```
package kr.ac.soongsil.assignment;

/*******************************************************************************
 * This class is a homework assignment;
 * A DoubleLinkedSeq is a collection of double numbers.
 * The sequence can have a special "current element," which is specified and
 * accessed through four methods that are not available in the sequence class
 * (start, getCurrent, advance and isCurrent).
 *
 * Limitations:
 *   Beyond Int.MAX_VALUE elements, the size method
 *   does not work.
 *
 * Note:
 *   This file contains only blank implementations ("stubs")
 *   because this is a Programming Project for my students.
 *******************************************************************************/
public class DoubleLinkedSeq implements Cloneable {
    private Node head;      // head : 리스트의 시작
    private Node tail;      // tail : 리스트의 끝
    private Node cursor;    // cursor : 항상 마지막에 추가된 노드를 가르킴
    private int size;       // 노드 개수


    /**
     * Initialize an empty sequence.
     * 리스트의 초기화
     *
     * @param - none
     *
     * Postcondition:   This sequence is empty.
     *                 생성된 리스트는 비어있습니다.
     **/
    public DoubleLinkedSeq() {
        head = null;
        tail = null;
        cursor = head;
        size = 0;
    }


    /**
     * Add a new element to this sequence, after the current element.
     * 새로운 노드를 리스트 끝에 생성
     *
     * @param data     the new element that is being added
     *                 새로운 노드를 생성하는데
     *                 A new copy of the element has been added to this sequence.
     *                 If there was no current element, then the new element is placed at the end of the
sequence.
     *                 리스트에 노드가 존재하지 않을 경우 리스트의 새로운 노드가 리스트의 끝이 됨
     *                 If there was a current element, then the new element is placed after the current
element.
     *                 리스트에 노드가 존재할 경우 새로운 노드를 리스트의 끝에 삽입
     *                 In all cases, the new element becomes the new current element of this sequence.
     *                 위의 모든 경우가 진행되면 가장 마지막 노드는 새로운 노드여야 함.
     *
     * @throws OutOfMemoryError Indicates insufficient memory for a new node.
     **/
```

```
    public void addAfter(double data) {
        Node newNode = new Node(data);         // 새로운 노드 생성
        if (size == 0) {                      // 리스트에 노드가 없을 경우
            head = newNode;         // head의 다음 노드 주소를 새로운 노드로 지정
            tail = newNode;         // tail의 이전 노드 주소를 새로운 노드로 지정
            cursor = newNode;
        } else if(cursor == tail) {
            newNode.setPrevNode(cursor);       // 새로운 노드의 이전 노드 주소를 커서의 위치로 지정
            cursor.setNextNode(newNode);       // cursor가 가르키는 노드의 다음 노드 주소를 새로운 노드로 지정
            tail = newNode;         // tail의 이전 노드 주소를 새로운 노드로 지정
            cursor = newNode;
        } else {
            newNode.setPrevNode(cursor);
            newNode.setNextNode(cursor.getNextNode());
            cursor.getNextNode().setPrevNode(newNode);
            cursor.setNextNode(newNode);
            cursor = newNode;
        }
        size++;                               // size + 1
    }



    /**
     * Add a new element to this sequence, before the current element.
     * 새로운 노드를 리스트의 시작에 생성
     *
     * @param data     the new element that is being added
     *                 새로운 노드를 생성하는데
     *                 A new copy of the element has been added to this sequence.
     *                 If there was no current element, then the new element is placed at the start of the
sequence.
     *                 리스트에 노드가 존재하지 않을 경우 새로운 노드가 리스트의 시작이 됨.
     *                 If there was a current element, then the new element is placed before the current
element.
     *                 리스트에 노드가 존재할 경우 새로운 노드를 리스트의 시작에 삽입.
     *                 In all cases, the new element becomes the new current element of this sequence.
     *                 위의 모든 경우가 진행되면 시작 노드는 새로운 노드여야 함.
     *
     * @throws OutOfMemoryError Indicates insufficient memory for a new node.
     **/
    public void addBefore(double data) {
        Node newNode = new Node(data);             // 새로운 노드 생성
        if (size == 0) {                          // 리스트에 노드가 없을 경우
            head = newNode;         // head의 다음 노드 주소를 새로운 노드로 지정
            tail = newNode;         // tail의 이전 노드 주소를 새로운 노드로 지정
            cursor = newNode;
        } else if(cursor == head){
            newNode.setNextNode(cursor);           // 새로운 노드의 다음 노드 주소를 cursor가 가르키는 노드로
지정
            head.setPrevNode(newNode);           // cursor가 가르키는 노드의 이전 노드 주소를 새로운 노드로 지정
