자료구조: 2022년 1학기 [강의]

Array List, Linked List

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

Class “List”

리스트

- 원소들이 순서 있게 나열
  - Bag 과 Set 의 원소는 순서가 없다.
  - 순서의 의미는 상황이나 필요에 따라 달라진다.
    - 그러므로, 특정 값 (이를 테면, 학번 등) 에 의해 정렬되어 있을 수도 있다.
- 원소가 중복될 수 있다.
  - Set 에서는 원소가 중복될 수 없다.
- 예:
  - 학번 순으로 나열되어 있는 우리 학과 학생들

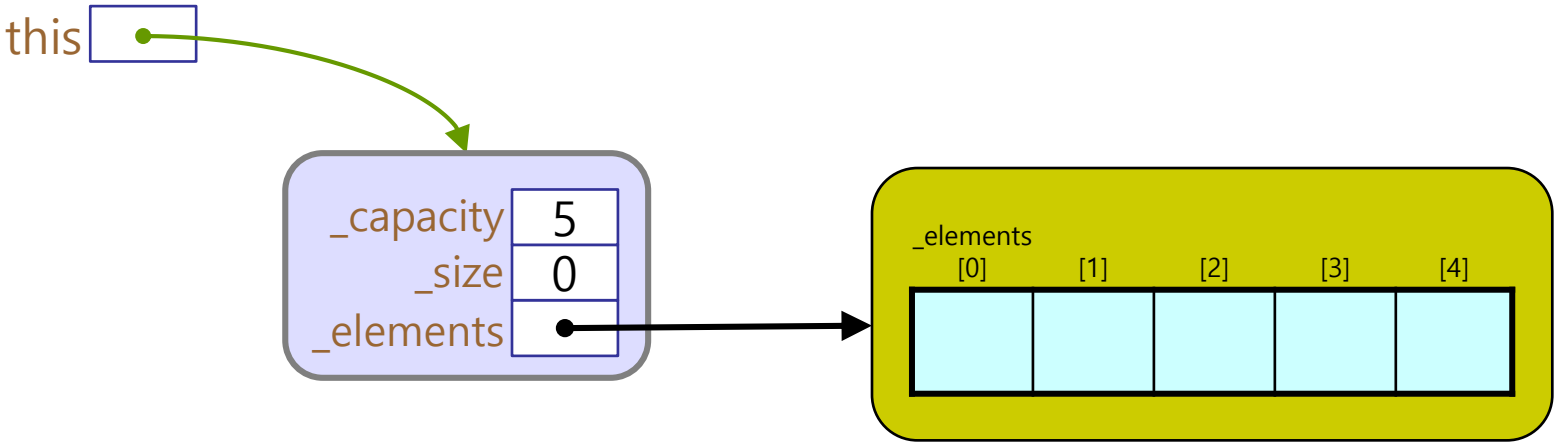
List 의 공개함수

- List<T> 객체 사용법
  - public List () { }
  - public boolean isEmpty () { }
  - public boolean isFull () { }
  - public boolean size () { }
  - public boolean doesContain (T anElement) { }
  - public T elementAt (int order) { }
  - public T first () { }
  - public T last () { }
  - public int orderOf (T anElement) { }
  - public boolean addTo (T anElement, int order) { }
  - public boolean addToFirst (T anElement) { }
  - public boolean addToLast (T anElement) { }
  - public boolean add (T anElement) { }
  - public T removeFrom (int order) { }
  - public T removeFirst () { }
  - public T removeLast () { }
  - public T removeAny () { }
  - public boolean remove (T anElement) { }
  - public boolean replaceAt (T anElement, int order) { }
  - public void clear () { }

Class "ArrayList"

List 로서의 "ArrayList"

- 추상적인 List 를 Array 를 이용하여 구현
  - ArrayList myList = new ArrayList() ;
    ..... // myList 를 이용하여 일을 한다



ArrayList 의 공개함수

- ArrayList<T> 객체 사용법:
  - public ArrayList () { }
  - public ArrayList (int givenCapacity) { }
  - public boolean isEmpty () { }
  - public boolean isFull () { }
  - public boolean size () { }
  - public T elementAt (int order) { }
  - public T first() { }
  - public T last () { }
  - public boolean doesContain (T anElement) { }
  - public int frequencyOf (T anElement) { }
  - public int orderOf (T anElement) { }
  - public boolean addTo (T anElement, int order) { }
  - public boolean addToFirst (T anElement) { }
  - public boolean addToLast (T anElement) { }
  - public boolean add (T anElement) { }
  - public T removeFrom (int order) { }
  - public T removeFirst () { }
  - public T removeLast () { }
  - public T removeAny () { }
  - public boolean remove(T anElement)
  - public boolean replaceAt (T anElement, int order) { }
  - public void clear () { }

Class "ArrayList"의 구현

ArrayList: 비공개 인스턴스 변수

```
public class ArrayList<T>
{
    // 비공개 상수
    private static final int DEFAULT_CAPACITY = 25 ;

    // 비공개 인스턴스 변수
    private int       _capacity ;
    private int       _size ;
    private T[]       _elements ;
```

ArrayList: Getter/Setter

```
public class ArrayList<T>
{
    // 비공개 상수
    private static final int DEFAULT_CAPACITY = 25 ;

    // 비공개 인스턴스 변수
    private int     _capacity ;
    private int     _size ;
    private T[]     _elements ;

    // Getters / Setters
    private int  capacity() {
        return  this._capacity ;
    }
    private void  setCapacity (int newCapacity) {
        this._capacity = newCapacity ;
    }
    public int  size() {
        return  this._size ;
    }
    private void  setSize (int newSize) {
        this._size = newSize ;
    }
    private T[]  elements () {
        return  this._elements ;
    }
    private void  setElements (T[] newElements) {
        this._elements = newElements ;
    }
```

ArrayList 의 비공개함수

- 여러 함수에서 공통으로 자주 사용

  - private boolean anElementDoesExistAt (int position) { }

  - private void makeRoomAt (int position) { }
  - private void removeGapAt (int position) { }

    - "position" 은 리스트에서의 순서 (order) 가 아니라, 리스트의 원소들이 저장되어 있는 배열에서의 위치를 의미한다.
    - 이번 Class "ArrayList" 의 구현에서의 position 은, order 와 동일한 의미가 된다.
    - 구현의 목적으로만 사용된다. (private method)

ArrayList 의 생성자

```
public class ArrayList < T>
{
    // 비공개 인스턴스 변수
    .....

    // 생성자
    public ArrayList ( )
    {
        this ( ArrayList.DEFAULT_CAPACITY ) ;
    }

    @SuppressWarnings ("Unchecked") ;
    public ArrayList ( int givenCapacity )
    {
        this.setCapacity (givenCapacity) ;
        this.setElements ( (T[ ]) new Object[this.capacity()] ) ;
        this.setSize (0) ;
    }
```

ArrayList 의 생성자

```
public class ArrayList < T>
{
    // 비공개 인스턴스 변수
    ......

    // 생성자
    public ArrayList ( )
    {
        this ( ArrayList.DEFAULT_CAPACITY ) ;
    }

    public ArrayList ( int givenCapacity )
    {
        @SuppressWarnings("Unchecked") ;
        this.setCapacity = givenCapacity ;
        this.setElements = ( (T[ ]) new Object[this.capacity()] ) ;
        this.setSize (0) ;
    }
```

여기서의 "this"는 객체 생성자

ArrayList: 상태 알아보기

```
public class ArrayList<T>
{
    // 비공개 인스턴스 변수
    ......


    // 상태 알아보기
    public boolean  isEmpty()
    {
        return (this.size() == 0) ;
    }

    public boolean  isFull()
    {
        return (this.size() == this.capacity()) ;
    }

    // public int  size()
    // {
    //      return  this._size ;
    // }
```

ArrayList: 내용 알아보기

```
public T  elementAt (int order)
{
    if ( this.anElementDoesExistAt (order) ) {
        int  position = order ;
        return  this.elements() [position] ;
    }
    else {
        return  null ;
    }
}


private boolean  anElementDoesExistAt (int order)
{
    return  ( (order >= 0) && (order < this.size()) ) ;
}
```

ArrayList: 내용 알아보기

```
// Version 1
public T  last ()
{
    return  this.elementAt (this.size()-1) ;

    // 다음 줄과 같이 하면 안되는 이유는?
    // return  this.elements()[this.size()-1] ; // 리스트가 empty 라면?
}


// version 2
public T  last ()
{
    if ( this.isEmpty() ) {
        return  null ;
    }
    else {
        return  this.elements()[this.size()-1] ;
}
```

ArrayList: 내용 알아보기

```
public int  orderOf (T anElement)
{
    // 원소 anElement 가 리스트 안에 존재하면 해당 위치를 돌려준다
    // 존재하지 않으면 -1 을 돌려준다
    for ( int order = 0 ; order < this.size() ; order++) {
        if ( this.elements()[order].equals(anElement) ) {
            return  order ;
        }
    }
    return  -1 ; // 주어진 원소 anElement가 리스트 안에 없다
}
```

ArrayList: 내용 알아보기

```
public boolean  doesContain (T anElement) // Version 1
{
    return  (this.orderOf (anElement) != -1) ;
}


public boolean  doesContain (T anElement) // Version 2
{
    // 원소 anElement 가 리스트 안에 존재하면 해당 위치를 돌려준다
    // 존재하지 않으면 -1을 돌려준다
    for ( int order = 0 ; order < this.size() ; order++) {
        if ( this.elements()[order].equals(anElement) ) {
            return  true ;
        }
    }
    return  false ; // 주어진 원소 anElement가 리스트 안에 없다
}
```

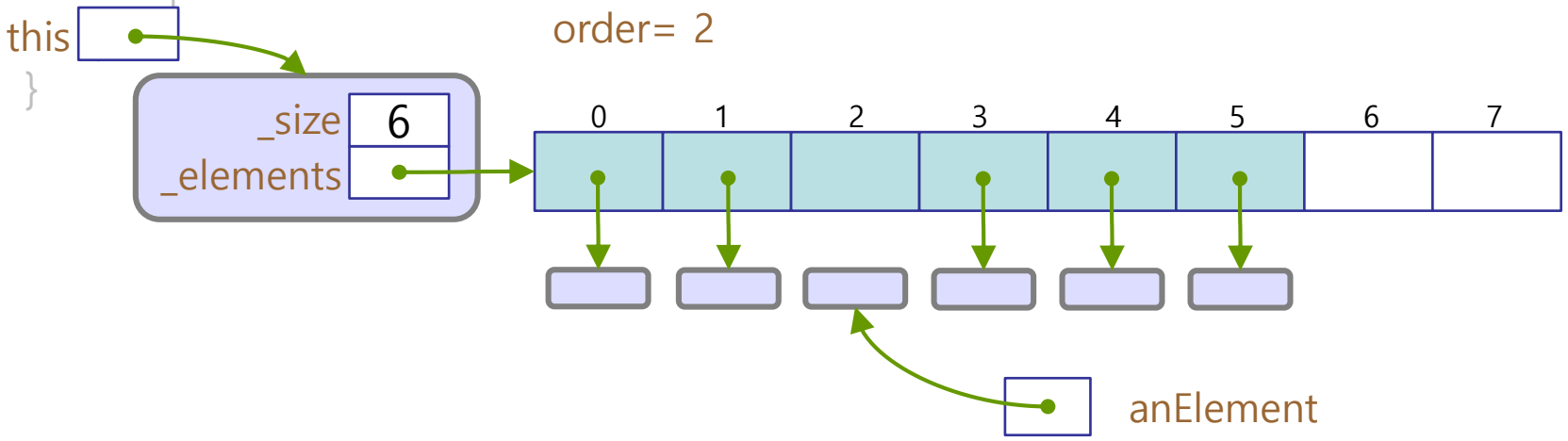
ArrayList: 원소 삽입하기

```
// 원소 삽입
public boolean  addTo (T anElement, int order) {
    if ( this.isFull() ) {
        return  false ;
    }
    else {
        if ( (order >= 0) && (order <= this.size() ) ) {
            this.makeRoomAt (order) ;
            this.elements()[order] = anElement ;
            this.setSize (this.size()+1) ;
            return  true ;
        }
        else {
            return  false ; // 잘못된 삽입 위치
        }
    }
}
```

"<" 가 아닌 "<="을 사용한 이유는?

