4. 컬렉션 프레임워크 - LinkedList

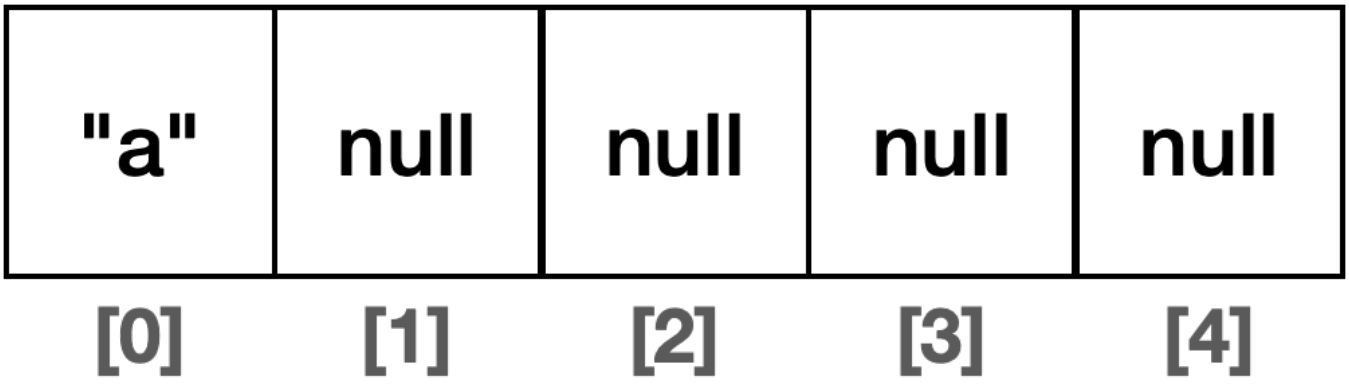
#0.강의/1.자바로드맵/4.자바-중급2편

노드와 연결1

배열 리스트의 단점

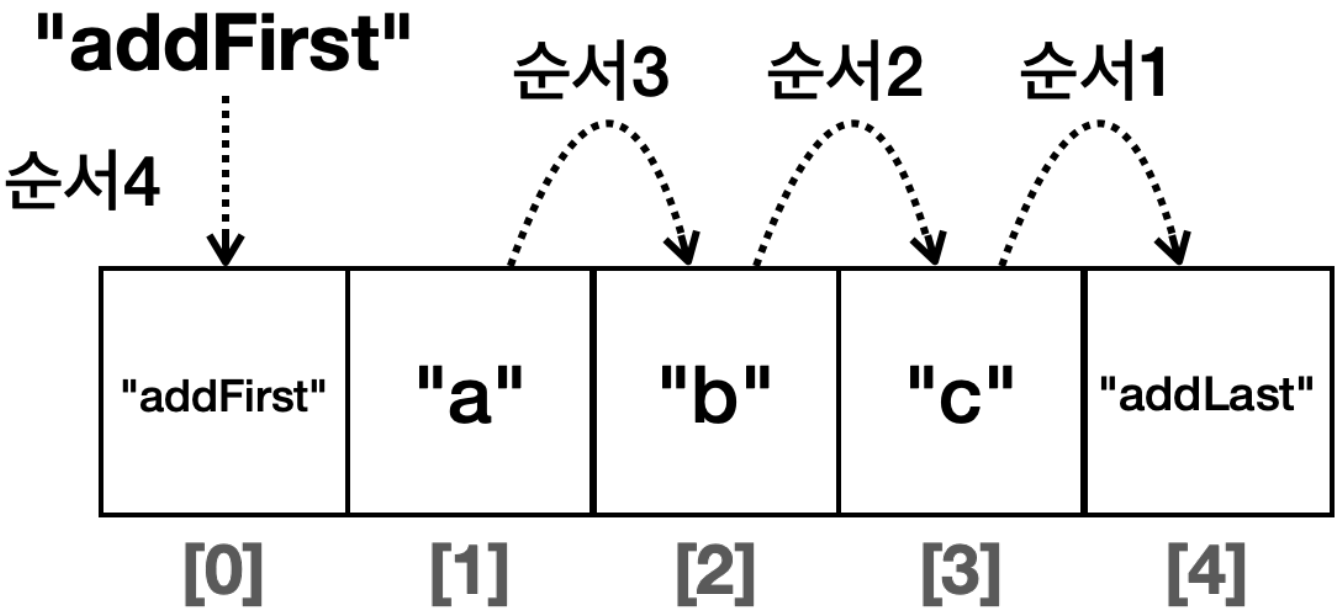
배열 리스트는 내부에 배열을 사용해서 데이터를 보관하고 관리한다. 이로 인해 다음과 같은 단점을 가진다.

배열의 사용하지 않는 공간 낭비



배열은 필요한 배열의 크기를 미리 확보해야 한다. 데이터가 얼마나 추가될지 예측할 수 없는 경우 나머지는 공간은 사용되지 않고 낭비된다.

배열의 중간에 데이터 추가



배열의 앞이나 중간에 데이터를 추가하면 추가할 데이터의 공간을 확보하기 위해 기존 데이터들을 오른쪽으로 이동해야 한다. 그리고 삭제의 경우에는 빈 공간을 채우기 위해 왼쪽으로 이동해야 한다.
이렇게 앞이나 중간에 데이터를 추가하거나 삭제하는 경우 많은 데이터를 이동해야 하기 때문에 성능이 좋지 않다.

노드와 연결

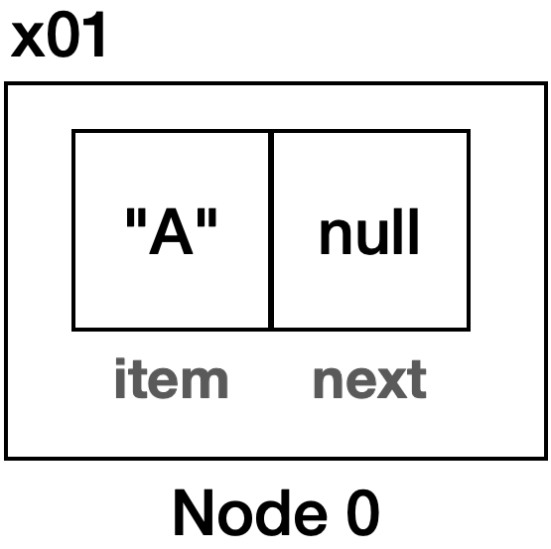
낭비되는 메모리 없이 딱 필요한 만큼만 메모리를 확보해서 사용하고, 또 앞이나 중간에 데이터를 추가하거나 삭제할 때도 효율적인 자료 구조가 있는데, 바로 노드를 만들고 각 노드를 서로 연결하는 방식이다.

노드 클래스

```
public class Node {
    Object item;
    Node next;
}
```

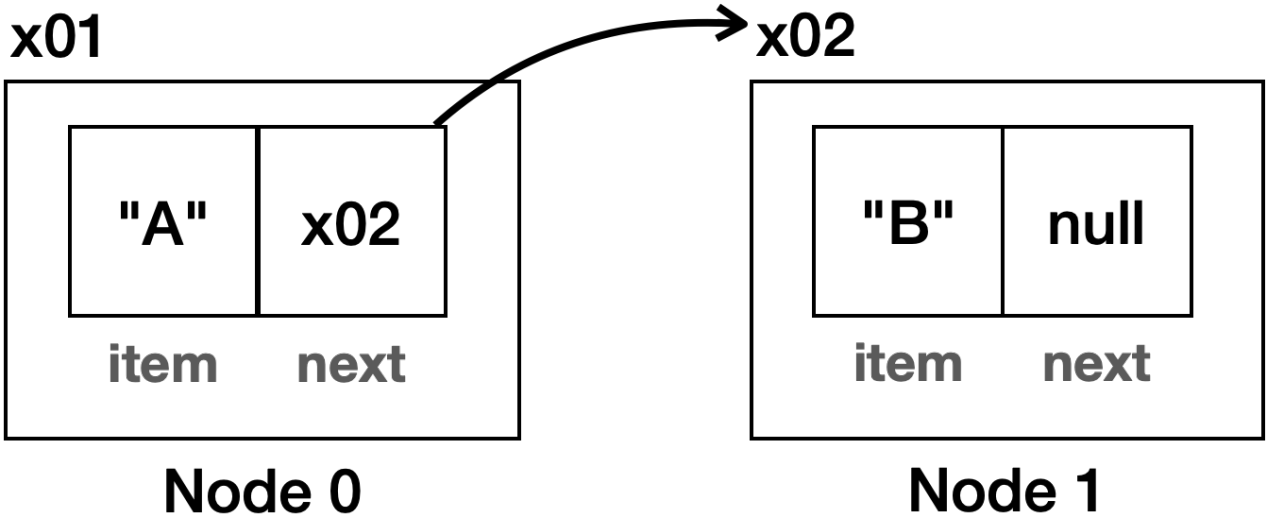
노드 클래스는 내부에 저장할 데이터인 `item`과, 다음으로 연결할 노드의 참조인 `next`를 가진다.
이 노드 클래스를 사용해서 데이터 A, B, C를 순서대로 연결해보자.

노드에 데이터 A 추가

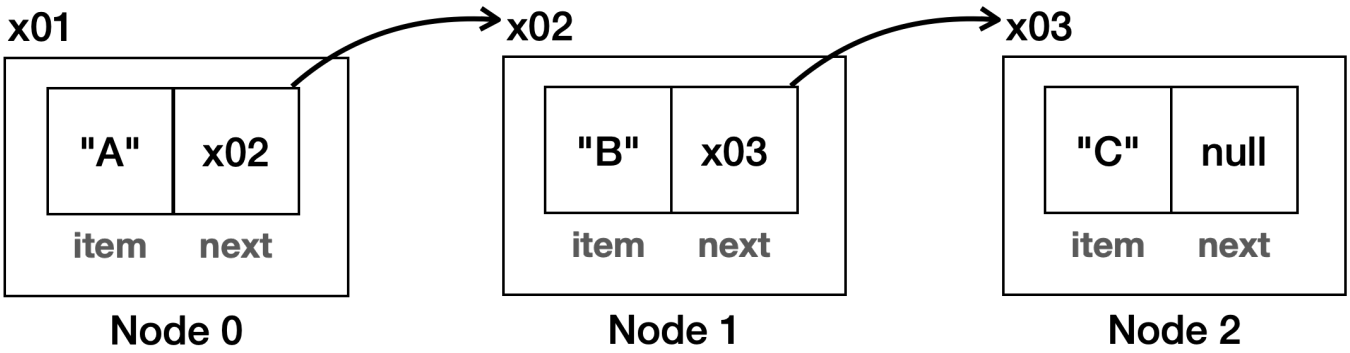


- `Node` 인스턴스를 생성하고 `item`에 "A"를 넣어준다.