            head = newNode;             // head의 다음 노드 주소를 새로운 노드로 지정
            cursor = newNode;
        } else {
            newNode.setNextNode(cursor);
            newNode.setPrevNode(cursor.getPrevNode());
            cursor.getPrevNode().setNextNode(newNode);
            cursor.setPrevNode(newNode);
```

```
            cursor = newNode;
        }
        size++;                                 // size + 1
    }


    /**
     * Place the contents of another sequence at the end of this sequence.
     * 파라미터로 전달된 리스트의 내용을 호출된 리스트의 끝에 배치해주는 함수
     *
     * @param list     a sequence whose contents will be placed at the end of this sequence
     *                 호출된 리스트의 끝에 배치 될 리스트
     *
     *
     * Precondition:    The parameter, list, is not null.
     *                 파라미터로 받는 list가 null이 아닐 때
     *
     * Postcondition:   The elements from list have been placed at the end of this sequence.
     *                 파라미터로 전해진 list가 호출된 리스트의 끝에 배치된다.
     *                 The current element of this sequence remains where it was, and the list is also
unchanged.
     *                 호출된 리스트의 내용은 유지가 되며, 파라미터로 전달된 list도 변경이 되지 않는다.
     *
     * @throws IllegalArgumentException Indicates that list is null.
     *                                 추가하려는 리스트가 비어있을 경우
     *
     * @throws NullPointerException    Indicates that reference list is null.
     *                                 추가하는 목적지가 되는 리스트가 비어있을 경우
     *
     * @throws OutOfMemoryError         Indicates insufficient memory to increase the size of this
sequence.
     *                                  리스트에 추가하려고 할때 메모리가 충분하지 못할 경우
     *
     **/
    public void addAll(DoubleLinkedSeq list) {
        if(list == null) {
            throw new IllegalArgumentException("addAll: list is null");
        }
        if(list.size() > 0) {
            if((head == null) || (tail == null)) {
                head = list.head;
                tail = list.tail;
            } else {
                DoubleLinkedSeq listclone = (DoubleLinkedSeq) list.clone();
                tail.setNextNode(listclone.head);
                listclone.head.setPrevNode(tail);
                tail = listclone.tail;
            }
            size += list.size();
        }
    }


    /**
     * Move forward, so that the current element is now the next element in this sequence.
     * cursor가 가리키는 노드를 다음 노드로 변경
     *
     * @param - none
     *
     * Precondition:    isCurrent() returns true.
     *                 isCurrent()가 true를 반환할 때 (cursor가 가리키는 노드가 있음)
     *
     * Postcondition:   If the current element was already the end element of this sequence
```

```
     *                만약 cursor가 리스트의 끝에 존재하는 노드를 가리키고 있을 경우
     *                (with nothing after it), then there is no longer any current element.
     *                (아무것도 하지 않고) cursor는 null을 가리키게 된다.
     *                Otherwise, the new element is the element immediately after the
     *                그렇지 않으면 cursor는 원래 cursor가 가리키던 노드의 다음노드를 가리키게 된다.
     *                original current element.
     *
     * @throws IllegalStateException    Indicates that there is no current element, so advance may not be
called.
     *                                cursor가 아무것도 가리키지 않아서 advance() 메소드가 호출되지 않을 경우
     **/
    public void advance() {
        if(isCurrent() != true) {
            throw new IllegalStateException("advance: cursor is null");
        }
        cursor = cursor.getNextNode();
    }

    public void retreat() {
        if(isCurrent() != true) {
            throw new IllegalStateException("advance: cursor is null");
        }
        cursor = cursor.getPrevNode();
    }

    /**
     * Generate a copy of this sequence.
     * 호출된 리스트를 복사합니다.
     *
     * @param - none
     *
     * @return  The return value is a copy of this sequence. Subsequent changes to the
     *          반환되는 값은 호출된 리스트의 복사본이다. 복사본의 후속 변경 사항은 원본에 영향을 주지 않으며
     *          copy will not affect the original, nor vice versa. Note that the return
     *          반대의 경우에도 마찬가지입니다. 리턴 값은 사용하기 전에 DoubleLinkedSeq 자료형으로 이루어진
     *          value must be type cast to a DoubleLinkedSeq before it can be used.
     *          새로운 변수에 할당되어야 사용이 가능합니다.
     *
     * @throws OutOfMemoryError Indicates insufficient memory for creating the clone.
     *                          복사본 생성에 필요한 메모리가 부족할 경우
     **/
    /*
    public Object clone() {
        DoubleLinkedSeq answer;
        try {
            answer = (DoubleLinkedSeq) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("clone: This class does not implement Cloneable");
