자료구조 실습 보고서

[제 5주] : 리스트 기본 기능

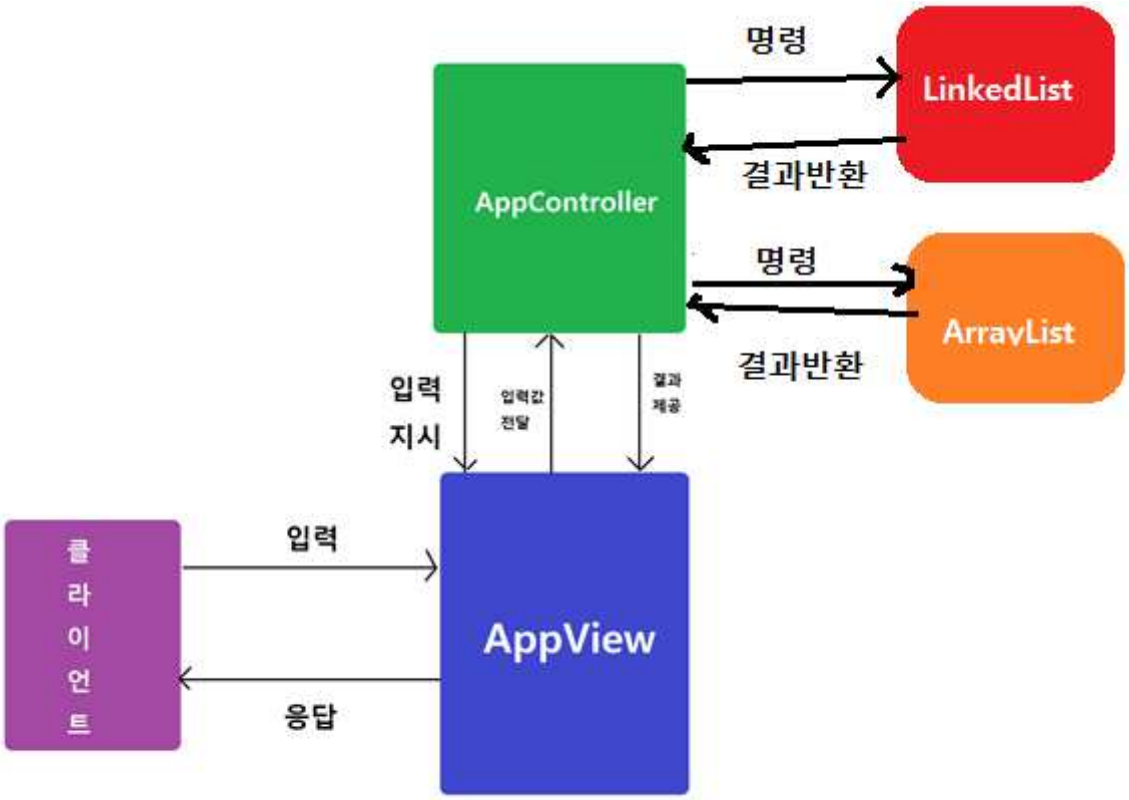
제출일: 2022-04-09(토)

201902708 신동훈

프로그램 설명

1. 프로그램의 전체 설계 구조

간단하게 표현하면 다음과 같다.



MVC(Model - View - Controller)구조를 따르며, 각각의 역할은 다음과 같다.

AppController

AppView에게 입력 요청을 보내며, 그에 대한 응답으로 반환된 입력값을 받아온다.
List에 Student객체를 전달하며 보관, 삭제 등의 계산을 요청하고, 결과를 받아온다.
AppView에게 출력 정보를 제공하며, 출력을 요청한다.

AppView

화면(콘솔)에 메시지를 출력하는 역할과, 화면으로부터 입력값을 받아와 다른 객체에게 전달하는 역할을 담당한다.

LinkedList, ArrayList

Student 객체를 받아서 저장하거나 삭제하는 등의 로직을 수행한다.

2. 함수 설명(주요 알고리즘/ 자료구조의 설명 포함)

2-1. 사용된 자료구조

LinkedList은 내부적으로 LinkedNode객체를 사용하여 Student 객체를 저장한다.
LinkedNode는 내부적으로 자신이 가진 원소인 element 와, 자신과 연결된, 즉 자신 이후의 LinkedNode에 대한 주소를 가지고 있다.
따라서 LinkedList은 가장 앞부분의 LinkedNode만 알고 있어도, 해당 LinkedNode와 연결된 LinkedNode들을 통해, 모든 원소를 순회할 수 있으며, 원소를 추가할 때도 용량 제한 없이 계속해서 원소를 추가할 수 있다.