노드에 데이터 B 추가



- `Node` 인스턴스를 생성하고 `item`에 "B"를 넣어준다.
- 처음 만든 노드의 `Node next` 필드에 새로 만든 노드의 참조값을 넣어준다.
- 이렇게하면 첫 번째 노드와 두 번째 노드가 서로 연결된다.
- 첫 번째 노드의 `node.next`를 호출하면 두 번째 노드를 구할 수 있다.



- `Node` 인스턴스를 생성하고 `item`에 "C"를 넣어준다.

- 두 번째 만든 노드의 `Node next` 필드에 새로 만든 노드의 참조값을 넣어준다.
- 이렇게하면 두 번째 노드와 세 번째 노드가 서로 연결된다.
- 첫 번째 노드의 `node.next`를 호출하면 두 번째 노드를 구할 수 있다.
- 두 번째 노드의 `node.next`를 호출하면 세 번째 노드를 구할 수 있다.
- 첫 번째 노드의 `node.next.next`를 호출하면 세 번째 노드를 구할 수 있다.

지금까지 설명한 내용을 코드로 직접 만들어보자.

```
package collection.link;

public class Node {

    Object item;
    Node next;

    public Node(Object item) {
        this.item = item;
    }

}
```

- 필드의 접근 제어자는 `private`으로 선언하는 것이 좋지만, 이 예제에서는 설명을 단순하게 하기 위해 디폴트 접근 제어자를 사용했다.

```
package collection.link;

public class NodeMain1 {

    public static void main(String[] args) {
        //노드 생성하고 연결하기: A -> B -> C
        Node first = new Node("A");
        first.next = new Node("B");
        first.next.next = new Node("C");

        System.out.println("모든 노드 탐색하기");
        Node x = first;
        while (x != null) {
            System.out.println(x.item);
            x = x.next;
        }
```

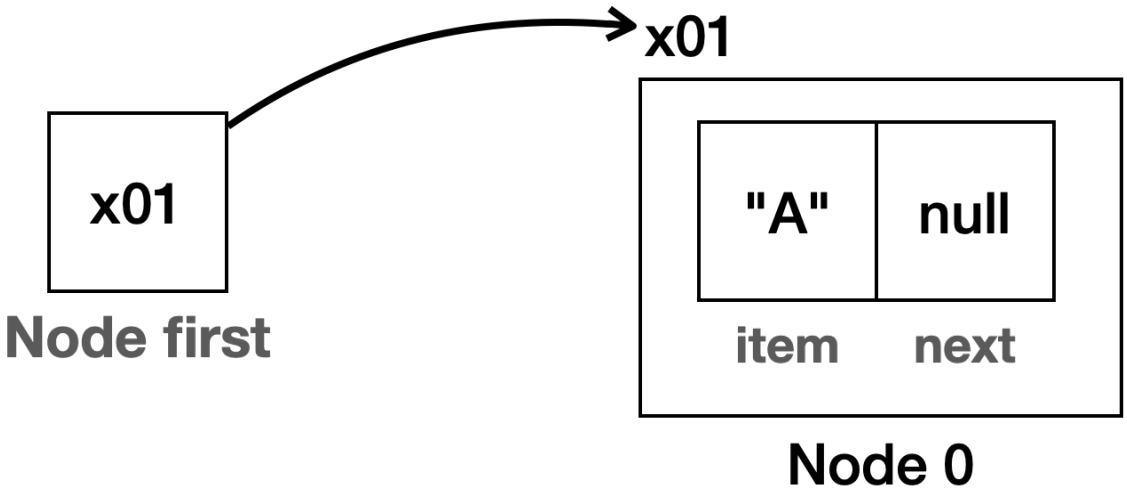
```
    }

}
```

실행 결과

```
모든 노드 탐색하기
A
B
C
```

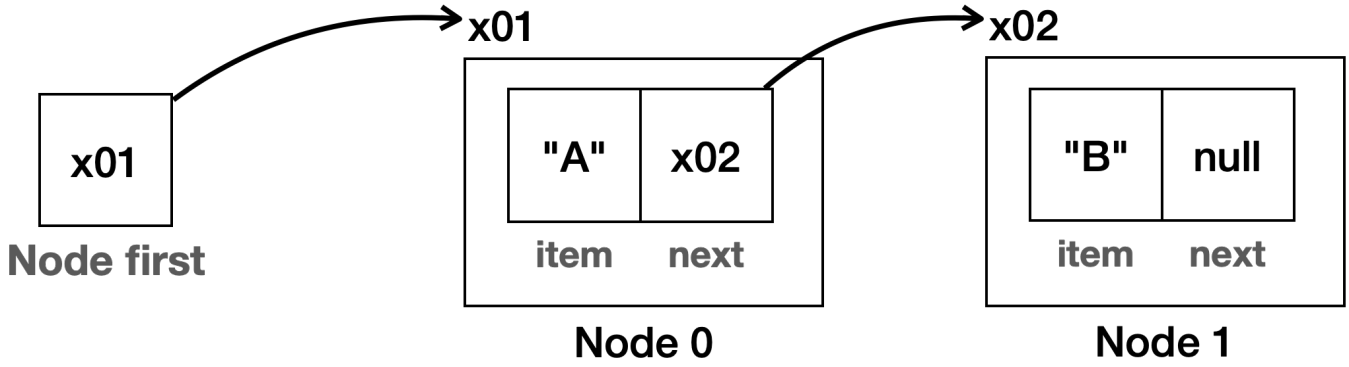
코드를 분석해보자.

노드 연결하기



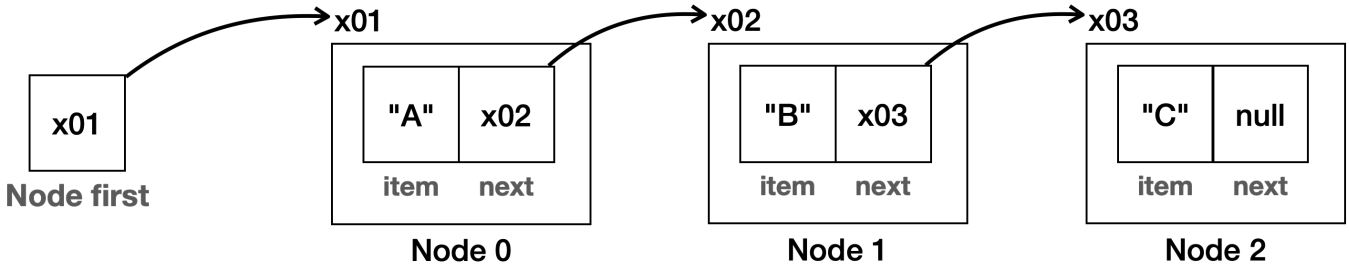
- `1. Node first = new Node("A")`
  - `Node` 인스턴스를 생성하고 `item`에 "A"를 넣어준다.
- `2. Node first = x01`
  - `first` 변수에 생성된 노드의 참조를 보관한다. `first` 변수는 이름 그대로 첫 번째 노드의 참조를 보관한다.

노드에 데이터 B 추가



- `1. first.next = new Node("B")`
  - `Node` 인스턴스를 생성하고 `item`에 "B"를 넣어준다.
- `2. first.next = x02`
  - 처음 만든 노드의 `next` 필드에 새로 만든 노드의 참조값을 넣어준다.
  - 이렇게하면 첫 번째 노드와 두 번째 노드가 서로 연결된다.

노드에 데이터 C 추가



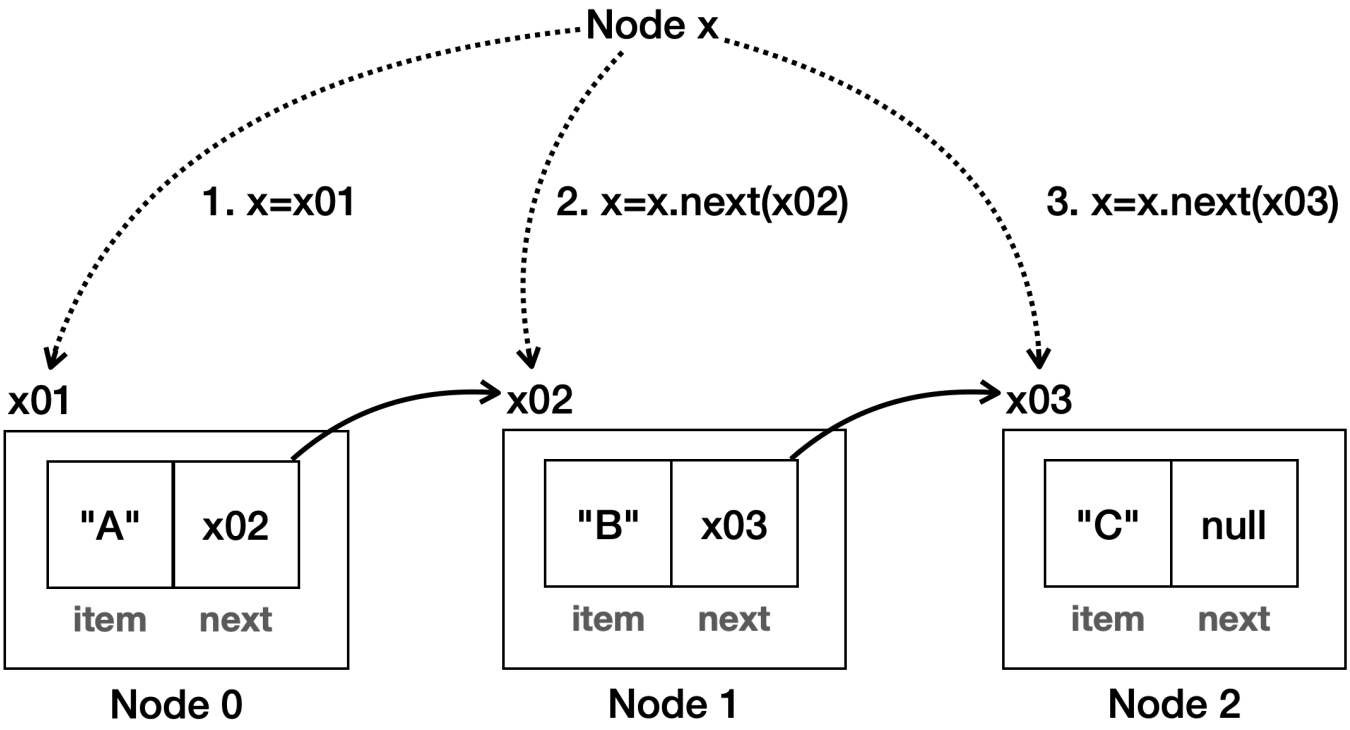
- `1. first.next.next = new Node("C")`
- `2. first.next.next = x03`
- `3. x01.next.next = x03`
- `4. x02.next = x03`
  - 두 번째 만든 노드의 `Node next` 필드에 새로 만든 노드의 참조값을 넣어준다.
- 이렇게하면 두 번째 노드와 세 번째 노드가 서로 연결된다.