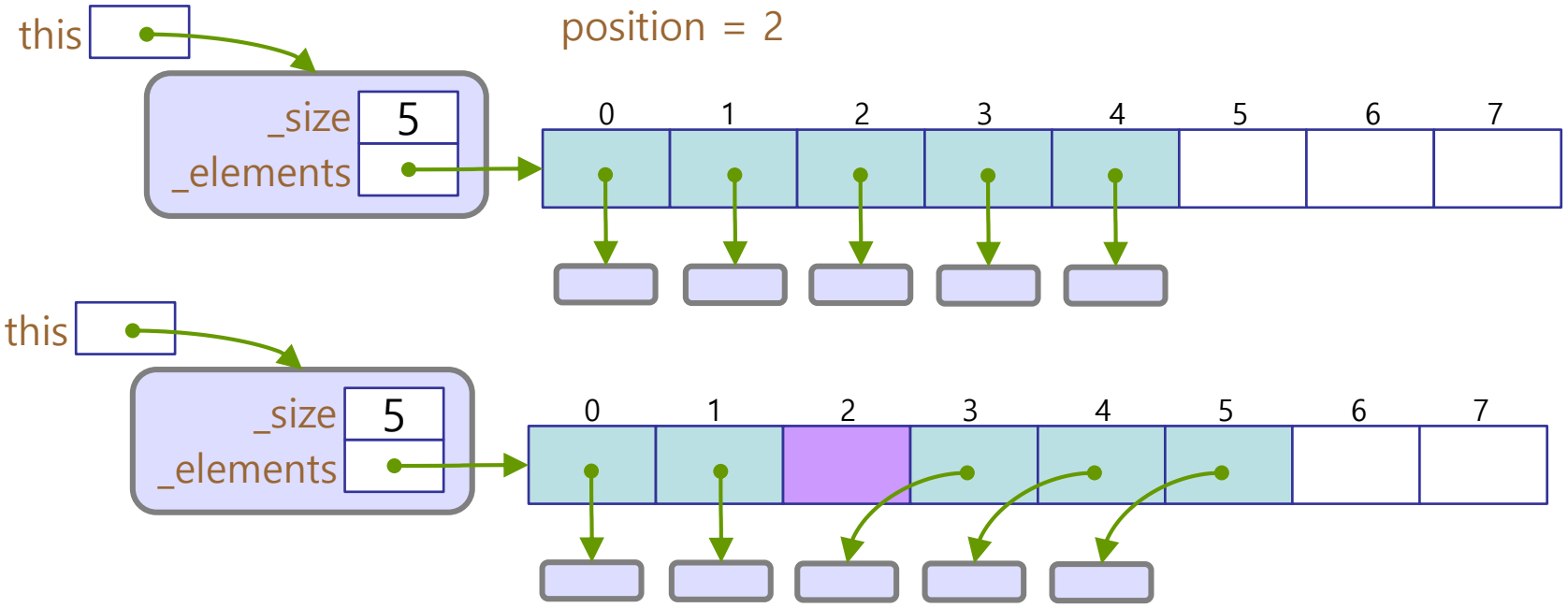
ArrayList: 원소 삽입하기

```
public boolean  addTo (T anElement, int order)
{
    ...
    else {
        if ( (order >= 0) && (order <= this.size() ) ) {
            this.makeRoomAt (order) ;
            this.elements()[order] = anElement ;
            this.setSize (this.size()+1) ;
            return  true ;
        }
        else {
            return  false ; // 잘못된 삽입 위치
        }
}
```



ArrayList: 삽입 공간 만들기

```
private void  makeRoomAt ( int position )
{
    for ( int i = this.size() ; i > position ; i-- ) {
        this.elements()[i] = this.elements()[i-1] ;
    }
}
```



ArrayList: addToFirst()

```
public boolean  addToFirst (T anElement)  // Version 1
{
    return  this.addTo ( anElement, 0 ) ;
}

public boolean  addToFirst (T anElement)  // Version 2
{
    if ( this.isFull() ) {
        return  false ;
    }
    else {
        this.makeRoomAt(0) ;
        this.elements()[0] = anElement ;
        this.setSize (this.size()+1) ;
        return  true ;
}
```

ArrayList: addToLast, add()

```
public boolean  addToLast (T anElement)
{
    return  this.addTo ( anElement, this.size() ) ;
}


public boolean  add (T anElement)
{
    return  this.addToLast (anElement) ; // 가장 효과적인 곳에 삽입
}
```

ArrayList: 원소 삭제하기 (Version 1)

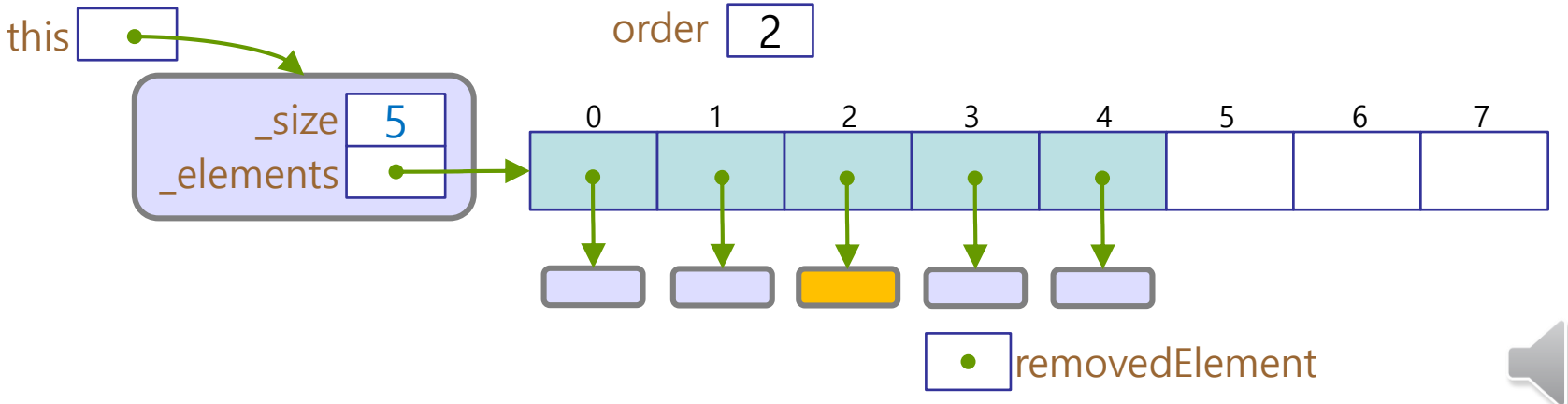
```
// Version 1:
public T  removeFrom ( int order )
{
    // 주어진 위치 order 에 원소가 없으면 null 을 return 한다
    // 원소가 있으면 리스트에서 제거하여 return 한다.
    if ( this.isEmpty() ) { // 이 검사가 꼭 필요한가?
        return  null ;
    }
    else {
        T  removedElement = null ;
        if ( this.anElementDoesExistAt (order) ) {
            removedElement = this.elements()[order] ;
            this.removeGapAt (order) ;
            this.setSize (this.size()-1) ;
        }
        return  removedElement ;
    }
}
```

ArrayList: 원소 삭제하기 (Version 2)

```
// Version 2:
public T  removeFrom ( int order)
{
    // 주어진 순서 order 에 원소가 없으면 null 을 return 한다
    // 원소가 있으면 리스트에서 제거하여 return 한다.
    T  removedElement = null ;
    if ( this.anElementDoesExistAt (order) ) {
        // 리스트가 empty 이면 이 조건은 false 를 얻는다.
        // 따라서, 별도의 empty 검사를 하지 않아도 안전하다.
        removedElement = this.elements()[order] ;
        this.removeGapAt (order) ;
        this.setSize (this.size()-1) ;
    }
    return  removedElement ;
}
```

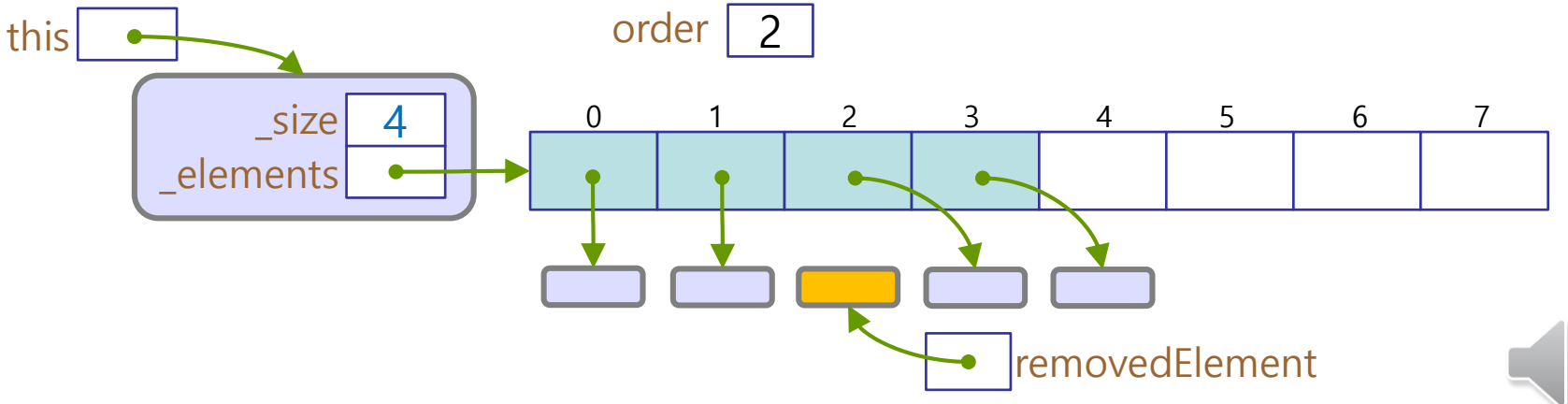
ArrayList: 원소 삭제하기 [전]

```
public T  removeFrom ( int order)
{
      T  removedElement = null ;
      if (  this.anElementDoesExistAt (order) ) {
            T  removedElement = this.elements()[order] ;
            this.removeGapAt (order) ;
            this.setSize (this.size()-1) ;
      }
      return  removedElement ;
}
```



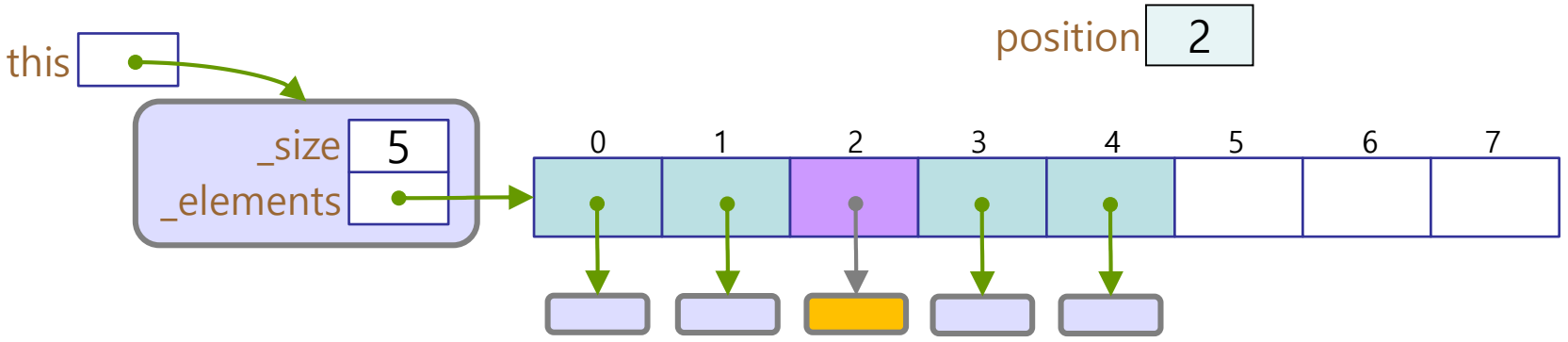
ArrayList: 원소 삭제하기 [후]

```
public T  removeFrom ( int order)
{
      T  removedElement = null ;
      if (  this.anElementDoesExistAt (order) ) {
            T  removedElement = this.elements()[order] ;
            this.removeGapAt (order) ;
            this.setSize (this.size()-1) ;
      }
      return  removedElement ;
}
```