ArrayList은 내부적으로 1차원 배열을 사용한다.
배열이기 때문에, 생성할 당시 지정해준 용량을 초과해서 값을 담을 수 없으며, 담기 위해서는 크기가 더 큰 배열을 새로 만들어준 후 기존의 값을 이동시켜서 구현하는 방식으로 늘릴 수 있겠지만, 이번 ArrayBag에서는 그렇게 하지 않았다.

2-2. 주요 함수

ArrayList 클래스

```
/**
     * ArrayList 가 비어있는지 확인한다.
     * @return 비어있다면 true
     */
    public boolean isEmpty(){
        return (this.size() == 0);
    }

    /**
     * ArrayList 가 가득 찬 상태인지 확인한다.
     * @return 가득 차 있다면 true
     */
    public boolean isFull(){
        return (this.size() == this.capacity());
    }



    /**
     * 매개변수로 들어온 원소에 대해, 해당 원소가 ArrayList 안에 존재하는지
     * 여부를 반환한다.
     * @param anElement 존재하는지 확인할 원소
     * @return 존재한다면 true
     */
    public boolean doesContain(E anElement){
        for(int i =0; i < this.size(); i++){//처음부터 ArrayList 의 size 만큼 배열을 순환
            if(this.elements()[i].equals(anElement)){//equals 를 통해 같은 원소인지 비교
                return true;
            }
        }
        return false;
    }

    /**
     * 매개변수로 들어온 원소가 ArrayList 속에 몇 개 존재하는지 확인한다.
     * @param anElement 빈도를 확인할 원소
     * @return 빈도수
     */
    public int frequencyOf(E anElement){
        int frequencyCount = 0;
        for (int i =0; i < this.size(); i++){
            if(this.elements()[i].equals(anElement)){
                frequencyCount++;
            }
        }
        return frequencyCount;
    }
```

```
/**
 * 매개변수로 주어진 숫자에 해당하는 ArrayList 의 원소를 가져온다.
 * @param order 인덱스 번호
 * @return 해당 인덱스 번호에 해당하는 요소
 */
public E elementAt(int order) {
    if(this.anElementDoesExistAt(order)){
        int position = order;
        return this.elements()[position];
    }
    else {
        return null;
    }
}

/**
 * 입력으로 주어진 인덱스 번호가 올바른지 확인한다.
 * @param order 인덱스 번호
 * @return 올바르다면 true
 */
private boolean anElementDoesExistAt(int order) {
    return ((order >= 0) && (order < this.size()));
}

/**
 * ArrayList 속 존재하는 가장 처음 요소를 반환한다
 * @return 가장 처음 요소
 */
public E first(){
    if(this.isEmpty()){
        return null;
    }
    else {
        return this.elements()[1];
    }
}

/**
 * ArrayList 속 존재하는 가장 마지막 요소를 반환한다
 * @return 가장 마지막 요소
 */
public E last(){
    if(this.isEmpty()){
        return null;
    }
    else {
        return this.elements()[this.size() - 1];
    }
}
```

```
/**
     * 주어진 원소가 존재하는 인덱스의 번호를 반환한다.
     * @param anElement 찾을 원소
     * @return 원소의 인덱스 혹은 존재하지 않는다면 -1
     */
    public int orderOf(E anElement){
        //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 order;
            }
        }
        return -1;
    }


    /**
     * 주어진 순서에 주어진 원소를 삽입한다.
     * @param anElement 주어진 원소
     * @param anOrder 주어진 순서
     * @return 삽입에 성공하면 true
     */
    public boolean addTo(E anElement, int anOrder) {
        if (this.isFull()) {
            return false;
        } else if (anOrder < 0 || anOrder > this.size()) {
            return false;
        } else {
            this.makeRoomAt(anOrder);
            this.elements()[anOrder] = anElement;
            this.setSize(this.size() +1);
            return true;
        }
    }

    /**
     * 주어진 순서의 위치를 빈칸으로 만들기 위해
     * 주어진 순서에 존재하는 원소부터 마지막 원소까지 옆으로 한 칸씩 이동시킨다.
     * @param aPosition 빈 공간을 만들 위치
     */
    private void makeRoomAt(int aPosition) {
        for (int i = this.size(); i > aPosition; i--) {
            this.elements()[i] = this.elements()[i-1];
        }
    }
```

```
/**
 * ArrayList 의 가장 첫번째 부분에 주어진 원소를 삽입한다.
 * @param anElement 삽입할 원소
 * @return 삽입에 성공하면 true
 */
public boolean addToFirst(E anElement){
    return this.addTo(anElement, 0);
}

/**
 * ArrayList 의 가장 마지막 부분에 주어진 원소를 삽입한다.
 * @param anElement 삽입할 원소
 * @return 삽입에 성공하면 true
 */
public boolean addToLast(E anElement){
    return this.addTo(anElement, this.size());
}


/**
 * 가장 효과적으로 삽입할 수 있는 곳에 주어진 원소를 삽입한다.
 * ArrayList 의 경우 가장 마지막 부분에 주어진 원소를 삽입한다.
 * @param anElement 삽입할 원소
 * @return 삽입에 성공하면 true
 */
public boolean add(E anElement){
    return this.addToLast(anElement);
}
/**
 * 주어진 순서에 존재하는 원소를 삭제한다.
 * @param order 삭제할 원소의 순서
 * @return 삭제된 원소, 없다면 null
 */
public E removeFrom(int order) {
    //주어진 순서 order 에 원소가 없으면 null 을 return 한다
    //원소가 있으면 리스트에서 제거하여 return 한다.

    E 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;
}
```

```
/**
    * 주어진 순서에 해당하는 빈 공간을 제거한다.
    * @param position 주어진 순서
    */
   private void removeGapAt(int position) {

       // 해당 메서드가 불러졌을 때 리스트는 empty 가 아니다.
       // 즉 언제나 (this.size() >0)
       // position 은 valid, 즉 언제나 (0 <= position < this.size())
       for (int i = position + 1; i < this.size(); i++) {
           this.elements()[i-1] = this.elements()[i];
       }

       //마지막 원소에 대한 참조를 지워 GC의 대상이 되도록 한다.
       this.elements()[this.size()-1] = null;

   }


   /**
    * ArrayList 의 가장 처음 원소를 삭제한다.
    * @return 삭제한 원소
    */
   public E removeFirst(){
       return removeFrom(0);
   }
   /**
    * ArrayList 의 가장 마지막 원소를 삭제한다.
    * @return 삭제한 원소
    */
   public E removeLast(){
       return removeFrom(this.size() -1);
   }

   /**
    * 아무 원소나 삭제한다.
    * ArrayList 의 경우 효율적으로 삭제하기 위해 가장 마지막 원소를 삭제한다.
    */
   public E removeAny(){
       return removeLast();
   }
```

```
    /**
     * 주어진 순서의 원소를 주어진 원소로 변경한다.
     * @param anElement 변경할 원소
     * @param anOrder 변경할 순서
     * @return 변경에 성공하면 true
     */
    public boolean replaceAt(E anElement, int anOrder) {
        if(!anElementDoesExistAt(anOrder)){
            return false;
        }
        this.elements()[anOrder] = anElement;
        return true;
    }
    /**
     * 반복자를 반환한다
     * @return ArrayList 의 반복자
     */
    public Iterator<E> iterator() {
        return (new ListIterator());
    }
```

