> *first, *last ; //조작 편의를 위해 last 정의
};
```

```
template<class T>
void Chain<T>::insertBack(const T& e)
{
      if ( first ) //공백이 아닌 체인
      {
            last→link = new ChainNode<T>(e);
            last = last→link;
      }
      else first = last = new ChainNode<T>(e);
}
```

# SList 클래스에서의 탐색

- **탐색은 인자로 주어지는 target을 찾을 때까지 연결리스트의 노드들을 첫 노드부터 차례로 탐색**

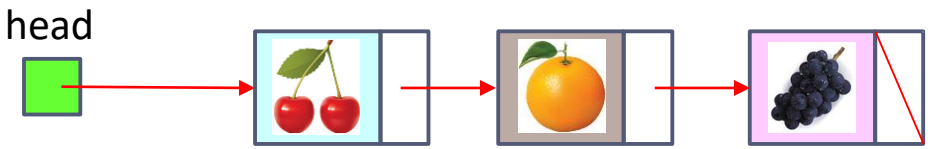
```
01  public int search(E target) {  // target을 탐색
02      Node p = head;
03      for (int  k = 0; k < size ;k++){
04          if (target == p.getItem()) return k;
05          p = p.getNext();
06      }
07      return -1; // 탐색을 실패한 경우 -1 리턴
08  }
```

  - Line 02: 지역변수 p가 연결리스트의 첫 노드를 참조
  - Line 03:  for-루프를 통해 target을 찾으면 line 04에서 target이 k번째 인덱스에 있음을 리턴
  - 탐색에 실패하면 line 07에서 '-1'을 리턴

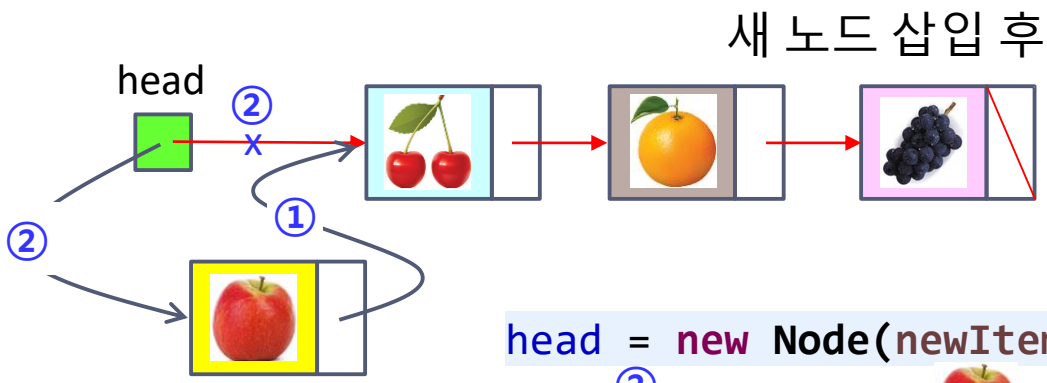
# SList 클래스에서의 제일 앞에 삽입

```
01 public void insertFront(E newItem){ // 연결리스트 맨 앞에 새 노드 삽입
02     head = new Node(newItem, head);
03     size++;
04 }
```

- **insertFront() 메소드: 새 노드를 리스트의 첫 번째 노드가 되도록 연결**
  - Line 02: 그림 참조




새 노드 삽입 전



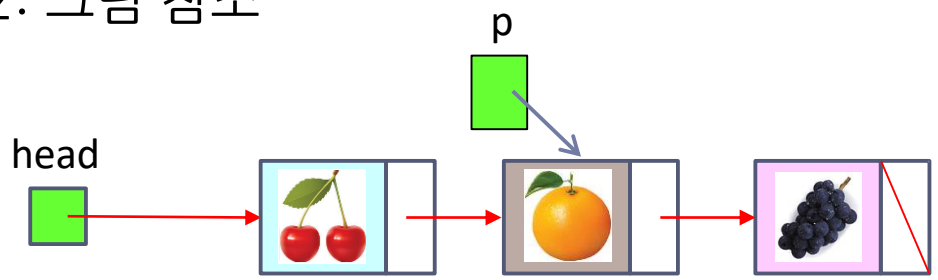

새 노드 삽입 후