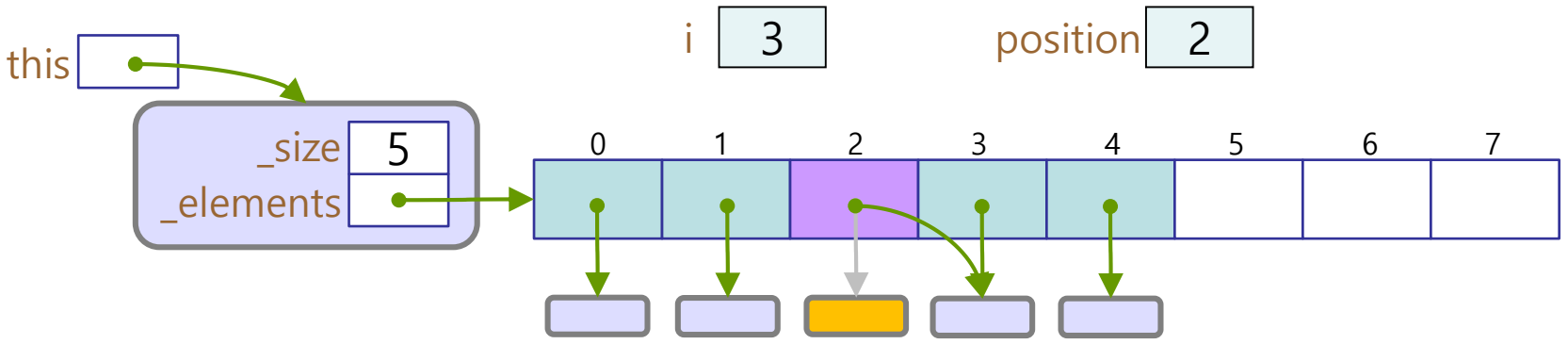
ArrayList: 삭제된 빈 공간 제거하기[1]

```
private void  removeGapAt ( int position )
{
    // 리스트는 empty 가 아님: 언제나 (this.size() > 0)
    // position 은 valid: 언제나 (0 <= position < this.size())

    for ( int i = position+1 ; i < this.size() ; i++ ) {
        this.elements() [i-1] = this.elements() [i] ;
    }
    this.elements() [this.size()-1] = null ;
}
```



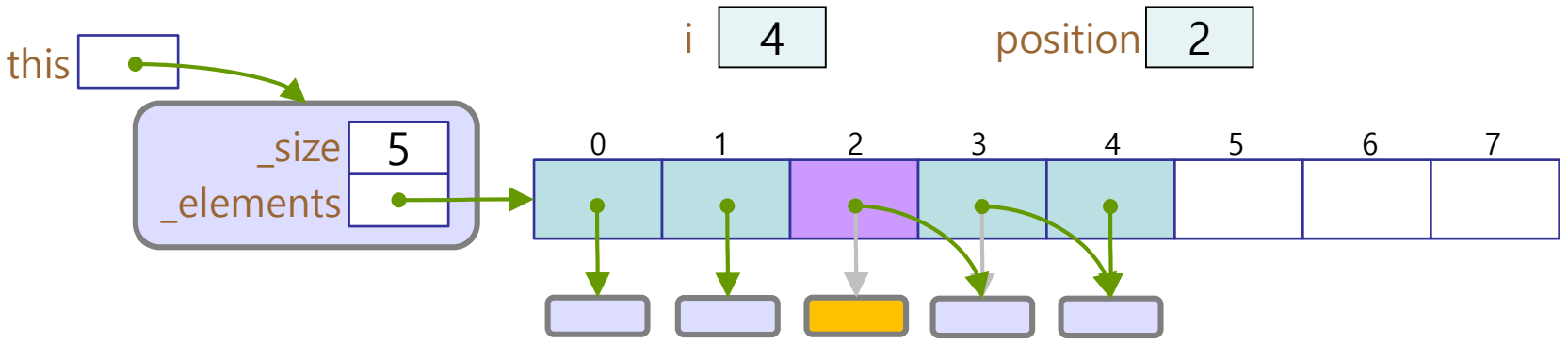
ArrayList: 삭제된 빈 공간 제거하기[2]

```
private void  removeGapAt ( int position )
{
    // 리스트는 empty 가 아님: 언제나 (this.size() > 0)
    // position 은 valid: 언제나 (0 <= position < this.size())

    for ( int i = position+1 ; i < this.size() ; i++ ) {
        this.elements() [i-1] = this.elements() [i] ;
    }
    this.elements() [this.size()-1] = null ;
}
```



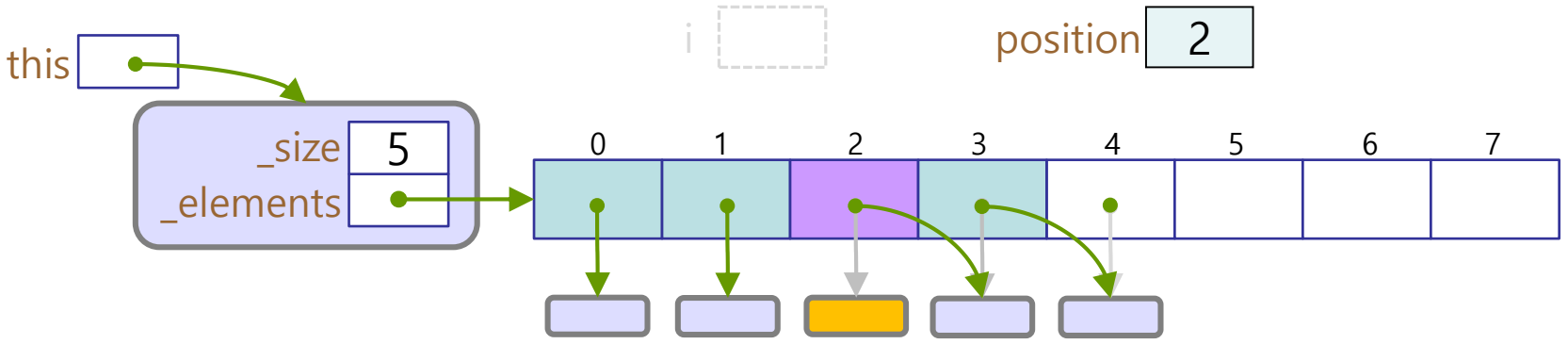
ArrayList: 삭제된 빈 공간 제거하기[3]

```
private void  removeGapAt ( int position )
{
    // 리스트는 empty 가 아님: 언제나 (this.size() > 0)
    // position 은 valid: 언제나 (0 <= position < this.size())

    for ( int i = position+1 ; i < this.size() ; i++ ) {
        this.elements() [i-1] = this.elements() [i] ;
    }
    this.elements() [this.size()-1] = null ;
}
```



ArrayList: 삭제된 빈 공간 제거하기[4]

```
private void  removeGapAt ( int position )
{
    // 리스트는 empty 가 아님: 언제나 (this.size() > 0)
    // position 은 valid: 언제나 (0 <= position < this.size())

    for ( int i = position+1 ; i < this.size() ; i++ ) {
        this.elements() [i-1] = this.elements() [i] ;
    }
    this.elements() [this.size()-1] = null ;
}
```



ArrayList: removeFirst(),removeLast(), removeAny()

```
public T  removeFirst ()
{
    return  removeFrom (0) ;
}


public T  removeLast ()
{
    return  removeFrom (this.size()-1) ;
}


public T  removeAny ()
{
    return  removeLast () ;
}
```

ArrayList: remove()

```
public boolean  remove (T anElement)
{
    int  orderOfRemove = this.orderOf(anElement) ;
    if ( orderOfRemove < 0 ) {
        return  false ;
    }
    else {
        this.removeGapAt (orderOfRemove);
        this.setSize (this.size()-1) ;
        return  true ;
    }
}
```

ArrayList: 원소 대체하기

```
// Version 1:
public boolean  replaceAt (T anElement, int order)
{
     if ( this.isEmpty() ) {
          return  false ;
     }
     else {
          if ( this.anElementDoesExistAt (order) ) {
               this.elements()[order] = anElement ;
               return  true ;
          }
          else {
               return  false ;
          }
     }
}

// Version 2:
public boolean  replaceAt (T anElement, int order)
{
     if ( this.anElementDoesExistAt (order) ) {
          this.elements()[order] = anElement ;
          return  true ;
     }
     else {
          return  false ;
     }
}
```

ArrayList: 리스트 비우기

```
public void   clear ()
{
    for ( int   i = 0 ; i < this.size() ; i++ ) {
        this.elements()[i] = null ;
    }
    this.setSize (0) ;
}
```

Class “LinkedList”

List 로서의 "LinkedList"

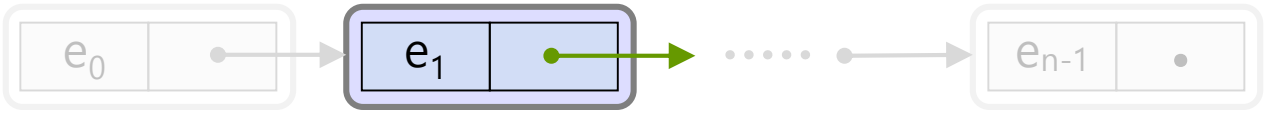
- 추상적인 List를 연결 체인을 이용하여 구현
  - LinkedList myList = new LinkedList() ;
    ..... // myList 를 이용하여 일을 한다

LinkedList<T> 의 공개함수

- LinkedList<T> 객체 사용법
  - public LinkedList () { }
  - public boolean isEmpty () { }
  - public boolean isFull () { }
  - public boolean size () { }
  - public boolean doesContain (T anElement) { }
  - public T elementAt (int order) { }
  - public T firstElement () { }
  - public T lastElement () { }
  - public int orderOf (T anElement) { }
  - public boolean addTo (T anElement, int order) { }
  - public boolean addToFirst (T anElement) { }
  - public boolean addToLast (T anElement) { }
  - public boolean add (T anElement) { }
  - public T removeFrom (int order) { }
  - public T removeFirst () { }
  - public T removeLast () { }
  - public T removeAny () { }
  - public boolean remove(T anElement)
  - public boolean replaceAt (T anElement, int order) { }
  - public void clear () { }

Linked List 구현에 필요한

Class "LinkedNode"



LinkedNode<T> 의 공개함수

- LinkedNode 객체 사용법을 Java로 구체적으로 표현
  - public LinkedNode() { }
  - public T element() { }
  - public LinkedNode<T> next() { }
  - public void setElement (T newElement) { }
  - public void setNext (LinkedNode<T> newNext) { }

Class LinkedNode: 비공개 인스턴스 변수

```
public class  LinkedNode<T>
{
    // 비공개 인스턴스 변수
    private  T                  _element ;
    private  LinkedNode<T>     _next ;
```

Class "LinkedNode"의 구현: 생성자

```
public class  LinkedNode<T>
{
    // 비공개 멤버 변수
    ......

    // 생성자:
    public  LinkedNode ( )
    {
        this.setElement (null) ;
        this.setNext (null) ;
    }

    public  LinkedNode (T givenElement, LinkedNode<T> givenNext)
    {
        this.setElement (givenElement) ;
        this.setNext (givenNext) ;
    }
```

Class "LinkedNode"의 구현: 생성자

```
public class  LinkedNode<T>
{
    // 비공개 멤버 변수
    ……

    // 생성자: 다른 생성자를 활용한 구현
    public  LinkedNode ( )
    {
        this (null, null) ;
    }


    public  LinkedNode (T givenElement, LinkedNode<T> givenNext)
    {
        this.setElement (givenElement) ;
        this.setNext (givenNext) ;
    }
```

Class "LinkedNode"의 구현: Getters

```
public class  LinkedNode<T>
{
    // 비공개 멤버 변수
    ......

    // Getters
    public T  element ( )
    {
        return  this._element ;
    }

    public LinkedNode<T>  next ( )
    {
        return  this._next ;
    }
```