ArrayList의 ListIterator 클래스

```
        /**
         * 다음 위치를 반환한다.
         * @return 다음 위치 index
         */
        private int nextPosition() {
            return _nextPosition;
        }

        /**
         * 다음 위치를 지정한다
         * @param newNextPosition 지정할 다음 위치
         */
        private void setNextPosition(int newNextPosition) {
            this._nextPosition = newNextPosition;
        }

        private ListIterator(){
            this.setNextPosition(0);
        }
```

```
/**
 * 다음 순서의 원소가 존재하는지 확인한다.
 * @return 존재하면 true
 */
@Override
public boolean hasNext() {
    return (this.nextPosition() < ArrayList.this.size());
}


/**
 * 다음 순서의 원소를 반환한다
 * @return 다음 순서의 원소
 */
@Override
public E next() {
    E nextElement = null;
    if(this.hasNext()){
        nextElement = ArrayList.this.elements()[this.nextPosition()];
        this.setNextPosition(this.nextPosition() +1);
    }
    return nextElement;
}
```

LinkedList 클래스

```
/**
 * 리스트가 비었는지 확인한다
 * @return 비었으면 true
 */
public boolean isEmpty() {
    return (this.size() == 0);
}

/**
 * 리스트가 가득차있는지 확인한다.
 * @return LinkedList 는 용량 제한이 없으므로 false
 */
public boolean isFull() {
    return false;
}
/**
 * 주어진 순서의 원소를 반환한다
 *
 * @param order 원소의 순서
 * @return 주어진 순서에 있는 원소
 */
public E elementAt(int order) {
    if (this.anElementDoesExistAt(order)) {
        LinkedNode<E> currentNode = this.head();
        int nodeCount = 0;
        while (nodeCount < order) {
            currentNode = currentNode.next();
            nodeCount++;
        }
        return currentNode.element();
    } else {
        return null;
    }
}
```

```
    /**
     * 리스트의 맨 앞 원소를 반환한다
     *
     * @return 맨 앞 원소
     */
    public E first() {
        if (this.isEmpty()) {
            return null;
        } else {
            return this.elementAt(0);
        }
    }

    /**
     * 리스트의 마지막 원소를 반환한다
     *
     * @return 마지막 원소
     */
    public E last() {
        if (this.isEmpty()) {
            return null;
        } else {
            return this.elementAt(this.size() - 1);
        }
    }
    /**
     * 주어진 원소가 존재하는 위치를 반환한다
     *
     * @param anElement 찾을 원소
     * @return 원소의 위치, 존재하지 않으면 -1
     */
    public int orderOf(E anElement) {
        int order = 0;
        LinkedNode<E> currentNode = this.head();
        while ((currentNode != null) && (!currentNode.element().equals(anElement))) {
            order++;
            currentNode = currentNode.next();
        }
        //찾지 못한 경우
        if(currentNode == null){
            return -1;
        }else {
            return order;
        }
    }
```

```
/**
     * 주어진 원소가 들어있는지 여부를 반환한다.
     * @param anElement 찾을 원소
     * @return 있다면 true
     */
    public boolean doesContain(E anElement){
        LinkedNode<E> currentNode = this.head();

        while (currentNode != null){
            if(currentNode.element().equals(anElement)){
                return true;
            }
            currentNode = currentNode.next();
        }
        return false;
    }

    /**
     * 주어진 순서에 주어진 원소를 삽입한다.
     * @param anElement 삽입할 원소
     * @param order 삽입할 순서
     * @return 삽입에 성공하면 true
     */
    public boolean addTo(E anElement, int order) {
        if ((order < 0) || (order > this.size())) {//order 가 유효한지 검사
            return false;
        }
        else if (this.isFull()) {
            return false;
        }
        else {
            LinkedNode<E> nodeForAdd = new LinkedNode<>(anElement, null);
            if (order == 0){//맨 앞 순서에 삽입, 앞 (previous) 노드가 존재하지 않는다.
                nodeForAdd.setNext(this.head());//추가할 노드의 다음 노드에 현재의 head Node 를 설정한다.