```
head = new Node(newItem, head);
       ②                 ①
```

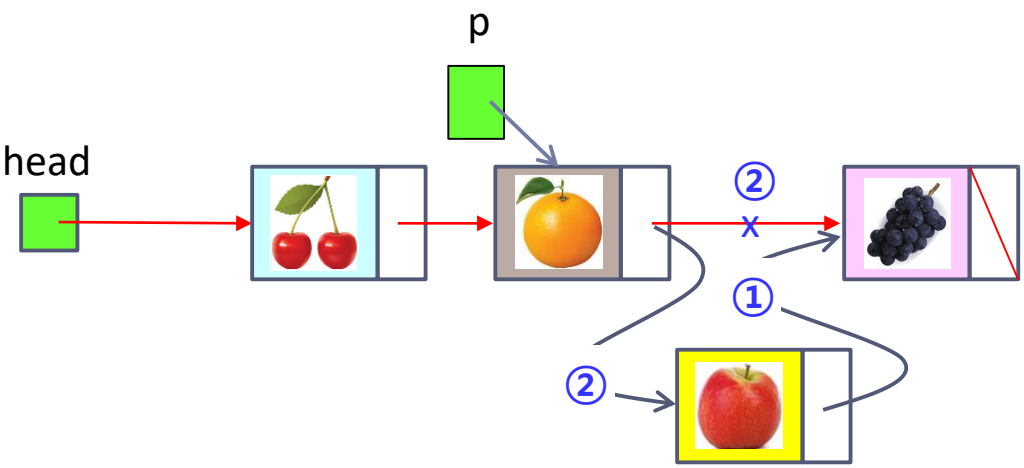
# SList 클래스에서의 특정 노드 뒤에 삽입

```
01 public void insertAfter(E newItem, Node p){ // 노드 p 바로 다음에 새 노드 삽입
02     p.setNext(new Node(newItem, p.getNext()));
03     size++;
04 }
```

- **insertAfter() 메소드는 p가 가리키는 노드의 다음에 새 노드 삽입**
  - Line 02: 그림 참조




새 노드 삽입 전



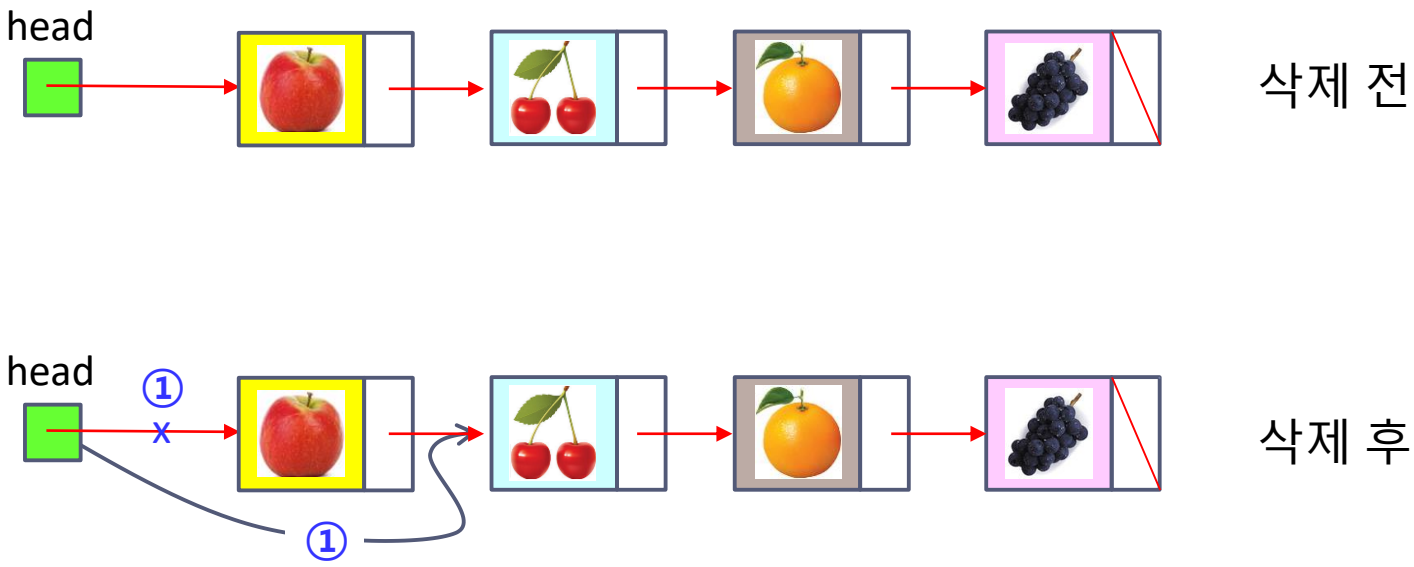

새 노드 삽입 후

