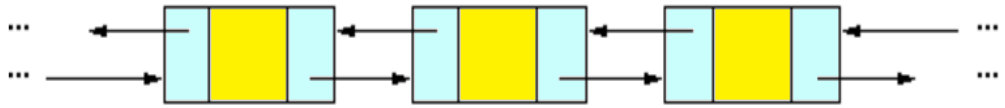
2.3 이중연결리스트

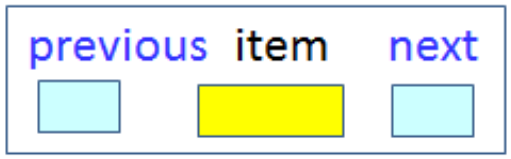
- **이중연결리스트(Doubly Linked List)는 각 노드가 두 개의 레퍼런스로 각각 이전 노드와 다음 노드를 가리키는 연결리스트**

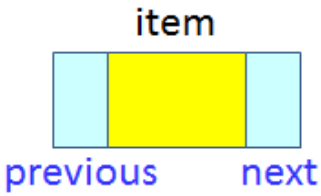


  - 단순연결리스트는 삽입이나 삭제할 때 반드시 이전 노드를 가리키는 레퍼런스를 추가로 알아내야 하고, 역방향으로 노드들을 탐색 불가
  - 이중연결리스트는 단순연결리스트의 이러한 단점을 보완하나, 각 노드마다 추가로 한 개의 레퍼런스를 추가로 저장해야 한다는 단점

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 }
```



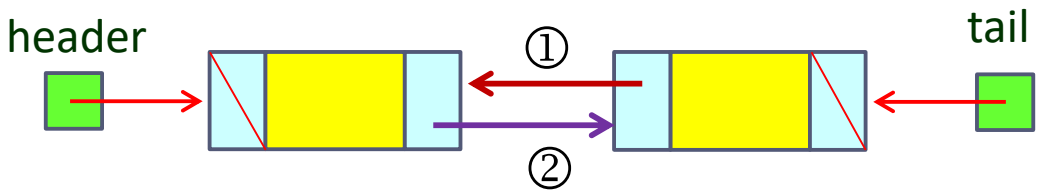


이중연결리스트의 노드인 DNode 객체를 만드는 생성자

이중연결리스트를 위한 DList 클래스

```
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       }
```

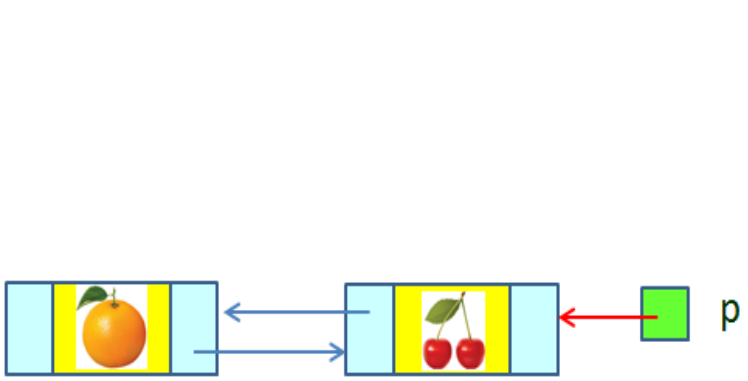
- head, tail, size를 가지는 DList 객체로, 생성자에서 head에 연결리스트의 첫 노드를 가리키는 레퍼런스를 저장
- tail: 연결리스트의 마지막 노드를 가리키는 레퍼런스를 저장
- head와 tail이 가리키는 노드는 생성자에서 아래와 같이 초기화.
- **이 두 노드들은 실제로 항목을 저장하지 않는 Dummy 노드**



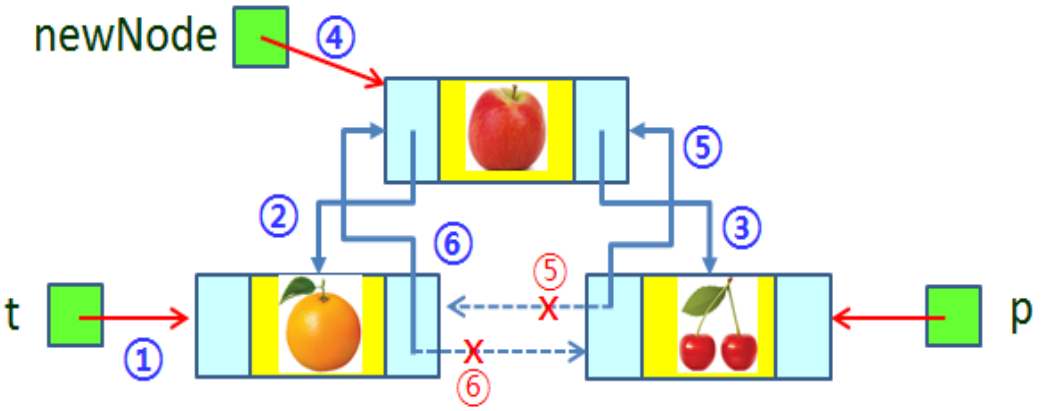
DList에서 특정 노드 앞에 삽입