Class "LinkedNode"의 구현: Setters

```
public class  LinkedNode<T>
{
    // 비공개 멤버 변수
    ......

    // Getters
    ......

    // Setters
    public void  setElement (T newElement)
    {
        return  this._element = newElement ;
    }

    public void  setNext (LinkedNode<T> newNext)
    {
        this._next =  newNext ;
    }
```

Class “LinkedList”의 구현

LinkedList: 비공개 인스턴스 변수

```
public class  LinkedList<T>
{
    // 비공개 인스턴스 변수
    private int                 _size ; // 리스트가 가지고 있는 원소의 개수
    private LinkedNode<T>  _head ; // LinkedChain 의 맨 앞 노드
```

LinkedList: Getter/Setter

```
public class  LinkedList<T>
{
    // 비공개 인스턴스 변수
    private int                 _size ; // 리스트가 가지고 있는 원소의 개수
    private LinkedNode<T>       _head ; // LinkedChain 의 맨 앞 노드

    // Getters/Setters
    public int  size() {
        return  this._size ;
    }
    private void  setSize (int newSize) {
        this._size = newSize ;
    }

    private LinkedNode<E>  head() {
        return  this._head ;
    }
    private void  setSize (LinkedNode<E> newHead) {
        this._head = newHead ;
    }
```

LinkedList: 비공개함수

- 여러 함수에서 공통으로 자주 사용

```
private boolean  anElementDoesExistAt (int order) {
        return  ( (order >= 0) && (order < this.size()) ) ;
}
```

LinkedList: 생성자

```
public class  LinkedList<T>
{
    // 비공개 인스턴스 변수
    ……

    // 생성자
    public  LinkedList ( )
    {
        this.setHead (null) ;
        this.setSize (0) ;
    }
```

LinkedList: 상태 알아보기

```
public class  LinkedList<T>
{
    ……

    // 상태 알아보기
    public boolean  isEmpty ()
    {
        return  (this.size() == 0) ;
    }

    public boolean  isFull ()
    {
        // 시스템 메모리가 모자라는 경우는 없다고 가정
        return  false ; // 언제나 full 이 아니다
    }

    // public int  size ()
    // {
    //      return  this._size ;
    // }
```

LinkedList: order 의 표현

- order:
  - 리스트에서의 원소의 순서
  - 사용자의 관점
  - 구현과는 무관

- LinkedList 에서의 구현
  - 여러 방법이 있을 수 있다.
  - 우리의 구현:
    - order를, 연결체인 에서의 노드의 순서와 일치시킨다.
    - 따라서, head node 에 들어있는 원소가 order 0.

LinkedList: 내용 알아보기

```
public T  elementAt (int order)
{
    if ( this.anElementDoesExistAt (order) ) {
        LinkedNode<T>  currentNode = this.head() ;
        int  nodeCount = 0 ;
        while ( nodeCount < order ) {
            currentNode = currentNode.next() ;
            nodeCount++ ;
        }
        return  currentNode.element() ;
    }
    else {
        return  null ;
    }
}
```

LinkedList: 내용 알아보기

```
public T  first () ;
{
    if  ( this.isEmpty() ) {
        return  null ; // 마지막 원소가 존재할 수 없으므로
    }
    else {
        return  this.elementAt (0) ;
        // 또는 이렇게: return this.head().element() ;
    }
}


public T  last () ;
{
    if  ( this.isEmpty() ) {
        return  null ; // 마지막 원소가 존재할 수 없으므로
    }
    else {
        return  this.elementAt (this.size()-1) ;
    }
}
```

LinkedList: 내용 알아보기

```
public int   orderOf (T anElement)
{ // 순차 검색
    int                  order = 0 ;
    LinkedNode<T>  currentNode = this.head() ;
    while ( (currentNode != null) &&
              (! currentNode.element().equals(anElement)) )
    {
        order++ ;
        currentNode = currentNode.next() ;
    }
    if ( currentNode == null ) { // Not Found
        return  -1 ; // 존재하지 않으면 -1 을 돌려주기로 한다
    }
    else {
        return  order ;
    }
}
```

LinkedList: 내용 알아보기

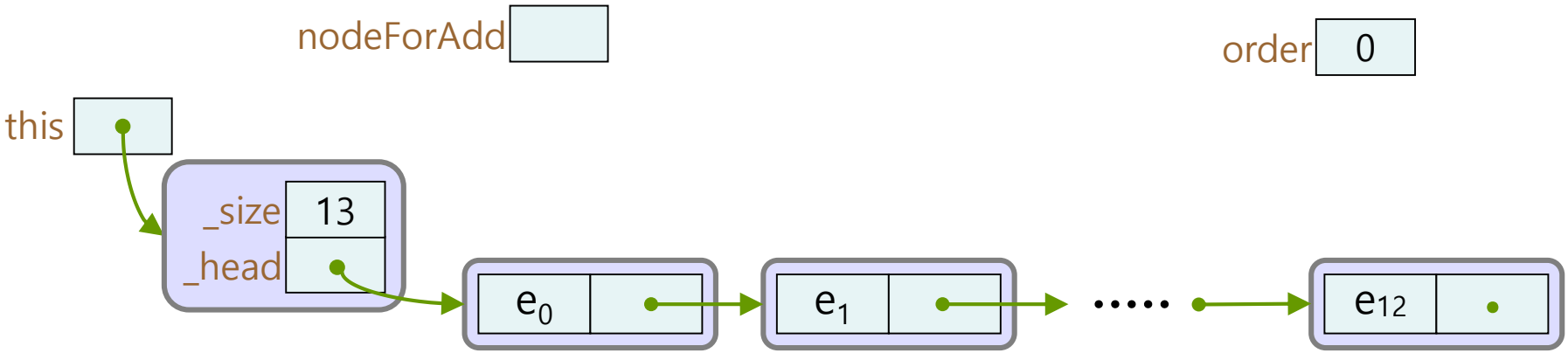
```
public boolean  doesContain (T anElement)  { // Version 1
    return  ( this.orderOf(anElement)  != -1 ) ;
}

public boolean  doesContain (T anElement)  { // Version 2
    LinkedNode<T>  currentNode = this.head() ;
    while ( currentNode != null &&
            (! currentNode.element().equals(anElement)) )
    {
        currentNode = currentNode.next() ;
    }
    return  (currentNode != null) ;
}

public boolean  doesContain (T anElement)  { // Version 3
    LinkedNode<T>  currentNode = this.head() ;
    while ( currentNode != null ) {
        if ( currentNode.element().equals(anElement) )  {
            return true ;
        }
        currentNode = current.next() ;
    }
    return  false ;
}
```

LinkedList: 원소 삽입하기

```
public boolean  addTo (T anElement, int order) {
    if ( (order < 0) || (order > this.size()) ) { // order 가 유효한지 검사
        return  false ;
    }
    else if ( this.isFull() ) {
        return  false ;
    }
    else {
        LinkedNode<T>  nodeForAdd = new LinkedNode<T> (anElement, null) ;
        if ( order == 0 ) {  // 맨 앞 순서에 삽입. 앞 (previous) 노드가 존재하지 않는다
            nodeForAdd.setNext (this.head()) ;
            this.setHead (nodeForAdd) ;
        }
        else { // 순서가 맨 앞이 아니므로, 반드시 앞 (previous) 노드가 존재한다.
            LinkedNode<T>  previousNode = this.head() ;
            for ( int i = 1 ; i < order ; i++ ) {
                previousNode = previousNode.next() ; // 삽입할 위치의 앞 노드를 찾는다
            }
            nodeForAdd.setNext (previousNode.next()) ;
            previousNode.setNext (nodeForAdd) ;
        }
        this.setSize (this.size()+1) ;
        return  true ;
    }
}
```

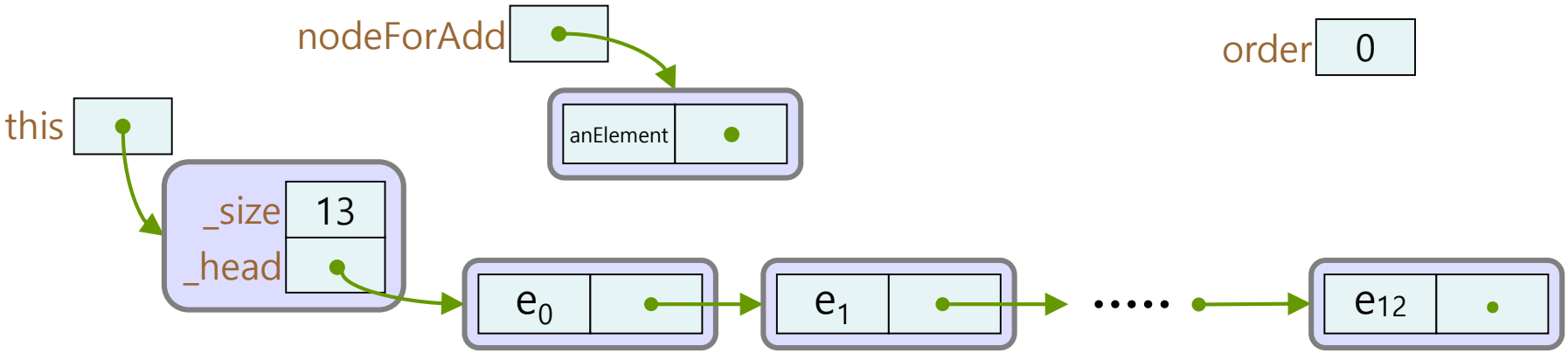
LinkedList: addTo() [1]

```
public boolean  addTo (T anElement, int order) {
    if ( (order < 0) || (order > this.size()) ) {
        return  false ;
    }
    ......
    else {
        LinkedNode<T>  nodeForAdd = new LinkedNode<T> (anElement, null) ;
        if ( order == 0 ) {
            nodeForAdd.setNext (this.head()) ;
            this.setHead (nodeForAdd) ;
        }
        else {
            LinkedNode<T>  previousNode = this.head() ;
```



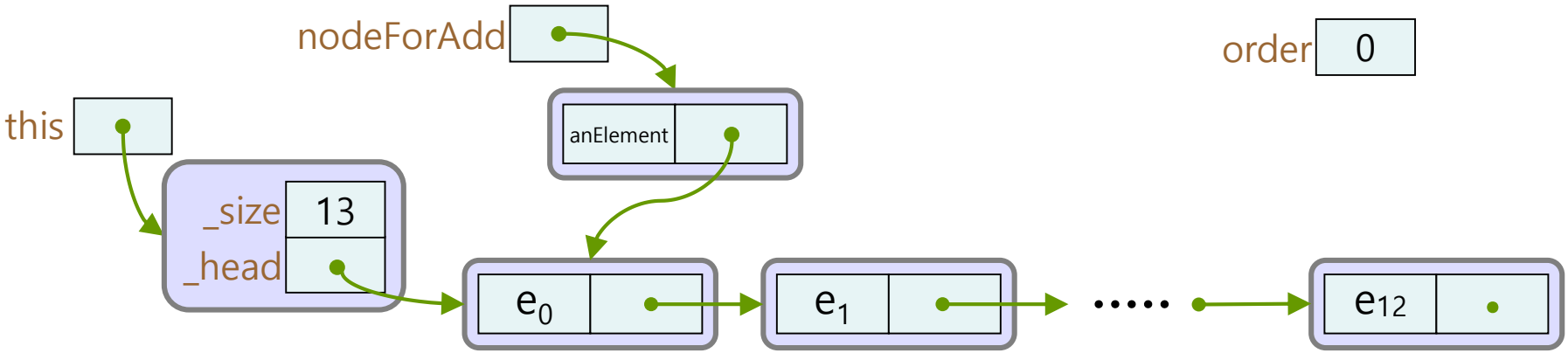
LinkedList: addTo() [2]

```
public boolean  addTo (T anElement, int order) {
    if ( (order < 0) || (order > this.size()) ) {
        return  false ;
    }
    ......
    else {
        LinkedNode<T>  nodeForAdd = new LinkedNode<T> (anElement, null) ;
        if ( order == 0 ) {
            nodeForAdd.setNext (this.head()) ;
            this.setHead (nodeForAdd) ;
        }
        else {
            LinkedNode<T>  previousNode = this.head() ;
```