```
p.setNext(new Node(newItem, p.getNext()));
```

# SList 클래스에서의 처음 요소 삭제

```
01 public void deleteFront(){          // 리스트의 첫 노드 삭제
02     if(size == 0) throw new NoSuchElementException();
03     head = head.getNext();
04     size--;
05 }
```

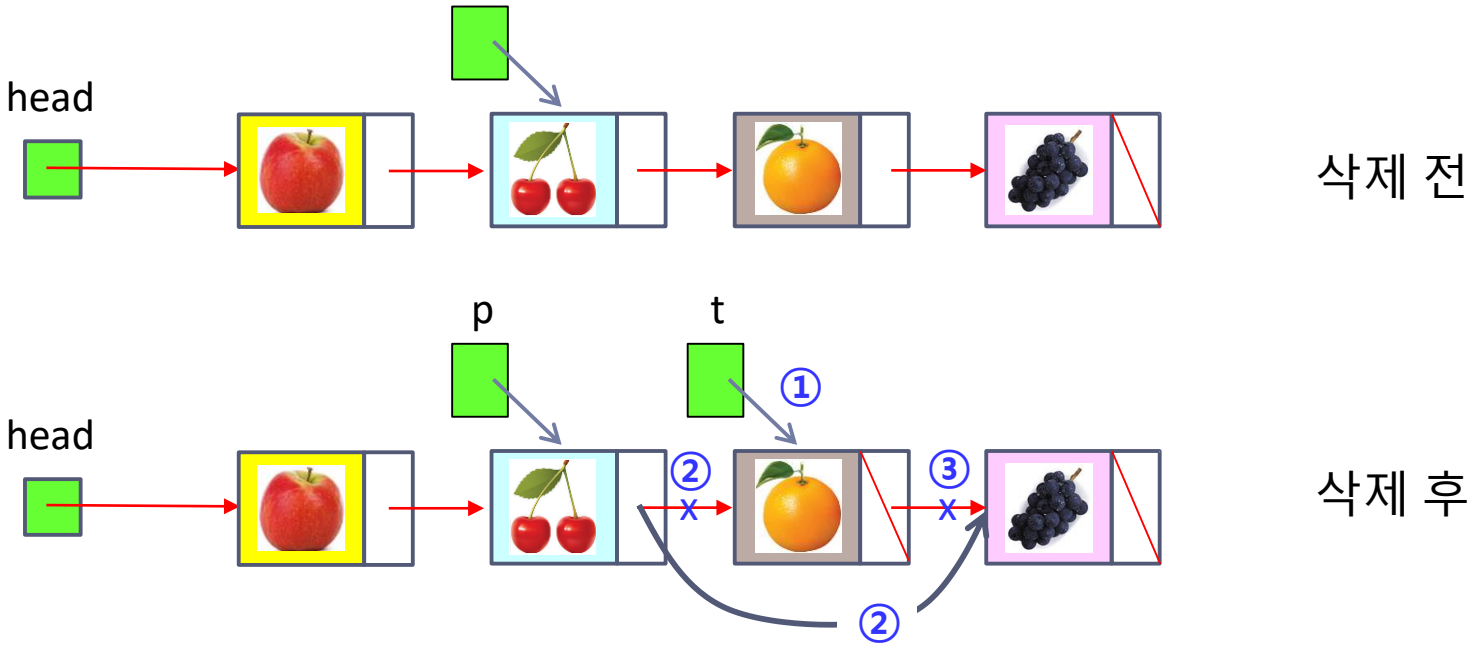
- deleteFront() 메소드는 리스트가 empty가 아닐 때, 리스트의 첫 노드를 삭제



# SList 클래스에서의 특정 노드 위에 삭제