                this.setHead(nodeForAdd);//LinkedList 의 Head Node 를 추가할 노드로 설정해준
            }
            else {//순서가 맨 앞이 아니기 때문에, 반드시 앞 노드가 존재한다.
                LinkedNode<E> previousNode = this.head();
                for (int i = 1; i < order; i++) {//0이 아니라 1부터 시작하여 삽입할 위치의 앞 노드를 찾는
                    previousNode = previousNode.next();
                }
                nodeForAdd.setNext(previousNode.next());
                previousNode.setNext(nodeForAdd);
            }
            this.setSize(this.size() + 1);
            return true;
        }
    }
```

```
/**
 * 주어진 원소를 맨 앞에 삽입한다.
 * @param anElement 주어진 원소
 * @return 삽입에 성공하면 true
 */
public boolean addToFirst(E anElement){
    return this.addTo(anElement, 0);
}

/**
 * 주어진 원소를 맨 뒤에 삽입한다.
 * @param anElement 주어진 원소
 * @return 삽입에 성공하면 true
 */
public boolean addToLast(E anElement){
    return this.addTo(anElement, this.size());
}

/**
 * 주어진 원소를 임의의 위치에 삽입한다.
 * LinkedList의 경우 맨 앞에 삽입하는 경우 시간이 가장 짧게 걸리므로
 * 맨 앞에 삽입한다.
 * @param anElement 삽입할 원소
 * @return 삽입에 성공하면 true
 */
public boolean add(E anElement){
    return this.addToFirst(anElement);
}
```

```
/**
 * 주어진 순서의 원소를 삭제한다
 * @param order 삭제할 원소의 순서
 * @return 삭제된 원소
 */
public E removeFrom(int order) {
    if (!this.anElementDoesExistAt(order)) {//삭제할 원소가 없거나 잘못된 위치
        return null;
    }
    else {
        //리스트는 비어 있지 않으며, 삭제할 원소가 있음
        LinkedNode<E> removeNode = null;

        if (order == 0) {//삭제할 원소가 맨 앞 원소
            removeNode = this.head();
            this.setHead(this.head().next());
        }

        else {//삭제할 원소의 위치는 맨 앞이 아니며, 따라서 원소가 두 개 이상
            LinkedNode<E> previousNode = this.head();
            for (int i = 1; i < order; i++) {
                previousNode = previousNode.next();
            }
            removeNode = previousNode.next();
            previousNode.setNext(removeNode.next());
        }
        this.setSize(this.size() - 1);
        return removeNode.element();
    }
}
/**
 * 맨 앞 원소를 삭제한다.
 * @return 삭제된 원소
 */
public E removeFirst(){
    return this.removeFrom(0);
}

/**
 * 맨 뒤 원소를 삭제한다.
 * @return 삭제된 원소
 */
public E removeLast(){
    return this.removeFrom(this.size()-1);
}
```

```
/**
 * 아무 원소나 삭제한다.
 * LinkedList 의 경우 맨 앞 노드를 삭제하는 것이 가장 빠르므로
 * 맨 앞 원소를 삭제한다.
 * @return 삭제된 원소
 */
public E removeAny(){
    return this.removeFirst();
}


/**
 * 주어진 원소를 리스트에서 삭제한다.
 * @param anElement 삭제할 원소
 * @return 삭제에 성공하면 true
 */
public boolean remove(E anElement) {
    LinkedNode<E> previousNode = null;
    LinkedNode<E> currentNode = this.head();
    while ((currentNode != null) && (!currentNode.element().equals(anElement))) {
        previousNode = currentNode;
        currentNode = currentNode.next();
    }

    if(currentNode == null){
        return false;
    }
    else {
        if(currentNode == this.head()){
            this.setHead(this.head().next());
        }
        else {
            previousNode.setNext(currentNode.next());
        }
        this.setSize(this.size() - 1);
        return true;
    }

}
```

```
/**
 * 주어진 순서의 원소를 주어진 원소로 교체한다.
 * @param anElement 교체할 원소
 * @param order 교체할 순서
 * @return 교체에 성공하면 true
 */
public boolean replaceAt(E anElement, int order) {
    if (!this.anElementDoesExistAt(order)) {
        return false;
    }
    else {
        LinkedNode<E> currentNode = this.head();
        for (int i = 0; i < order; i++) {
            currentNode = currentNode.next();
        }
        currentNode.setElement(anElement);
        return true;
    }
}

/**
 * 반복자를 반환한다.
 * @return LinkedList 의 반복자
 */
public Iterator<E> iterator() {
    return (new ListIterator());
}
```

