제 2장 리스트

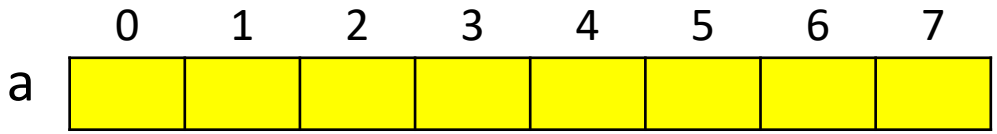
리스트

- 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

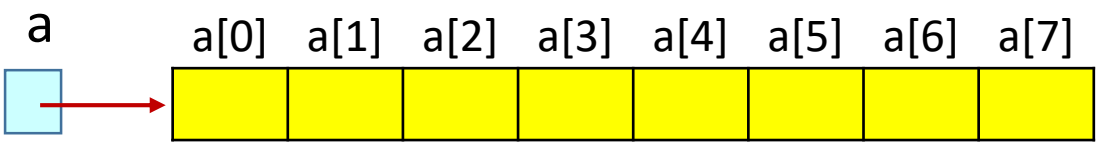
2.1 배열

- 배열(Array)은 동일한 타입의 원소들이 연속적인 메모리 공간에 할당되어 각 항목이 하나의 원소에 저장되는 기본적인 자료구조이다.
- 특정 원소에 접근할 때에는 배열의 인덱스를 이용하여 O(1) 시간에 접근할 수 있다.
- 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 없다.

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

- 각 원소 a[i]는 a가 가지고 있는 레퍼런스에 원소의 크기(바이트) x i 를 더하여 a[i]의 레퍼런스를 계산
- a[i] = a + (원소의 크기 x i)
- char 배열의 원소의 크기 = 2바이트, int 배열의 원소의 크기 = 4바이트

Overflow

- 배열은 미리 정해진 크기의 메모리 공간을 할당 받은 뒤 사용해야 하므로, 빈자리가 없어 새 항목을 삽입할 수 없는 상황(Overflow) 발생
- Overflow 가 발생하면 에러 처리를 하여 프로그램을 정지시키는 방법이 주로 사용된다. 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.

- 동적배열(Dynamic Array): 프로그램이 실행되는 동안에 할당된 배열

ArrList 클래스

- 리스트를 배열로 구현: ArrList 클래스

```
01 import java.util.NoSuchElementException;
02 public class ArrList <E> {
03     private  E a[];      // 리스트의 항목들을 저장할 배열
04     private  int size;   // 리스트의 항목 수
05     public  ArrList() { // 생성자
06         a = (E[]) new Object[1];  // 최초로 1개의 원소를 가진 배열 생성
07         size = 0;                 // 항목 수를 0으로 초기화
08     }
       // 탐색, 삽입, 삭제 연산을 위한 메소드 선언
   }
```

- Line 01: `java.util` 라이브러리에 선언되어 있는 `NoSuchElementException` 클래스를 이용하여 리스트가 empty인 상황에서 항목을 읽으려고 하면(즉, underflow가 발생하면) 프로그램을 정지시키는 예외처리
- Line 05-08: `ArrList` 클래스의 생성자는 크기가 1인 generic 타입의 배열과 배열에 저장된 항목 수를 저장하는 size를 0으로 초기화

```
01  public E peek(int k) {  // k번째 항목을 리턴, 단순히 읽기만 한다.
02      if ( size == 0 )
03          throw new NoSuchElementException(); // underflow 경우에 프로그램 정지
04      return a[k];
05    }
```

- peek() 메소드: k번째 저장된 항목을 탐색, k = 0, 1, ….
- Line 04: a[k]를 리턴, k < size라고 가정

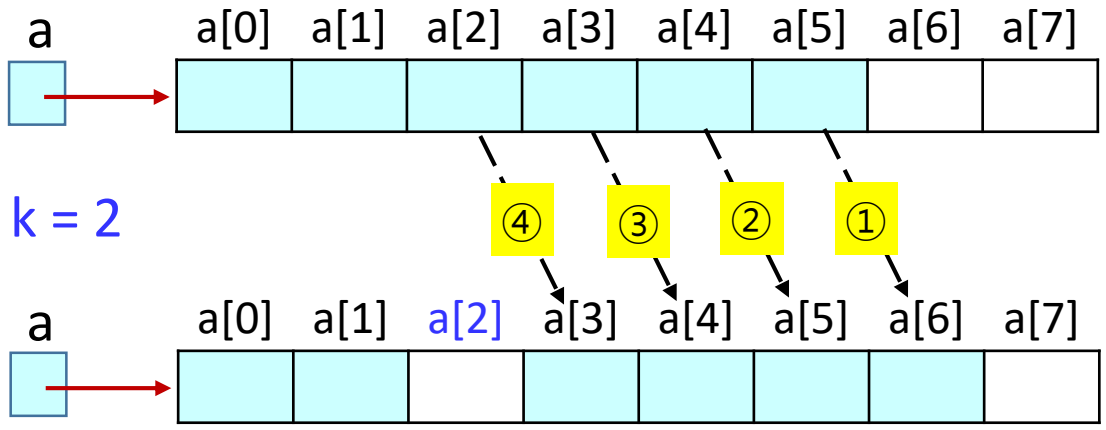
```
01  public void insertLast(E newItem) { // 가장 뒤에 새 항목 삽입
02      if (size == a.length)          // 배열에 빈 공간이 없으면
03          resize(2*a.length);        // 배열 크기 2배로 확장
04       a[size++] = newItem;          // 새 항목 삽입
05  }
```

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

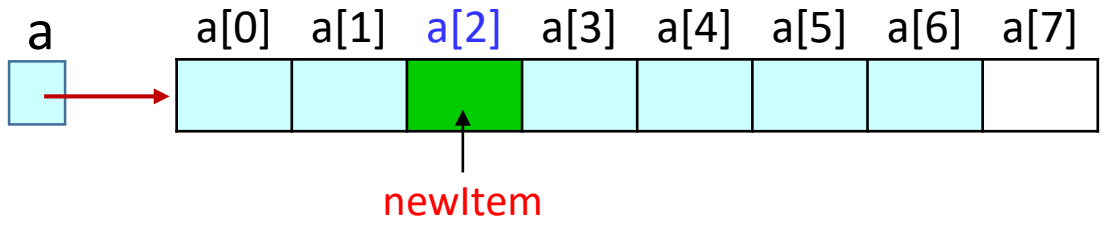
```
06 public void insert(E newItem, int k) { // 새 항목을 k-1번째 항목 다음에 삽입
07     if (size == a.length)              // 배열에 빈 공간이 없으면
08         resize(2*a.length);            // 배열 크기 2배로 확장
09     for (int i = size-1; i >= k; i--)  a[i+1] = a[i];  // 한 칸씩 뒤로 이동
10     a[k] = newItem;
11     size++;
12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

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

k번째 항목부터 마지막 항목까지 한 칸씩 이동



새 항목 a[k]에 삽입



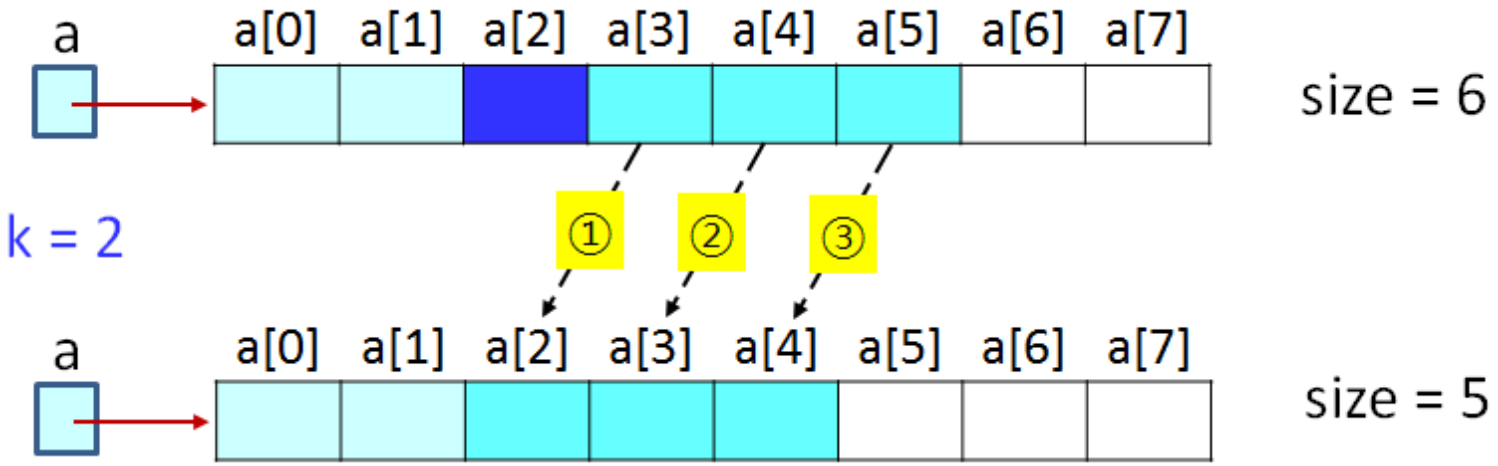
```
01 private void resize(int newSize) {        // 배열 크기 조절
02     Object[] t = new Object[newSize];     // newSize 크기의 새로운 배열 t 생성
03     for (int i = 0; i < size; i++)
04         t[i] = a[i];                      // 배열 s를 배열 t로 복사
05     a = (E[]) t;                          // 배열 t를 배열 s로
06 }
```

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