연결된 노드를 찾는 방법

- `Node first`를 통해 첫 번째 노드를 구할 수 있다.
- 첫 번째 노드의 `node.next`를 호출하면 두 번째 노드를 구할 수 있다.
- 두 번째 노드의 `node.next`를 호출하면 세 번째 노드를 구할 수 있다.
- 첫 번째 노드의 `node.next.next`를 호출하면 세 번째 노드를 구할 수 있다.

모든 노드 탐색하기

모든 노드를 탐색하는 방법을 알아보자.




```
Node x = first;
while (x != null) {
    System.out.println(x.item);
    x = x.next;
}
```

```
A
B
C
```

순서대로 분석해보자.

`Node x`는 처음 노드부터 순서대로 이동하면서 모든 노드를 가리킨다. 처음에 `Node x`는 `x01`을 참조한다. 그리고 while 문을 통해 반복해서 다음 노드를 가리킨다. while 문은 다음 노드가 없을 때 까지 반복한다. `Node.next`의 참조값이 `null`이면 노드의 끝이다.

반복 실행 순서

1. `Node x = x01`
    1. `x.item` 출력 → "A"
    2. `x = x.next(x02)`

2. `Node x = x02`
    1. `x.item` 출력 ➜ "B"
    2. `x = x.next(x03)`
3. `Node x = x03`
    1. `x.item` 출력 ➜ C
    2. `x = x.next(null)`
4. `Node x = null`
    1. `while(x != null)` 조건에서 `x=null` 이므로 종료

노드와 연결2

toString() - IDE

노드의 연결 상태를 더 편하게 보기 위해 `toString()`을 오버라이딩 해보자.

먼저 IDE의 도움을 받아서 구현해보자.

```
package collection.link;

public class Node {

    Object item;
    Node next;

    public Node(Object item) {
        this.item = item;
    }

    //IDE 생성 toString()
    @Override
    public String toString() {
        return "Node{" +
                "item=" + item +
                ", next=" + next +
                '}';
    }

}
```

```
package collection.link;

public class NodeMain2 {

    public static void main(String[] args) {
        //노드 생성하고 연결하기: A -> B -> C
        Node first = new Node("A");
        first.next = new Node("B");
        first.next.next = new Node("C");

        System.out.println("연결된 노드 출력하기");
        System.out.println(first);
    }
}
```

실행 결과

```
연결된 노드 출력하기
Node{item=A, next=Node{item=B, next=Node{item=C, next=null}}}
```

IDE의 도움을 받아서 만든 `toString()`으로 필요한 정보를 확인할 수는 있지만, 한눈에 보기에는 좀 복잡하다.
대신에 `[A->B->C]`와 같이 필요한 정보만 편리하게 확인할 수 있게 `toString()`을 직접 구현해보자.

toString() - 직접 구현

`[A->B->C]`와 같이 데이터와 연결 구조를 한눈에 볼 수 있도록 `toString()`을 직접 구현해보자.

```
package collection.link;

public class Node {

    Object item;
    Node next;

    public Node(Object item) {
        this.item = item;
```

```
    }

/*
    //IDE 생성 toString()
    @Override
    public String toString() {
        return "Node{" +
                "item=" + item +
                ", next=" + next +
                '}';
    }
*/

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        Node x = this;
        sb.append("[");
        while (x != null) {
            sb.append(x.item);
            if (x.next != null) {
                sb.append("->");
            }
            x = x.next;
        }
        sb.append("]");
        return sb.toString();
    }
}
```

- 이전에 작성한 `toString()`을 주석처리 한다.
- 직접 만든 `toString()`은 연결된 모든 노드를 탐색해서 출력하고 `[A->B->C]`와 같이 출력한다.
- 반복문 안에서 문자를 더하기 때문에 `StringBuilder`를 사용하는 것이 효과적이다.
- 구현은 앞서 살펴본 모든 노드 탐색하기와 같다. while을 사용해서 다음 노드가 없을 때 까지 반복한다.
  - `x = x.next`: 탐색의 위치를 다음으로 이동한다. 현재 탐색중인 노드가 `x`이다. `x.next`를 통해 `x`의 참조값을 다음 노드로 변경한다.

NodeMain2 - 실행 결과

```
연결된 노드 출력하기
[A->B->C]
```

이제 내부 데이터와 연결 구조를 깔끔하게 볼 수 있다.

노드와 연결3

노드와 연결을 활용해서 다양한 기능을 만들어보자.

- 모든 노드 탐색하기
- 마지막 노드 조회하기
- 특정 index의 노드 조회하기
- 노드에 데이터 추가하기

```
package collection.link;

public class NodeMain3 {

    public static void main(String[] args) {
        //노드 생성하고 연결하기: A -> B -> C
        Node first = new Node("A");
        first.next = new Node("B");
        first.next.next = new Node("C");

        System.out.println(first);

        //모든 노드 탐색하기
        System.out.println("모든 노드 탐색하기");
        printAll(first);

        //마지막 노드 조회하기
        Node lastNode = getLastNode(first);
        System.out.println("lastNode = " + lastNode);

        //특정 index의 노드 조회하기
        int index = 2;
        Node index2Node = getNode(first, index);
        System.out.println("index2Node = " + index2Node.item);

        //데이터 추가하기
```

```
        System.out.println("데이터 추가하기");
        add(first, "D");
        System.out.println(first);
        add(first, "E");
        System.out.println(first);
        add(first, "F");
        System.out.println(first);
    }

    private static void printAll(Node node) {
        Node x = node;
        while (x != null) {
            System.out.println(x.item);
            x = x.next;
        }
    }

    private static Node getLastNode(Node node) {
        Node x = node;
        while (x.next != null) {
            x = x.next;
        }
        return x;
    }

    private static Node getNode(Node node, int index) {
        Node x = node;
        for (int i = 0; i < index; i++) {
            x = x.next;
        }
        return x;
    }

    private static void add(Node node, Object param) {
        Node lastNode = getLastNode(node);
        lastNode.next = new Node(param);
    }
}
```

참고

- 영상에서 `getNode()` 코드를 작성할 때, `getLastNode()`의 코드가 잠깐 다르게 보일 수 있습니다. 이때는 메뉴얼의 코드를 따라하시면 됩니다.

- 영상과 다르게 `add()` 메서드의 매개변수를 `String param` ➜ `Object param`으로 변경했습니다. (이렇게 변경하는 것이 더 범용성이 있습니다.)

실행 결과

```
[A->B->C]
모든 노드 탐색하기
A
B
C
lastNode = [C]
index2Node = C
데이터 추가하기
[A->B->C->D]
[A->B->C->D->E]
[A->B->C->D->E->F]
```

모든 노드 탐색하기

`printAll(Node node)` : 다음 노드가 없을 때 까지 반복해서 노드의 데이터를 출력한다.

마지막 노드 조회하기

`Node getLastNode(Node node)` : 마지막 노드를 조회한다.

`Node.next`의 참조값이 `null`이면 노드의 끝이다.

`getLastNode()`는 노드를 순서대로 탐색하면서 `Node.next`의 참조값이 `null`인 노드를 찾아서 반환한다.

여기서는 마지막에 있는 C 값을 가지고 있는 노드가 출력된다.

```
lastNode = [C]
```

특정 index의 노드 조회하기

`getNode(Node node, int index)` : index로 특정 위치의 노드를 찾는다.

`x = x.next`를 호출할 때 마다 x가 참조하는 노드의 위치가 순서대로 하나씩 증가한다.

`index`의 수 만큼만 반복해서 이동하면 원하는 위치의 노드를 찾을 수 있다.

- `index 0` : A
- `index 1` : B

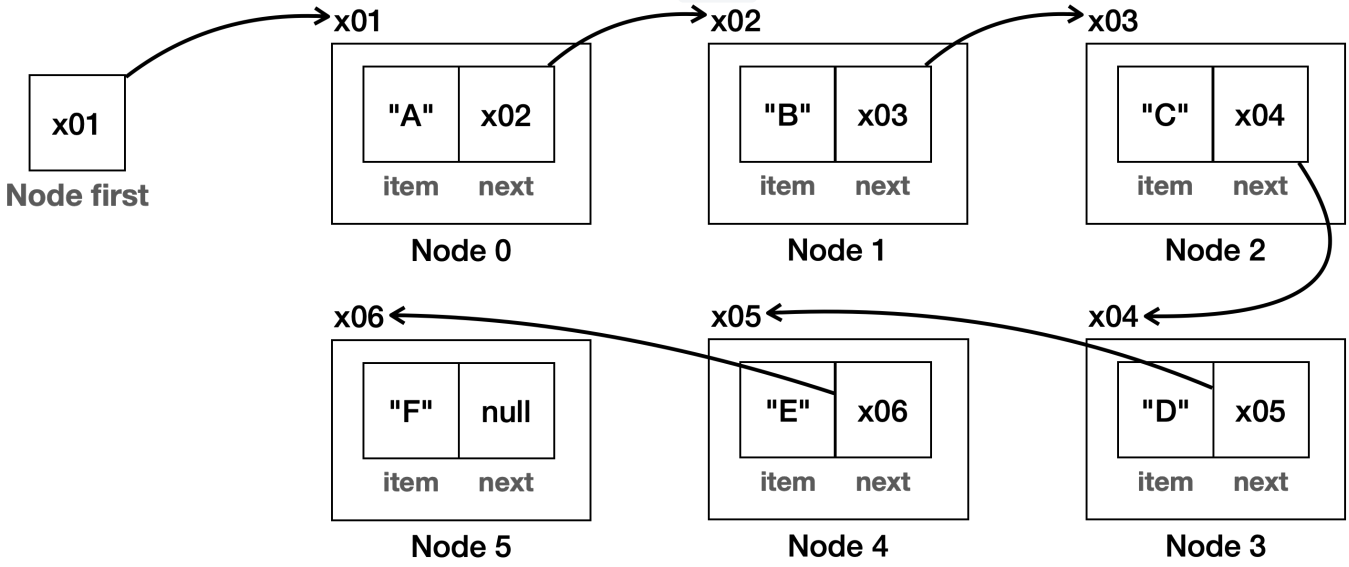
- `index 2` : C

데이터 추가하기

```
void add(Node node, Object param) {
    Node lastNode = getLastNode(node);
    lastNode.next = new Node(param);
}
```

데이터를 추가할 때는 새로운 노드를 만들고, 마지막 노드에 새로 만든 노드를 연결하면 된다.

순서대로 설명하면 먼저 마지막 노드를 찾고, 마지막 노드의 `next`에 새로운 노드를 연결하면 된다.