```
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 }
```

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



삽입 전

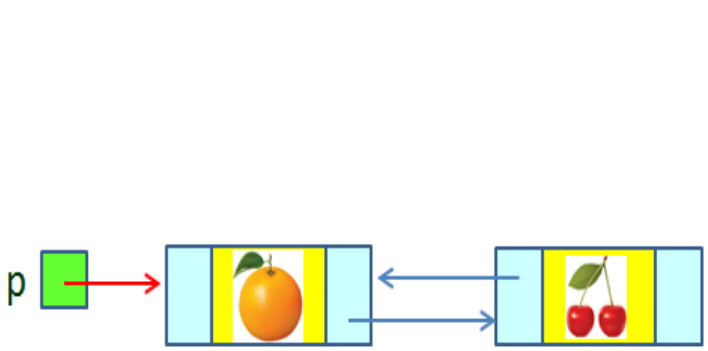


삽입 후

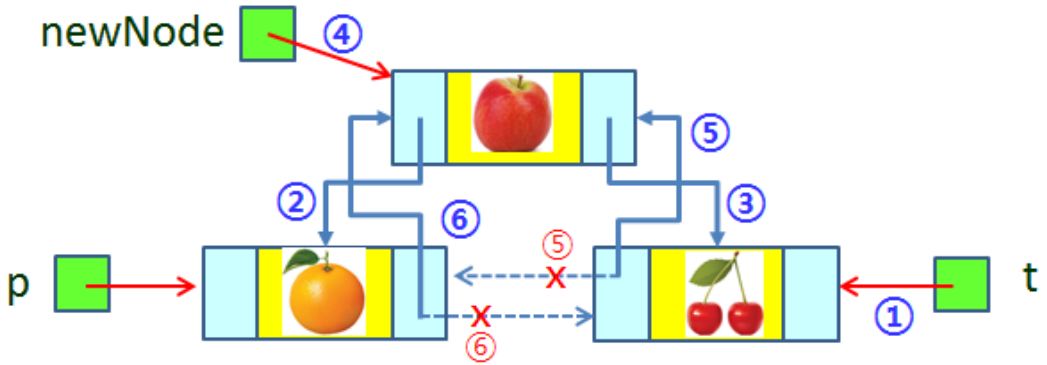
DList에서 특정 노드 뒤에 삽입

