자료구조 실습 보고서

[제10주] 큐(Queue)

2021년 05월 10일

201702039 오명주

1. 프로그램 설명서

(1) 프로그램의 전체 설계 구조

➔ MVC (Model – View – Controller) 구조

Model : 프로그램이 "무엇"을 할 것인지 정의. 사용자의 요청에 맞는 알고리즘을 처리하고 DB와 상호작용하여 결과물을 산출하고 Controller에게 전달.

View : 화면에 무엇인가를 "보여주기 위한" 역할. 최종 사용자에게 "무엇"을 화면으로 보여줌.

Controller : 모델이 "어떻게" 처리할 지 알려주는 역할. 사용자로부터 입력을 받고 중개인 역할. Model과 View는 서로 직접 주고받을 수 없음. Controller을 통해 이야기함.

➔ ArrayQueue 프로그램에서의 각 클래스 별 MVC 구조 역할

Model :

- Interface Queue : Queue의 변수와 속성을 구성한다.
- CircularArrayQueue : Queue를 이용한 순환 배열 큐를 구성한다.
- Iterator : 반복자의 변수와 함수로 이루어져있다.

View :

- AppView : 프로그램의 입/출력을 담당한다.

Controller :

- AppController : Model을 통해 생성된 결과물을 AppView를 통해 출력한다.

➔ LinkedQueue 프로그램에서의 각 클래스 별 MVC 구조 역할

Model :

- Interface Queue : Queue의 변수와 속성을 구성한다.
- CircularlyLinkedQueue : Queue를 이용한 순환 연결리스트 큐를 구성한다.
- Iterator : 반복자의 변수와 함수로 이루어져 있다.
- LinkedNode<E> : 연결 노드에 대한 변수와 속성을 구성한다.

(2) 함수 설명서

➔ 주요 알고리즘

1) CircularArrayQueue

```
public class CircularArrayQueue<E> implements Queue<E> {
    private static final int DEFAULT_CAPACITY = 100;

    private int _maxLength; // capacity + 1
    private int _frontPosition;
    private int _rearPosition;
    private E[] _elements;

    // 생성자
    @SuppressWarnings("unchecked")
    public CircularArrayQueue(int givenCapacity) {
        this.setMaxLength(givenCapacity + 1);
        this.setFrontPosition(0);
        this.setRearPosition(0);
        this.setElements((E[]) new Object[this.maxLength()]);
    }
```

배열로 구현한 순환 원형 큐 클래스이다.

⇨ 배열의 크기가 10이라면 maxlength를 10으로 설정, capacity는 9로 설정함으로써 9만큼 원소가 들어가면 배열이 가득 찼다고 판단한다. rear=0, front=0이면 empty인지 full인지 모르기 때문에 front=0, rear=9면 가득 찼다고 판단한다.

⇨ 생성자 초기화는 maxlength로 진행한다.

```
@Override
public boolean enQueue(E anElement) { // 큐 rear 위치에 원소 추가
    if (this.isFull()) { // 큐가 가득 찼다면
        return false; // false를 반환
    } else {
        if (this.isEmpty()) { // 만약 큐가 비어있을 경우
            this.elements()[this.rearPosition()] = anElement; // rear 현재 위치에 원소를 추가
            this.setFrontPosition((this.frontPosition() + this.maxLength() - 1) % this.maxLength()); // front를 하나 전으로 밀고 rear을 현재 위치에 삽입
            return true;
        } // 만약 큐에 원소가 있다면
        this.setRearPosition((this.rearPosition() + 1) % this.maxLength()); // rear을 다음 위치로 설정
        this.elements()[this.rearPosition()] = anElement; // rear위치에 원소를 추가
        return true;
    }
}
```

rear 위치에 원소를 추가하는 enQueue 함수.

⇨ 만약 큐가 full 상태라면 false를 반환한다.

⇨ 만약 큐가 비어 있는 상태라면, rear와 front 위치가 같기 때문에 front를 뒤로 밀고 현재 rear자리에 원소를 삽입한다. 이때 front 자리를 0인경우가 있기 때문에 maxlength를 더한 후 빼준다. 나머지 연산을 통해 0->10으로 가는 경우를 계산한다.