정리

- 노드는 내부에 데이터와 다음 노드에 대한 참조를 가지고 있다.
- 지금까지 설명한 구조는 각각의 노드가 참조를 통해 연결(Link, 링크) 되어 있다.
- 데이터를 추가할 때 동적으로 필요한 만큼의 노드만 만들어서 연결하면 된다. 따라서 배열과 다르게 메모리를 낭비하지 않는다.
    - 물론 `next` 필드를 통해 참조값을 보관해야 하기 때문에 배열과 비교해서 추가적인 메모리 낭비도 발생한다.
- 노드와 연결 구조를 처음 공부하면 이해하기 쉽지않다. 하지만 반복해서 학습하고 또 코드도 직접 만들어보면서 반드시 이해해야 한다. 이해가 어렵다면 코드를 따라서 종이에 천천히 노드를 그려보면 이해가 될 것이다.
- 이렇게 각각의 노드를 연결(링크)해서 사용하는 자료 구조로 리스트를 만들 수 있는데, 이것을 연결 리스트라 한다.

직접 구현하는 연결 리스트1 - 시작

우리는 앞서 배열을 통해서 리스트를 만들었는데 이것을 배열 리스트(`ArrayList`)라 한다.
이번에는 배열이 아닌 앞서 학습한 노드와 연결 구조를 통해서 리스트를 만들어보자. 이런 자료 구조를 연결 리스트(`LinkedList`)라 한다. 참고로 링크드 리스트, 연결 리스트라는 용어를 둘다 사용한다.

연결 리스트는 배열 리스트의 단점인, 메모리 낭비, 중간 위치의 데이터 추가에 대한 성능 문제를 어느정도 극복할 수 있다.

리스트 자료 구조
순서가 있고, 중복을 허용하는 자료 구조를 리스트(List)라 한다.

우리는 앞서 `MyArrayList` 시리즈를 만들어보았다. 배열 리스트도, 연결 리스트도 모두 같은 리스트이다. 리스트의 내부에서 배열을 사용하는가 아니면 노드와 연결 구조를 사용하는가의 차이가 있을 뿐이다.
배열 리스트를 사용하든 연결 리스트를 사용하든 둘다 리스트 자료 구조이기 때문에 리스트를 사용하는 개발자 입장에서는 거의 비슷하게 느껴져야 한다. 쉽게 이야기해서 리스트를 사용하는 개발자 입장에서 `MyArrayList`를 사용하든 `MyLinkedList`를 사용하든 내부가 어떻게 돌아가지는 몰라도, 그냥 순서가 있고, 중복을 허용하는 자료 구조구나 생각하고 사용할 수 있어야 한다.

연결 리스트를 직접 구현해보자.

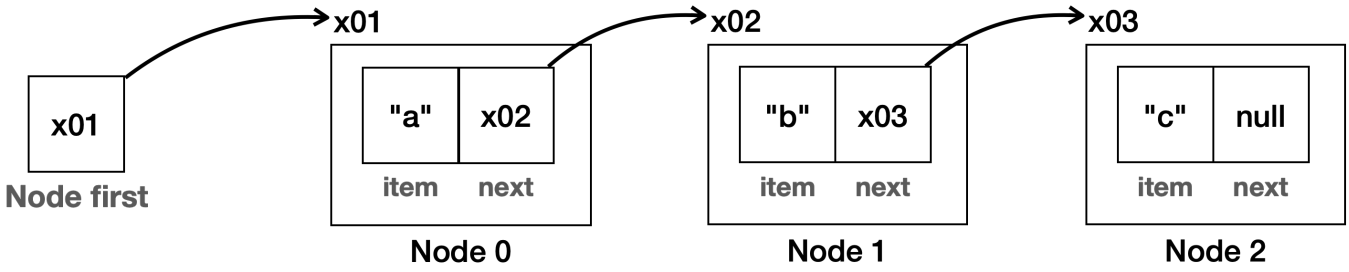
```
package collection.link;

public class MyLinkedListV1 {

    private Node first;
    private int size = 0;

    public void add(Object e) {
        Node newNode = new Node(e);
        if (first == null) {
            first = newNode;
        } else {
            Node lastNode = getLastNode();
            lastNode.next = newNode;
        }
        size++;
```

```
    }

    private Node getLastNode() {
        Node x = first;
        while (x.next != null) {
            x = x.next;
        }
        return x;
    }

    public Object set(int index, Object element) {
        Node x = getNode(index);
        Object oldValue = x.item;
        x.item = element;
        return oldValue;
    }

    public Object get(int index) {
        Node node = getNode(index);
        return node.item;
    }

    private Node getNode(int index) {
        Node x = first;
        for (int i = 0; i < index; i++) {
            x = x.next;
        }
        return x;
    }

    public int indexOf(Object o) {
        int index = 0;
        for (Node x = first; x != null; x = x.next) {
            if (o.equals(x.item))
                return index;
            index++;
        }
        return -1;
    }

    public int size() {
        return size;
    }
```

```
    @Override
    public String toString() {
        return "MyLinkedListV1{" +
                "first=" + first +
                ", size=" + size +
                '}';
    }
}
```

- `Node first`: 첫 노드의 위치를 가리킨다.
- `size`: 자료 구조에 입력된 데이터의 사이즈, 데이터가 추가될 때 마다 하나씩 증가한다.

구조 예시



void add(Object e)

- 마지막에 데이터를 추가한다.
- 새로운 노드를 만들고, 마지막 노드를 찾아서 새로운 노드를 마지막에 연결한다.
- 만약 노드가 하나도 없다면 새로운 노드를 만들고 `first`에 연결한다.

Object set(int index, Object element)

- 특정 위치에 있는 데이터를 찾아서 변경한다. 그리고 기존 값을 반환한다.
- `getNode(index)`를 통해 특정 위치에 있는 노드를 찾고, 단순히 그 노드에 있는 `item` 데이터를 변경한다.

Object get(int index)

- 특정 위치에 있는 데이터를 반환한다.
- `getNode(index)`를 통해 특정 위치에 있는 노드를 찾고, 해당 노드에 있는 값을 반환한다.

int indexOf(Object o)

- 데이터를 검색하고, 검색된 위치를 반환한다.
- 모든 노드를 순회하면서 `equals()`를 사용해서 같은 데이터가 있는지 찾는다.

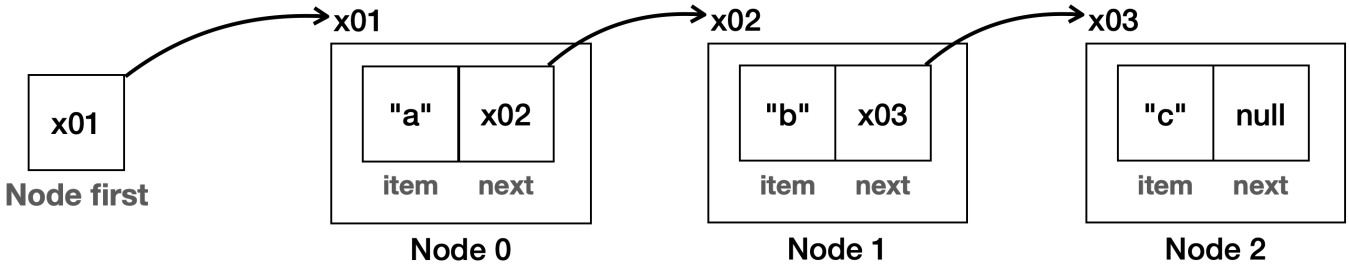
```java
package collection.link;

public class MyLinkedListV1Main {

    public static void main(String[] args) {
        MyLinkedListV1 list = new MyLinkedListV1();
        System.out.println("==데이터 추가==");
        System.out.println(list);
        list.add("a");
        System.out.println(list);
        list.add("b");
        System.out.println(list);
        list.add("c");
        System.out.println(list);

        System.out.println("==기능 사용==");
        System.out.println("list.size(): " + list.size());
        System.out.println("list.get(1): " + list.get(1));
        System.out.println("list.indexOf('c'): " + list.indexOf("c"));
        System.out.println("list.set(2, 'z'), oldValue: " + list.set(2, "z"));
        System.out.println(list);

        System.out.println("==범위 초과==");
        list.add("d");
        System.out.println(list);
        list.add("e");
        System.out.println(list);
        list.add("f");
        System.out.println(list);
    }
}
```

- `MyArrayListV1Main`에 있는 코드를 거의 그대로 사용했다.
- `MyArrayListV1` 대신에 `MyLinkedListV1 list = new MyLinkedListV1()`를 사용한 부분에 주의하자.
- 참고로 기존에 만든 `MyArrayListV1` 리스트와 제공하는 기능이 같기 때문에 메서드 이름도 같게 맞추어두었다.
- 연결 리스트는 데이터를 추가할 때 마다 동적으로 노드가 늘어나기 때문에 범위를 초과하는 문제는 발생하지 않는다.

실행 결과

```
==데이터 추가==
MyLinkedListV1{first=null, size=0}
MyLinkedListV1{first=[a], size=1}
MyLinkedListV1{first=[a->b], size=2}
MyLinkedListV1{first=[a->b->c], size=3}
==기능 사용==
list.size(): 3
list.get(1): b
list.indexOf('c'): 2
list.set(2, 'z'), oldValue: c
MyLinkedListV1{first=[a->b->z], size=3}
==범위 초과==
MyLinkedListV1{first=[a->b->z->d], size=4}
MyLinkedListV1{first=[a->b->z->d->e], size=5}
MyLinkedListV1{first=[a->b->z->d->e->f], size=6}
```

연결 리스트와 빅오



Object get(int index)

- 특정 위치에 있는 데이터를 반환한다.
- O(n)
  - 배열은 인덱스로 원하는 데이터를 즉시 찾을 수 있다. 따라서 배열을 사용하는 배열 리스트(`ArrayList`)도 인덱스로 조회시 O(1)의 빠른 성능을 보장한다. 하지만 연결 리스트에서 사용하는 노드들은 배열이 아니다. 단지 다음 노드에 대한 참조가 있을 뿐이다. 따라서 인덱스로 원하는 위치의 데이터를 찾으려면 인덱스 숫자 만큼 다음 노드를 반복해서 찾아야 한다. 따라서 인덱스 조회 성능이 나쁘다.
  - 특정 위치의 노드를 찾는데 O(n)이 걸린다.

void add(Object e)

- 마지막에 데이터를 추가한다.
- O(n)
  - 마지막 노드를 찾는데 O(n)이 소요된다. 마지막 노드에 새로운 노드를 추가하는데 O(1)이 걸린다. 따라서

O(n)이다.

Object set(int index, Object element)

- 특정 위치에 있는 데이터를 찾아서 변경한다. 그리고 기존 값을 반환한다.
- O(n)
  - 특정 위치의 노드를 찾는데 O(n)이 걸린다.

int indexOf(Object o)

- 데이터를 검색하고, 검색된 위치를 반환한다.
- O(n)
  - 모든 노드를 순회하면서 `equals()`를 사용해서 같은 데이터가 있는지 찾는다.