LinkedList 의 ListIterator

```
    /**
     * LinkedList 의 head 를 처음 노드로 성정한다.
     */
    private ListIterator(){
        this._nextNode = LinkedList.this._head;
    }
    /**
     * 다음 순서의 원소가 존재하는지 확인한다.
     * @return 존재하면 true
     */
    @Override
    public boolean hasNext() {
        return (this._nextNode != null);
    }
```

```
/**
 * 다음 순서의 원소를 반환한다
 * @return 다음 순서의 원소
 */
@Override
public E next() {
    if(this._nextNode == null){
        return null;
    }
    else {
        E e = this._nextNode.element();
        this._nextNode = this._nextNode.next();
        return e;
    }
}
```

3. 종합 설명

해당 프로그램은 리스트 속에 학생들을 추가하거나 삭제하는 등의 작업을 하는 프로그램이다.
앱을 실행하기 위해서는 '_DS05_Main_201902708_신동훈' 이라는 이름의 클래스에 속한 main 메서드를 실행시켜주면 된다.

프로그램을 시작하면 가능한 작업의 번호를 보여주며, 사용자는 해당 번호들 중 하나를 선택하여 원하는 작업을 실행시키면 된다.

리스트 속에 학생이 존재하는지 확인하거나, 특정 순서의 학생을 확인하거나, 학생을 추가, 삭제하는 등 여러 작업을 할 수 있으며, 99를 입력받으면 프로그램이 종료된다.

프로그램 장단점/특이점

1. 프로그램의 장점

MVC 패턴을 사용하였기에 유지보수성과, 이후 요구사항의 변경 시 코드의 변경의 파급효과가 적을 뿐더러 LinkedList 클래스와 ArrayList 클래스 모두 제네릭을 사용하여 다양한 객체들을 가방에 담을 수 있음과 동시에 잘못된 객체를 담게되었을 때 런타임이 아닌 컴파일 시예 예외가 발생하여 실수를 방지할 수 있다는 장점이 있다.

2 프로그램의 단점

제네릭을 사용하였긴 했지만, 가방에는 담을 수 없는 것들, 혹은 담아서는 안 되는 것들이 존재한다. 그러나 현재 ArrayBag은 어떠한 객체던 차별 없이 가방에 보관할 수 있다는 문제점을 갖는다.
이를 해결하기 위해서는 간단한 예시로 '담을 수 있는'이라는 속성을 인터페이스로 정의해둔 후, 제네릭의 extends 키워드를 사용하여 상한 제한을 걸 수 있다.

