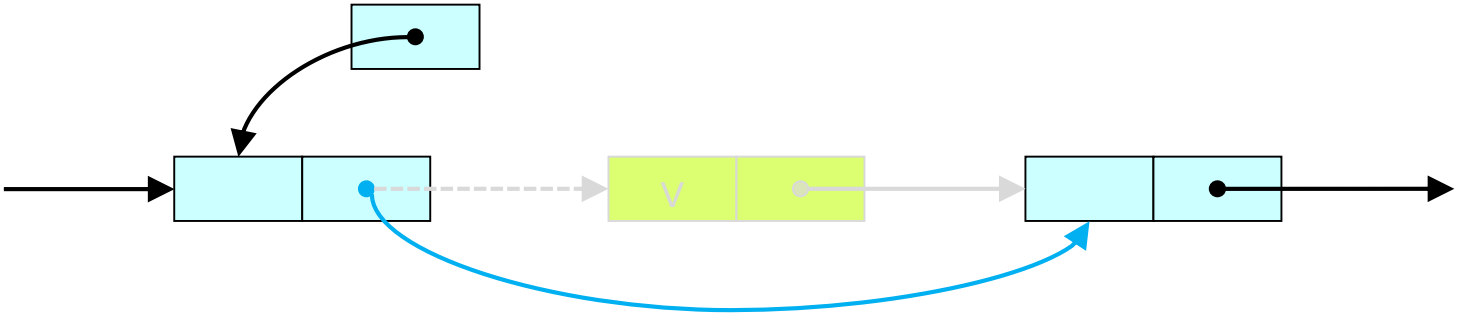
자료구조: 2022년 1학기 [강의]

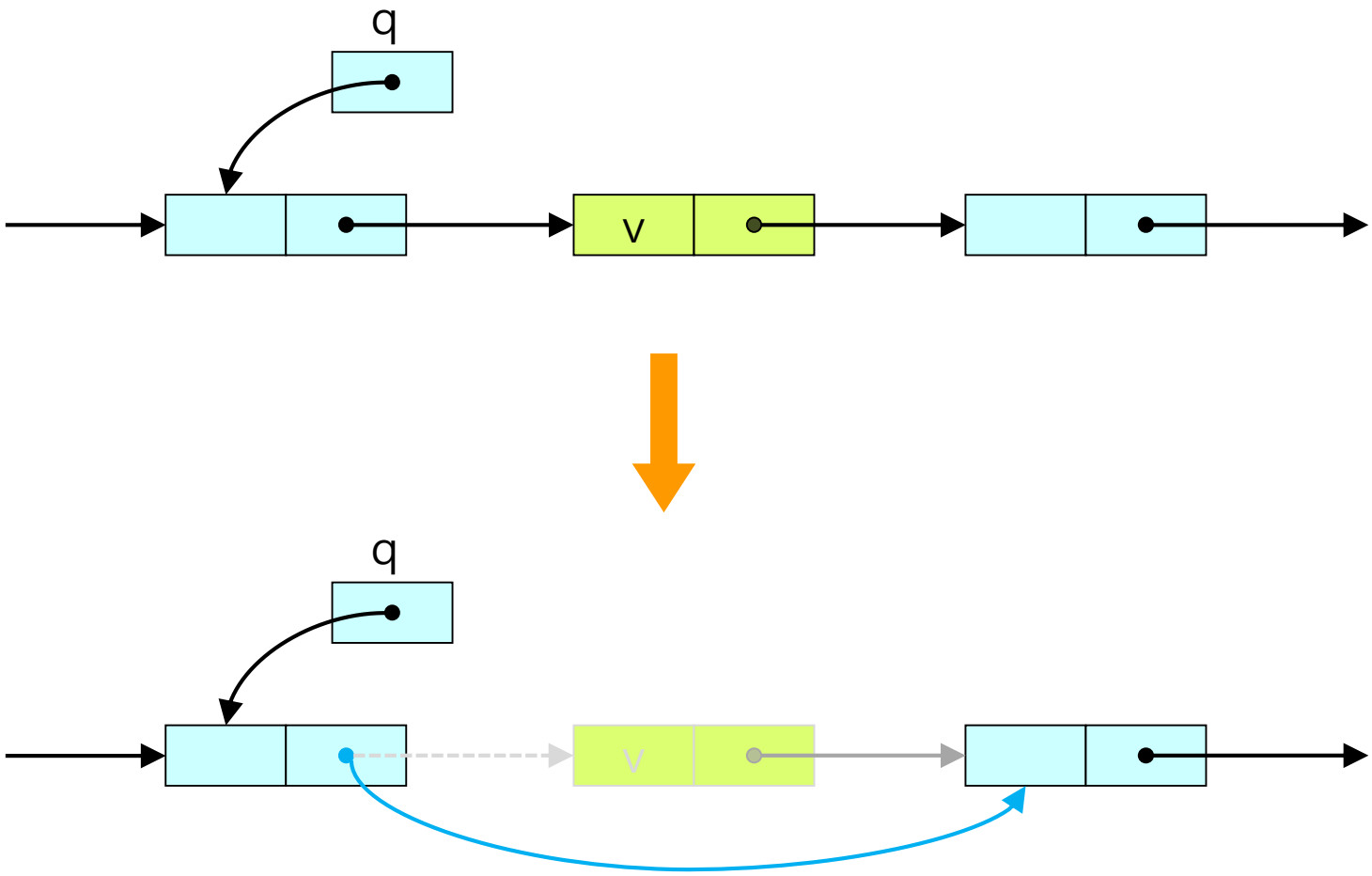
Linked Bag (2)

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

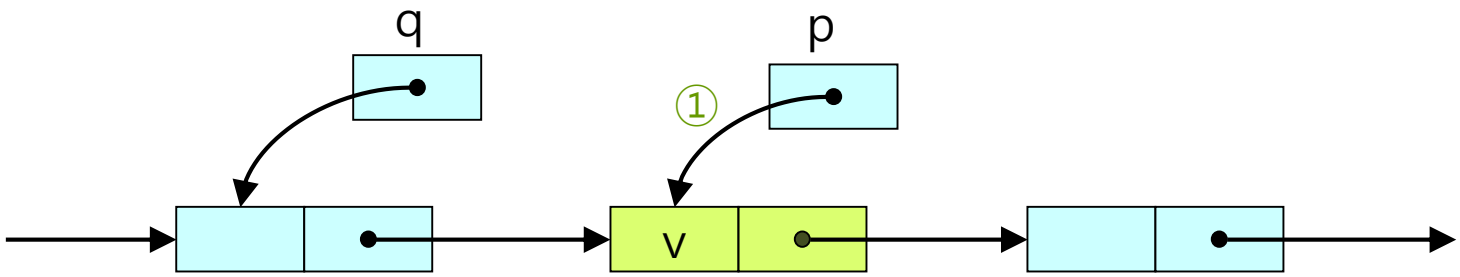
연결 체인에서의 삭제



노드의 삭제 [1]