정리

- 앞서 노드와 연결의 개념을 이해했다면 충분히 이해할 수 있을 것이다. 만약 이해가 어렵다면 노드와 연결을 다시 복습하자. 그리고 코드를 따라서 종이에 천천히 노드를 그려보면 이해가 될 것이다.
- 연결 리스트를 통해 데이터를 추가하는 방식은 꼭 필요한 메모리만 사용한다. 따라서 배열 리스트의 단점인 메모리 낭비를 해결할 수 있었다.
  - 물론 연결을 유지하기 위한 추가 메모리가 사용되는 단점도 함께 존재한다.

배열 리스트는 중간에 데이터를 추가하거나 삭제할 때 기존 데이터를 한 칸씩 이동해야 하는 문제가 있었다. 연결 리스트는 이 문제를 어떻게 해결하는지 알아보자.

직접 구현하는 연결 리스트2 - 추가와 삭제1

특정 위치에 있는 데이터를 추가하고, 삭제하는 기능을 만들어보자.

다음 두 기능을 추가하면 된다.

void add(int index, Object e)

- 특정 위치에 데이터를 추가한다.
- 내부에서 노드도 함께 추가된다.

Object remove(int index)

- 특정 위치에 있는 데이터를 제거한다.
- 내부에서 노드도 함께 제거된다.

기능을 구현하기 전에 먼저 어떤 식으로 구현해야 할지 알아보자.

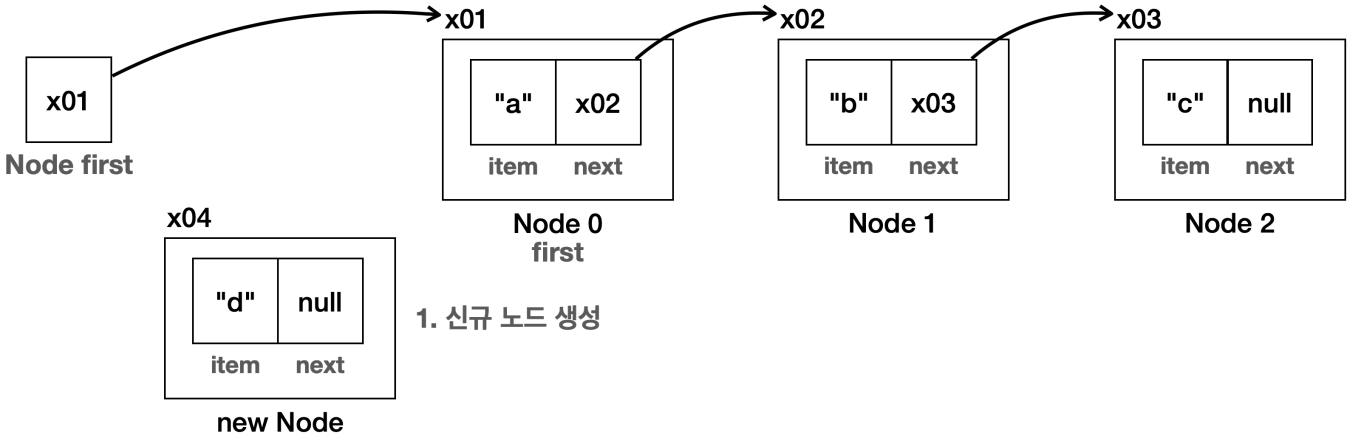
첫 번째 위치에 데이터 추가

노드에 다음과 같은 데이터가 있다고 가정해보자.

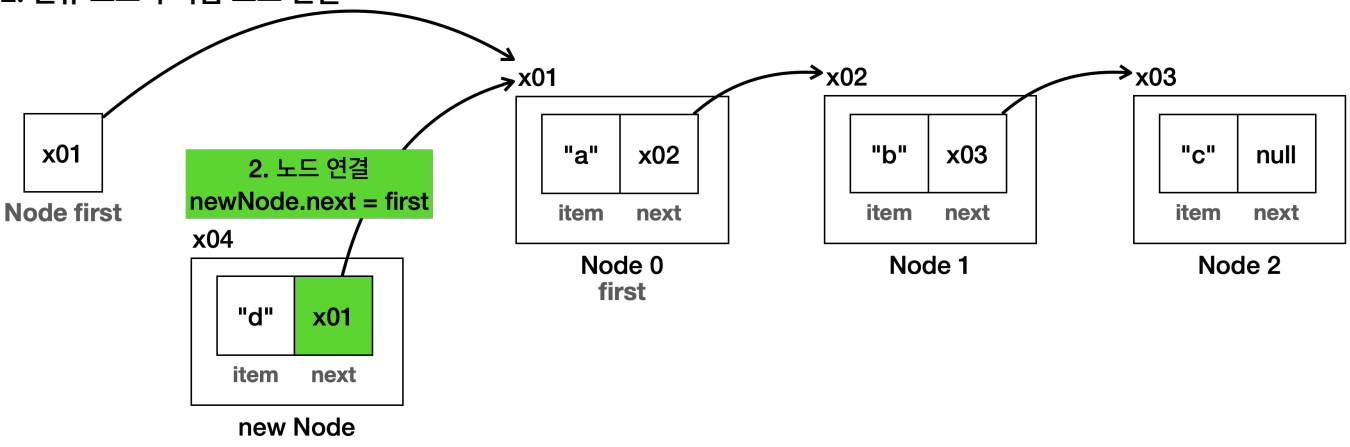
`[a->b->c]`

첫 번째 항목에 `"d"`를 추가해서 `[d->a->b->c]`로 만드는 코드를 분석해보자.

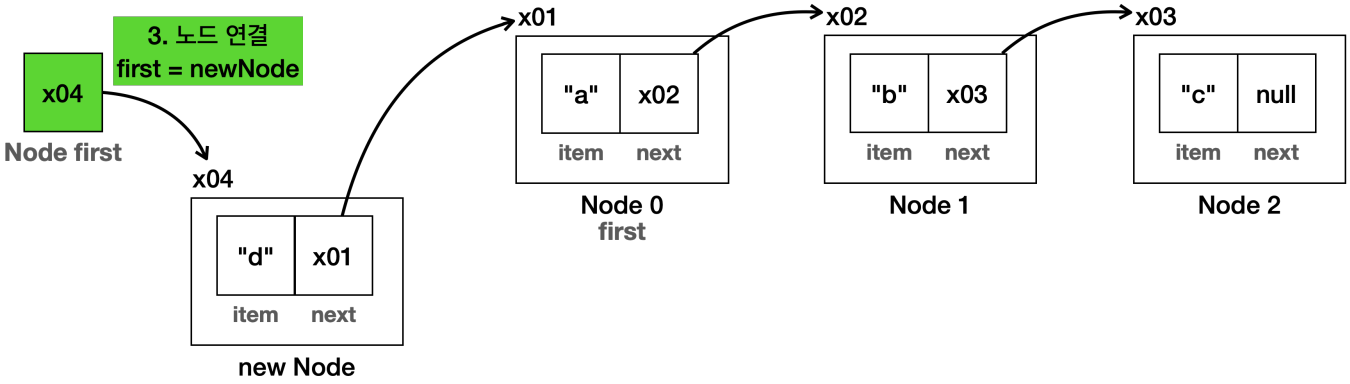
1. 신규 노드 생성



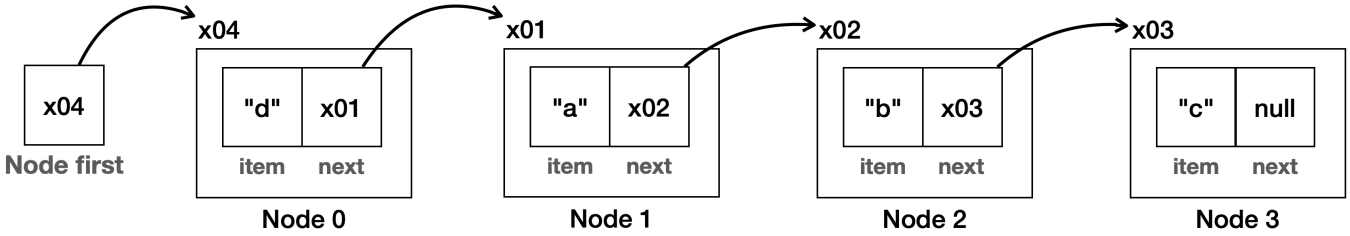
2. 신규 노드와 다음 노드 연결



3. first에 신규 노드 연결



최종 결과



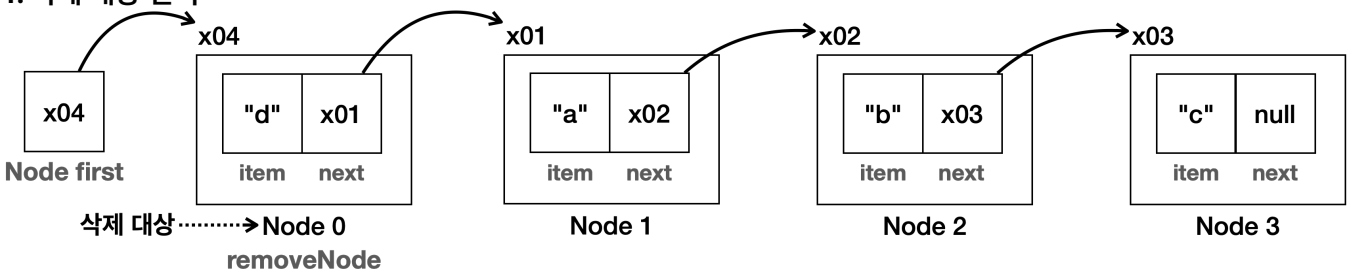
- 노드를 추가했으므로 오른쪽 노드의 index가 하나씩 뒤로 밀려난다.
  - 연결 리스트는 배열처럼 실제 index가 존재하는 것이 아니다. 처음으로 연결된 노드를 index 0, 그 다음으로 연결된 노드를 index 1로 가정할 뿐이다. 연결 리스트에서 index는 연결된 순서를 뜻한다.
- 배열의 경우 첫 번째 항목에 데이터가 추가되면 모든 데이터를 오른쪽으로 하나씩 밀어야 하지만 연결 리스트는 새로 생성한 노드의 참조만 변경하면 된다. 나머지 노드는 어떤 일이 일어난지도 모른다.
- 연결 리스트의 첫 번째 항목에 값을 추가하는 것은 매우 빠르다. O(1)로 표현할 수 있다.