```
01 public void deleteAfter(Node p){ // p가 가리키는 노드의 다음 노드를 삭제
02     if (p == null) throw new NoSuchElementException();
03     Node t = p.getNext();          ①
04     p.setNext(t.getNext());        ②
05     t.setNext(null);               ③
06     size--;
07 }
```

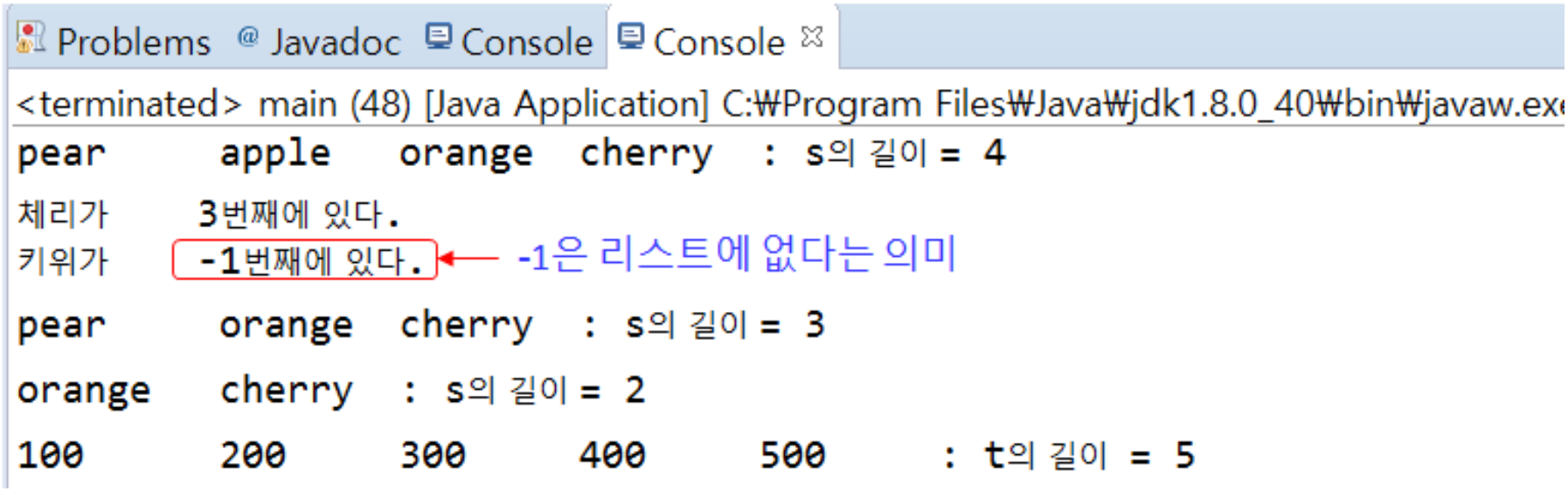
- deleteAfter() 메소드는 p가 가리키는 노드의 다음 노드를 삭제