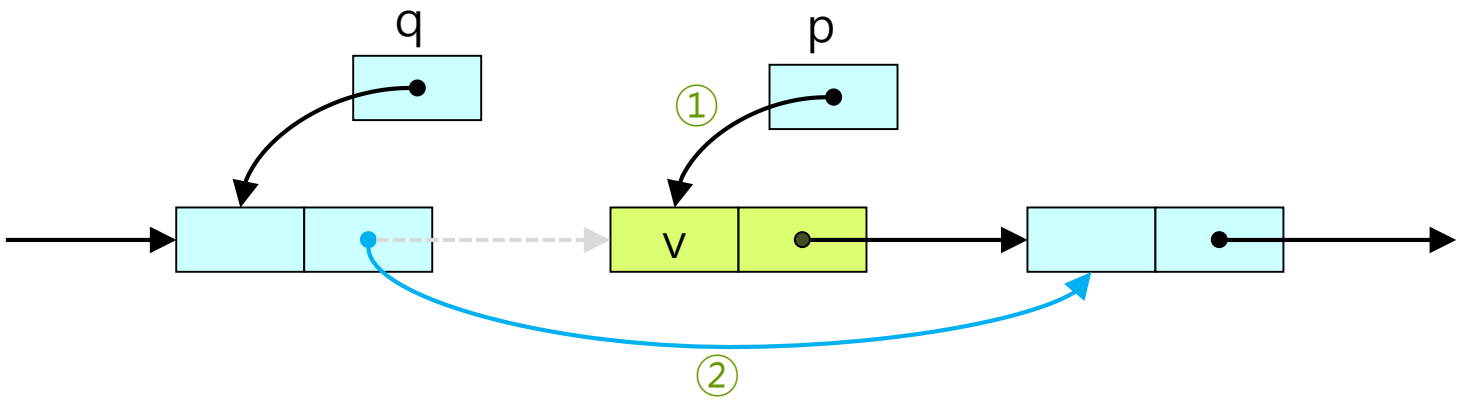
노드의 삭제 [2]



① p = q.next() ;

② q.setNext(p.next()) ;
// (또는) q.setNext((q.next()).next()) ;

노드의 삭제 [3]



① p = q.next() ;

② q.setNext(p.next()) ;
// (또는) q.setNext((q.next()).next()) ;

“LinkedBag” 에서의 삭제

LinkedBag: removeAny()

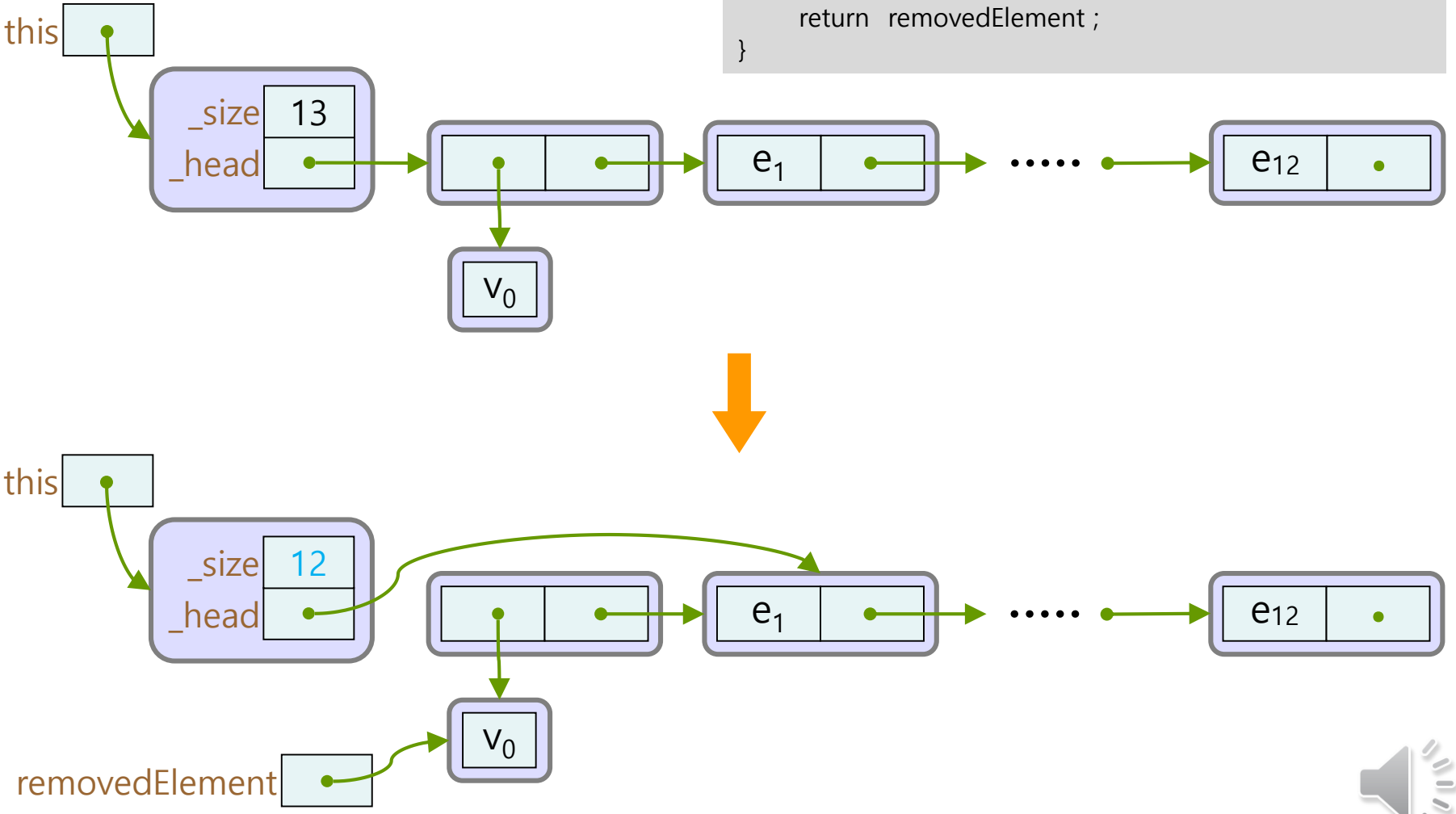
```
// 내용 바꾸기
.....

// 아무 노드나 하나 삭제하면 된다.
// 그렇다면, 어느 노드를 삭제하는 것이 효율적일까?
//
public E  removeAny ()
{
    if ( this.isEmpty() ) {
        return  null ;
    }
    else {
        E  removedElement = this.head().element() ;
        this.setHead (this.head().next()) ;
        this.setSize (this.size()-1) ;
        return  removedElement ;
    }
}
```

LinkedBag: removeAny()