```
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                ② ③
04   ⑤ t.setPrevious(newNode);
05   ⑥ p.setNext(newNode);
06     size++;
07 }
```

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



삽입 전

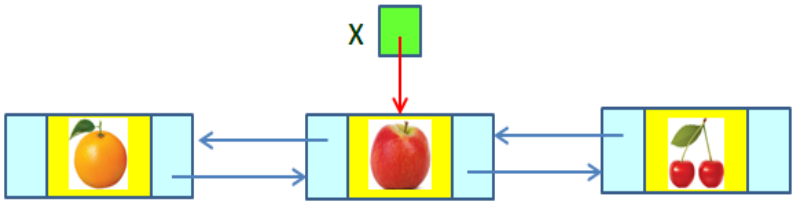


삽입 후

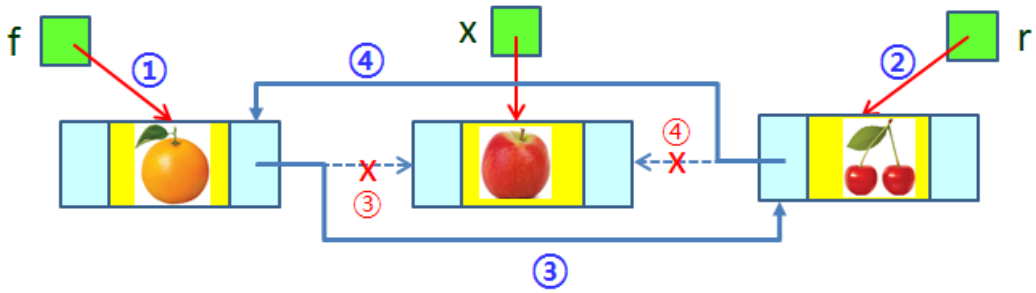
DList에서 특정 노드를 삭제

```
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 }
```

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



삭제 전

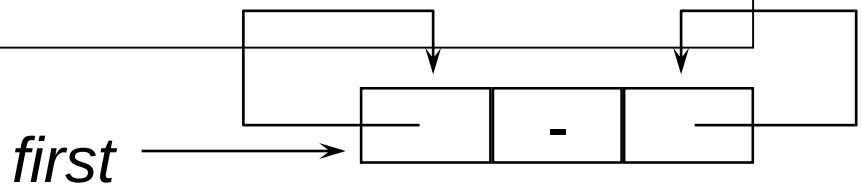


삭제 후

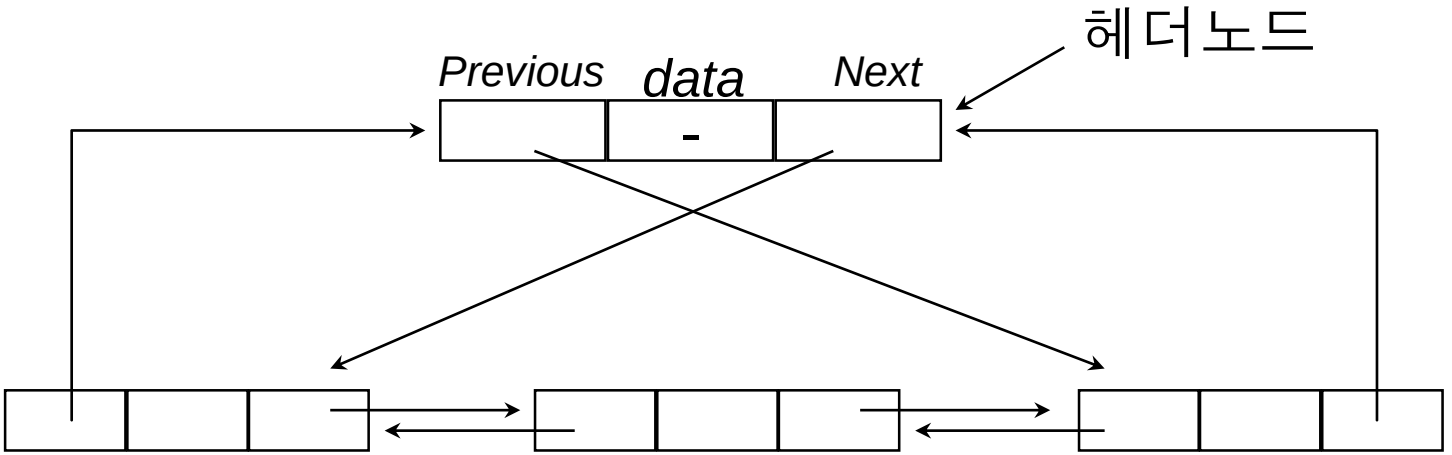
다른 방식의 dummy head를 가진 이중연결리스트