예를 들면 다음과 같다.
'public class ArrayBag<E extends Containable> {...}'

실행 결과 분석

1. 입력과 출력

```
<<<< 리스트 기능 확인 프로그램을 시작합니다 >>>>

! 현재의 리스트 원소들: [ ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 9

! Add 작업을 실행합니다:
? 점수를 입력하시오: 60
! 입력된 점수 (60)의 학생을 [임의의 순서]에 삽입하는 작업을 성공하였습니다.

! 현재의 리스트 원소들: [ 60 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 9

! Add 작업을 실행합니다:
? 점수를 입력하시오: 50
! 입력된 점수 (50)의 학생을 [임의의 순서]에 삽입하는 작업을 성공하였습니다.

! 현재의 리스트 원소들: [ 50 60 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
```

```
? 작업 번호를 입력하시오: 9

! Add 작업을 실행합니다:
? 점수를 입력하시오: 80
! 입력된 점수 (80)의 학생을 [임의의 순서]에 삽입하는 작업을 성공하였습니다.

! 현재의 리스트 원소들: [ 80 50 60 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 7

! AddToFirst 작업을 실행합니다:
? 점수를 입력하시오: 30
! 입력된 점수 (30)의 학생을 [맨 앞]에 삽입하는 작업을 성공하였습니다.

! 현재의 리스트 원소들: [ 30 80 50 60 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 8

! AddToLast 작업을 실행합니다:
? 점수를 입력하시오: 90
! 입력된 점수 (90)의 학생을 [맨 뒤]에 삽입하는 작업을 성공하였습니다.
```

```
! 현재의 리스트 원소들: [ 30 80 50 60 90 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 6

! AddTo 작업을 실행합니다:
? 리스트에서 순서 번호를 입력하시오: 2
? 점수를 입력하시오: 40
! 입력된 순서 [2]에 입력된 점수 (40)의 학생을 삽입하는 작업을 성공하였습니다.

! 현재의 리스트 원소들: [ 30 80 40 50 60 90 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 6

! AddTo 작업을 실행합니다:
? 리스트에서 순서 번호를 입력하시오: 6
? 점수를 입력하시오: 95
! 입력된 순서 [6]에 입력된 점수 (95)의 학생을 삽입하는 작업을 성공하였습니다.

! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 6

! AddTo 작업을 실행합니다:
? 리스트에서 순서 번호를 입력하시오: 8
! 입력된 순서 [8]가 정상 범위 [0..7]에 있지 않습니다.
```

```
! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 1

! DoesContain 작업을 실행합니다:
? 점수를 입력하시오: 70
! 입력된 점수 (70)의 학생이 리스트에 존재하지 않습니다.

! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 1

! DoesContain 작업을 실행합니다:
? 점수를 입력하시오: 90
! 입력된 점수 (90)의 학생이 리스트에 존재합니다.
? 작업 번호를 입력하시오: 2

! ElementAt 작업을 실행합니다:
? 리스트에서 순서 번호를 입력하시오: 7
! 입력된 순서 [7]에 존재하는 학생은 없습니다.

! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 2

! ElementAt 작업을 실행합니다:
? 리스트에서 순서 번호를 입력하시오: 6
! 입력된 순서 [6]의 학생의 점수는 [95]입니다.
```

```
! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 3

! First 작업을 실행합니다:
! [맨 앞] 학생의 점수는 (30) 입니다.

! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 4

! Last 작업을 실행합니다:
! [맨 뒤] 학생의 점수는 (95) 입니다.
! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 5

! OrderOf 작업을 실행합니다:
? 점수를 입력하시오: 85
! 입력된 점수 (85)의 학생이 리스트에 존재하지 않습니다.

! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 5

! OrderOf 작업을 실행합니다:
? 점수를 입력하시오: 80
! 입력된 점수 (80)의 학생의 순서는 [1]입니다.
```

```
! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 10

! RemoveFrom 작업을 실행합니다:
? 리스트에서 순서 번호를 입력하시오: 8
! 입력된 순서 [8]가 정상 범위 [0..6]에 있지 않습니다.
! 입력된 순서 [8]에서 학생을 삭제하는 작업을 실패하였습니다.

! 현재의 리스트 원소들: [ 30 80 40 50 60 90 95 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 10

! RemoveFrom 작업을 실행합니다:
? 리스트에서 순서 번호를 입력하시오: 6
! 입력된 순서 [6]에서 삭제된 학생의 성적은 (95) 입니다.
! 현재의 리스트 원소들: [ 30 80 40 50 60 90 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 11

! RemoveFirst 작업을 실행합니다.
! 삭제된 [맨 앞] 학생의 성적은 (30) 입니다.

! 현재의 리스트 원소들: [ 80 40 50 60 90 ]
```

```
! 현재의 리스트 원소들: [ 80 40 50 60 90 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 12

! RemoveLast 작업을 실행합니다:
! 삭제된 [맨 뒤] 학생의 성적은 (90) 입니다.

! 현재의 리스트 원소들: [ 80 40 50 60 ]
? 작업 번호를 입력하시오: 14

! ReplaceAt 작업을 실행합니다:
? 리스트에서 순서 번호를 입력하시오: 4
! 입력된 순서 [4]가 정상 범위 [0..3]에 있지 않습니다.

! 현재의 리스트 원소들: [ 80 40 50 60 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 14

! ReplaceAt 작업을 실행합니다:
? 리스트에서 순서 번호를 입력하시오: 3
? 점수를 입력하시오: 75
! 주어진 순서 [3]의 학생의 점수가 (75)로 바뀌었습니다.

! 현재의 리스트 원소들: [ 80 40 50 75 ]
```

```
! 현재의 리스트 원소들: [ 80 40 50 75 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 13

! RemoveAny 작업을 실행합니다:
! 삭제된 [임의]의 학생의 성적은 (80) 입니다.

! 현재의 리스트 원소들: [ 40 50 75 ]
> 해야 할 작업의 번호를 선택해야 합니다:
  DoesContain=1, ElementAt=2, First=3, Last=4, OrderOf=5,
  AddTo=6, AddToFirst=7, AddToLast=8, Add=9,
  RemoveFrom=10, RemoveFirst=11, RemoveLast=12, RemoveAny=13, ReplaceAt=14, EndOfRun=99
? 작업 번호를 입력하시오: 99

> 리스트 정보 입니다:
! 학생 수: 3
! 현재의 리스트 원소들: [ 40 50 75 ]

<<< 리스트 기능 확인 프로그램을 종료합니다 >>>
```