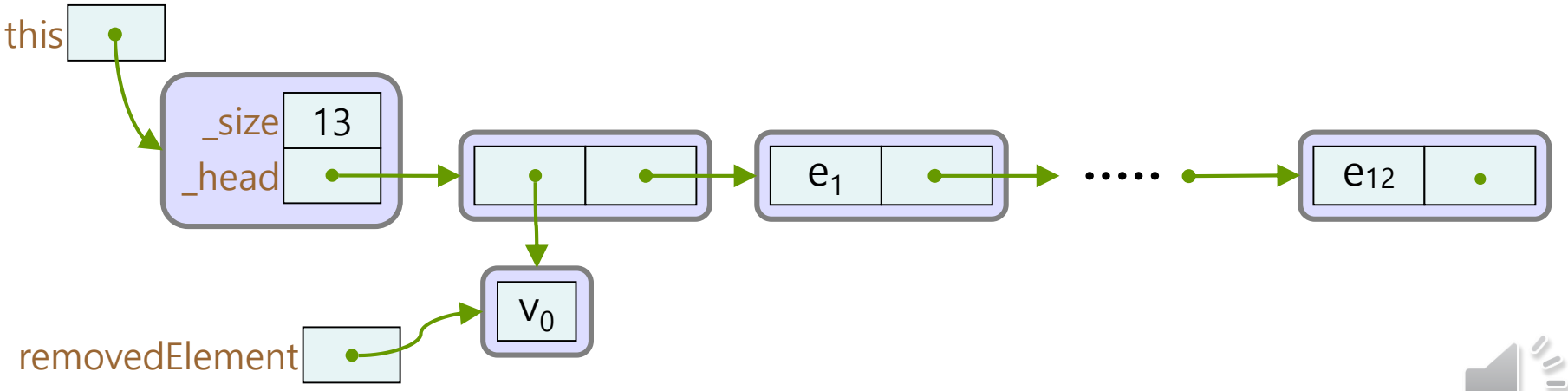
- 삭제 위치는?

```
else {
        E  removedElement = this.head().element() ;
        this.setHead (this.head().next()) ;
        this.setSize (this.size()-1) ;
        return  removedElement ;
}
```



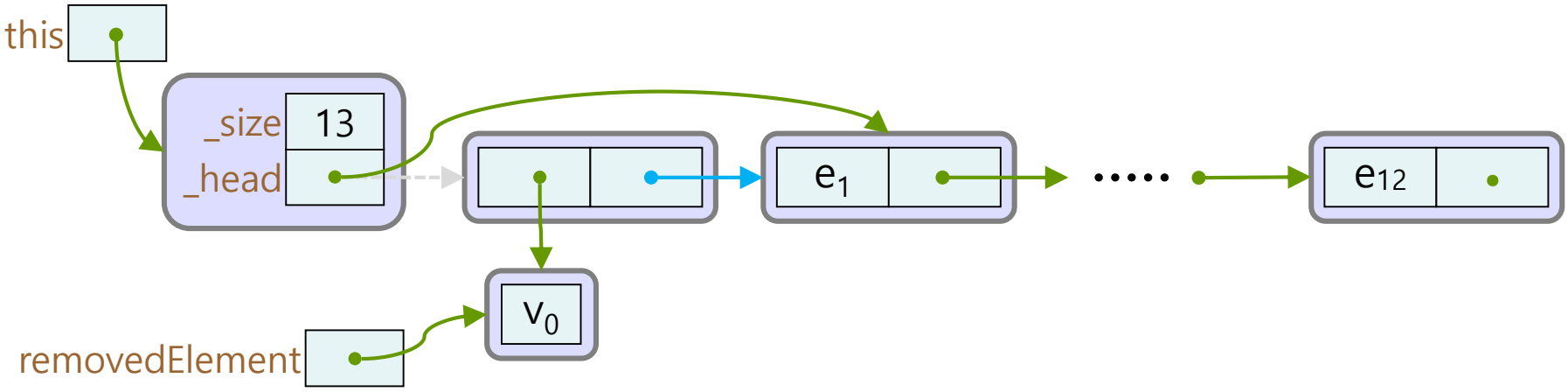
LinkedBag: removeAny()

```
// 내용 바꾸기
.....
public E  removeAny ()
{
    if ( this.isEmpty() ) {
        return  null ;
    }
    else {
        E  removedElement  =  this.head().element() ;
        this.setHead (this.head().next()) ;
        this.setSize (this.size()-1) ;
        return  removedElement ;
    }
}
```



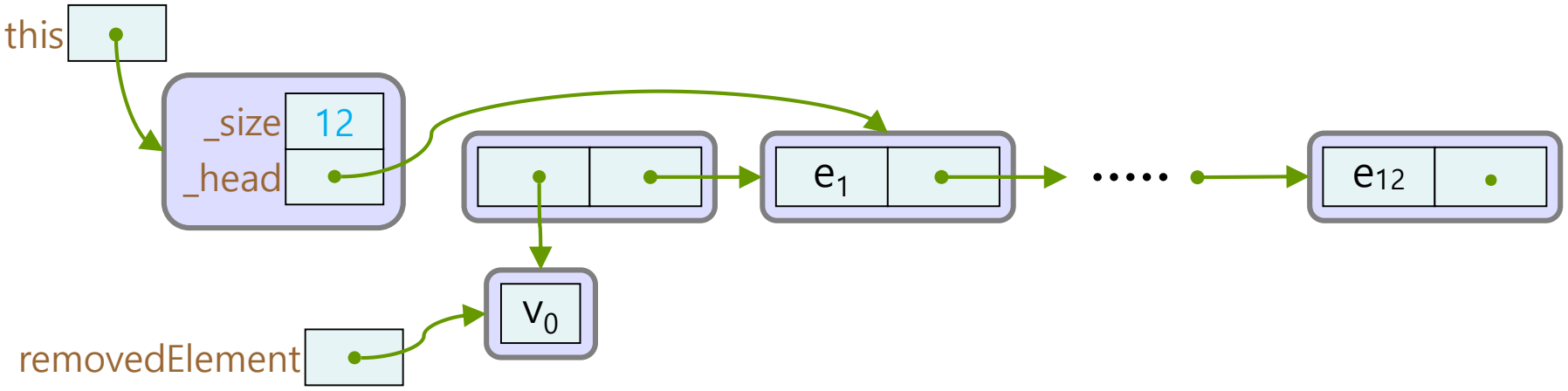
LinkedBag: removeAny()

```
// 내용 바꾸기
.....
public E  removeAny ()
{
    if ( this.isEmpty() ) {
        return  null ;
    }
    else {
        E  removedElement  =  this.head().element() ;
        this.setHead (this.head().next()) ;
        this.setSize (this.size()-1) ;
        return  removedElement ;
    }
}
```



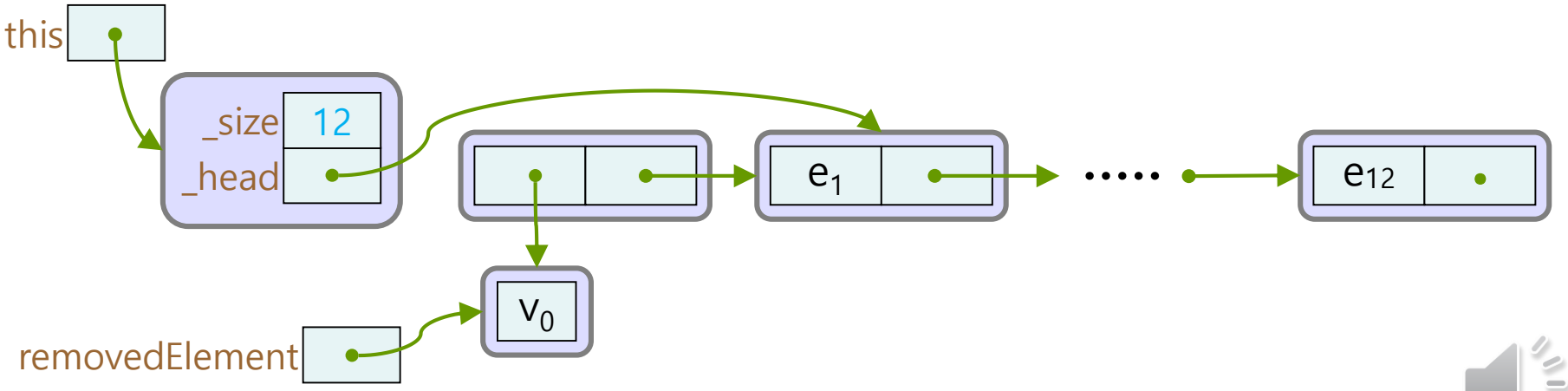
LinkedBag: removeAny()

```
// 내용 바꾸기
.....
public E  removeAny ()
{
    if ( this.isEmpty() ) {
        return  null ;
    }
    else {
        E  removedElement  =  this.head().element() ;
        this.setHead (this.head().next()) ;
        this.setSize (this.size()-1) ;
        return  removedElement ;
    }
}
```