```
template <class T>
class DblListNode
{
private:
    T data;
    DblListNode *previous, *next;
  ….
}
```

```
template <class T>
class DblList
{
private:
    DblListNode<T> *first;
        // 헤더 노드를 가리킴
  …..
}
```



Dummy Header Node를 가진 공백 이중 연결 원형 리스트



C++에서의 삭제와 삽입

```
template <class T>
void DblList::delete(DblListNode<T> *x)
{
   if ( x == first )
      throw "Deletion of header node not permitted“;
   else
   {
        x→left→right = x→right;
        x→right→left = x→left;
        delete x;
   }
}
```

```
template <class T>
void DblList::insertAfter(DblListNode<T> *p, DblListNode<T> *x)
{// 노드 x의 오른편에 노드 p 삽입

     p→left = x;
     p→right = x→right;

     x→right→left = p;
     x→right = p;
}
```

```
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 }
```

Problems @ Javadoc Console Console

<terminated> main (50) [Java Application] C:₩P

```
apple    pear    orange  cherry

apple    pear    orange

apple    pear    orange  grape

pear     orange  grape
orange   grape
grape
리스트 비어있음
```

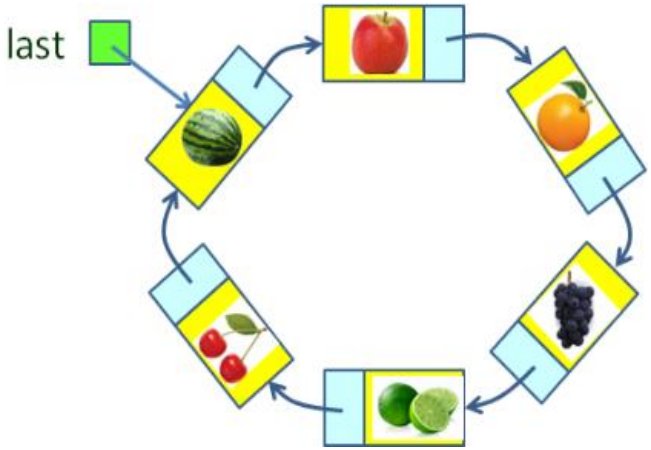
- 4개의 항목을 insertAfter()와 insertBefore()를 이용하여 리스트에 삽입한 후, 리스트를 출력
- delete() 메소드로 마지막 노드를 삭제하고, 리스트 출력
- 한 개의 새 항목을 삽입한 후 연속적으로 삭제 연산 수행

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

- **마지막 노드가 첫 노드와 연결된 단순연결리스트**
  - 마지막 노드의 레퍼런스가 저장된 last가 단순연결리스트의 head 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
    - 이중 원형 연결리스트로 해결할 수 있음



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) 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
   }
```

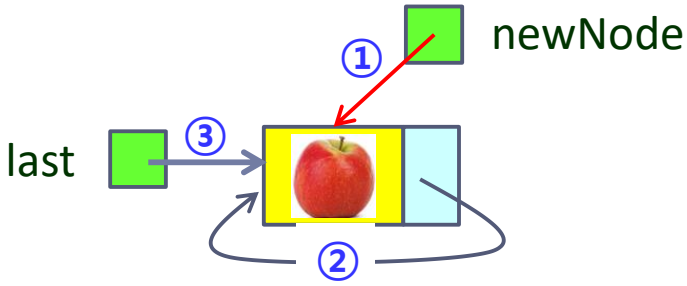
- CList 객체: 마지막 노드를 가리키는 last와 노드 수를 저장할 size
- Node 클래스: 단순연결리스트의 Node 클래스와 동일