첫 번째 위치의 데이터 삭제

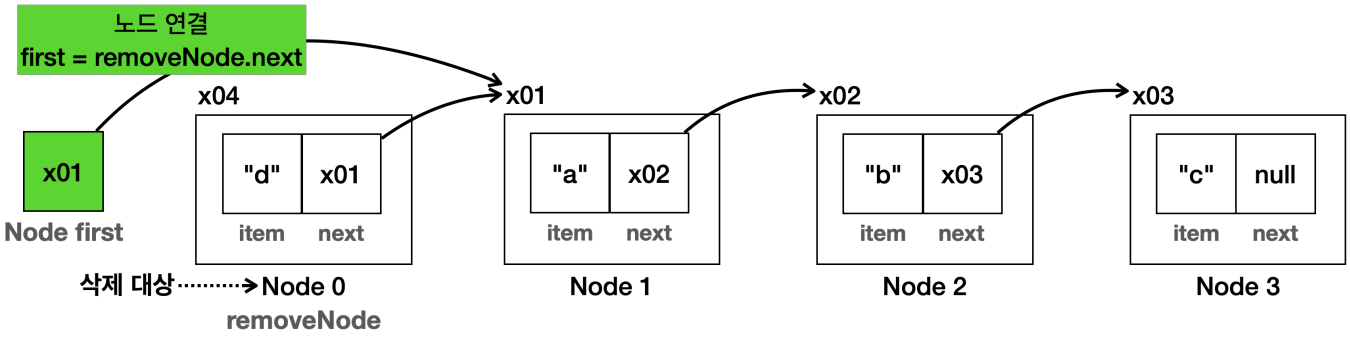
첫 번째 항목을 삭제하는 코드를 분석해보자.

`[d->a->b->c]`로 만들어진 노드를 `[a->b->c]`로 변경하면 된다.

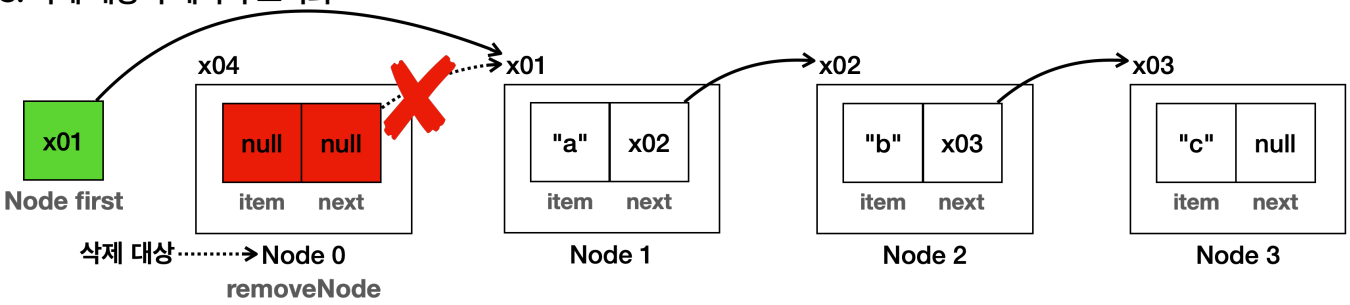
1. 삭제 대상 선택



2. first에 삭제 대상의 다음 노드 연결

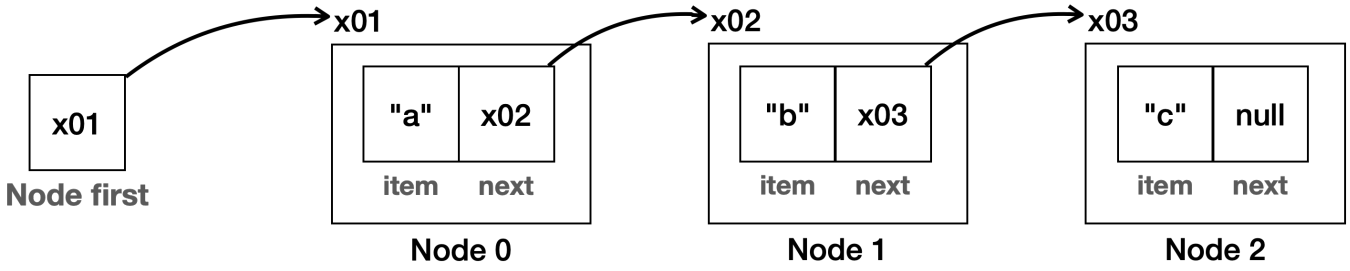


3. 삭제 대상의 데이터 초기화



- 더는 삭제 노드를 참조하는 곳이 없다. 이후 삭제 노드는 GC의 대상이 되어서 제거된다.

최종 결과



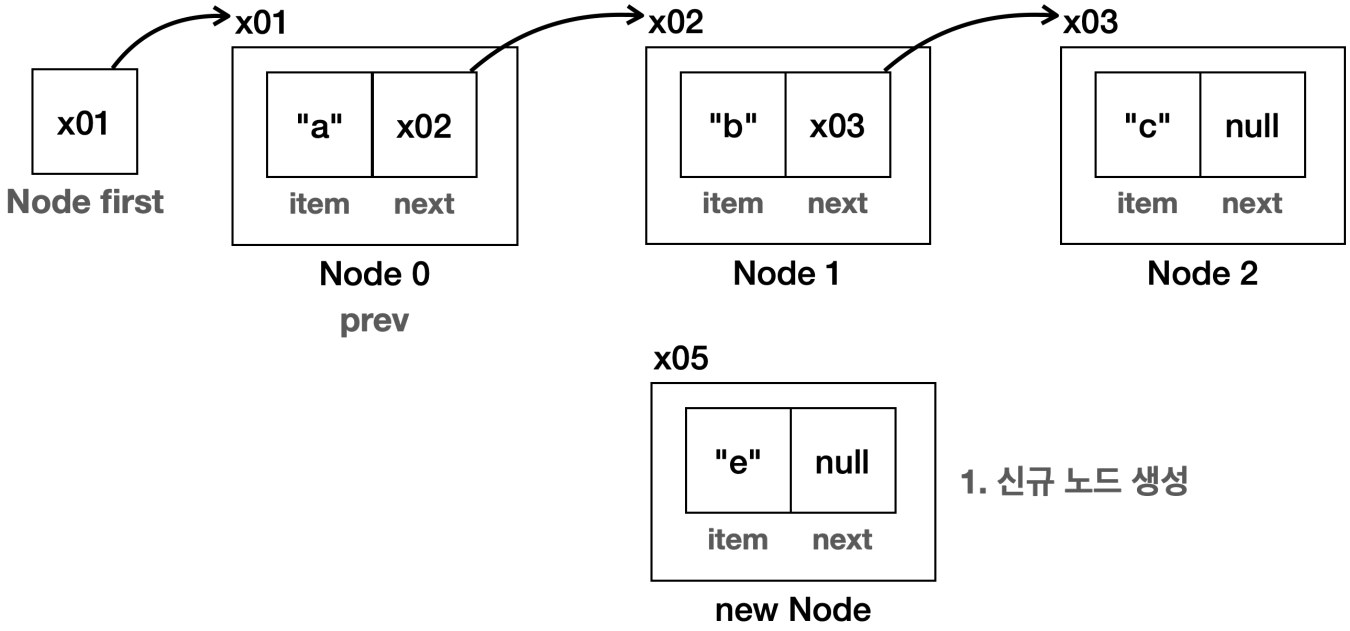
- 노드를 삭제했으므로 오른쪽 노드의 index가 하나씩 당겨진다.
- 배열의 경우 첫 번째 항목이 삭제되면 모든 데이터를 왼쪽으로 하나씩 밀어야 하지만 연결 리스트는 일부 참조만 변경하면 된다. 나머지 노드는 어떤 일이 일어난지도 모른다.
- 연결 리스트의 첫 번째 항목에 값을 삭제하는 것은 매우 빠르다. O(1)로 표현할 수 있다.

중간 위치에 데이터 추가

중간 항목에 `"e"`를 추가하는 코드를 분석해보자.

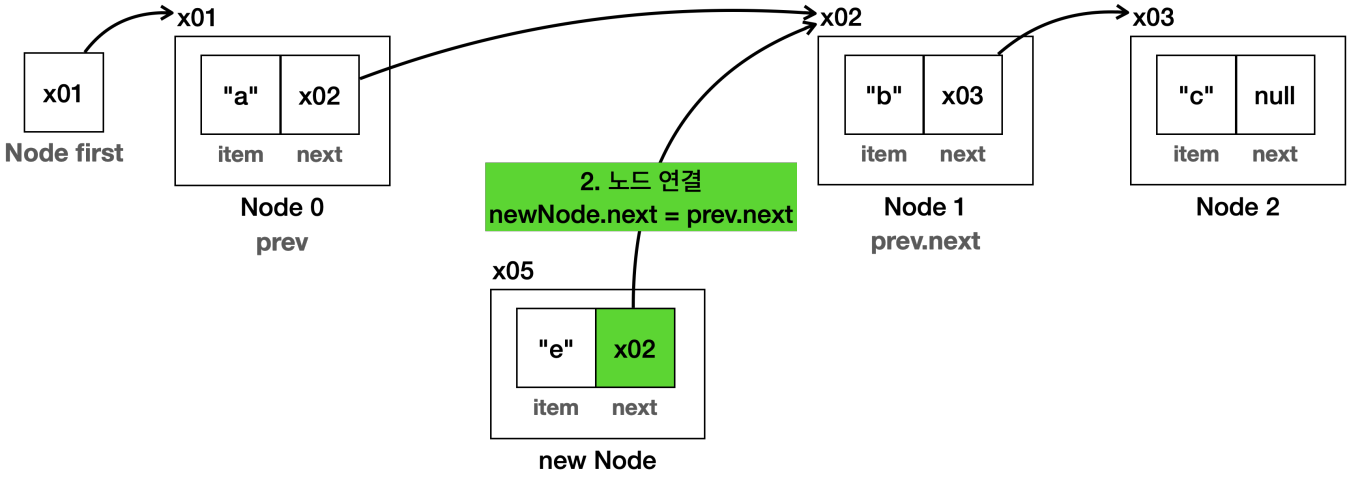
`[a->b->c]`로 만들어진 노드의 1번 인덱스 위치에 `e`를 추가해서 `[a->e->b->c]`로 변경하면 된다. 참고로 인덱스는 0부터 시작한다.

1. 새로운 노드를 생성하고, 노드가 입력될 위치의 직전 노드(prev)를 찾아둔다.