LinkedBag: removeAny()

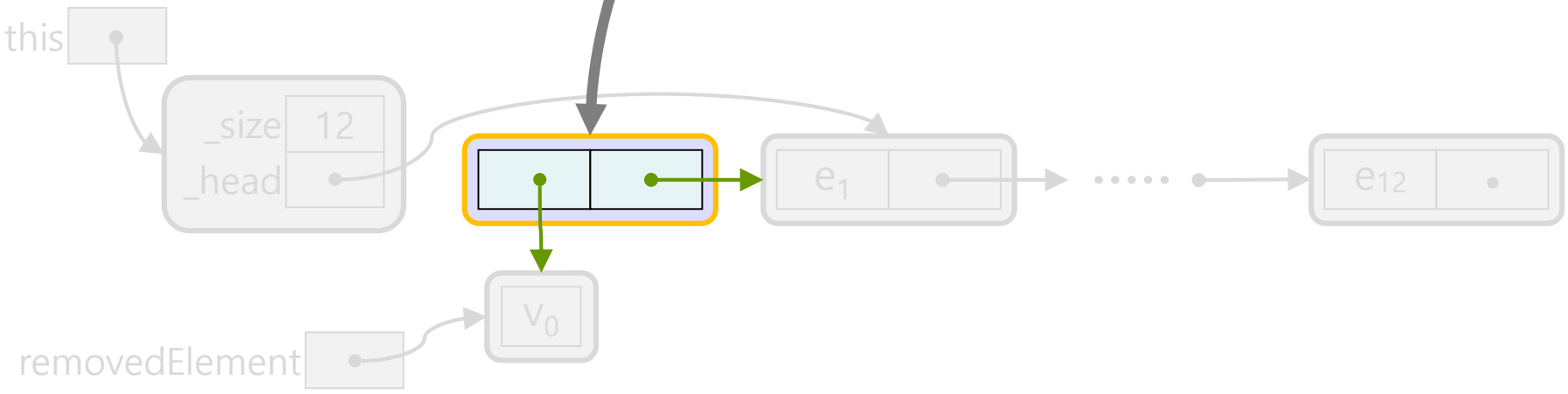
```
// 내용 바꾸기
.....
public E  removeAny ()
{
    if ( this.isEmpty() ) {
        return  null ;
    }
    else {
        E  removedElement  =  this.head().element() ;
        this.setHead (this.head().next()) ;
        this.setSize (this.size()-1) ;
        return  removedElement ;
    }
}
```



LinkedBag: removeAny()

```
// 내용 바꾸기
.....
public E   removeAny ()
{
    if ( this.isEmpty() ) {
        return  null ;
    }
    else {
        E  removedElement  =  thi
        this.setHead (this.head().
        this.setSize (this.size()-1)
        return  removedElement
    }
}
```

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

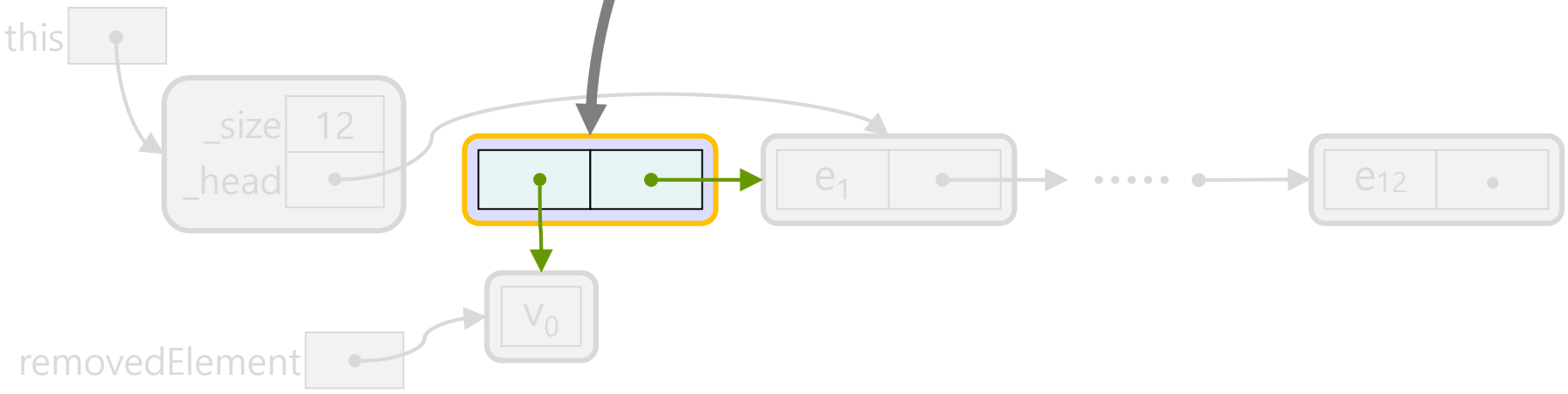


LinkedBag: removeAny()

```
// 내용 바꾸기
.....
public E   removeAny ()
{
    if ( this.isEmpty() ) {
        return  null ;
    }
    else {
        E  removedElement  =  thi
        this.setHead (this.head().
        this.setSize (this.size()-1)
        return  removedElement
    }
}
```

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

➡ "Garbage Collection"



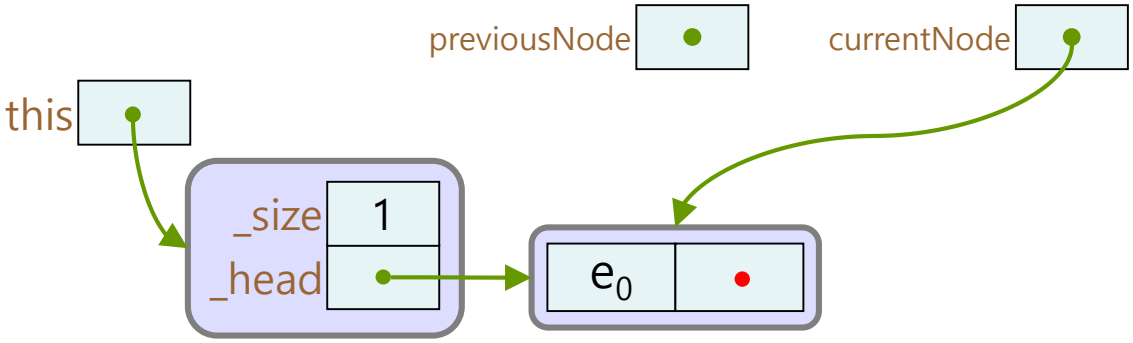
LinkedBag: remove()

```
public boolean  remove (E anElement)
{
        if ( this.isEmpty() ) {
                return  false ;
        }
        else {
                LinkedNode<E>  previousNode = null ;
                LinkedNode<E>  currentNode = this.head() ;
                boolean  found = false ;
                // 첫 번째 단계: 삭제할 위치 찾기
                while ( currentNode != null && !found ) {
                        if ( currentNode.element().equals(anElement) ) {
                                found = true ;
                        }
                        else {
                                previousNode = currentNode  ;
                                currentNode = currentNode.next() ;
                        }
                }

                // 두 번째 단계: 삭제하기
                if ( ! found ) {
                        return  false ;
                }
                else {
                        if ( currentNode == this.head() ) {
                                this.setHead (this.head().next()) ;
                        }
                        else {
                                previousNode.setNext(currentNode.next()) ;
                        }
                        this.setSize (this.size()-1) ;
                        return  true ;
                }
        }
}
```