```
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 }
```

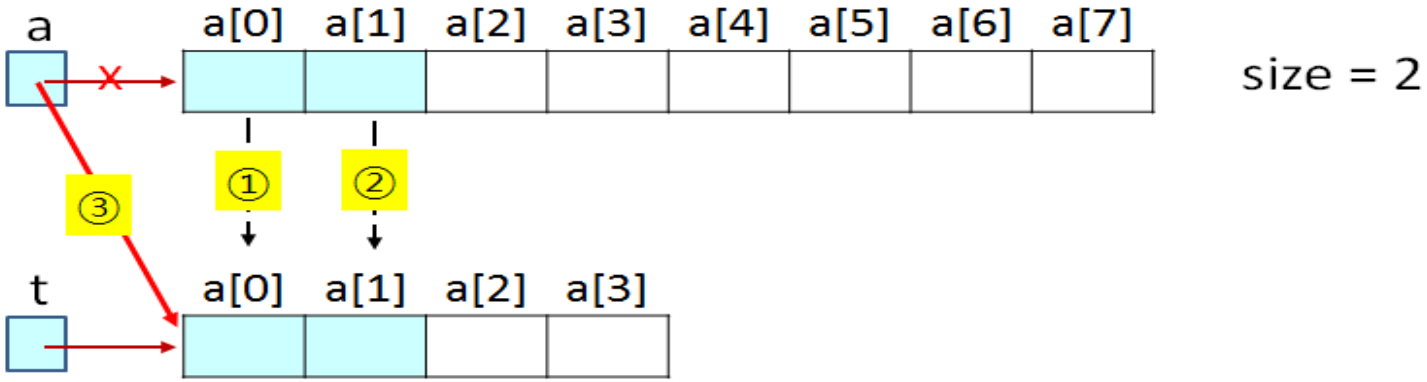
- delete(): k번째 항목을 삭제
- Line 02: underflow를 검사
- Line 03: 삭제되는 항목을 지역변수인 item에 저장
- Line 04: a[k+1]부터 a[size-1]까지 한 칸씩 앞으로 이동하여 a[k]의 빈칸을 메움

삭제된 원소의 공간 메우기



- Line 05: size를 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

배열의 크기 축소

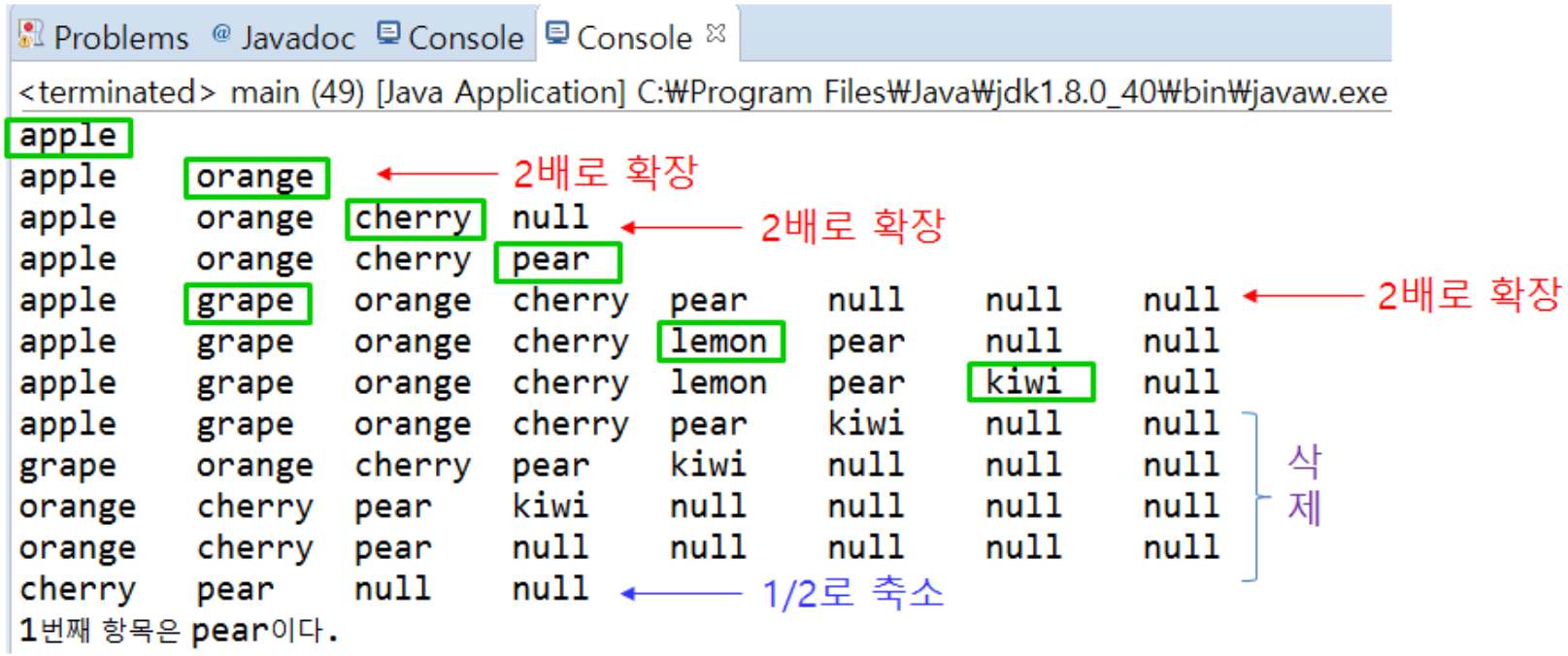


- 배열 크기가 8이고, 실제 두 개의 항목만 있는 경우, 배열 크기를 4로 축소
- 항목 수가 배열 크기의 1/4일 때 배열 크기를 1/2로 축소시키는 것은 축소된 배열에 1/2은 항목들로 차있고, 나머지 1/2은 비어있는 상태로 만들기 위해

```
1  public class main {
2      public static void main(String[] args) {
3          ArrList<String> s = new ArrList<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.