CList에서의 제일 앞에 새 원소 삽입

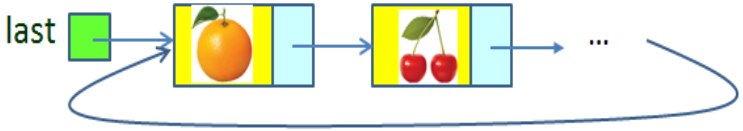
```
01 public void insert(E newItem) {  // last가 가리키는 노드의 다음에 새노드 삽입
02   ① Node newNode = new Node(newItem, null);  // 새 노드 생성
03     if  (last == null) {                      // 리스트가 empty일때
04       ② newNode.setNext(newNode);
05       ③ last = newNode;
06          }
07     else {
08       ② newNode.setNext(last.getNext());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

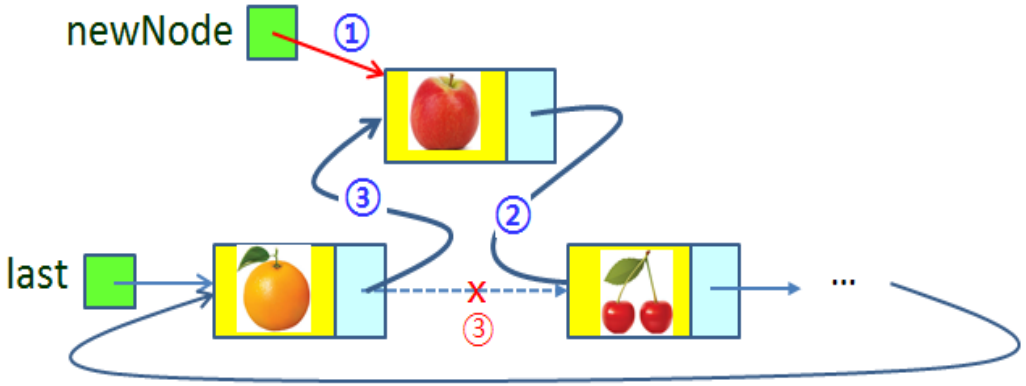
- insert() 메소드: 새 항목을 리스트의 첫 노드로 삽입한
- Line 02: 새 항목을 저장할 노드를 생성
- 리스트가 empty인 경우와 그렇지 않은 경우로 나누어 삽입

리스트가 empty인 경우



리스트가 empty가 아닌 경우





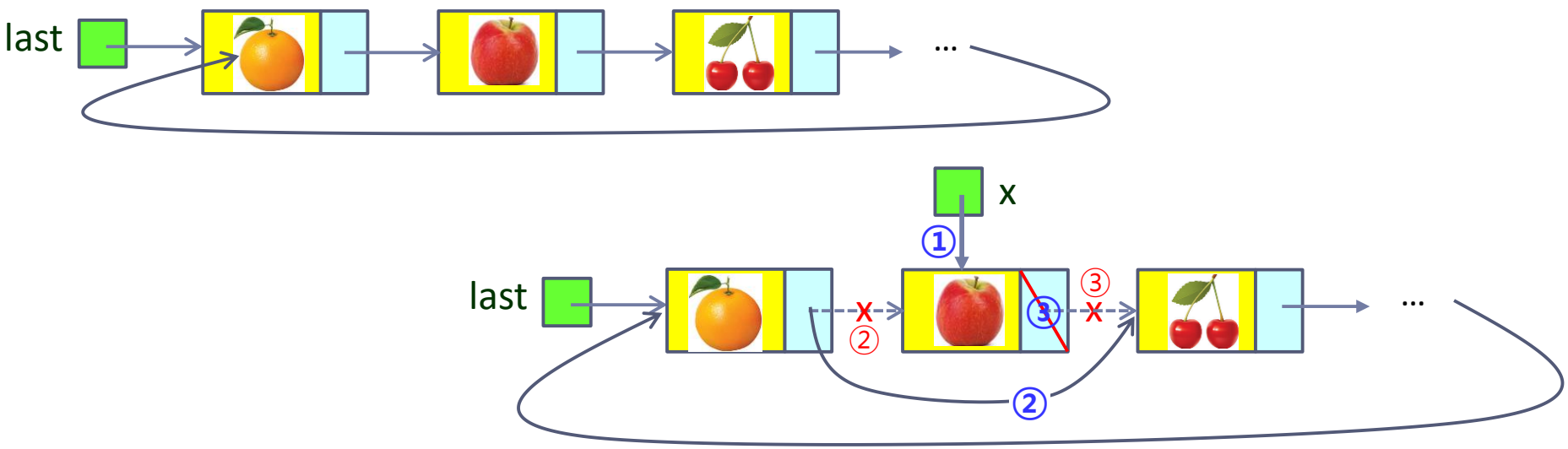
CList에서의 제일 앞의 원소 삭제