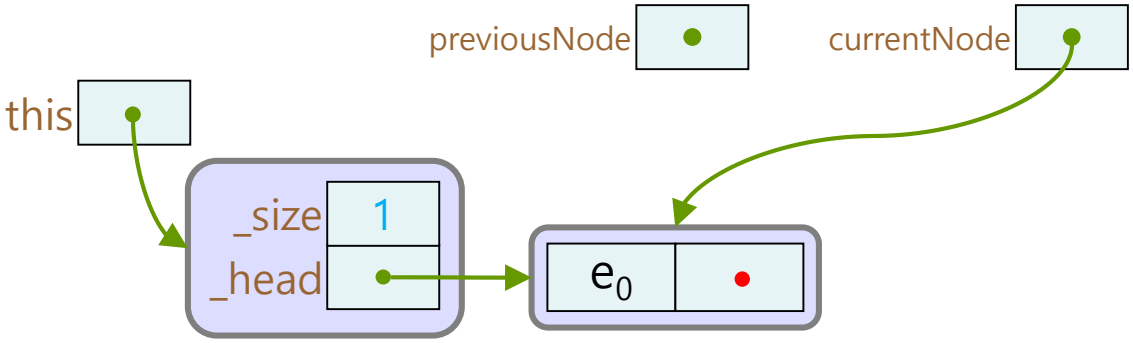
LinkedBag: remove() [single node]

```
public boolean  remove (E anElement)
{
    ......
        LinkedNode<E>  previousNode = null ;
        LinkedNode<E>  currentNode = this.head() ;
        boolean  found = false ;
         // 첫 번째 단계: 삭제할 위치 찾기
        while ( currentNode != null && !found ) {
            if ( currentNode.element().equals(anElement) ) {
                found = true ;
            }
            else {
                previousNode = currentNode  ;
                currentNode = currentNode.next() ;
            }
        }
        // 두 번째 단계: 삭제하기
    ......
```



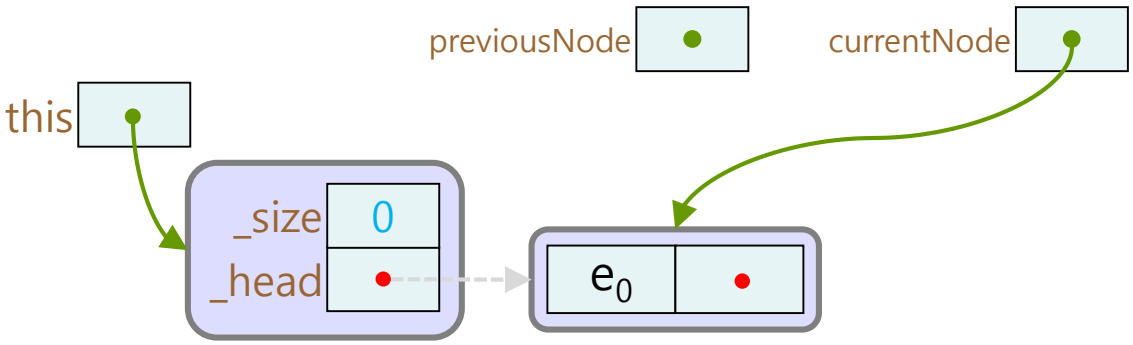
LinkedBag: remove() [single node]

```
public boolean  remove (E anElement)
{
    ......
        LinkedNode<E>  previousNode = null ;
        LinkedNode<E>  currentNode = this.head() ;
        boolean  found = false ;
        // 첫 번째 단계: 삭제할 위치 찾기
        while ( currentNode != null && !found) {
            if ( currentNode.element().equals(anElement) ) {
                found = true ;
            }
            else {
                previousNode = currentNode  ;
                currentNode = currentNode.next() ;
            }
        }
        // 두 번째 단계: 삭제하기
    ......
```



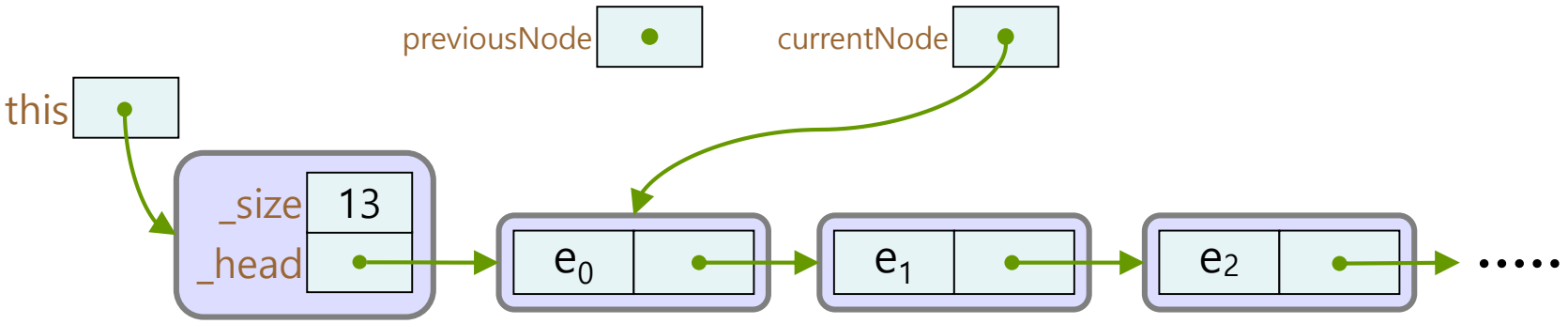
LinkedBag: remove() [single node]

```
public boolean  remove (E anElement)
{
    ……
        // 두 번째 단계: 삭제하기
        if (! found ) {
            return false ;
        }
        else {
            if ( currentNode == this.head() ) {
                this.setHead (this.head().next()) ;
            }
            else {
                previousNode.setNext(current.next()) ;
            }
            this.setSize (this.size()-1) ;
    ……
}
```