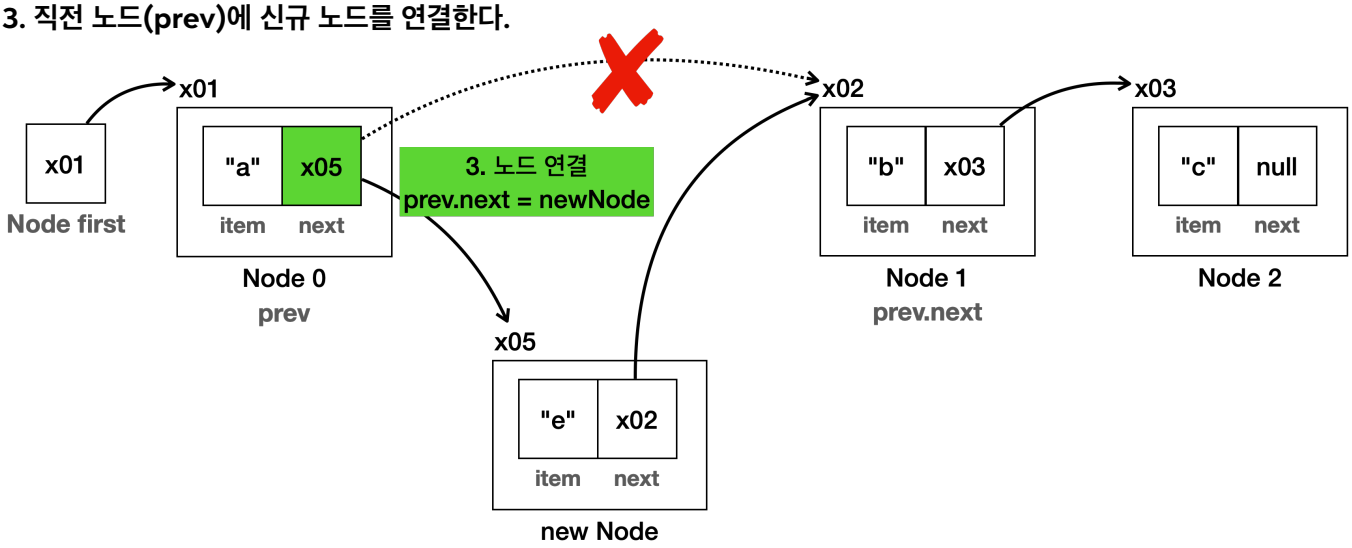


- 여기서 인덱스 1번의 직전 노드는 인덱스 0번 노드이다.

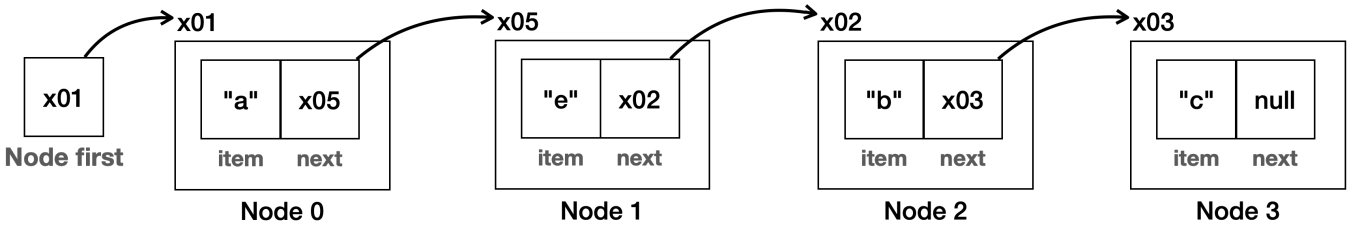
2. 신규 노드와 다음 노드를 연결한다. 직전 노드(prev)의 다음 노드를 연결하면 된다.



3. 직전 노드(prev)에 신규 노드를 연결한다.



최종 결과

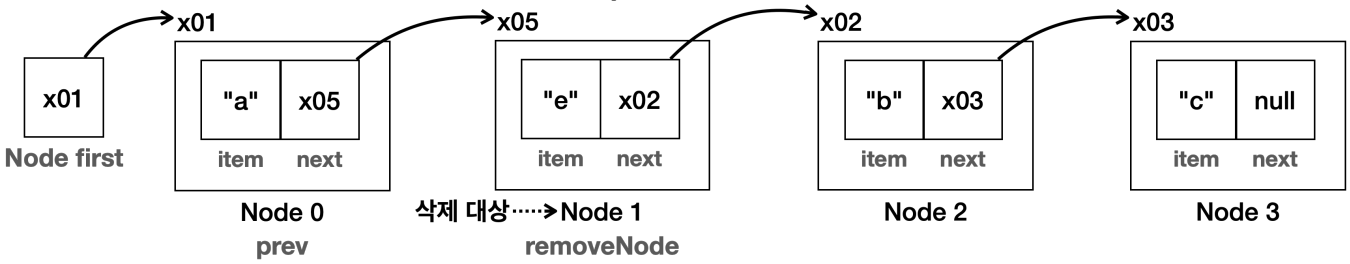


- 노드를 추가했으므로 추가한 노드 오른쪽에 있는 노드들의 index가 하나씩 뒤로 밀려난다.
- 배열의 경우 데이터가 추가되면 인덱스 위치 부터 모든 데이터를 오른쪽으로 하나씩 밀어야 하지만 연결 리스트는 새로 생성한 노드의 참조만 변경하면 된다. 나머지 노드는 어떤 일이 일어난지도 모른다.
- O(n)의 성능이다.
  - 연결 리스트는 인덱스를 사용해서 노드를 추가할 위치를 찾는데 O(n)이 걸린다.
  - 위치를 찾고 노드를 추가하는데는 O(1)이 걸린다.
  - 따라서 둘을 합하면 O(n)이 걸린다.

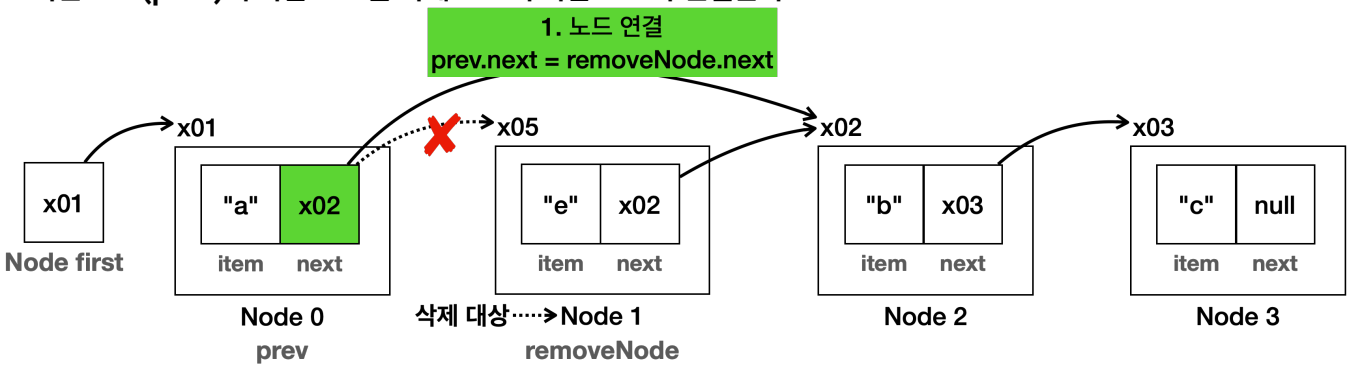
중간 위치의 데이터 삭제

중간 항목을 삭제하는 코드를 분석해보자.

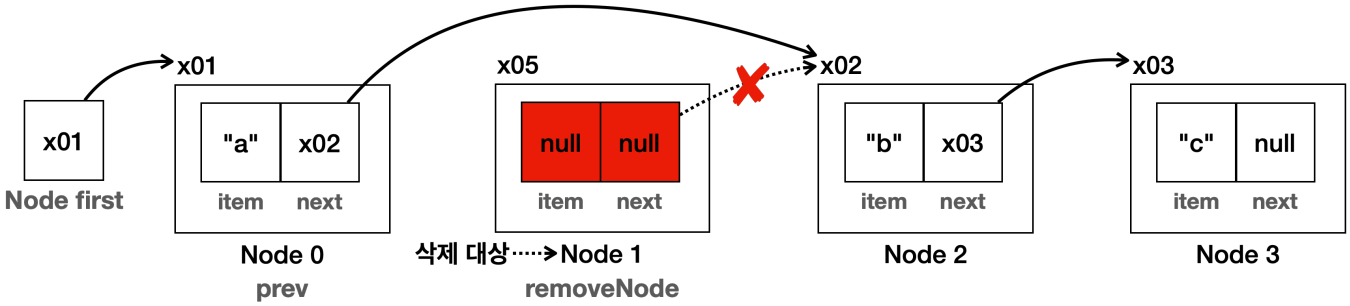
1. 삭제 대상을 찾는다. 그리고 삭제 대상의 직전 노드(prev)도 찾아둔다.



2. 직전 노드(prev)의 다음 노드를 삭제 노드의 다음 노드와 연결한다.

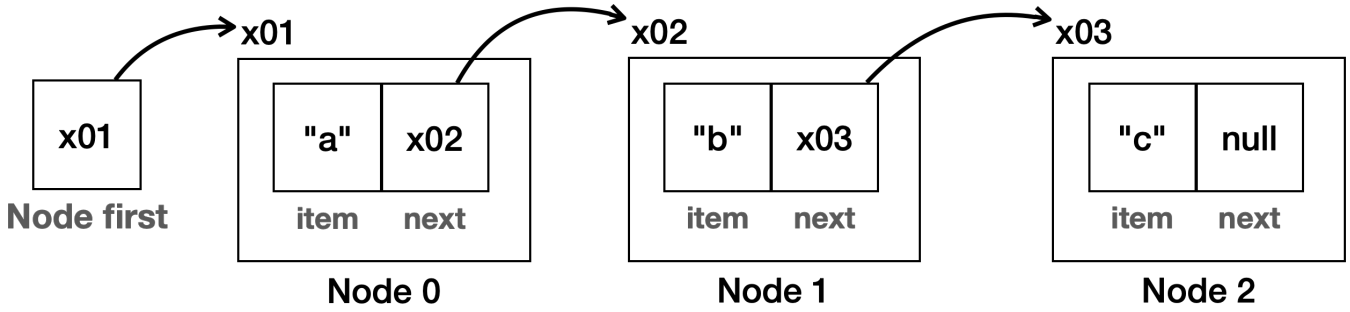


3. 삭제 노드의 데이터를 초기화 한다.



- 더는 삭제 노드를 참조하는 곳이 없다. 삭제 노드는 이후 GC의 대상이 되어서 제거된다.

최종 결과



- 노드를 삭제했으므로 오른쪽 노드의 index가 하나씩 당겨진다.
- O(n)의 성능이다.
  - 연결 리스트에서 인덱스로 삭제할 항목을 찾는데 O(n)이 걸린다.
  - 연결 리스트에서 항목을 삭제하는 것은 매우 빠르다. O(1)로 표현할 수 있다.

연결 리스트와 배열 리스트 둘다 중간에 있는 항목을 추가하거나 삭제하는 경우 둘다 같은 O(n)이다.