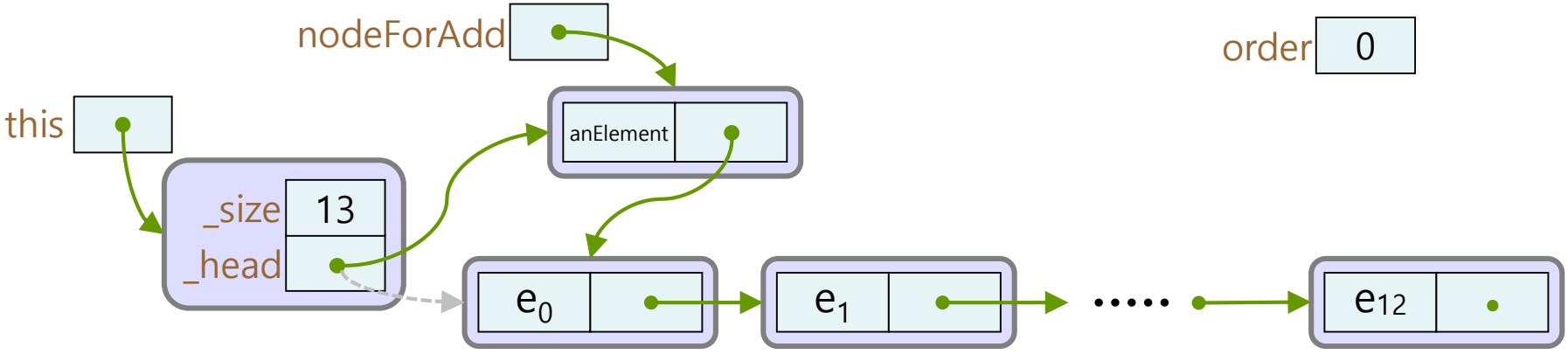
LinkedList: addTo() [3]

```
public boolean  addTo (T anElement, int order) {
    if ( (order < 0) || (order > this.size()) ) {
        return  false ;
    }
    ......
    else {
        LinkedNode<T>  nodeForAdd = new LinkedNode<T> (anElement, null) ;
        if ( order == 0 ) {
            nodeForAdd.setNext (this.head()) ;
            this._head = nodeForAdd ;
        }
        else {
            LinkedNode<T>  previousNode = this._head ;
```



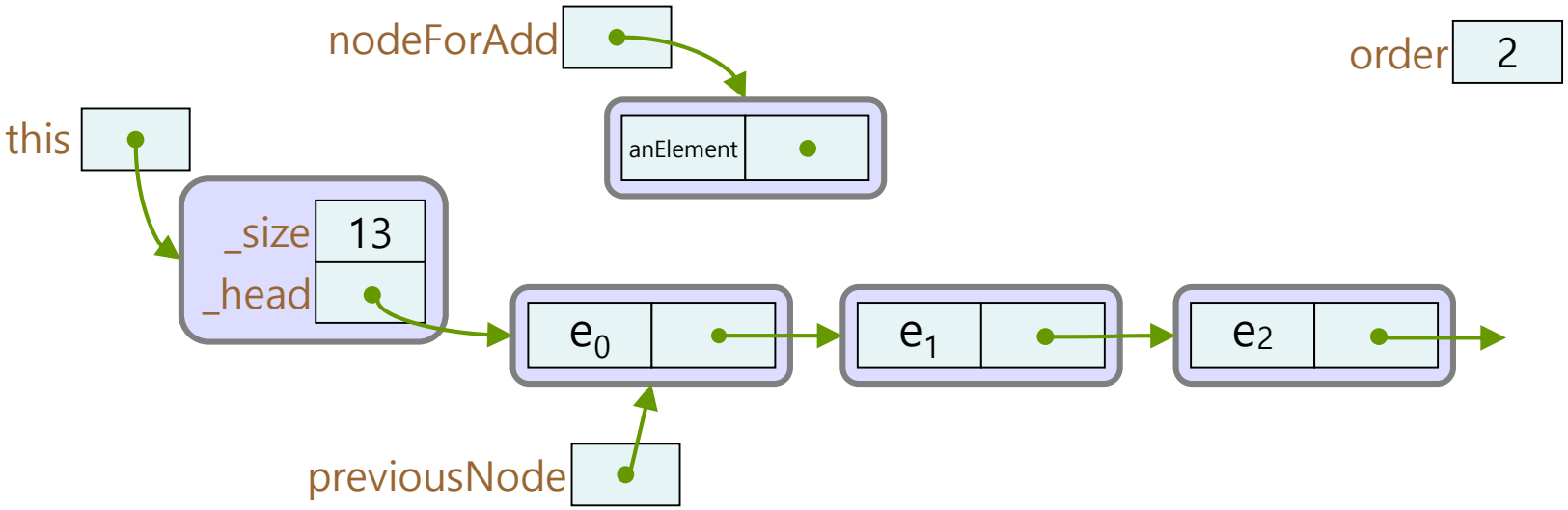
LinkedList: addTo() [4]

```
public boolean  addTo (T anElement, int order) {
    if ( (order < 0) || (order > this.size()) ) {
        return  false ;
    }
    ......
    else {
        LinkedNode<T>  nodeForAdd = new LinkedNode<T> (anElement, null) ;
        if ( order == 0 ) {
            nodeForAdd.setNext (this.head()) ;
            this.setHead (nodeForAdd) ;
        }
        else {
            LinkedNode<T>  previousNode = this.head()  ;
```



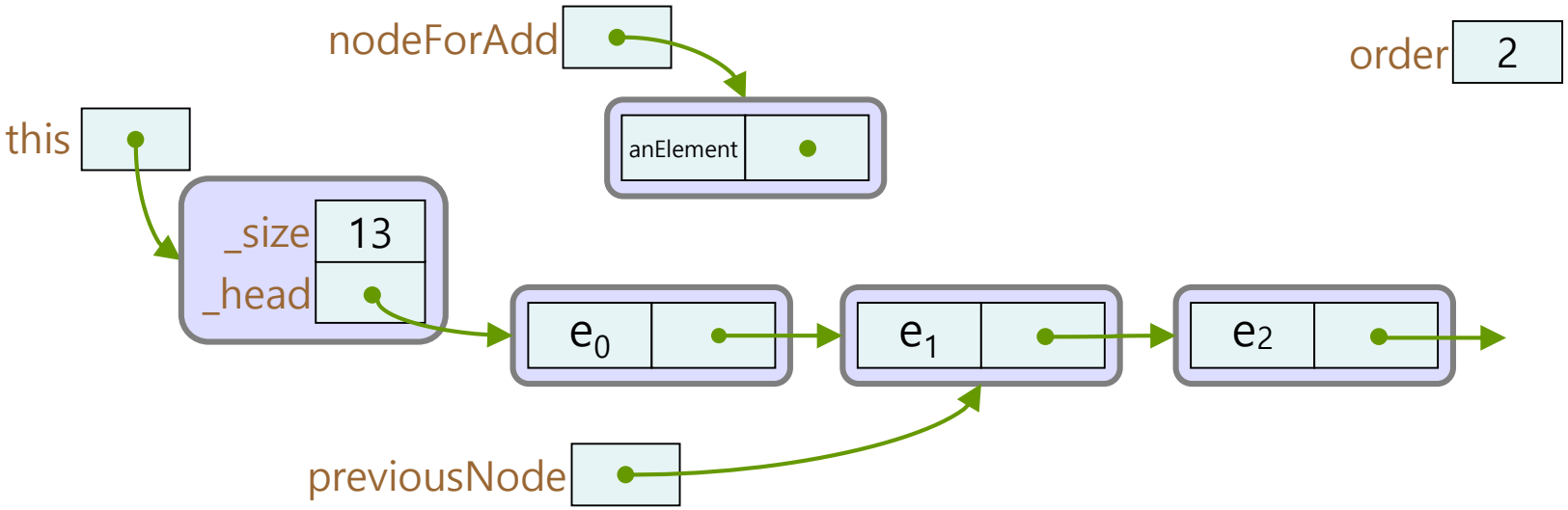
LinkedList: addTo() [5]

```
public boolean  addTo (T anElement, int order)
{
        ......
        else {
            LinkedNode<T>  previousNode = this.head() ;
            for ( int i = 1 ; i < order ; i++ ) {
                previousNode = previousNode.next() ;
            }
            nodeForAdd.setNext (previousNode.next()) ;
            previousNode.setNext (nodeForAdd) ;
        }
```



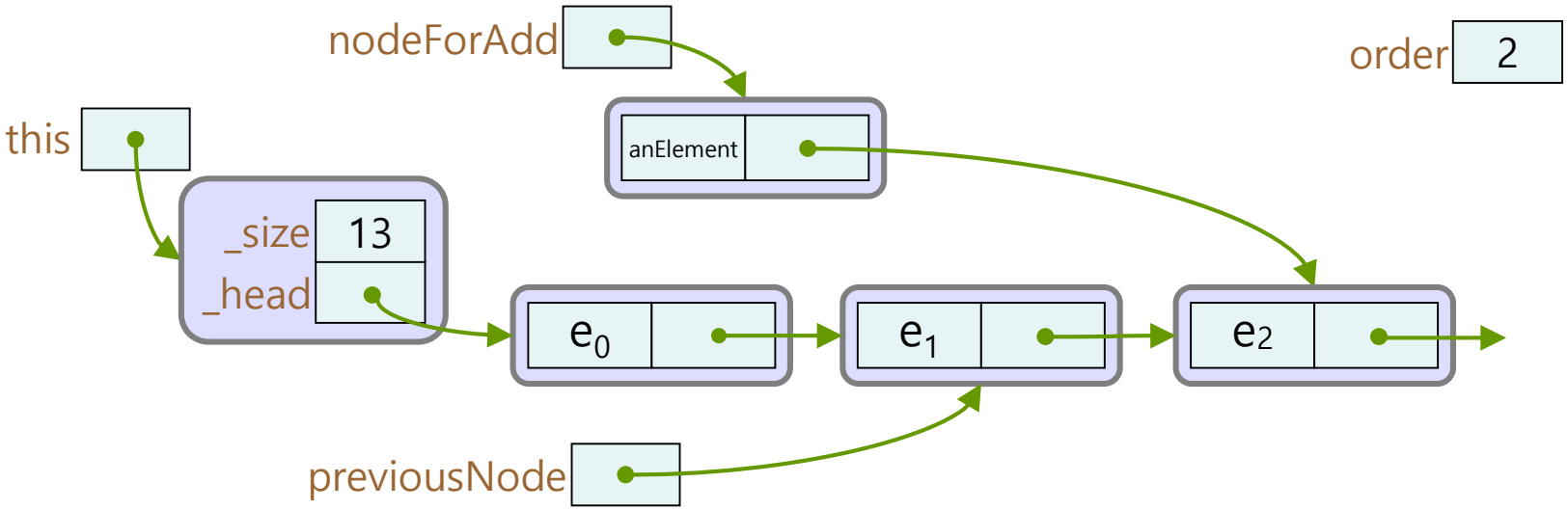
LinkedList: addTo() [6]

```
public boolean  addTo (T anElement, int order)
{
        ......
        else {
            LinkedNode<T>  previousNode = this.head() ;
            for ( int i = 1 ; i < order ; i++ ) {
                previousNode = previousNode.next() ;
            }
            nodeForAdd.setNext (previousNode.next()) ;
            previousNode.setNext (nodeForAdd) ;
        }
```