프로그램 실행 결과

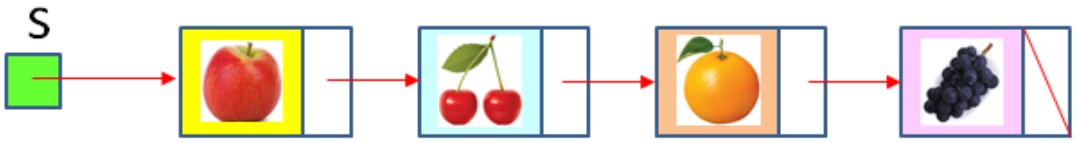

Problems
Javadoc
Console
Console
<terminated> main (49) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
apple
apple orange
apple orange cherry null
apple orange cherry pear
apple grape orange cherry pear null null null
apple grape orange cherry lemon pear null null
apple grape orange cherry lemon pear kiwi null
apple grape orange cherry pear kiwi null null
grape orange cherry pear kiwi null null null
orange cherry pear kiwi null null null null
orange cherry pear null null null null null
cherry pear null null
1번째 항목은 pear이다.
2배로 확장
2배로 확장
2배로 확장
삭제
1/2로 축소

수행시간

- peek() 메소드는 인덱스를 이용하여 배열 원소를 직접 접근하므로 O(1) 시간에 수행 가능
- 삽입이나 삭제는 새 항목을 중간에 삽입하거나 중간에 있는 항목을 삭제한 후에 뒤 따르는 항목들을 한 칸씩 앞이나 뒤로 이동해야 하므로 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)

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

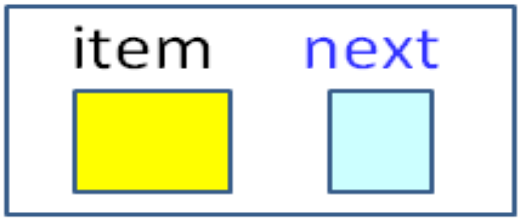


- 연결리스트에서는 삽입이나 삭제 시 항목들의 이동이 필요 없음
- 배열의 경우 최초에 배열의 크기를 예측하여 결정해야 하므로 대부분의 경우 배열에 빈 공간을 가지고 있으나, 연결리스트는 빈 공간이 존재하지 않음
- 연결리스트에서는 항목을 탐색하려면 항상 첫 노드부터 원하는 노드를 찾을 때까지 차례로 방문하는 순차탐색(Sequential Search)을 해야

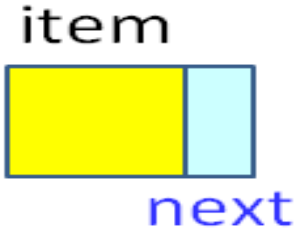
단순연결리스트의 노드를 위한 Node 클래스

```
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

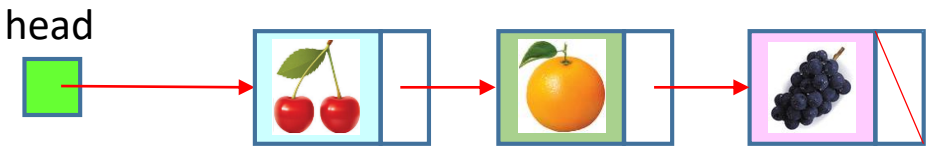
- 탐색은 인자로 주어지는 target을 찾을 때까지
  연결리스트의 노드들을 첫 노드부터 차례로 탐색

```
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  }
```

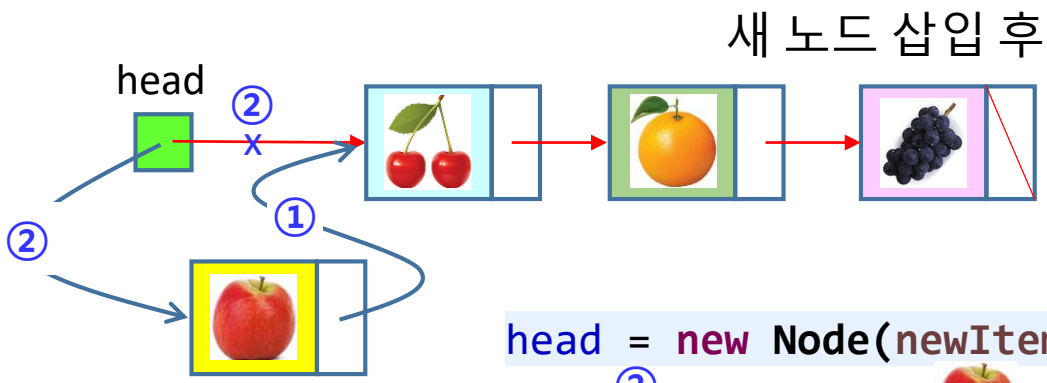
- Line 02: 지역변수 p가 연결리스트의 첫 노드를 참조
- Line 03: for-루프를 통해 target을 찾으면 line 04에서 target이 k번째 인덱스에 있음을 리턴
- 탐색에 실패하면 line 07에서 '-1'을 리턴