직접 구현하는 연결 리스트3 - 추가와 삭제2

특정 위치에 있는 데이터를 추가하고, 삭제하는 기능을 코드로 구현해보자.

```
package collection.link;

public class MyLinkedListV2 {
    private Node first;
    private int size = 0;

    public void add(Object e) {
        Node newNode = new Node(e);
```

```
        if (first == null) {
            first = newNode;
        } else {
            Node lastNode = getLastNode();
            lastNode.next = newNode;
        }
        size++;
    }

    private Node getLastNode() {
        Node x = first;
        while (x.next != null) {
            x = x.next;
        }
        return x;
    }

    //추가 코드
    public void add(int index, Object e) {
        Node newNode = new Node(e);
        if (index == 0) {
            newNode.next = first;
            first = newNode;
        } else {
            Node prev = getNode(index - 1);
            newNode.next = prev.next;
            prev.next = newNode;
        }
        size++;
    }

    public Object set(int index, Object element) {
        Node x = getNode(index);
        Object oldValue = x.item;
        x.item = element;
        return oldValue;
    }

    //추가 코드
    public Object remove(int index) {
        Node removeNode = getNode(index);
        Object removedItem = removeNode.item;
        if (index == 0) {
```

```
            first = removeNode.next;
        } else {
            Node prev = getNode(index - 1);
            prev.next = removeNode.next;
        }
        removeNode.item = null;
        removeNode.next = null;
        size--;
        return removedItem;
    }

    public Object get(int index) {
        Node node = getNode(index);
        return node.item;
    }

    private Node getNode(int index) {
        Node x = first;
        for (int i = 0; i < index; i++) {
            x = x.next;
        }
        return x;
    }

    public int indexOf(Object o) {
        int index = 0;
        for (Node x = first; x != null; x = x.next) {
            if (o.equals(x.item))
                return index;
            index++;
        }
        return -1;
    }

    public int size() {
        return size;
    }

    @Override
    public String toString() {
        return "MyLinkedListV2{" +
                "first=" + first +
                ", size=" + size +
```

```
                '}';
    }
}
```

void add(int index, Object e)

- 특정 위치에 데이터를 추가한다.
- 내부에서 노드도 함께 추가된다.

Object remove(int index)

- 특정 위치에 있는 데이터를 제거한다.
- 내부에서 노드도 함께 제거된다.

```
package collection.link;

public class MyLinkedListV2Main {

    public static void main(String[] args) {
        MyLinkedListV2 list = new MyLinkedListV2();
        //마지막에 추가 //O(n)
        list.add("a");
        list.add("b");
        list.add("c");
        System.out.println(list);

        //첫 번째 항목에 추가, 삭제
        System.out.println("첫 번째 항목에 추가");
        list.add(0,"d"); //O(1)
        System.out.println(list);

        System.out.println("첫 번째 항목 삭제");
        list.remove(0); //remove First O(1)
        System.out.println(list);

        //중간 항목에 추가, 삭제
        System.out.println("중간 항목에 추가");
        list.add(1,"e"); //O(n)
        System.out.println(list);
```

```
        System.out.println("중간 항목 삭제");
        list.remove(1);//remove O(n)
        System.out.println(list);
    }
}
```

실행 결과

```
MyLinkedListV2{first=[a->b->c], size=3}

첫 번째 항목에 추가
MyLinkedListV2{first=[d->a->b->c], size=4}

첫 번째 항목 삭제
MyLinkedListV2{first=[a->b->c], size=3}

중간 항목에 추가
MyLinkedListV2{first=[a->e->b->c], size=4}

중간 항목 삭제
MyLinkedListV2{first=[a->b->c], size=3}
```

정리

직접 만든 배열 리스트와 연결 리스트의 성능 비교 표

기능	배열 리스트	연결 리스트
인덱스 조회	O(1)	O(n)
검색	O(n)	O(n)
앞에 추가(삭제)	O(n)	O(1)
뒤에 추가(삭제)	O(1)	O(n)
평균 추가(삭제)	O(n)	O(n)

- 배열 리스트는 인덱스를 통해 추가나 삭제할 위치를 O(1)로 빠르게 찾지만, 추가나 삭제 이후에 데이터를 한 칸씩 밀어야 한다. 이 부분이 O(n)으로 오래 걸린다.
- 반면에 연결 리스트는 인덱스를 통해 추가나 삭제할 위치를 O(n)으로 느리게 찾지만, 찾은 이후에는 일부 노드의

참조값만 변경하면 되므로 이 부분이 O(1)로 빠르다.

- 앞에 추가하는 경우: 연결 리스트는 배열 리스트처럼 추가한 항목의 오른쪽에 있는 데이터를 모두 한 칸씩 밀지 않아도 된다. 단순히 일부 노드의 참조만 변경하면 된다. 따라서 데이터를 앞쪽에 추가하는 경우 보통 연결 리스트가 더 좋은 성능을 제공한다.
  - 배열 리스트: 데이터를 앞쪽에 추가하는 경우 모든 데이터를 오른쪽으로 한 칸씩 밀어야 한다. O(n)
  - 연결 리스트: 데이터를 앞쪽에 추가하는 경우 일부 노드의 참조만 변경하면 된다. O(1)
- 마지막에 데이터를 추가하는 경우
  - 배열 리스트
    - 인덱스로 마지막 위치를 바로 찾을 수 있다. O(1)
    - 데이터를 마지막에 추가하는 경우 데이터를 이동하지 않아도 된다. O(1)
    - 따라서 O(1)의 성능을 제공한다.
  - 연결 리스트
    - 노드를 마지막까지 순회해야 마지막 노드를 찾을 수 있다. 따라서 마지막 노드를 찾는데 O(n)의 시간이 걸린다.
    - 데이터를 추가하는 경우 일부 노드의 참조만 변경하면 된다. O(1)
    - 따라서 O(n)의 성능을 제공한다.

배열 리스트 vs 연결 리스트 사용

데이터를 조회할 일이 많고, 뒷 부분에 데이터를 추가한다면 배열 리스트가 보통 더 좋은 성능을 제공한다. 앞쪽의 데이터를 추가하거나 삭제할 일이 많다면 연결 리스트를 사용하는 것이 보통 더 좋은 성능을 제공한다.

참고 - 단일 연결 리스트, 이중 연결 리스트

우리가 구현한 연결 리스트는 한 방향으로만 이동하는 단일 연결 리스트다. 노드를 앞뒤로 모두 연결하는 이중 연결 리스트를 사용하면 성능을 더 개선할 수 있다.

뒤에서 설명하지만 자바가 제공하는 연결 리스트는 이중 연결 리스트다. 추가로 자바가 제공하는 연결 리스트는 마지막 노드를 참조하는 변수를 가지고 있어서, 뒤에 추가하거나 삭제하는 삭제하는 경우에도 O(1)의 성능을 제공한다. 자세한 내용은 뒤에서 설명한다.

이중 연결 리스트 예시

```
public class Node {
    Object item;
    Node next; //다음 노드 참조
    Node prev; //이전 노드 참조
}
```

마지막 노드를 참조하는 연결 리스트

```
public class LinkedList {
    private Node first;
    private Node last; //마지막 노드 참조
    private int size = 0;
}
```

직접 구현하는 연결 리스트4 - 제네릭 도입

지금까지 만든 연결 리스트에 제네릭을 도입해서 타입 안전성을 높여보자.

추가로 여기서 사용하는 `Node`는 외부에서 사용되는 것이 아니라 연결 리스트 내부에서만 사용된다. 따라서 중첩 클래스로 만들자.

```
package collection.link;

public class MyLinkedListV3<E> {
    private Node<E> first;
    private int size = 0;

    public void add(E e) {
        Node<E> newNode = new Node<>(e);
        if (first == null) {
            first = newNode;
        } else {
            Node<E> lastNode = getLastNode();
            lastNode.next = newNode;
        }
        size++;
    }

    private Node<E> getLastNode() {
        Node<E> x = first;
        while (x.next != null) {
            x = x.next;
        }
        return x;
```

```
    }

    public void add(int index, E e) {
        Node<E> newNode = new Node<>(e);
        if (index == 0) {
            newNode.next = first;
            first = newNode;
        } else {
            Node<E> prev = getNode(index - 1);
            newNode.next = prev.next;
            prev.next = newNode;
        }
        size++;
    }

    public E set(int index, E element) {
        Node<E> x = getNode(index);
        E oldValue = x.item;
        x.item = element;
        return oldValue;
    }

    public E remove(int index) {
        Node<E> removeNode = getNode(index);
        E removedItem = removeNode.item;
        if (index == 0) {
            first = removeNode.next;
        } else {
            Node<E> prev = getNode(index - 1);
            prev.next = removeNode.next;
        }
        removeNode.item = null;
        removeNode.next = null;
        size--;
        return removedItem;
    }

    public E get(int index) {
        Node<E> node = getNode(index);
        return node.item;
    }

    private Node<E> getNode(int index) {
```

```
        Node<E> x = first;
        for (int i = 0; i < index; i++) {
            x = x.next;
        }
        return x;
    }

    public int indexOf(E o) {
        int index = 0;
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item))
                return index;
            index++;
        }
        return -1;
    }

    public int size() {
        return size;
    }

    @Override
    public String toString() {
        return "MyLinkedListV3{" +
                "first=" + first +
                ", size=" + size +
                '}';
    }

    private static class Node<E> {
        E item;
        Node<E> next;

        public Node(E item) {
            this.item = item;
        }

        @Override
        public String toString() {
            StringBuilder sb = new StringBuilder();
            Node<E> temp = this;
            sb.append("[");
            while (temp != null) {
```

```
                sb.append(temp.item);
                if (temp.next != null) {
                    sb.append("->");
                }
                temp = temp.next;
            }
            sb.append("]");
            return sb.toString();
        }
    }
}
```

- `MyLinkedListV3<E>`로 제네릭 클래스를 선언했다.
- `Object`로 처리하던 부분을 타입 매개변수 `<E>`로 변경했다.
- 정적 중첩 클래스로 새로 선언한 `Node<E>`도 제네릭 타입으로 선언했다.

참고 - 영상 오류 정정

영상에서는 `public Object set(int index, E element)`으로 `Object`를 반환하는데, 메뉴얼과 같이 `E`를 반환하는 것이 맞다. 메뉴얼은 `public E set(int index, E element)`으로 `E`를 반환한다.

중첩 클래스 예시)

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

- 이렇게 클래스 안에서 클래스를 선언하는 것을 중첩 클래스라 한다.
- 중첩 클래스는 특정 클래스 안에서만 사용될 때 주로 사용한다.
- `Node` 클래스는 `MyLinkedList` 안에서만 사용된다. 외부에서는 사용할 이유가 없다.
- 이럴 때 중첩 클래스를 사용하면 특정 클래스 안으로 클래스 선언을 숨길 수 있다.
- 중첩 클래스를 사용하면 `MyLinkedListV3` 입장에서 외부에 있는 `Node` 클래스보다 내부에 선언한 `Node` 클래스를 먼저 사용한다.

```
package collection.link;
```

```
public class MyLinkedListV3Main {
    public static void main(String[] args) {
        MyLinkedListV3<String> stringList = new MyLinkedListV3<>();
        stringList.add("a");
        stringList.add("b");
        stringList.add("c");
        String string = stringList.get(0);
        System.out.println("string = " + string);

        MyLinkedListV3<Integer> intList = new MyLinkedListV3<>();
        intList.add(1);
        intList.add(2);
        intList.add(3);
        Integer integer = intList.get(0);
        System.out.println("integer = " + integer);
    }

}
```

실행 결과

```
string = a
integer = 1
```

제네릭의 덕분에 타입 안전성 있는 자료 구조를 만들 수 있다.

정리