```
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 }
```

- 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

삭제 후 리스트 empty인 경우

삭제 후 리스트 empty가 안되는 경우



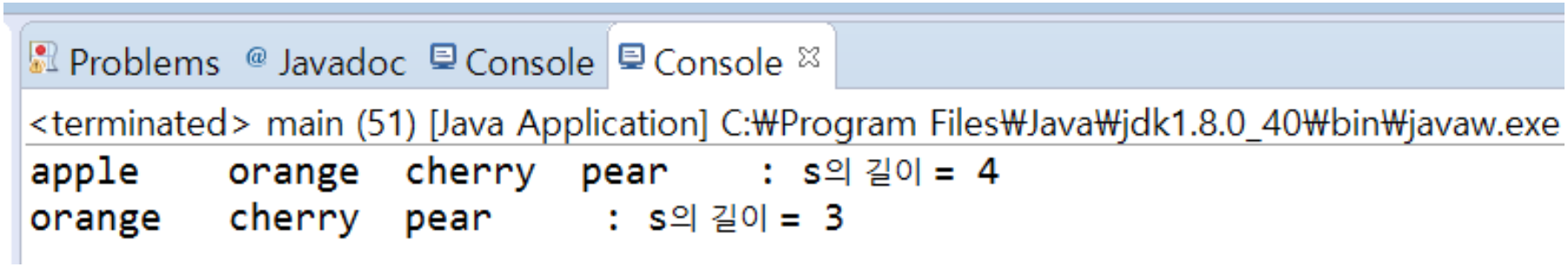
```java
public class main {
    public static void main(String[] args) {
        CList<String> s = new CList<String>();  // 연결 리스트 객체 s 생성

        s.insert("pear");     s.insert("cherry");
        s.insert("orange");   s.insert("apple");
        s.print();
        System.out.print(": s의 길이 = "+s.size()+"\n");