```
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

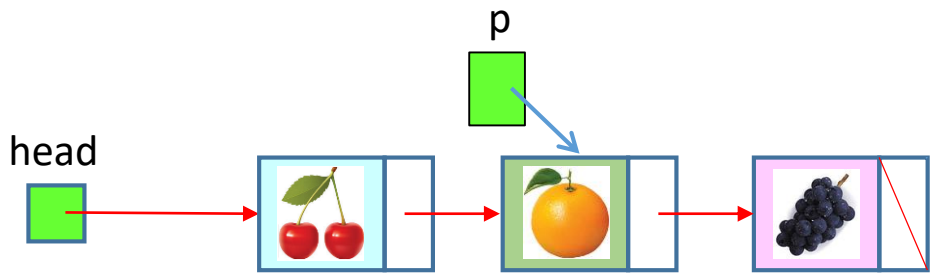


새 노드 삽입 후

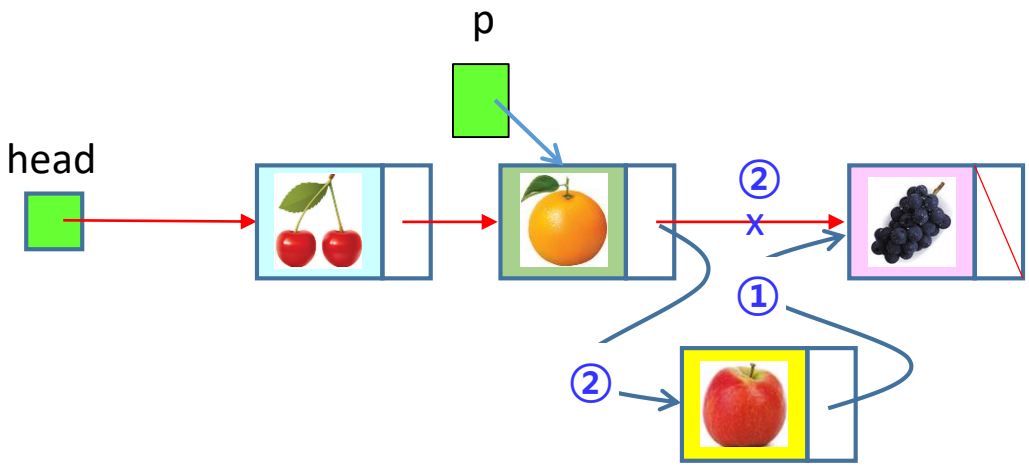
```
head = new Node(newItem, head);
       ②                 ①
```

```
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 노드를 삽입
- Line 02: 그림 참조(다음 슬라이드)



새 노드 삽입 전



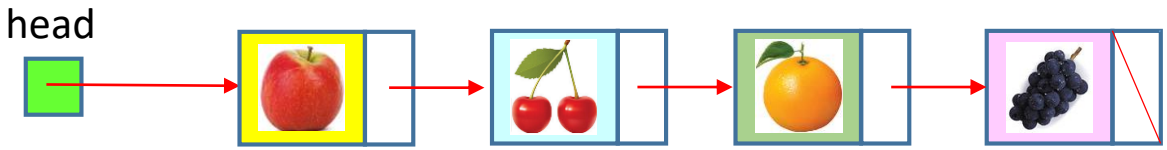
새 노드 삽입 후

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

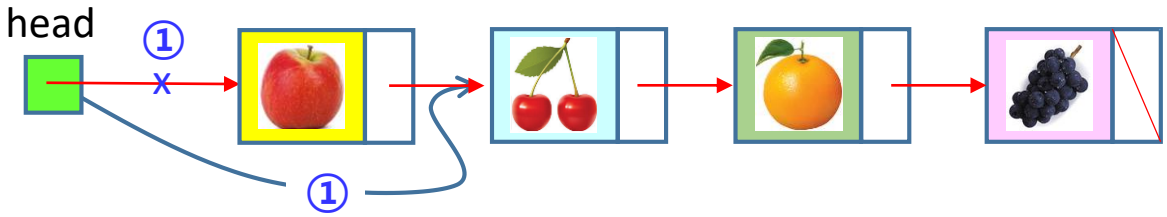
②　　①

```
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



삭제 전



삭제 후

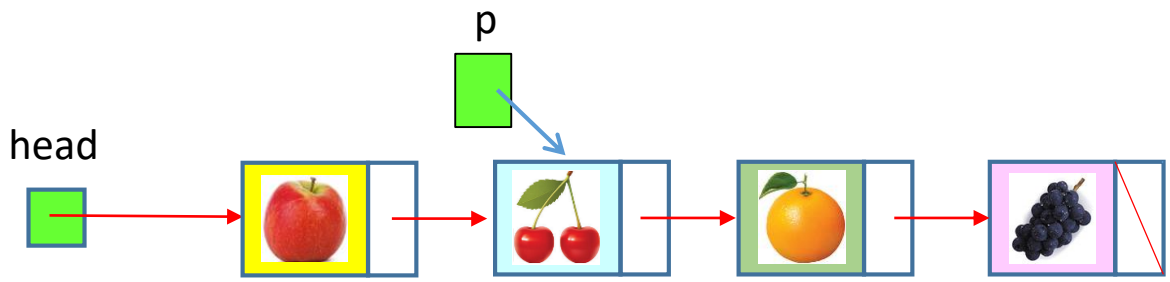
```
head = head.getNext();
```

①

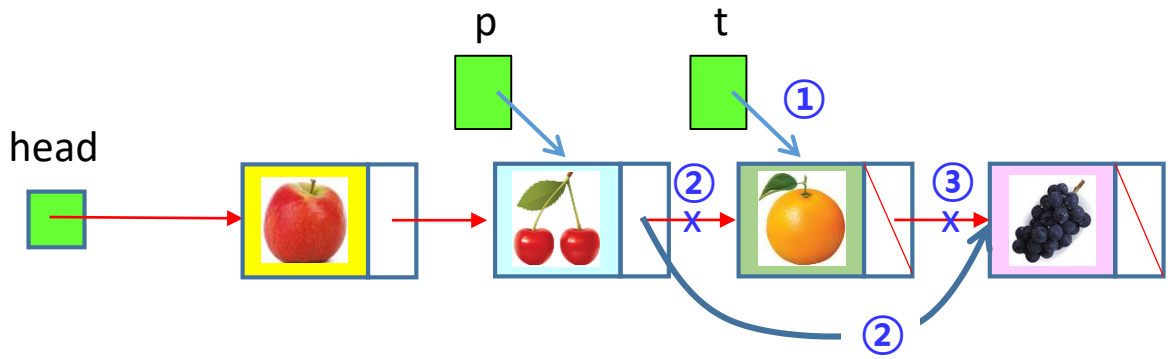
```
01 public void deleteAfter(Node p){ // p가 가리키는 노드의 다음 노드를 삭제
02     if (p == null) throw new NoSuchElementException();
03     Node t = p.getNext();
04     p.setNext(t.getNext());
05     t.setNext(null);
06     size--;
07 }
```

- deleteAfter() 메소드는 p가 가리키는 노드의 다음 노드를 삭제

```
① Node t = p.getNext();
② p.setNext(t.getNext());
③ t.setNext(null);
```