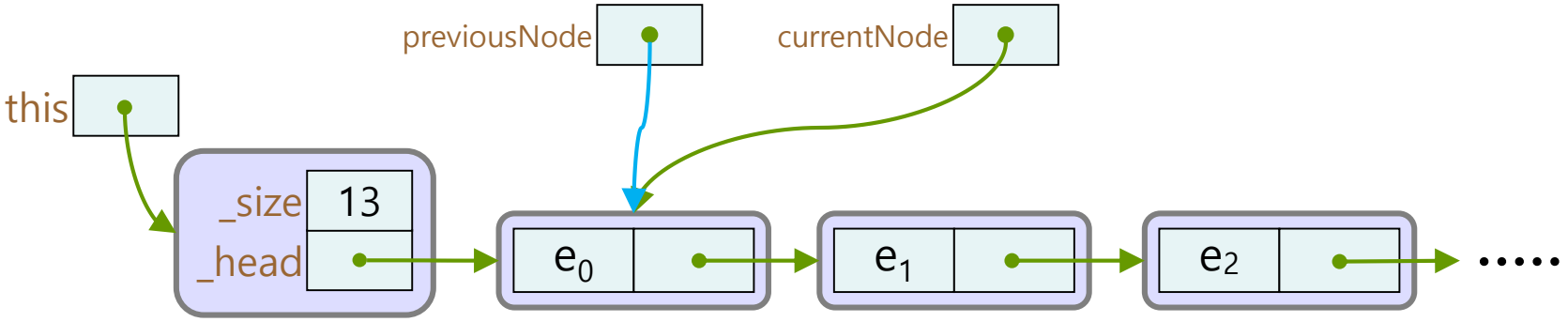
LinkedBag: remove() [multiple nodes]

```
public boolean  remove (E anElement)
{
    ……
        LinkedNode<E>  previousNode = null ;
        LinkedNode<E>  currentNode = this.head() ;
        boolean  found = false ;
        // 첫번째 단계: 삭제할 위치 찾기
        while ( currentNode != null && !found ) {
            if ( currentNode.element().equals(anElement) ) {
                found = true ;
            }
            else {
                previousNode = currentNode  ;
                currentNode = currentNode.next() ;
            }
        }
        // 두 번째 단계: 삭제하기
    ……
```



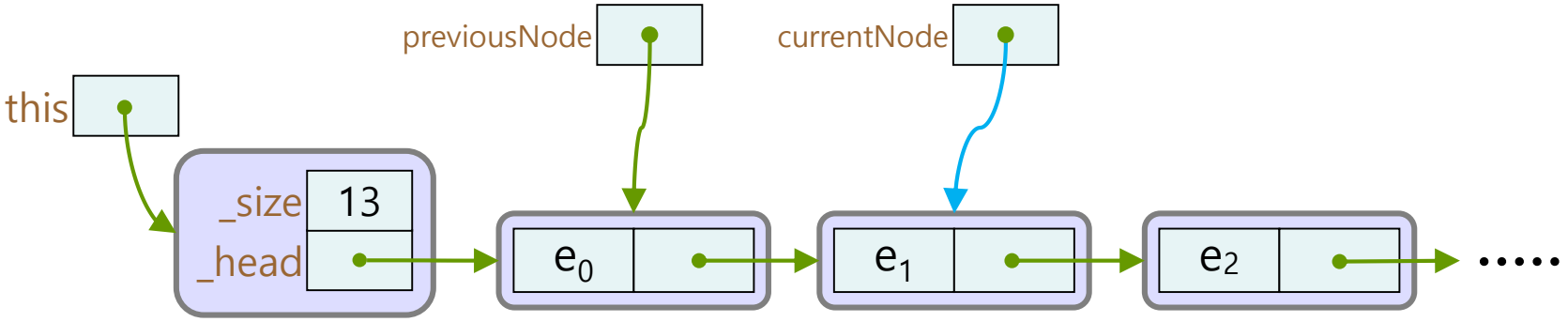
LinkedBag: remove() [multiple nodes]

```
public boolean  remove (E anElement)
{
    ……
        LinkedNode<E>  previousNode = null ;
        LinkedNode<E>  currentNode = this.head() ;
        boolean  found = false ;
        // 첫 번째 단계: 삭제할 위치 찾기
        while ( currentNode != null && !found ) {
            if ( currentNode.element().equals(anElement) ) {
                found = true ;
            }
            else {
                previousNode = currentNode  ;
                currentNode = currentNode.next() ;
            }
        }
        // 두 번째 단계: 삭제하기
    ……
```



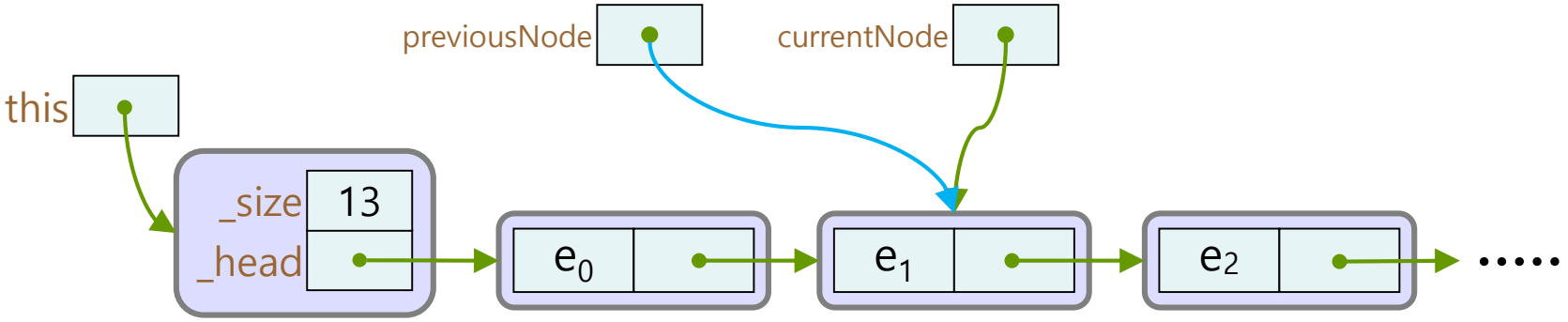
LinkedBag: remove() [multiple nodes]

```
public boolean  remove (E anElement)
{
    ……
        LinkedNode<E>  previousNode = null ;
        LinkedNode<E>  currentNode = this.head() ;
        boolean  found = false ;
        // 첫 번째 단계: 삭제할 위치 찾기
        while ( currentNode != null && !found ) {
            if ( currentNode.element().equals(anElement) ) {
                found = true ;
            }
            else {
                previousNode = currentNode ;
                currentNode = currentNode.next() ;
            }
        }
        // 두 번째 단계: 삭제하기
    ……
```



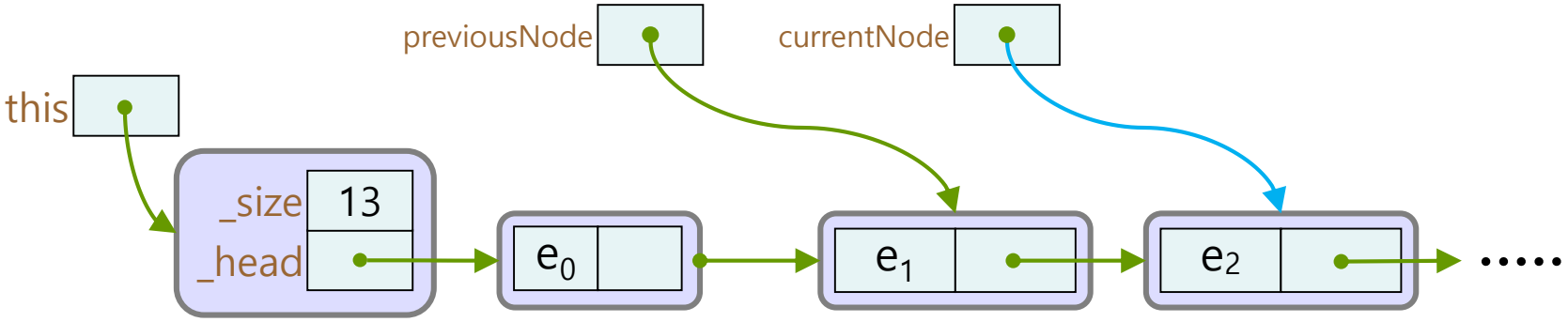
LinkedBag: remove() [multiple nodes]

```
public boolean  remove (E anElement)
{
    ……
        LinkedNode<E>  previousNode = null ;
        LinkedNode<E>  currentNode = this.head() ;
        boolean  found = false ;
        // 첫 번째 단계: 삭제할 위치 찾기
        while ( currentNode != null && !found) {
            if ( currentNode.element().equals(anElement) ) {
                found = true ;
            }
            else {
                previousNode = currentNode ;
                currentNode = currentNode.next() ;
            }
        }
        // 두 번째 단계: 삭제하기
    ……
```