        }
        answer.head = Node.listCopy(head);
        answer.cursor = answer.head;
        return answer;
    }
    */
    public Object clone() {
        DoubleLinkedSeq answer;
        try {
            answer = (DoubleLinkedSeq) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("clone: This class does not implement Cloneable");
        }
        //Sequence has no current element
        if (cursor == null) {
```

```
            Node[] newList = Node.listCopyWithTail(head);

            answer.head = newList[0];
            answer.tail = newList[1];
        }
        //Sequence with a current element equal to head
        else if (cursor == head) {
            Node[] newList = Node.listCopyWithTail(head);
            answer.head = newList[0];
            answer.tail = newList[1];

            answer.cursor = answer.head;
        }

        else if (cursor != null) {
            this.start();
            Node[] newList = Node.listCopyWithTail(head);
            answer.head = newList[0];
            answer.tail = newList[1];

            answer.cursor = answer.head;
        }

        return answer;
    }


    /**
     * Accessor method to get the current element of this sequence.
     * 리스트에 존재하는 cursor의 data를 얻기 위한 Accessor 메소드임.
     *
     * @param - none
     *
     * Precondition:    isCurrent() returns true.
     *                 isCurrent()가 참을 반환할 때
     *
     * @return  cursor가 가리키는 노드의 data
     *
     * @throws IllegalStateException    Indicates that there is no current element, so getCurrent may not
be called.
     *                                cursor가 아무것도 가리키지 않음으로 인해 getCurrent()가 호출되지 않을 수
있음을 처리
     **/
    public double getCurrent() {
        if (!isCurrent()) {
            throw new IllegalStateException("getCurrent: isCurrent() is null");
        } else {
            return cursor.getData();
        }
    }


    /**
     * Accessor method to determine whether this sequence has a specified
     * 이 리스트에 getCurrent()로 cursor의 data를 얻기 위해
     * current element that can be retrieved with the getCurrent method.
     * cursor의 노드 지정 유무를 판별하는 메소드.
     *
     * @param - none
     *
     * @return true(cursor가 가리키는 노드가 있음) or false(cursor가 가리키는 노드가 없음)
     **/
    public boolean isCurrent() {
```

```
       return cursor != null;
    }

    /**
     * Remove the current element from this sequence.
     * 리스트에서 cursor가 가리키는 노드를 삭제합니다.
     *
     * @param - none
     *
     * Precondition:    isCurrent() returns true.
     *                 isCurrent()가 true를 반환할 때
     *
     * Postcondition:   The current element has been removed from this sequence, and the
     *                 cursor가 지정하는 노드가 리스트에서 제거됩니다. 그리고 cursor는 제거된 노드의
     *                 following element (if there is one) is now the new current element.
     *                 다음 노드의 위치로 이동합니다. 만약 제거된 노드의 다음 노드가 없을 경우 cursor는 null이
된니다.
     *                 If there was no following element, then there is now no current element.
     *
     * @throws IllegalStateException Indicates that there is no current element, so removeCurrent may not
be called.
     **/
    public void removeCurrent() {
       if(!isCurrent()) {         // cursor가 가리키는 노드가 없음
           throw new IllegalStateException("removeCurrent: isCurrent() is null");
       } else if(size() == 0) {    // 리스트가 비어있음
           throw new IllegalStateException("removeCurrent: list is empty");
       } else if(size() == 1) {    // 리스트에 노드가 1개밖에 없음
           head = null;
           tail = null;
           cursor = null;
       } else if(cursor == head) { // cursor가 시작 노드에 있을 경우
           head = head.getNextNode();
           cursor = cursor.getNextNode();
           cursor.setPrevNode(null);
       } else if(cursor == tail) { // cursor가 끝 노드에 있을 경우
           tail = tail.getPrevNode();
           cursor = cursor.getPrevNode();
           cursor.setNextNode(null);
       } else {                   // cursor가 리스트 중간 노드에 있을 경우
           Node temp = head;
           while(temp.getNextNode() != cursor) {
               temp = temp.getNextNode();
           }
           temp.setNextNode(cursor.getNextNode());
           temp = temp.getNextNode();
           temp.setPrevNode(cursor.getPrevNode());
           cursor = cursor.getNextNode();
       }
       size--;
    }


    /**
     * Determine the number of elements in this sequence.
     * 리스트의 노드의 개수를 결정합니다.
     *
     * @param - none
     * @return  the number of elements in this sequence
     *          리스트에 존재하는 노드의 개수 반환
     **/
```