삭제 전

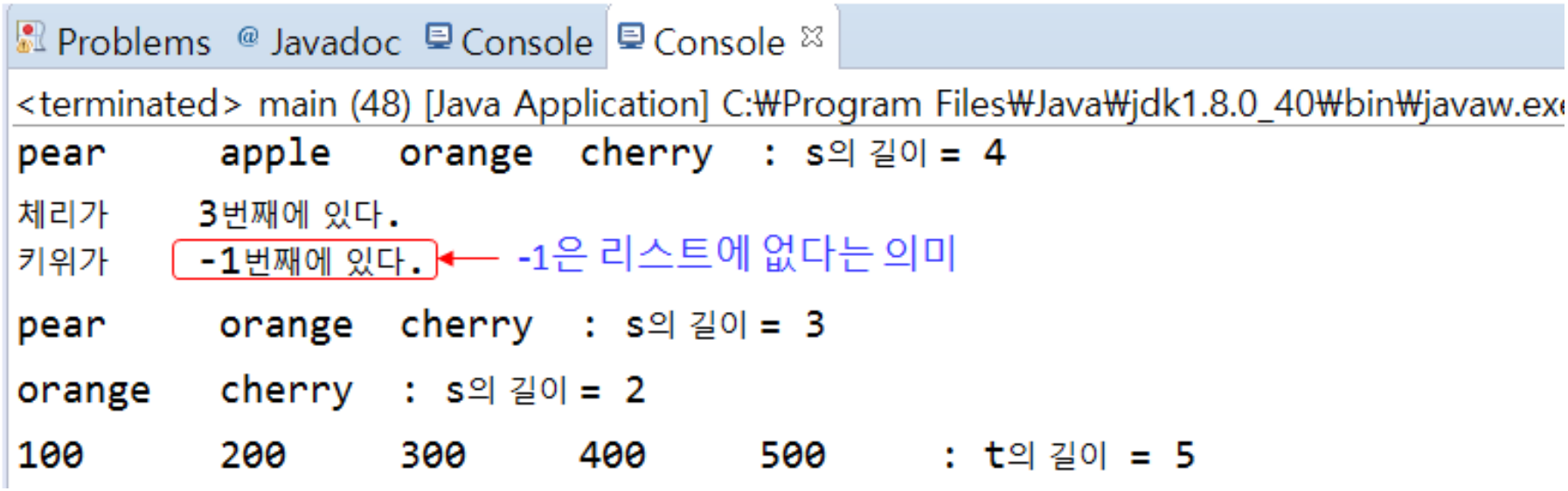


삭제 후

```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

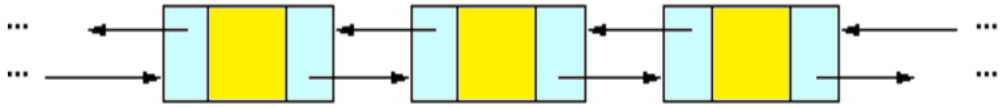
수행시간

- 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

2.3 이중연결리스트

- 이중연결리스트(Doubly Linked List)는 각 노드가 두 개의 레퍼런스를 가지고 각각 이전 노드와 다음 노드를 가리키는 연결리스트

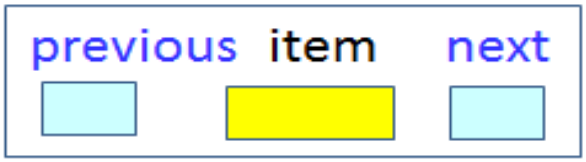


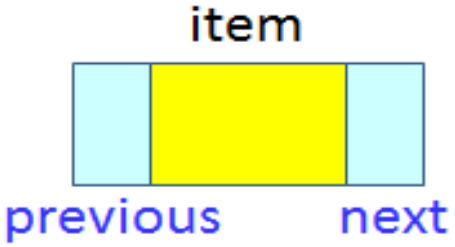
- 단순연결리스트는 삽입이나 삭제할 때 반드시 이전 노드를 가리키는 레퍼런스를 추가로 알아내야 하고, 역방향으로 노드들을 탐색할 수 없음
- 이중연결리스트는 단순연결리스트의 이러한 단점을 보완하나, 각 노드마다 추가로 한 개의 레퍼런스를 추가로 저장해야 한다는 단점을 가짐

이중연결리스트의 노드를 위한 DNode 클래스

```
01 public class DNode <E> {
02     private E     item;
03     private DNode previous;
04     private DNode next;
05     public DNode(E newItem, DNode p, DNode q){ // 노드 생성자
06         item     = newItem;
07         previous = p;
08         next     = q;
09     }
10     // get 메소드와 set 메소드
11     public E     getItem()     { return item;}
12     public DNode getPrevious() { return previous;}
13     public DNode getNext()     { return next;}
14     public void  setItem(E newItem)   { item     = newItem;}
15     public void  setPrevious(DNode p) { previous = p;}
16     public void  setNext(DNode q)     { next     = q;}
17 }
```

- Line 05~09: 이중연결리스트의 노드인 DNode 객체를 만드는 생성자이고, 다음과 같이 노드가 생성된다. 생성된 노드를 오른쪽 그림과 같이 간략히 표현



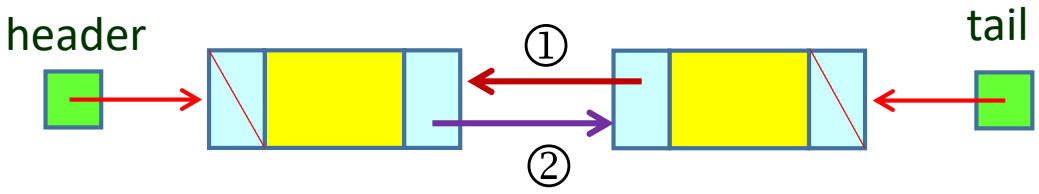


- Line 11~16: item과 next에 대한 get, set 메소드들

이중연결리스트를 위한 DList클래스

```
01 import java.util.NoSuchElementException;
02 public class DList <E> {
03       protected DNode head, tail;
04       protected int size;
05       public DList(){   //생성자
06            head = new DNode (null, null, null);
07            tail = new DNode (null, head, null);  // tail의 이전 노드를 head로 만든다.
08            head.setNext(tail);   //head의 다음 노드를 tail로 만든다.
09            size = 0;
10       }
   // 삽입, 삭제 연산을 위한 메소드 선언

   }
```

- head, tail, size를 가지는 DList 객체로, 생성자에서 head에 연결리스트의 첫 노드를 가리키는 레퍼런스를 저장
- tail: 연결리스트의 마지막 노드를 가리키는 레퍼런스를 저장
- head와 tail이 가리키는 노드는 생성자에서 아래와 같이 초기화. 이 두 노드들은 실제로 항목을 저장하지 않는 Dummy 노드

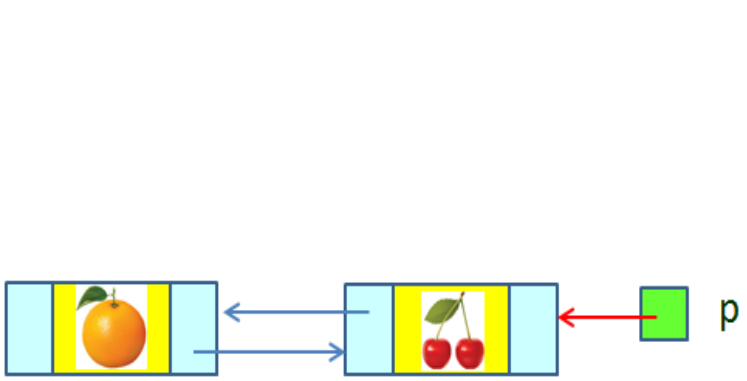



```
07  head = new DNode (null, null, null);
08  tail = new DNode (null, head, null);
09  head.setNext(tail);
```

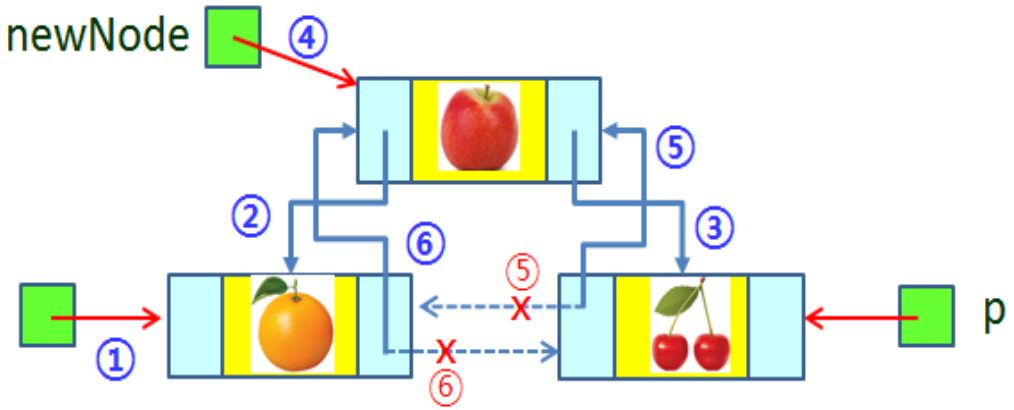
① ②

- insertBefore() 메소드로서 새 노드를 인자로 주어지는 노드 p 앞에 삽입한다.