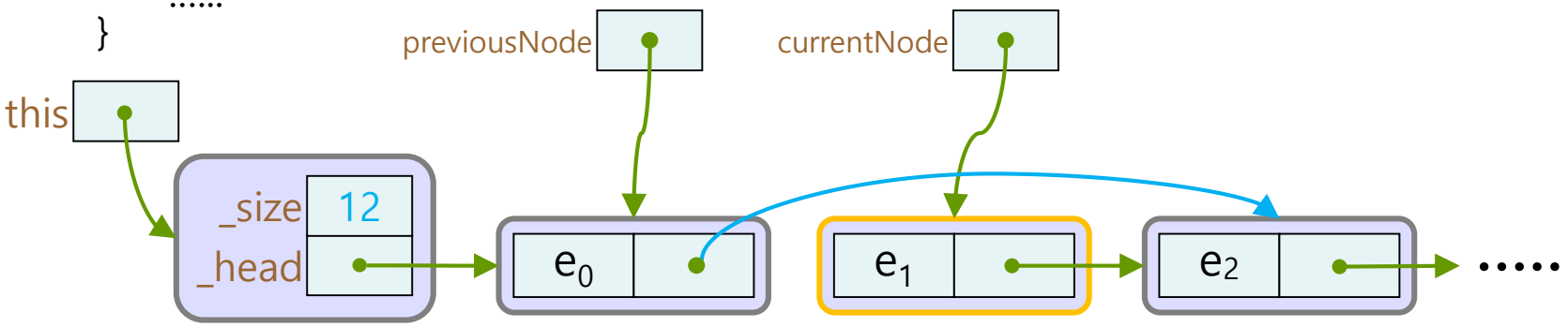
LinkedBag: remove() [multiple nodes]

```
public boolean  remove (E anElement)
{
    ......
        LinkedNode<E>  previousNode = null ;
        LinkedNode<E>  currentNode = this.head() ;
        boolean  found = false ;
        // 첫 번째 단계: 삭제할 위치 찾기
        while ( currentNode != null && !found ) {
            if ( currentNode.element().equals(anElement) ) {
                found = true ;
            }
            else {
                previousNode = currentNode ;
                currentNode = currentNode.next() ;
            }
        }
        // 두 번째 단계: 삭제하기
    ......
```



LinkedBag: remove() [multiple nodes]

```
public boolean  remove (Et anElement)
{
      ……
          // 두 번째 단계: 삭제하기
          if ( ! found ) {
                    return false ;
          }
          else {
               if (currentNode = this.head()) {
               {
                    this.setHead (this.head().next()) ;
               }
               else {
                    previous.setNext(current.next()) ;
               }
               this.setSize (this.size()-1) ;
                return true ;
          }
      ……
}
```



LinkedBag: clear()

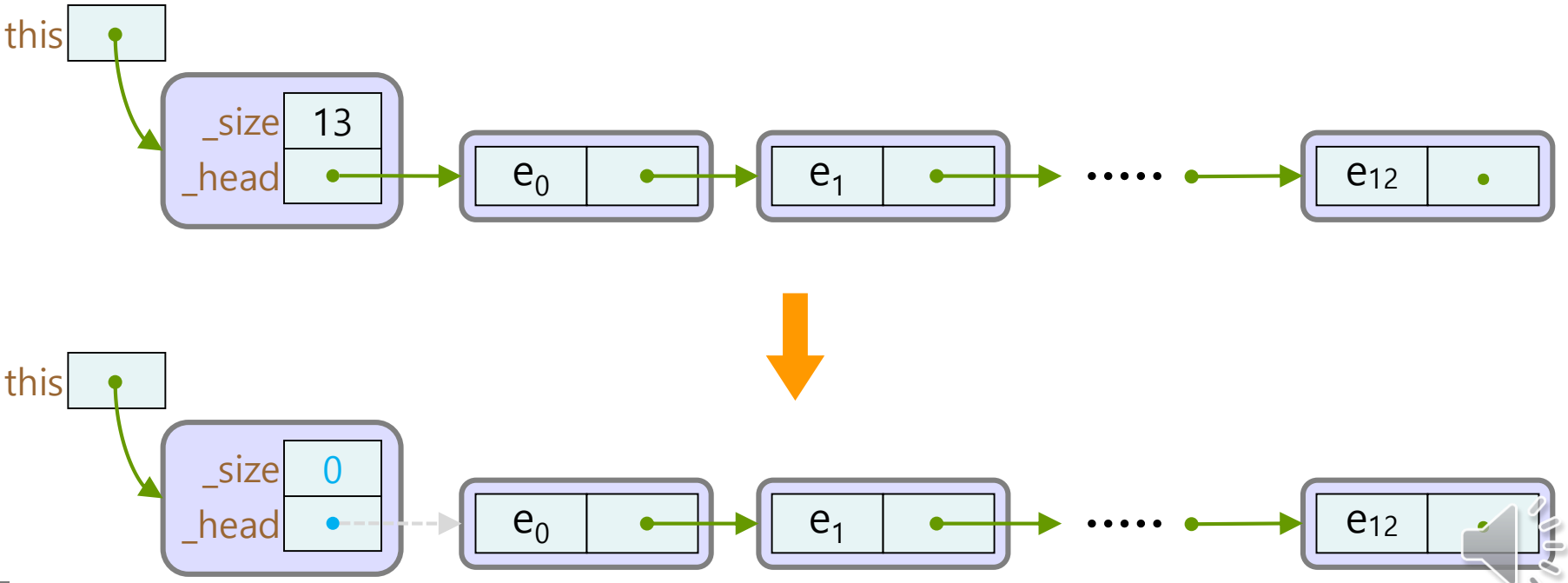
```
// 내용 바꾸기
.....

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

LinkedBag: clear()

```
// 내용 바꾸기
.....