2. 결과 분석

학생을 리스트에 추가하거나 삭제하는 등의 모든 작업이 정상적으로 잘 작동한다.

생각해 볼 점에 대한 의견

1. ArrayList 와 LinkedList 의 구현의 차이점, 장단점은?

ArrayList는 내부적으로 배열을 사용해서 구현되었고, LinkedList는 내부적으로 LinkedNode 를 사용하여 구현였다.
자바에서는 배열을 생성하는 당시 배열의 자료형과 배열의 크기를 가지고, 원소를 저장할 공간을 연속적으로 미리 할당한다.
미리 할당해두는 특성으로 인하여, 배열을 생성할 때 배열의 크기와 자료형을 알고있어야 한다.
따라서 프로그램 실행 중에 동적으로 배열의 크기를 변경시키는 것은 불가능하다.
만약 용량을 동적으로 늘리고 싶다면, 용량이 다 찼을 때, 더 큰 용량을 가진 배열을 생성하고, 기존에 존재하던 배열의 원소들을 모두 복사한 뒤 더 큰 용량의 배열을 사용하는 방법을 사용할 수 있지만, 이번 과제에서는 그렇게 하지는 않았다.
배열은 메모리 공간을 연속적으로 사용하기에 크기가 고정적이라는 단점이 있지만, 특정 원소에 인덱스를 통해 임의로 접근할 때는 매우 빠른 속도의 연산이 가능하다. 배열의 시작 주솟값에 (특정 인덱스 * 값 하나의 메모리 크기)를 더해주면, 접근하고자 하는 원소의 주소가 나오기 때문이다.

또한 내부 원소의 수정과, 맨 뒤 원소를 삭제하거나 추가하는 경우 매우 빠른 속도의 연산이 가능하다.
내부 원소의 수정과, 맨 마지막 위치에 원소를 추가하거나 삭제하는 작업은 특정 인덱스에 접근해서 값을 설정해주면 되기 때문에, 이는 인덱스를 통한 접근이 매우 빠른 배열의특성 상 빠른 것이 당연하다.
그러나 맨 뒤가 아닌 원소의 삽입과 삭제의 경우를 생각해 보면, 특정 순서에 원소를 삽입하거나 삭제하는 것은 빠른 속도로 가능하지만, 원소를 삭제하는 경우에는 삭제된 원소 이후 존재하는 원소들의 인덱스를 앞으로 한칸씩 이동시켜야 하고, 원소를 삽입하는 경우에도, 기존에 존재하던 원소들의 인덱스에 +1을 해주어야 하기에, 배열 속 존재하는 원소들이 많아질수록 오랜 시간이 걸리게 된다.