```
01 public void insertBefore(DNode p, E newItem){  // p가 가리키는 노드 앞에 삽입
02   ① DNode t = p.getPrevious();
03     DNode newNode = new DNode(newItem, t, p);
04   ⑤ p.setPrevious(newNode);
05   ⑥ t.setNext(newNode);
06     size++;
07 }
```



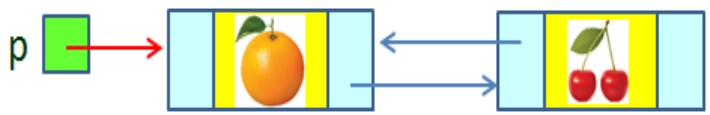
삽입 전



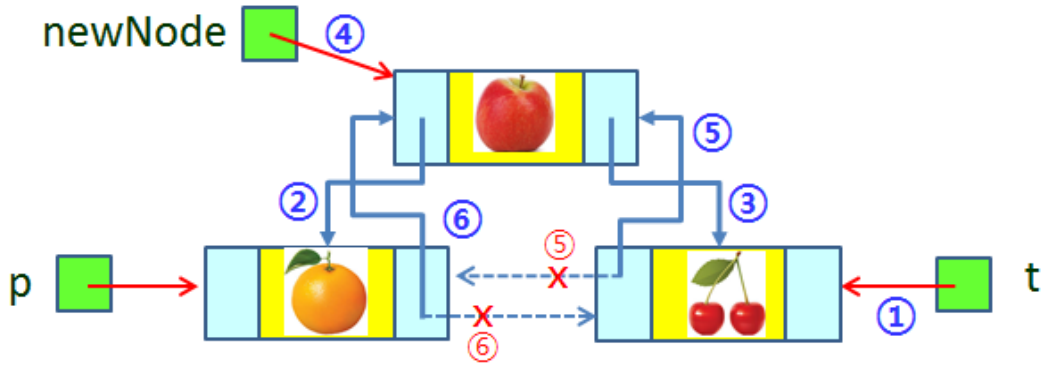
삽입 후

- insertAfter() 메소드: 인자로 주어지는 노드 p 다음에 새 노드를 삽입
- Line 02~05: 번호 순으로 레퍼런스들이 갱신

```
01 public void insertAfter(DNode p, E newItem){   // p가 가리키는 노드 뒤에 삽입
02   ① DNode t = p.getNext();
03     DNode newNode = new DNode(newItem, p, t);
                          ④
04   ⑤ t.setPrevious(newNode);
05   ⑥ p.setNext(newNode);
06     size++;
07 }
```



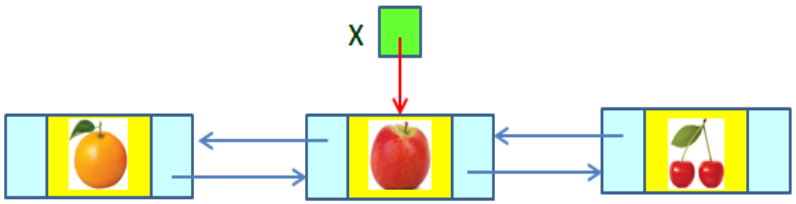
삽입 전



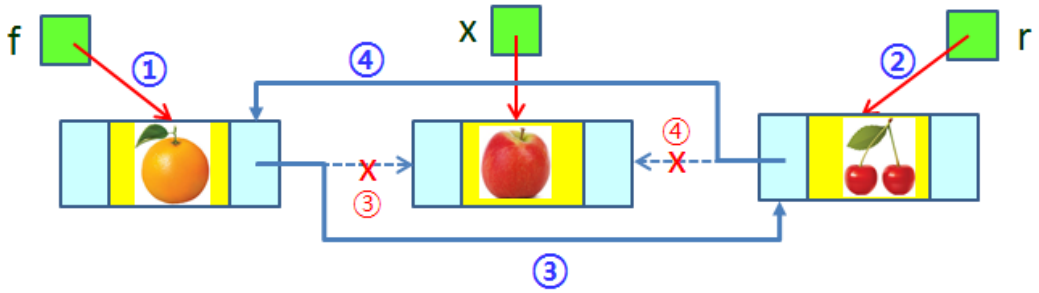
삽입 후

- delete() 메소드: 인자로 주어지는 노드 x를 삭제
- Line 03~08: 번호 순으로 레퍼런스를 갱신

```
01 public void delete(DNode x) {  // x가 가리키는 노드 삭제
02     if (x == null) throw new NoSuchElementException();
03   ① DNode f = x.getPrevious();
04   ② DNode r = x.getNext();
05   ③ f.setNext(r);
06   ④ r.setPrevious(f);
07     size--;
08 }
```



삭제 전

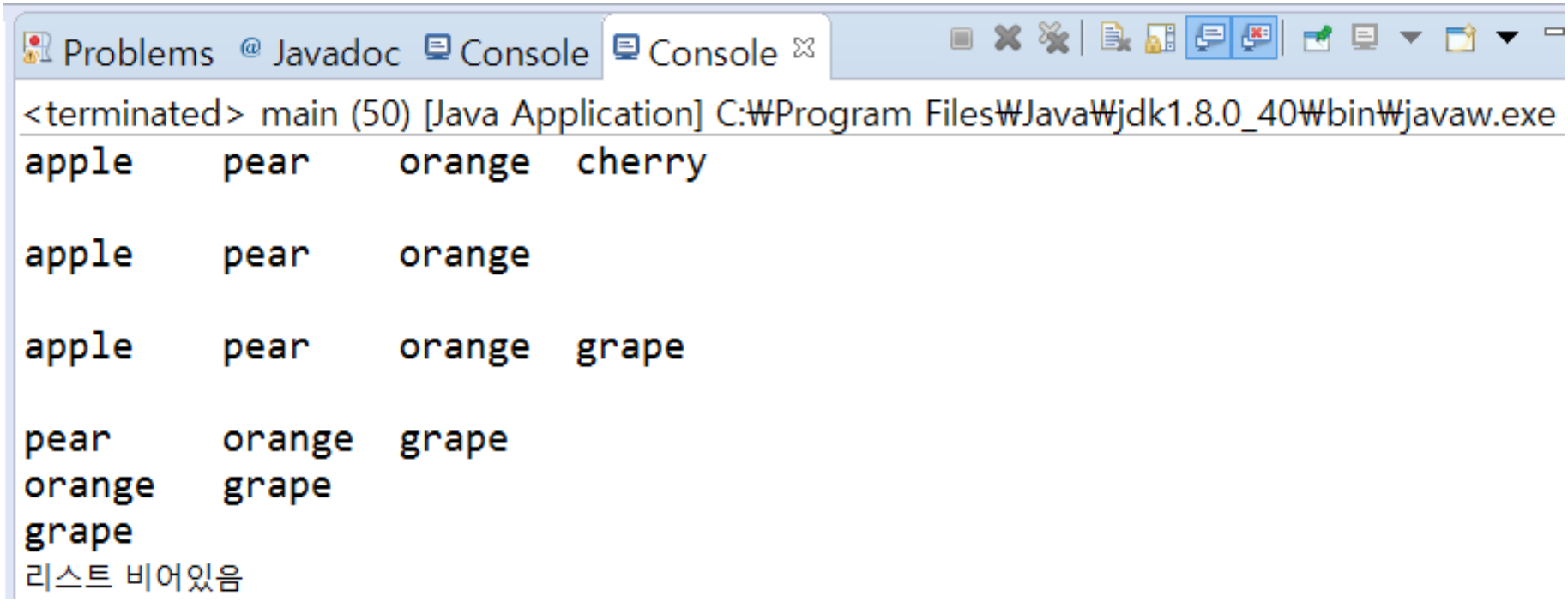


삭제 후