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


    /**
     * Set the current element at the front of this sequence.
     * 이 리스트의 맨 앞 노드에 cursor를 지정합니다.
     *
     * @param - none
     *
     * Postcondition:   The front element of this sequence is now the current element (but
     *                 cursor가 리스트의 맨 앞 노드를 가리킵니다. 그러나 리스트에 노드가 없는 경우
     *                 if this sequence has no elements at all, then there is no current element).
     *                 cursor는 null을 가리킵니다.
     **/
    public void start() {
        if(head == null) {
            cursor = null;
        }
        cursor = head;
    }

    /**
     * Set the current element at the end of this sequence.
     * 이 리스트의 맨 뒤에 cursor를 지정합니다.
     *
     * @param - none
     *
     * Postcondition:   The end element of this sequence is now the current element
     *                 cursor가 리스트의 맨 뒤 노드를 가리킵니다. 그러나 리스트에 노드가 없는 경우
     *                 (but if this sequence has no elements at all, then there is no current element).
     *                 cursor는 null을 가리킵니다.
     **/
    public void end() {
        if(tail == null) {
            cursor = null;
        }
        cursor = tail;
    }

    @Override
    public String toString() {
        String information = "Size: " + size() + "\n";
        information += "Current Node: " + (cursor != null ? cursor.getData() : "null") + "\n";
        information += "Nodes: [";
        Node cursor = head;
        while(cursor != null) {
            information += cursor.getData();
            if(cursor.getNextNode() != null) {
                information += ", ";
            }
            cursor = cursor.getNextNode();
        }
        information += "]\n";
        return information;
    }
}
```

Node.java

```
package kr.ac.soongsil.assignment;

public class Node {
    private double data;       // 데이터를 저장하는 공간
    private Node prevNode;     // 이전 노드
    private Node nextNode;     // 다음 노드

    Node(double data) {
        this.data = data;      // 데이터 지정
        this.nextNode = null;  // 다음 노드는 지정하지 않음.
    }

    public void addNodeAfter(double data) {
        Node newNode = new Node(data);
        newNode.setPrevNode(this);
        this.setNextNode(newNode);
    }

    public static Node listCopy(Node list) {
        Node copyHead = null;
        Node copyTail = null;
        Node cursor = list;
        if (list == null) {    // 리스트가 비어있을 경우
            return null;
        }
        copyHead = new Node(list.getData());    // 첫 노드 생성
        copyTail = copyHead;                   // copyHead = copyTail = 첫 노드
        while (cursor.getNextNode() != null) {   // cursor가 마지막 노드까지 이동
            cursor = cursor.getNextNode();
            copyTail.addNodeAfter(cursor.getData());
            copyTail = copyTail.getNextNode();
        }
        return copyHead;
    }

    public static Node[] listCopyWithTail(Node list) {
        Node[] answer = new Node[2];
        Node cursor = list;

        // Handle the special case of the empty list.
        if (list != null) {
            // Make the first node for the newly created list.
            Node copyHead = new Node(list.getData());
            Node copyTail = copyHead;

            // Make the rest of the nodes for the newly created list.
            while (cursor.getNextNode() != null) {
                cursor = cursor.getNextNode();
                copyTail.addNodeAfter(cursor.getData());
                copyTail = copyTail.getNextNode();
            }
            // Return the head and tail references.
            answer[0] = copyHead;
            answer[1] = copyTail;
        }
        return answer;
    }

    public void setData(double newdata) {
        data = newdata;
```

```
    }

    public void setPrevNode(Node node) {
        prevNode = node;
    }

    public void setNextNode(Node node) {
        nextNode = node;
    }

    public double getData() {
        return data;
    }

    public Node getNextNode() {
        return nextNode;
    }

    public Node getPrevNode() {
        return prevNode;
    }

}
```

Main.java

```
package kr.ac.soongsil.assignment;

public class Main {
    public static void main(String args[]) {
        DoubleLinkedSeq list[] = new DoubleLinkedSeq[3];
        DoubleLinkedSeq l1;

        for (int i = 0; i < 3; i++) {
            list[i] = new DoubleLinkedSeq();
        }

        //addBefore도 가능
        list[0].addAfter(1);
        list[0].addAfter(2);
        list[0].addAfter(3);
        list[1].addAfter(4);
        list[1].addAfter(5);
        list[1].addAfter(6);

        list[2] = (DoubleLinkedSeq) list[1].clone();
        l1 = (DoubleLinkedSeq) list[2].clone();


        list[0].start();
        list[1].end();
        list[0].removeCurrent();
        list[1].removeCurrent();


        list[2].addAll(list[1]);
        l1.addAll(list[0]);

        System.out.println("=========List[0]=========");
        printList(list[0]);
        System.out.println("=========List[1]=========");
        printList(list[1]);
        System.out.println("=========List[2]=========");
        printList(list[2]);
        System.out.println("========== l1 ==========");
        printList(l1);

    }

    public static void printList(DoubleLinkedSeq list) {
        String information = list.toString();
        System.out.println(information);
    }
}
```