LinkedList의 경우에는 내부적으로 LinkedNode를 사용하여 구현하였으며, LinkedNode는 데이터와, 연결된 다음 순서의 LinkedNode 객체를 가지고 있다.
배열처럼 연속적인 메모리 공간을 사용하는 것이 아니기 때문에, 메모리 공간을 효율적으로 사용하지 못하며, 특정 순서의 원소에 접근하기 위해서는, 처음부터 순차적으로 LinkedNode를 순회해야 하기 때문에, 많은 시간이 소요된다.
그러나 LinkedNode는 크기가 고정적이지 않기 때문에, 컴퓨터의 메모리가 버텨주는 한 계속해서 원소를 추가할 수 있으며, 또한 원소의 삽입과 삭제의 경우에도 LinkedNode가 가리키는 다음 LinkedNode의 주소만 바꾸어주면 되기 때문에, 빠른 연산이 가능하다.
그러나 앞서 이야기했듯이 특정 순서에 접근하는 과정에서 많은 시간이 걸리게 되므로, 항상 ArrayList보다 빠른 속도로 원소의 삽입과 삭제가 가능한 것은 아니다.

즉 ArrayList와 LinkedList 모두 장단점을 가지고 있으며, 따라서 List가 어떤 연산을 많이 수행하게 되는지에 따라 적절한 자료구조를 택하여 사용하여야 효율적이다.

List의 맨 앞, 혹은 중간 순서의 삽입과 삭제가 빈번한 경우, LinkedList로 구현하는 것이 효율적이다.
그러나 List의 맨 뒤의 원소를 삽입하거나 삭제하는 경우, 혹은 리스트 내부의 원소들을 조회하는 작업이 많을 경우,
ArrayList로 구현하는 것이 효율적이다.

2. 반복자의 개념, 반복자 class의 구현 방법, 그리고 사용자가 반복자 객체를 사용하는 방법은?

반복자는 List가 어떻게 구현되었는지에 무관하게, List 내부의 원소들을 일관된 방법으로 순회하기 위해 사용하는 개념이다.

반복자는 리스트 내부의 원소들에 직접 접근하여야 하기 때문에 inner class로 선언하여, 기존 List의 원소들에게 직접 접근할 수 있게 함과 동시에
List의 캡슐화도 지키는 방식으로 구현한다.
List를 어떻게 구현하였는지에 따라, 반복자의 구현 방식이 달라지기 때문에, 이는 개발자가 알아서 잘 구현해 주어야 한다.

사용자는 List로부터 Iterator 객체를 얻어와서, hasNext()와 next()메소드를 통하여 List의 원소들을 조회할 수 있다.
즉 개발자는 사용자가 Iterator 객체을 얻을 수 있도록 메서드를 제공해 주어야 한다.
이후 사용자는 리스트의 원소들을 순회하기 위해, 반복문을 사용할 수 있으며, 반복문의 조건으로 hasNext()를 사용하여, 다음 원소가 있는지
여부를 확인한 이후, next() 메서드를 통해 다음 원소를 가져올 수 있다.

3. Static class 란?

static class 란 객체를 생성하지 않고, 유용한 기능들을 사용할 수 있도록 모든 필드와 메서드를 static으로 설정한 클래스이며 소위 '유틸리티 클래스'라고 불린다.
이번 프로그램에서는 AppView가 static class 예시로 볼 수 있으며, 화면 출력과 관련된 여러 유용한 기능들을 객체를 생성하지 않고도 사용할 수 있게 해준다.
정적 멤버만 담은 유틸리티 클래스는 인스턴스로 만들어 쓰려고 설계한 게 아니다. 하지만 생성자를 명시하지 않으면 컴파일러가 자동으로 기본 생성자를 만들어준다.
즉 인스턴스화를 방지하기 위해서는 private 생성자를 따로 만들어 주어야 하며, 따라서 이번 프로그램에서도 private 생성자를 추가하여 클래스의 인스턴스화를 막았다.

static class 는 분명 나름의 유용한 쓰임세가 있으나, 이는 객체지향적이지 못하다고 생각한다.
이번 프로그램의 AppView를 예를 들자면, AppView는 화면과 관련된 기능을 수행한다. AppView의 정적 메서드들은 모두 콘솔에서 입력을 받거나, 콘솔에 출력하는 기능을 수행한다.
요구사항이 콘솔이 아니라 파일로 출력하도록 바뀌거나, 출력하는 방식을 고를 수 있도록 요구사항이 바뀐다면 AppView는 재사용이 거의 불가능하다.
AppView에 구현되어있는 모든 함수를 다시 작성해야 하며, 심지어는 똑같이 View 기능을 담당하는 클래스를 새로 만들어야 할 수도 있다.
또한 AppView를 사용하는 클래스인 AppController도 자신의 역할과는 상관없는 이유인 '출력 화면의 변경'이라는 이유로 변경될 수 있다.
로 선언한 후, 이를 구현하는 ConsoleAppView등의 구현체를 만들어 Console 화면에 출력하는 기능을 담당하도록 설계할 것이다.
이는 이후 출력되는 화면이 바뀌거나 하는 등의 상황에서, 다른 클래스에 영향을 끼치지 않고 자유롭게 변경될 수 있다.