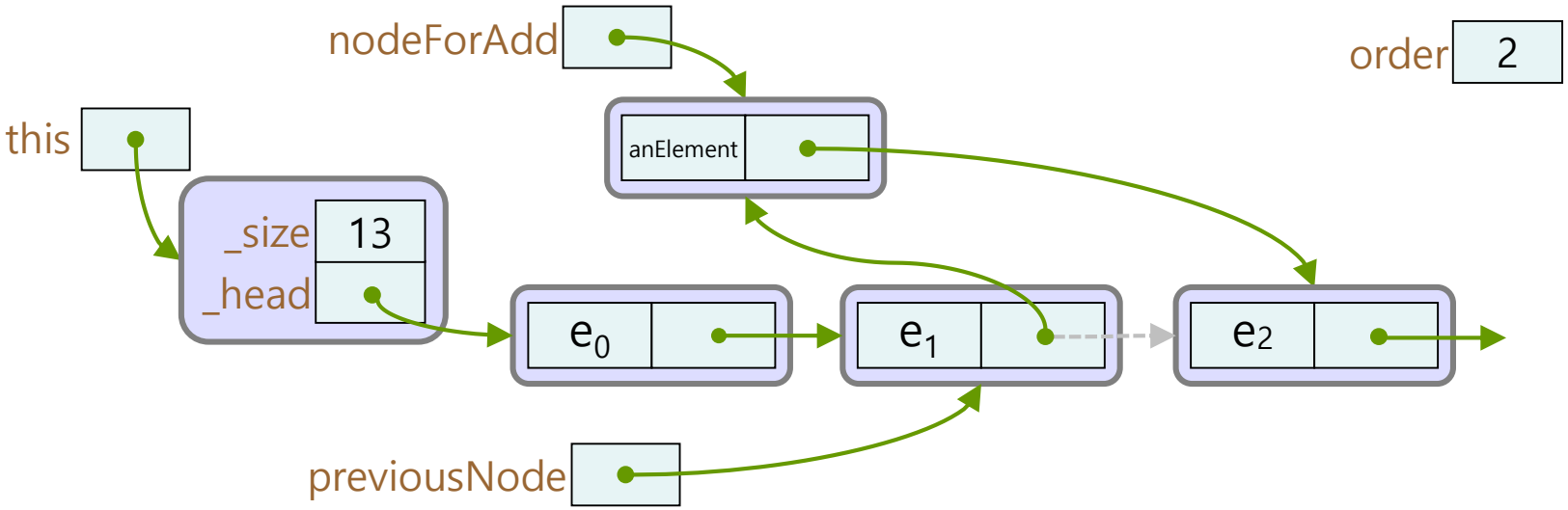
LinkedList: addTo() [7]

```
public boolean  addTo (T anElement, int order)
{
        ......
        else {
            LinkedNode<T>  previousNode = this.head() ;
            for ( int i = 1 ; i < order ; i++ ) {
                previousNode = previousNode.next() ;
            }
            nodeForAdd.setNext (previousNode.next()) ;
            previousNode.setNext (nodeForAdd) ;
        }
```



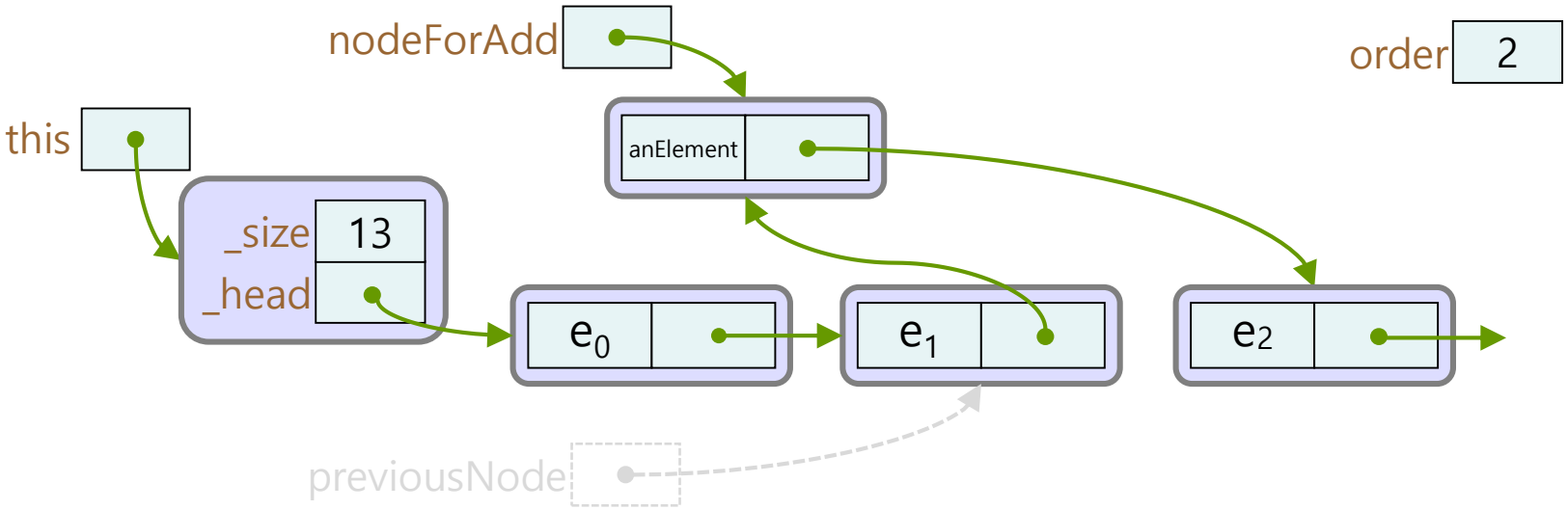
LinkedList: addTo() [8]

```
public boolean  addTo (T anElement, int order)
{
        ......
        else {
            LinkedNode<T>  previousNode = this.head() ;
            for ( int i = 1 ; i < order ; i++ ) {
                previousNode = previousNode.next() ;
            }
            nodeForAdd.setNext (previousNode.next()) ;
            previousNode.setNext (nodeForAdd) ;
        }
```



LinkedList: addTo() [9]

```
public boolean  addTo (T anElement, int order)
{
        ......
        else {
            LinkedNode<T>  previousNode = this.head() ;
            for ( int i = 1 ; i < order ; i++ ) {
                previousNode = previousNode.next() ;
            }
            nodeForAdd.setNext (previousNode.next()) ;
            previousNode.setNext (nodeForAdd) ;
        }
```



LinkedList: addToFirst()

```
public boolean  addToFirst (T anElement) { // Version 1
     return  this.addTo (anElement, 0) ;
}


public boolean  addToFirst (T anElement) { // Version 2
     if ( this.isFull() ) {
          return  false ;
     }
     else {
          LinkedNode<T> nodeForAdd = new LinkedNode<T>(anElement, this.head()) ;
          this.setHead (nodeForAdd) ;
          this.setSize (this.size()+1) ;
          return  true ;
     }
}
```

LinkedList: addToLast()

```
public boolean  addToLast (T anElement)   // Version 1
{
     return   this.addTo (anElement, this.size()) ;
}

public boolean  addToLast (T anElement)   // Version 2
{
     if ( this.isFull() ) {
          return   false ;
     }
     else {
          LinkedNode<T> nodeForAdd = new LinkedNode<T>(anElement, null) ;
          if ( this.isEmpty() ) {
               this.setHead (nodeForAdd) ;
          }
          else {
               LinkedNode<T> lastNode = this.head() ;
               while ( lastNode.next() != null ) {
                    lastNode = lastNode.next() ;
               }
               lastNode.setNext(nodeForAdd) ;
          }
          this.setSize (this.size()+1) ;
          return   true ;
     }
}
```

LinkedList: add()

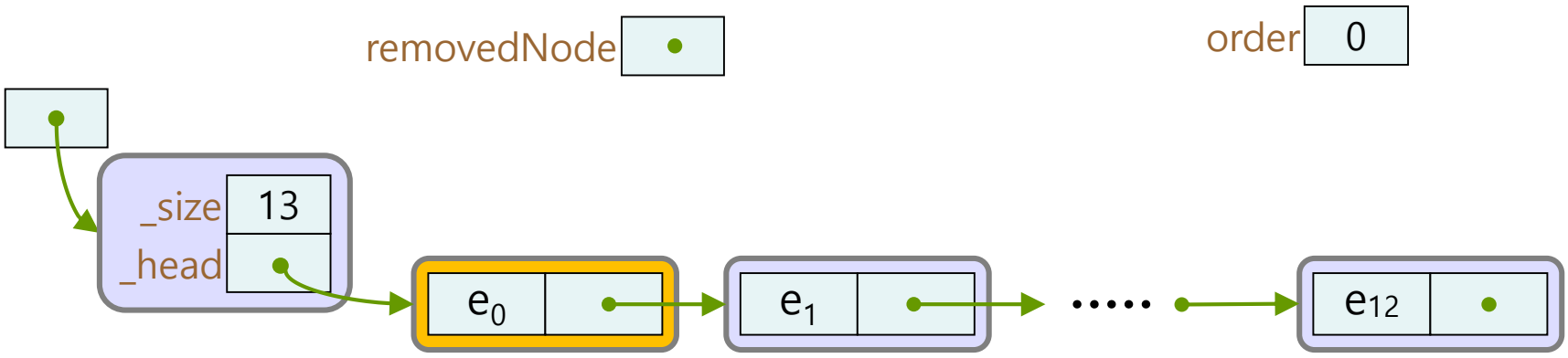
```
public boolean  add (T anElement)
{
    return  this.addToFirst (anElement) ;        // 왜 addToFirst() ?
                                                 // addToLast() 라면 ?
}
```

LinkedList: 원소 삭제하기

```
public T  removeFrom (int order)
{
    if ( ! this.anElementDoesExistAt (order) ) { // 삭제할 원소가 없거나, 잘못된 위치
        return  null ;
    }
    else {
        // 리스트는 비어 있지 않으며, 삭제할 원소가 있음
        LinkedNode<T>  removedNode = null ;
        if ( order == 0 ) { // 삭제할 원소가 맨 앞 원소
            removedNode = this.head() ;
            this.setHead (this.head().next()) ;
        }
        else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
            LinkedNode<T>  previousNode = this.head() ;
            for ( int i = 1 ; i < order ; i++ ) {
                previousNode = previousNode.next() ; // 삭제할 위치의 앞 노드를 찾는다
            }
            removedNode = previousNode.next() ;
            previousNode.setNext (removedNode.next()) ;
        }
        this.setSize (this.size()-1) ;
        return  removedNode.element() ;
    }
}
```

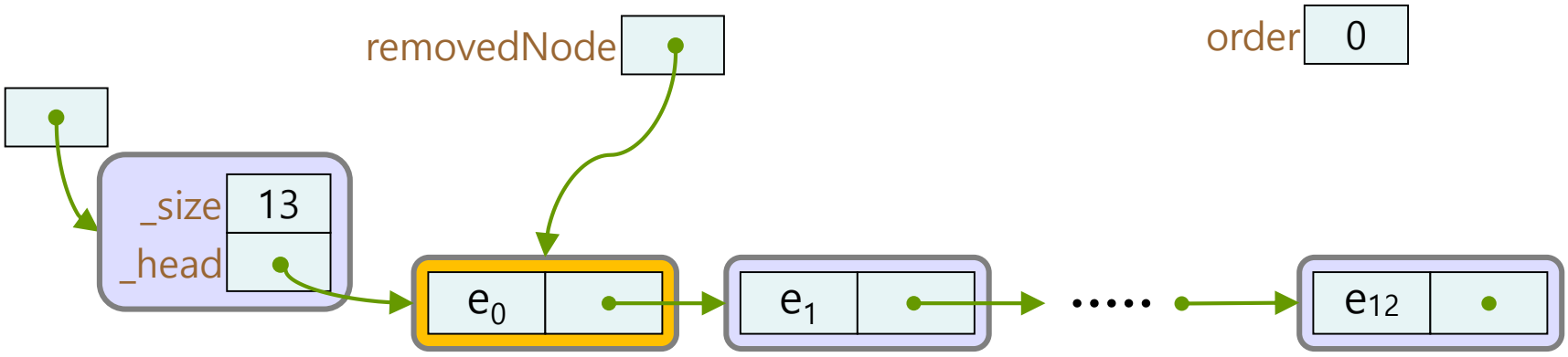
LinkedList: removeFrom() [1]

```
public T  removeFrom (int order)
{
      if ( ! this.anElementDoesExistAt (order) ) { // 삭제할 원소가 없거나, 잘못된 위치
            return null ;
      }
      else {
            // 리스트는 비어 있지 않으며, 삭제할 원소가 있음
            LinkedNode<T>  removedNode = null ;
            if ( order == 0 ) { // 삭제할 원소가 맨 앞 원소
                  removedNode = this.head() ;
                  this.setHead (this.head().next()) ;
            }
            else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
```