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



연결 체인에서의 순차검색 (Sequential Search)

순차 검색: Version 1

```
public boolean  doesContain (E anElement)
{
    boolean  found = false ;
    LinkedNode<E>  currentNode = this.head() ;
    while ( currentNode != null && ! found ) {
        if ( currentNode.element().equals(anElement) ) {
            found = true ;
        }
        currentNode = currentNode.next() ;
    }
    return  found ;
}
```

순차 검색: Version 1 의 성능

```
public boolean  doesContain (E anElement)
{
    boolean  found = false ;
    LinkedNode<E>  currentNode = this.head() ;
    while ( currentNode != null && ! found ) {
        if ( currentNode.element().equals(anElement) ) {
            found = true ;
        }
        currentNode = currentNode.next() ;
    }
    return  found ;
}
```

- 시간은 얼마나 걸릴까?
  - this._size 의 값을 n 이라고 하면…

순차 검색: Version 2

```
public boolean  doesContain (E anElement)
{
    boolean  found = false ;
    LinkedNode<E>  currentNode = this.head() ;
    while ( currentNode != null && ! found )  {
        if ( currentNode.element().equals(anElement) ) {
            found = true ;
            return  true ;
        }
        currentNode = currentNode.next() ;
    }
    return  found ;
    return  false ;
}
```

- "Version 1" 과 "Version 2"
  - 시간적 성능의 차이는?
  - 각각의 장단점은?
  - 어느 코드가 더 이해하기 좋을까?
  - 결과가 true이든 false이든 (이 예제의 경우 return 하기 전에) 공통적으로 해야 할 일이 더 있다면?

순차 검색: Version 3

```
public boolean  doesContain (E anElement)
{
    boolean  found = false ;
    LinkedNode<E>  currentNode ;
    for (   currentNode = this.head() ;
            currentNode != null && ! found ;
            currentNode = currentNode.next() )
    {
        if ( currentNode.element().equals(anElement) ) {
            found = true ;
        }
    }
    return  found ;
}
```

```
// 순차 검색: Version 1
public boolean  doesContain (E anElement)
{
    boolean  found = false ;
    LinkedNode<E>  currentNode = this.head() ;
    while ( currentNode != null && ! found ) {
        if ( currentNode.element().equals(anElement) ) {
            found = true ;
        }
        currentNode = currentNode.next() ;
    }
    return  found ;
}
```

순차 검색: remove()

```
public boolean  remove (E anElement)
{
      if ( this.isEmpty() ) {
            return false ;
      }
      else if {
            LinkedNode<E>  previousNode = null ;
            LinkedNode<E>  currentNode = this.head() ;
            boolean  found = false ;

            // 첫 번째 단계: 삭제할 위치 찾기
            while ( currentNode != null && !found ) {
                  if ( currentNode.element().equals(anElement) ) {
                        found = true ;
                  }
                  else {
                        previousNode = currentNode  ;
                        currentNode = currentNode.next() ;
                  }
            }

            // 두 번째 단계: 삭제하기
            if ( ! found ) {
                  return  false ;
            }
            else {
                  if ( currentNode = this.head() ) {
                        this.setHead (this.head().next()) ;
                  }
                  else {
                        previousNode.setNext(currentNode.next()) ;
                  }
                  this.setSize (this.size()-1) ;
                  return  true ;
            }
      }
}
```

배열과 연결체인

배열과 연결 체인

- 배열에서의 문제점:
  - 연속된 (contiguous) 메모리 공간에 원소들을 저장
  - 임의의 원소를 삽입하거나 삭제하는데 많은 시간이 걸린다.
  - 최대 크기를 미리 지정

- 연결 체인의 장점:
  - 배열과 달리 원소들이 메모리에 연속적으로 모여 있지 않아도 된다. 아무 곳이나 가능하다.
  - 저장할 원소의 개수에 제한을 받지 않는다.
  - 원소를 삽입/삭제 하는데 배열보다 유리하다.

ArrayBag 구현의 문제점 [1]

- Bag 을 Java 언어로 구현하는 손쉬운 방법은 배열 (array) 을 사용하는 것이다.
  - 예: 최대 100 개 동전 가방
    - ArrayBag<Coin> coinBag = new ArrayBag<Coin>() ;

```
public class ArrayBag<E>
{
    // 비공개 인스턴스 변수
    private int          _size ;
    private E[100]       _elements ;
```

ArrayBag 구현의 문제점 [2]

- 어떤 점이 불편한가?
  - 처음에는 동전 수가 최대 100 개면 충분했는데, 나중에 120 개로 늘어난다면?
    - 실행 중에는 대처 불능
      - 현재의 실행 프로그램으로는 120 개의 상황이 발생하면 실행이 불가능하다.
    - 사후 대처 방법
      - 프로그램 원시 코드에서 배열의 크기를 100 에서 120 으로 바꾸어주고, 다시 컴파일 하여, 새로운 실행 프로그램을 얻는다.
    - 동적 대응 방법
      - 실행 중에 배열의 크기를 확장
      - 원래 배열에 들어있는 데이터의 크기에 비례한 시간이 걸림
        - 원래 배열의 데이터를 새로 확장한 배열로 복사를 해야 하는 시간이 소요된다

ArrayBag 구현의 문제점 [2]

- 어떤 점이 불편한가? [계속]
  - 새로운 원소를 배열의 맨 앞에 삽입해야 한다면?
    - 즉 0 번째 위치에 삽입해야 한다면, 기존의 모든 원소를 맨 뒤부터 하나씩 뒤로 밀고 0 번째 위치를 비운 다음 삽입할 수 있다.

```
for ( i = this.size() ; i > 0 ; i-- ) {
    this._elements[i] = this._elements[i-1] ;
}
this._elements[0] = addedElement ;
this._size ++ ;
```

    - 리스트에 원소의 개수가 많아지면 그 수에 비례해 시간이 많이 걸리게 된다.
  - 배열의 맨 앞의 원소를 삭제하려면?
    - 즉 0 번째 원소를 삭제해야 한다면, 기존의 모든 원소를 맨 앞에서부터 하나씩 앞으로 당겨야 한다.

```
E  removedElement = this._elements[0] ;
for ( i = 1 ; i < this.size() ; i++ ) {
    this._elements[i-1] = this._elements[i] ;
}
this._size -- ;
```

    - 배열에 원소의 개수가 많아지면 그 수에 비례해 시간이 많이 걸리게 된다.

ArrayBag 구현의 문제점 [3]

- 어떤 점이 불편한가? [계속]
  - 새로운 원소를 k 번째 위치에 삽입하려면?

```
for ( i = this.size() ; i > k ; i-- ) {
    this._elements[i] = this._elements[i-1] ;
}
this._elements[k] = addedElement ;
```

    - (n-k) 개의 원소를 한 칸씩 뒤로 이동시켜야 한다.
  - k 번째 위치의 원소를 삭제하려면?

```
E  removedElement = this._elements[k] ;
for ( i = k+1 ; i < this.size() ; i++ ) {
    this._elements[i-1] = this._elementss[i] ;
}
```

    - (n-k-1) 개의 원소를 한 칸씩 앞으로 이동시켜야 한다.
  - 결국, 삽입이나 삭제의 경우 전체 리스트의 길이가 길어지면 그만큼 해야 할 일이 늘어난다.

ArrayBag 구현의 문제점 [4]

- 장점은?
  - 프로그램에서 다루기가 간편한 편이다.
    - 프로그램 코드 작성이 상대적으로 단순하며, 이해하기도 쉬운 편이다.
    - 이는 연결 체인을 이용한 구현과 비교된다.
  - 메모리 활용이 효율적일 수 있다.
    - 동일한 양의 자료를 저장할 경우, 메모리의 사용량이 연결 체인을 이용한 구현에 비해 적다.
      - 노드를 연결하는 연결 정보가 배열에서는 필요 없게 된다.

End of

"Linked Bag (2)"