        s.delete();
        s.print();
        System.out.print(": s의 길이 = "+s.size());System.out.println();
    }
}
```

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

수행 결과

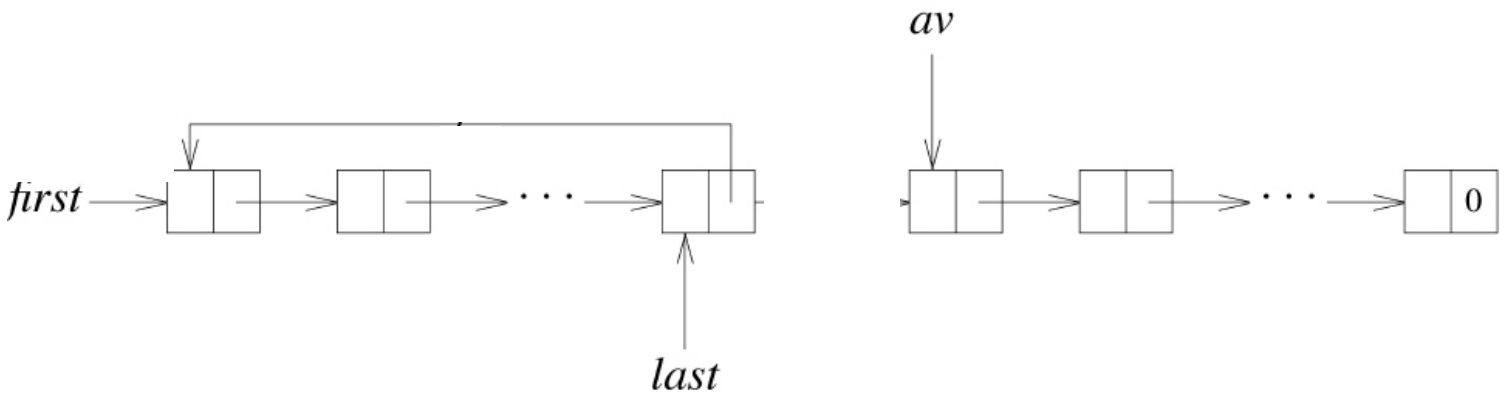


수행시간

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

가용 공간 리스트 (1)

- **자유(삭제된) 노드의 체인 (Available Space List)**
  - 반복적인 메모리 할당(new)과 해제(delete)의 수행은 비효율적
  - 해제된 메모리를 별도로 저장하였다가, 할당이 필요할 때 재 이용
    - 새 노드가 필요할 때는 자유 노드 체인 검사
    - 자유 노드 체인이 공백일 때만 new 호출
  - 파괴자의 실행 시간 O(1)로 감소 가능
    - 기존 Chain이나 CircularList의 소멸자는 O(n)만큼의 시간을 요구



가용 공간 리스트 (2)

- **노드 획득**

```
template <class T>
ChainNode<T>* CircularList<T>::getNode()
{//사용할 노드 생성
      ChainNode<T>* x;
      if(av)
     {
           x =av;
           av=av→link;
      }
      else
          x = new ChainNode<T>;
      return x;
}
```

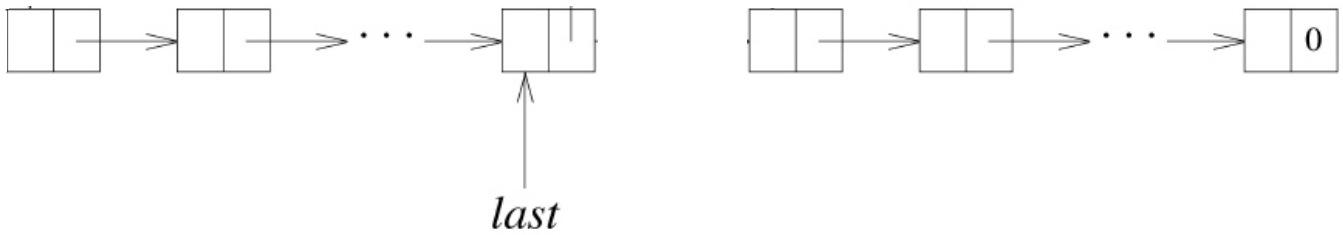
- **노드 반환**

```
template <class T>
void CircularList<T>::returnNode(ChainNode<T>*& x)
{//x가 가리키는 노드 반환
       x→link=av;
       av = x;
       x = 0;
}
```

가용 공간 리스트 (3)

- **원형 리스트의 삭제**

```
template <class T>
void CircularList<T>::~CricularList()
{//원형 리스트 삭제
     if(last){
          ChainNode<T>* first = last→link;
          last→link = av;   //마지막 노드가 av에 연결
          av = first;           //리스트의 첫 번째 노드가 av리스트의 첫 번째 노드가 됨
          last = 0;
     }
}
```



점선 화살표가 원형 리스트 삭제에 관련된 변경 표현

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 리스트는 새 항목을 삽입 또는 삭제하는 경우 뒤 따르는 항목들이 1 칸씩 뒤나 앞으로 이동해야 하는 경우로 복잡도가 O(N)이 되는 문제
  - 단순연결리스트에서는 삽입이나 삭제 시 항목들을 이동시킬 필요가 없어 O(1)으로 해결 가능 (삽입/삭제 위치가 정해져 있는 경우)

요약

- 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)	