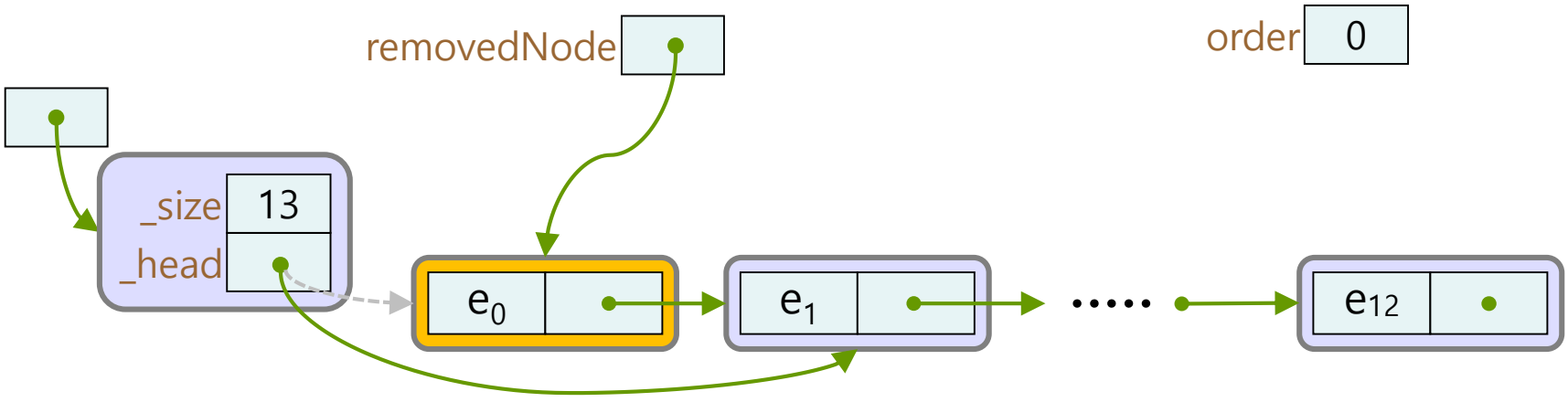
LinkedList: removeFrom() [2]

```
public T  removeFrom (int order)
{
      if ( ! this.anElementDoesExistAt (order) ) { // 삭제할 원소가 없거나, 잘못된 위치
            return null ;
      }
      else {
            // 리스트는 비어 있지 않으며, 삭제할 원소가 있음
            LinkedNode<T>  removedNode = null ;
            if ( order == 0 ) { // 삭제할 원소가 맨 앞 원소
                  removedNode = this.head() ;
                  this.setHead (this.head().next()) ;
            }
            else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
```



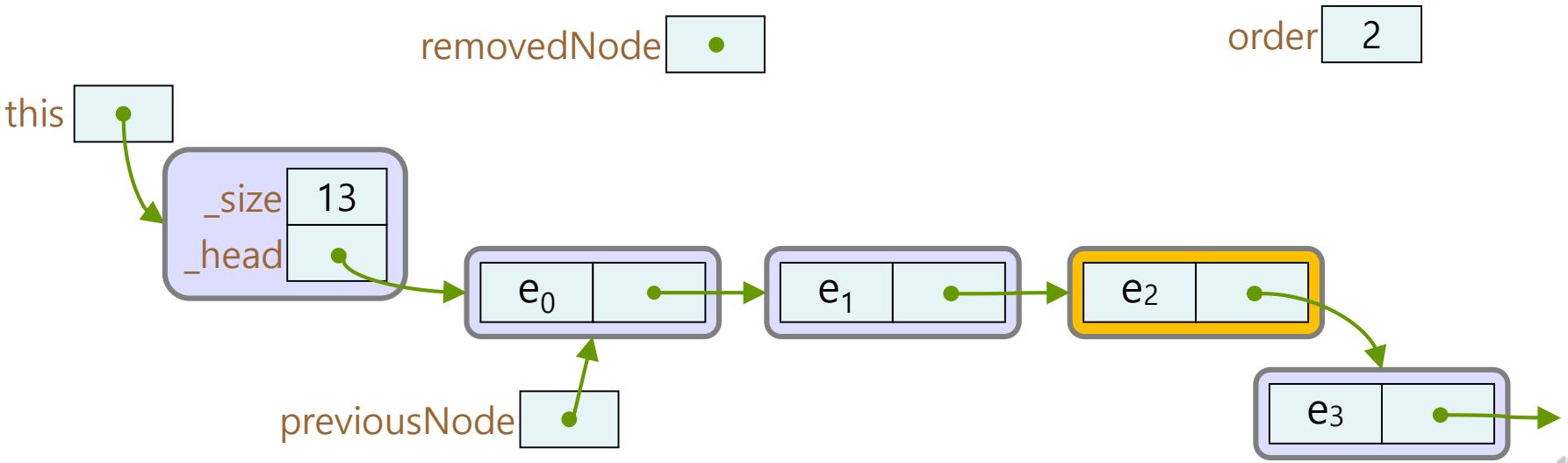
LinkedList: removeFrom() [3]

```
public T  removeFrom (int order)
{
      if ( ! this.anElementDoesExistAt (order) ) { // 삭제할 원소가 없거나, 잘못된 위치
            return null ;
      }
      else {
            // 리스트는 비어 있지 않으며, 삭제할 원소가 있음
            LinkedNode<T>  removedNode = null ;
            if ( order == 0 ) { // 삭제할 원소가 맨 앞 원소
                  removedNode = this.head() ;
                  this.setHead (this.head().next()) ;
            }
            else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
```



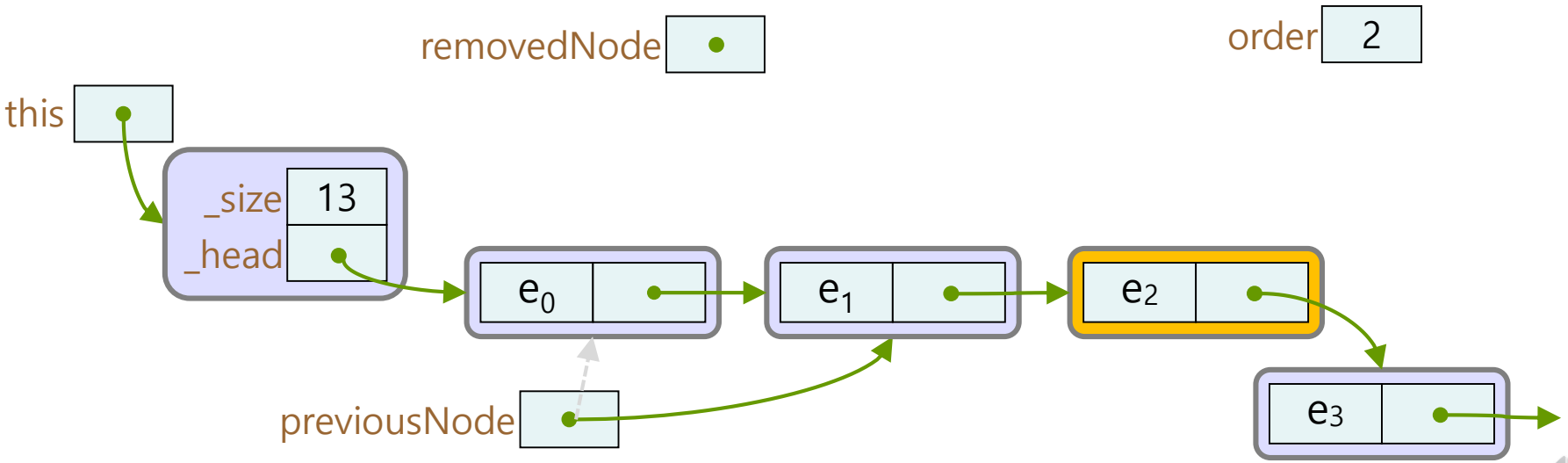
LinkedList: removeFrom() [4]

```
public T  removeFrom (int order)
{
        ......
        else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
              LinkedNode<T>  previousNode = this.head() ;
              for ( int i = 1 ; i < order ; i++ ) {
                    previousNode = previousNode.next() ; // 삭제할 위치의 앞 노드를 찾는다
              }
              removedNode = previousNode.next() ;
              previousNode.setNext (removedNode.next()) ;
        }
        this.setSize (this.size()-1) ;
        return  removedNode.element() ;
```



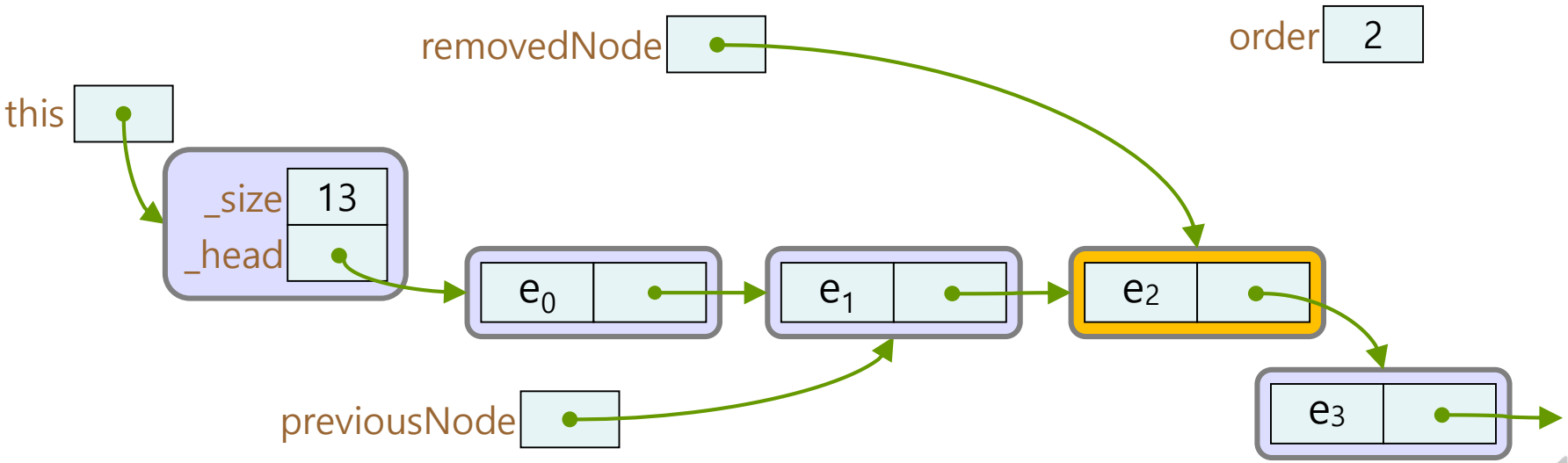
LinkedList: removeFrom() [5]

```
public T  removeFrom (int order)
{
        ......
        else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
              LinkedNode<T>  previousNode = this.head() ;
              for ( int i = 1 ; i < order ; i++ ) {
                    previousNode = previousNode.next() ; // 삭제할 위치의 앞 노드를 찾는다
              }
              removedNode = previousNode.next() ;
              previousNode.setNext (removedNode.next()) ;
        }
        this.setSize (this.size()-1) ;
        return  removed.element() ;
```