```java
public class main {
    public static void main(String[] args) {

        SList<String> s = new SList<String>();  // 연결 리스트 객체 s 생성
        s.insertFront("orange"); s.insertFront("apple");
        s.insertAfter("cherry",s.head.getNext());
        s.insertFront("pear");

        s.print();
        System.out.println(": s의 길이 = "+s.size()+"\n");
        System.out.println("체리가 \t"+s.search("cherry")+"번째에 있다.");
        System.out.println("키위가 \t"+s.search("kiwi")+"번째에 있다.\n");
        s.deleteAfter(s.head);
        s.print();
        System.out.println(": s의 길이 = "+s.size());System.out.println();
        s.deleteFront();
        s.print();
        System.out.println(": s의 길이 = "+s.size());System.out.println();

        SList<Integer> t = new SList<Integer>();  // 연결 리스트 객체 t 생성
        t.insertFront(500); t.insertFront(200);
        t.insertAfter(400,t.head);
        t.insertFront(100);
        t.insertAfter(300,t.head.getNext());
        t.print();
        System.out.println(": t의 길이 = "+t.size());
    }
}
```

item의 타입이 String인 연결리스트를 생성하여 다양한 연산을 수행하며, Integer타입의 연결리스트를 생성하고, 삽입 연산을 수행하여 항목이 5개인 리스트를 만듬

# 프로그램의 수행 결과


Problems
Javadoc
Console
Console
<terminated> main (48) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
pear apple orange cherry : s의 길이 = 4
체리가 3번째에 있다.
키위가 -1번째에 있다.
-1은 리스트에 없다는 의미
pear orange cherry : s의 길이 = 3
orange cherry : s의 길이 = 2
100 200 300 400 500 : t의 길이 = 5

# 수행시간

- **search() 연산: 탐색을 위해 연결리스트의 노드들을 첫 노드부터 순차적으로 방문해야 하므로 O(N) 시간이 소요**
- **삽입이나 삭제 연산: 각각 상수 개의 레퍼런스를 갱신하므로 O(1) 시간이 소요**
  - 단, insertAfter()나 deleteAfter()의 경우에 특정 노드 p의 레퍼런스가 주어지지 않으면 head로부터 p를 찾기 위해 search()를 수행해야 하므로 O(N) 시간이 소요

# 다양한 체인 연산(1)

Program 4.12:두 체인의 concatenation

```
template <class T>
void Chain<T>::concatenate(Chain<T>& b)
{// b가 현재 chain 뒤에 연결
   if (first)
   {
        last->link = b.first;
        last = b.last;
   }
   else
   {
        first = b.first;
        last = b.last;
   }
   b.first = b.last = 0;
}
```

Program 4.13:list를 역순으로 바꾸기

```
template <class T>
void Chain<T>::reverse()
{
    ChainNode<T> *current = first,  *previous = 0;
    while (current)
    {
        ChainNode<T> *r = previous;
        previous = current;       // 현재를 이전으로
        current = current->link;  // 현재는 다음으로
        previous->link = r;  // 이전 다음을 그 이전으로
  }
  first = previous;
}
```

# 다양한 체인 연산(2)

```
//체인 복사생성자
==================================
template <class T>
Chain<T>::Chain (Chain<T>& src)
{
   ChainNode<T> *curSrc = src->first ; // 원본
   ChainNode<T> *target ; // 복사본 따라가는 포인터
   if (!curSrc) first = null ;  // 비어있는 트리 처리
   else
   {
      first = new ChainNode(curSrc->data) ;
      for (ChainNode<T> *target = first ;
                curSrc -> link ; curSrc = curSrc->link, target = target->link)
            target->link = new ChainNode(curSrc->link->data, 0) ;
      }
}
==================================
```

# 다양한 체인 연산(3)

```
//체인 복사생성자. 재귀형태
================================
template <class T>
Chain<T>::Chain (Chain<T>& src)
{
   first = Copy(src.first) ;
}

template <class T>
ChainNode<T> * Chain<T>::Copy(ChainNode<T> *src)
{
   if (!src) return 0 ;
   else return  new ChainNode(src->data, Copy(src->link)) ;
}
================================
```