```
01 public class main {
02     public static void main(String[] args) {
03         DList<String> s = new DList<String>();  // 이중 연결 리스트 객체 s 생성
04 
05         s.insertAfter(s.head,"apple");
06         s.insertBefore(s.tail,"orange");
07         s.insertBefore(s.tail,"cherry");
08         s.insertAfter(s.head.getNext(),"pear");
09         s.print(); System.out.println();
10 
11         s.delete(s.tail.getPrevious());
12         s.print(); System.out.println();
13 
14         s.insertBefore(s.tail,"grape");
15         s.print(); System.out.println();
16         s.delete(s.head.getNext()); s.print();   s.delete(s.head.getNext());s.print();
17         s.delete(s.head.getNext()); s.print();   s.delete(s.head.getNext());s.print();
18     }
19 }
```

- 4개의 항목을 insertAfter()와 insertBefore()를 이용하여 리스트에 삽입한 후, 리스트를 출력
- delete() 메소드로 마지막 노드를 삭제하고, 리스트 출력
- 한 개의 새 항목을 삽입한 후 연속적으로 삭제 연산 수행

프로그램의 수행 결과

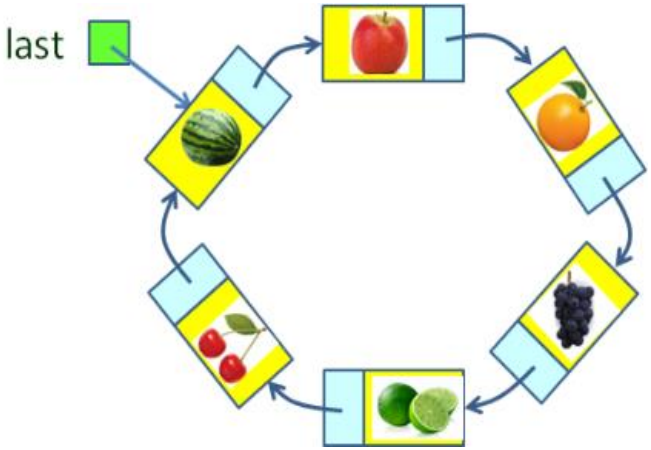

Problems
Javadoc
Console
Console
<terminated> main (50) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
apple pear orange cherry
apple pear orange
apple pear orange grape
pear orange grape
orange grape
grape
리스트 비어있음

수행시간

- 이중연결리스트에서 삽입이나 삭제 연산은 각각 상수 개의 레퍼런스만을 갱신하므로 O(1) 시간에 수행
- 탐색 연산: head 또는 tail로부터 노드들을 순차적으로 탐색해야 하므로 O(N) 시간 소요

2-4 원형연결리스트

- 원형연결리스트(Circular Linked List)는 마지막 노드가 첫 노드와 연결된 단순연결리스트
- 원형연결리스트에서는 마지막 노드의 레퍼런스가 저장된 last가 단순연결리스트의 head와 같은 역할



- 마지막 노드와 첫 노드를 O(1) 시간에 방문할 수 있는 장점
- 리스트가 empty가 아니면 어떤 노드도 null 레퍼런스를 가지고 있지 않으므로 프로그램에서 null 조건을 검사하지 않아도 되는 장점
- 원형연결리스트에서는 반대 방향으로 노드들을 방문하기 쉽지 않으며, 무한 루프가 발생할 수 있음에 유의할 필요

[원형연결리스트의 응용]

- 여러 사람이 차례로 돌아가며 하는 게임을 구현하는데 적합한 자료구조
- 많은 사용자들이 동시에 사용하는 컴퓨터에서 CPU 시간을 분할하여 작업들에 할당하는 운영체제에 사용
- 7장의 이항힙(Binomial Heap)이나 피보나치힙(Fibonacci Heap)과 같은 우선순위큐를 구현하는 데에도 원형연결리스트가 부분적으로 사용

환형연결리스트를 위한 CList 클래스

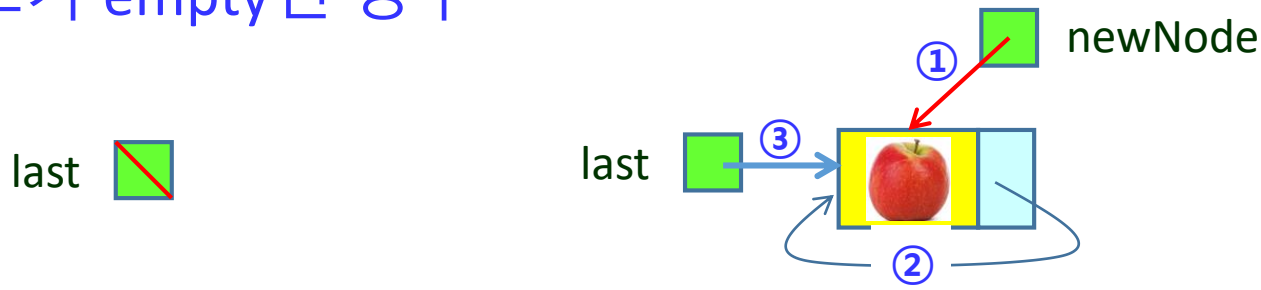
```
01 import java.util.NoSuchElementException;
02 public class CList <E> {
03     private Node last; // 리스트의 마지막 노드(항목)을 가리킨다.
04     private int  size; // 리스트의 항목(노드) 수
05     public CList() {   // 리스트 생성자
06         last = null;
07         size = 0;
08     }
   // 삽입, 삭제 연산을 위한 메소드 선언
   }
```

- CList 객체: 마지막 노드를 가리키는 last와 노드 수를 저장할 size를 가짐
- Node 클래스: 단순연결리스트의 Node 클래스와 동일

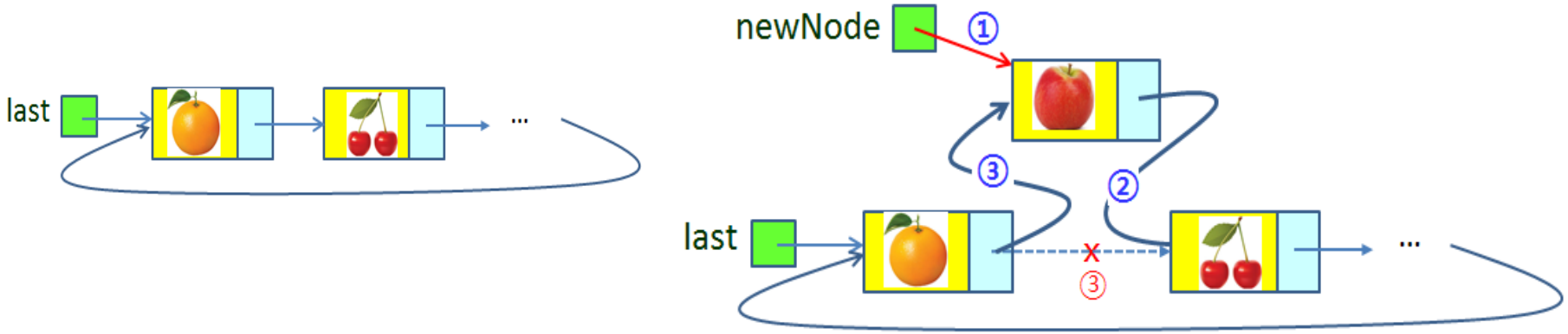
- insert() 메소드: 새 항목을 리스트의 첫 노드로 삽입한
- Line 02: 새 항목을 저장할 노드를 생성
- 리스트가 empty인 경우와 그렇지 않은 경우로 나누어 삽입