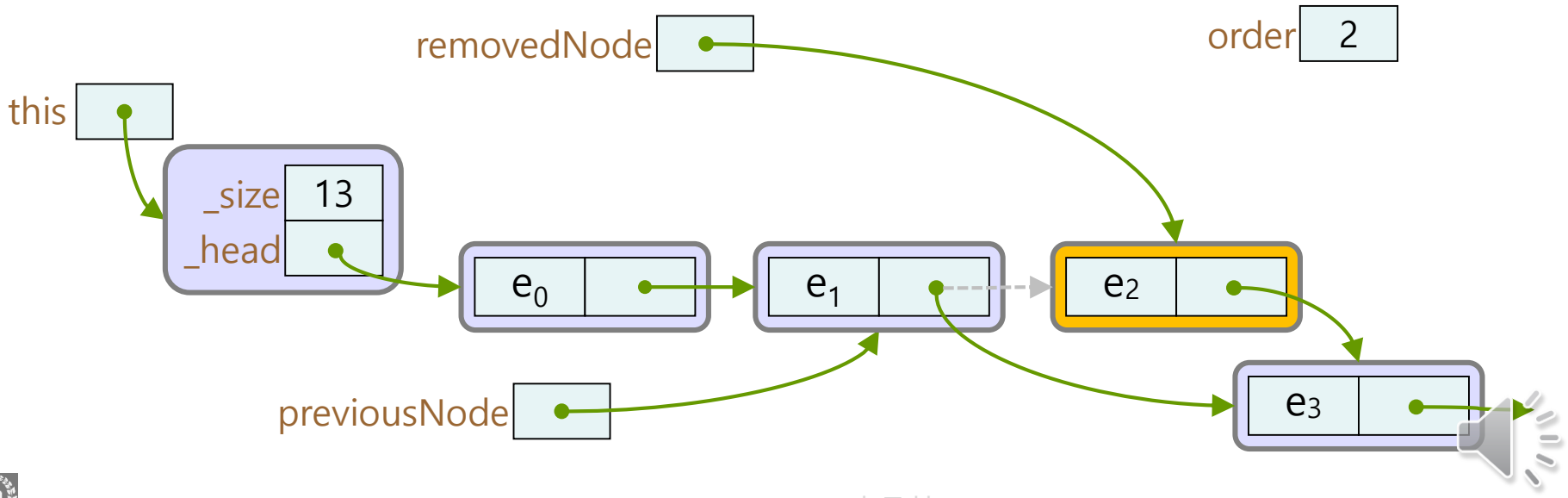
LinkedList: removeFrom() [6]

```
public T  removeFrom (int order)
{
        ......
        else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
              LinkedNode<T>  previousNode = this.head() ;
              for ( int i = 1 ; i < order ; i++ ) {
                    previousNode = previousNode.next() ; // 삭제할 위치의 앞 노드를 찾는다
              }
              removedNode = previousNode.next() ;
              previousNode.setNext (removedNode.next()) ;
        }
        this.setSize (this.size()-1);
        return  removedNode.element() ;
```



LinkedList: removeFrom() [7]

```
public T  removeFrom (int order)
{
        ......
        else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
              LinkedNode<T>  previousNode = this.head() ;
              for ( int i = 1 ; i < order ; i++ ) {
                    previousNode = previousNode.next() ; // 삭제할 위치의 앞 노드를 찾는다
              }
              removedNode = previousNode.next() ;
              previousNode.setNext (removedNode.next()) ;
        }
        this.setSize (this.size()-1);
        return  removedNode.element() ;
```



LinkedList: removeFirst()

```
public T  removeFirst () // Version 1
{
    return  (this.removeFrom(0)) ;
}


public T  removeFirst () // Version 2
{
    if ( this.isEmpty() ) {
        return  null ;
    }
    else {
        T  removedElement = this.head().element() ;
        this.setHead (this.head().next()) ;
        this.setSize (this.size()-1) ;
        return  removedElement ;
    }
}
```

LinkedList: removeLast()

```
public T  removeLast () {  // Version 1
      return  (removeFrom(this.size()-1)) ;
}

public T  removeLast () {  // Version 2
      if ( this.isEmpty() ) {
            return  null ;
      }
      else {
            T  removedElement = null ;
            if (this.head().next() == null ) { // single node in the list
                  removedElement = this.head().element() ;
                  this.setHead (null) ;
            }
            else { // At least 2 nodes in the list.
                  LinkedNode<T>  previousNode = this.head() ;
                  LinkedNode<T>  lastNode = previousNode.next()
                  while ( lastNode.next() != null ) {
                        previousNode = lastNode ;
                        lastNode = lastNode.next() ;
                  }
                  removedElement = lastNode.element() ;
                  previousNode.setNext(null) ;
            }
            this.setSize (this.size()-1) ;
            return  removedElement ;
      }
}
```

LinkedList: removeAny()

```
public T  removeAny ()
{
    return  this.removeFirst() ;   // 왜 removeFirst() ?
                                   // removeLast() 라면?
}
```

LinkedList: remove ()

```
public boolean  remove (T anElement) { // Version 1
        int orderOfRemove = this.orderOf(anElement) ;
        if ( orderOfRemove < 0 ) {
                return  false ; // Not found
        }
        else {
                this.removeFrom(orderOfRemove) ;
                return  true ;
        }
}

public boolean  remove (T anElement) { // Version 2
        // 단계 1: 주어진 원소의 위치를 찾는다
        LinkedNode<T>  previousNode = null ;
        LinkedNode<T>  currentNode = this.head() ;
        while ( (currentNode != null) && (!currentNode.element.equals(anElement)) ) {
                previousNode = currentNode ;
                currentNode = currentNode.next() ;
        }
        // 단계 2: 주어진 원소가 존재하면 삭제한다.
        if ( currentNode == null ) {
                return  false ; // Not Found
        }
        else {
                if ( currentNode == this.head() ) { // 삭제할 노드가 맨 앞 노드
                        this.setHead (this.head().next()) ;
                }
                else { // 삭제할 노드 앞에 노드 (previous) 가 존재
                        previousNode.setNext(currentNode.next()) ;
                }
                this.setSize (this.size()-1) ;
                return  true ;
        }
}
```

LinkedList: 원소 바꾸기

```
public boolean  replaceAt (T anElement, int order)
{
    if ( ! this.anElementDoesExistAt (order) ) {
        // 대체할 노드가 없거나, 잘못된 위치
        return  false ;
    }
    else {
        LinkedNode<T>  currentNode = this.head() ;
        for ( int i = 0 ; i < order ; i++ ) {
              currentNode = currentNode.next() ;
                 // 원소를 대체할 노드를 찾는다
        }
        currentNode.setElement (anElement) ;
        return  true ;
    }
}
```

LinkedList: clear()

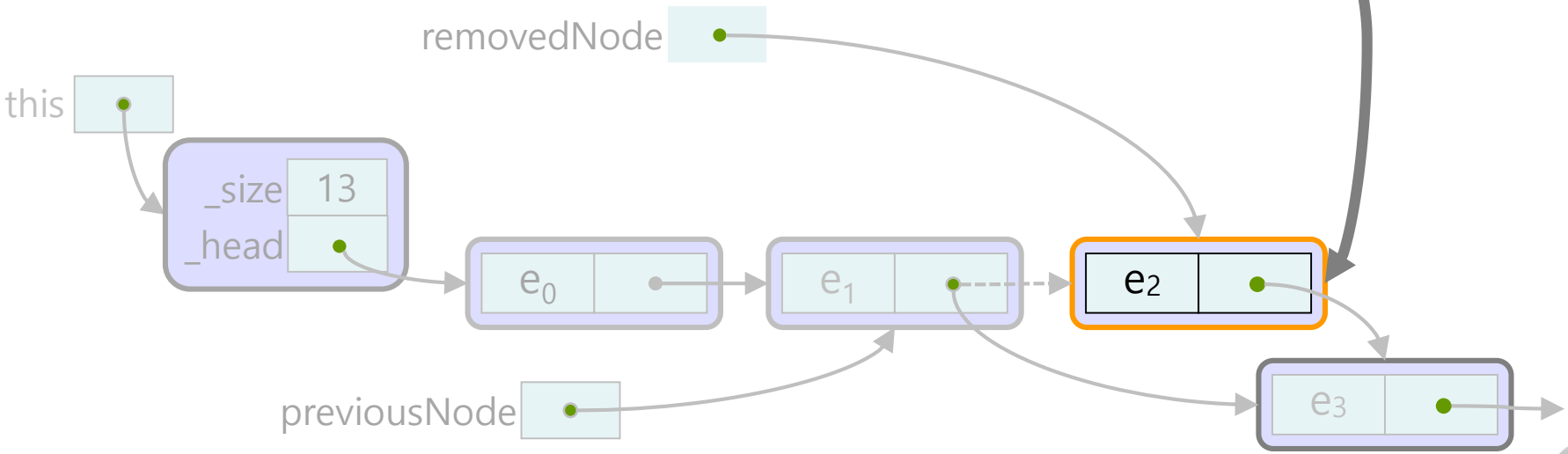
```
// 내용 바꾸기
.....

public void  clear()
{
    this.setHead (null) ;
    this.setSize (0) ;
}
```

쓰레기 줍기
(Garbage Collection)

쓰레기 줍기

```
public T  removeFrom (int order)
{
        ......
        else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
            LinkedNode<T>  previousNode = this._head ;
            for (int i = 1 ; i < order ; i++) {
                p                                              앞 노드를 찾는다
            }
            Linked
            previo
        }
        this._size--
        return  re
```

함수 종료 후에
이 노드는 어떻게 될까?
아무 곳에서도
이 노드를 가지고 있지 않음!!!

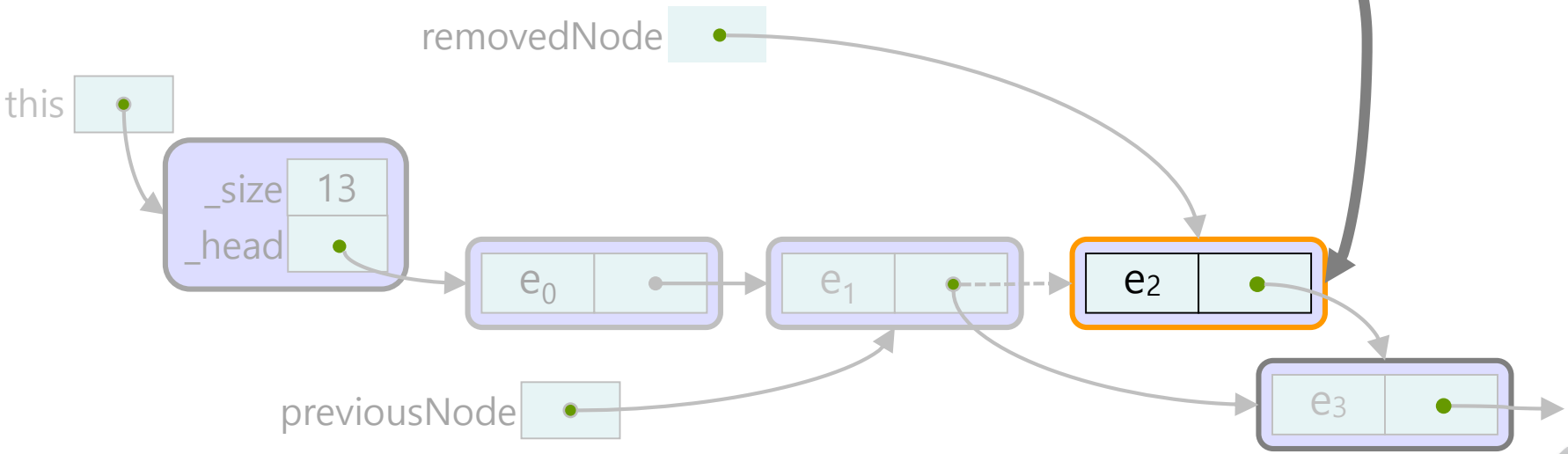


쓰레기 줍기

```
public T  removeFrom (int order)
{
        ......
        else { // 삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
            LinkedNode<T>  previousNode = this._head ;
            for (int i = 1 ; i < order ; i++) {
                p                                          앞 노드를 찾는다
            }
            Linked
            previo
        }
        this._size--
        return  re
```

Java 시스템은 이러한 메모리 조각들을
주기적으로 찾아 모아서
다시 사용할 수 있게 한다!

➡ "Garbage Collection"



모델-뷰-컨트롤러

입출력을 리스트 클래스 안에서?

```
public void  showAll()
{
    LinkedNode<T>  currentNode = this.head() ;
    while ( currentNode != null )
    {
        System.out.println (currentNode.element()) ; //  Why BAD??
        currentNode = currentNode.next() ;
    }
}
```

- 클래스의 역할의 구분:
  - 모델 (Model): 입출력과 무관한 순수한 알고리즘
  - 뷰 (View): 입출력만 담당
  - 컨트롤러 (Controller): 모델 객체와 뷰 객체를 소유하고 제어

End of

"ArrayList", "Linked List"