```
01 public void insert(E newItem) {  // last가 가리키는 노드의 다음에 새노드 삽입
02   ① Node newNode = new Node(newItem, null);  // 새 노드 생성
03      if  (last == null) {                    // 리스트가 empty일때
04        ② newNode.setNext(newNode);
05        ③ last = newNode;
06           }
07      else {
08        ② newNode.setNext(last.getNext()); // newNode의 다음 노드가 last가 가리키는 노드의 다음노드가 되도록
09        ③ last.setNext(newNode); // last가 가리키는 노드의 다음 노드가 newNode가 되도록
10      }
11      size++;
12 }
```

리스트가 empty인 경우



리스트가 empty가 아닌 경우



delete() 메소드: 리스트의 첫 노드를 삭제

Line 03: 첫 노드를 x가 가리키게 하고

Line 10: x를 리턴

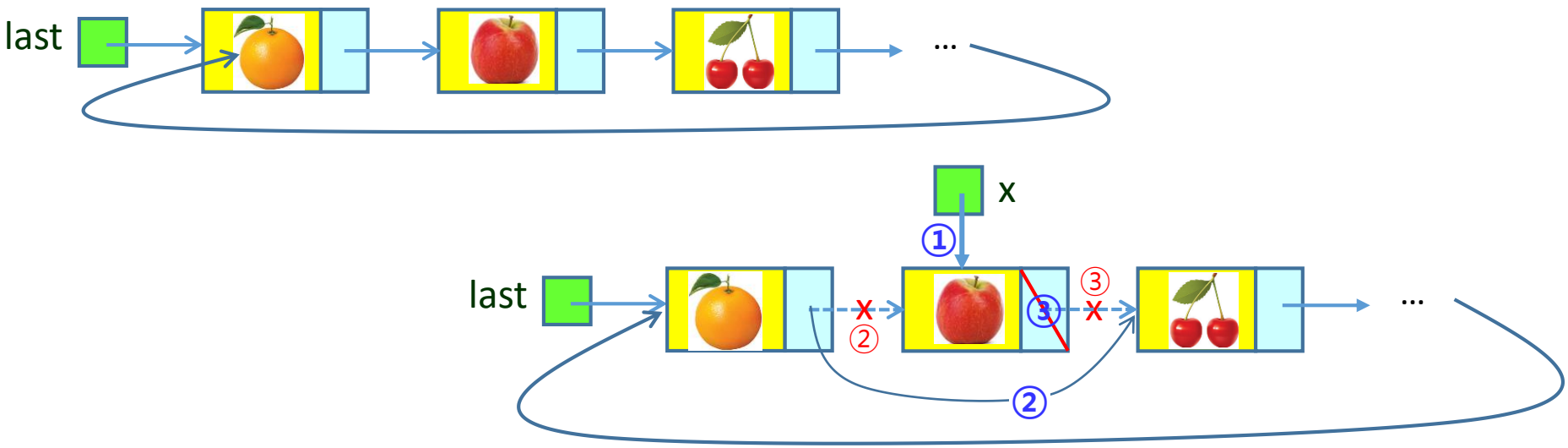
Line 04: 리스트에 노드가 1 개인 경우 last를 null로 만들며,

Line 05~08: 리스트에 노드가 2 개 이상인 경우로서 다음 슬라이드의 그림과 같이 x가 가리키는 노드를 리스트에서 분리

```
01 public Node delete() {  // last가 가리키는 노드의 다음 노드를 제거
02     if (isEmpty()) throw new NoSuchElementException();
03   ① Node x = last.getNext();        // x가 리스트의 첫 노드를 가리킴
04     if  (x == last) last = null;②  // 리스트에 1개의 노드만 있는 경우
05     else {
06       ② last.setNext(x.getNext()); // last가 가리키는 노드의 다음 노드가 x의 다음 노드가 되도록
07       ③ x.setNext(null);           // x의 next를 null로 만든다.
08     }
09     size--;
10     return x;
11 }
```

삭제 후 리스트 empty인 경우

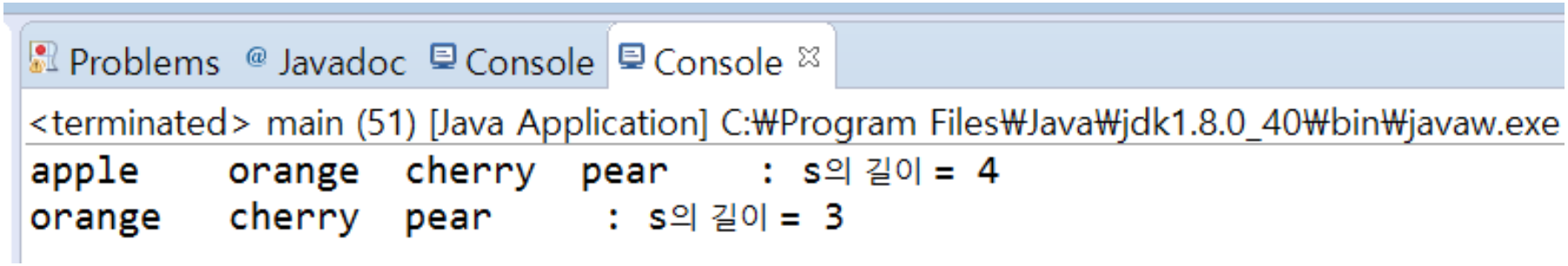
삭제 후 리스트 empty가 안되는 경우



```
01 public class main {
02     public static void main(String[] args) {
03         CList<String> s = new CList<String>();  // 연결 리스트 객체 s 생성
04 
05         s.insert("pear");     s.insert("cherry");
06         s.insert("orange");   s.insert("apple");
07         s.print();
08         System.out.print(": s의 길이 = "+s.size()+"\n");
09 
10         s.delete();
11         s.print();
12         System.out.print(": s의 길이 = "+s.size());System.out.println();
13     }
14 }
```

- 4개의 항목을 삽입 후, 리스트 와 리스트의 길이 출력
- 첫 항목을 삭제 후, 리스트와 그 길이 출력

프로그램의 수행 결과


Problems
Javadoc
Console
Console
<terminated> main (51) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
apple orange cherry pear : s의 길이 = 4
orange cherry pear : s의 길이 = 3

수행시간

- 원형연결리스트에서 삽입이나 삭제 연산 각각 상수 개의 레퍼런스를 갱신하므로 O(1) 시간에 수행
- 탐색 연산: last로부터 노드들을 순차적으로 탐색해야 하므로 O(N) 소요

요약

- 리스트: 일련의 동일한 타입의 항목들
- 배열: 동일한 타입의 원소들이 연속적인 메모리 공간에 할당되어 각 항목이 하나의 원소에 저장되는 기본적인 자료구조
- 단순연결리스트: 동적 메모리 할당을 이용해 리스트를 구현하는 가장 간단한 형태의 자료구조
- 배열로 구현된 리스트: 새 항목을 삽입 또는 삭제하는 경우 뒤 따르는 항목들이 1 칸씩 뒤나 앞으로 이동해야 하는 경우가 발생
- 단순연결리스트에서는 삽입이나 삭제 시 항목들을 이동시킬 필요 없음

- 단순연결리스트는 항목을 접근하기 위해서 순차탐색을 해야 하고, 삽입이나 삭제할 때에 반드시 이전 노드를 가리키는 레퍼런스를 알아야
- 이중연결리스트는 각 노드에 2 개의 레퍼런스를 가지며 각각 이전 노드와 다음 노드를 가리키는 방식의 연결리스트
- 원형연결리스트는 마지막 노드가 첫 노드와 연결된 단순연결리스트
- 원형연결리스트는 마지막 노드와 첫 노드를 O(1) 시간에 방문. 또한 리스트가 empty가 아닐 때, 어떤 노드도 null 레퍼런스를 갖지 않으므로 프로그램에서 null 조건을 검사하지 않아도 된다는 장점을 갖는다.

최악경우 수행시간 비교

자료구조	접근	탐색	삽입	삭제	비고
1 차원 배열	O(1)	O(N)	O(N)	O(N)	정렬된 배열에서 탐색은 O(logN)(부록 IV)
단순연결리스트	O(N)	O(N)	O(1)	O(1)	삽입될 노드의 이전 노드의 래퍼런스가 주어진 경우
이중연결리스트	O(N)	O(N)	O(1)	O(1)	
환형연결리스트	O(N)	O(N)	O(1)	O(1)	