⇨ 만약 큐에 원소가 있다면 setRearPosition을 이용하여 다음 rear위치를 잡는다. 이때 나머지 연산을 하는 이유는 10->0 인덱스를 계산해주기 위함이다. rearPosition에 원소를 추가하고 true를 반환한다.

```
@Override
public E deQueue() { // 큐 front 위치 원소 삭제
    E frontElement = null; // 비어있다면 null을 반환
    if (!this.isEmpty()) { // 비어있지않다면
        this.setFrontPosition((this.frontPosition() + 1) % this.maxLength()); // front를 다음 위치로 설정
        frontElement = this.elements()[this.frontPosition()]; // frontPosition의 원소를 변수에 저장
        this.elements()[this.frontPosition()] = null; // 삭제
    }
    return frontElement; // 삭제된 원소 반환
}
```

front 위치에 원소를 삭제하는 deQueue 함수.

⇨ 삭제할 원소를 저장할 변수를 선언하고 null로 초기화한다.

⇨ 큐가 empty 상태가 아니라면

⇨ frontPosition을 다음 위치로 설정한다. front는 null을 가리키는 상태. 이때 나머지 연산을 하는 이유는 10->0 인덱스를 계산해주기 위함이다.

⇨ frontPosition의 현재 위치를 변수에 저장하고 null로 삭제를 진행한다.

⇨ 저장한 변수를 반환한다.

```
// 큐의 모든 원소를 Front 부터 Rear까지 출력한다
private void showAllFromFront() {
    AppView.output("[Queue] <Front> ");
    Iterator<Character> queueIterator = this.queue().iterator(); // queue()의 반복자 생성
    while (queueIterator.hasNext()) { // queueIterator의 next가 있는 동안 반복
        Character element = queueIterator.next(); // 순서대로 element에 저장
        AppView.output(element.toString() + " "); // element의 문자열 출력
    }
    AppView.outputLine("<Rear> ");
}
```

큐의 모든 원소를 front부터 rear까지 출력하는 showAllFromFront 함수.

⇨ 반복자를 이용하여 큐의 next가 있는 동안 반복한다.

⇨ element에 원소를 저장하여 차례로 출력한다.

```
// 큐의 모든 원소를 Rear 부터 Front까지 출력한다
private void showAllFromRear() {
    AppView.output("[Queue] <Rear> ");
    if (this.queue().rearPosition() >= this.queue().frontPosition()) { // front - rear 순으로 있는 경우
        for (int order = this.queue().rearPosition(); order > this.queue().frontPosition(); order--) { // rearPosition부터 frontPosition까지
            AppView.output(this.queue().elementAt(order).toString() + " "); // 원소를 출력한다
```

큐의 모든 원소를 rear부터 front까지 출력하는 showAllFromRear 함수.

⇨ front가 rear보다 앞에있는 경우 (인덱스 숫자가 작은경우)

⇨ rearPosition부터 frontPosition까지 역순으로 출력

```
    } else { // rear - front 순으로 있는 경우
        for (int order = this.queue().rearPosition(); order >= 0; order--) { // 0 - rearPosition 를 먼저 출력
            AppView.output(this.queue().elementAt(order).toString() + " ");
        }
        for (int order = AppController.QUEUE_CAPACITY; order > this.queue().frontPosition(); order--) { // front - MAX 까지 출력
            AppView.output(this.queue().elementAt(order).toString() + " ");
        }
    }
    AppView.outputLine("<Front> ");
```

⇨ rear가 front보다 앞에있는 경우,

⇨ rearPosition부터 0까지 역순으로 출력 후, 배열의 MAX부터 frontPosition까지 역순으로 출력한다.

2) CircularlyLinkedQueue

```
public class CircularlyLinkedQueue<E> implements Queue<E> {

    private int _size;
    private LinkedNode<E> _rearNode;

    // 생성자
    public CircularlyLinkedQueue() {
        this.setSize(0);
        this.setRearNode(null);
    }
```

연결리스트로 구현한 순환 원형 큐 클래스이다.

⇨ frontNode를 rearNode의 next로 정의하기 때문에 rearNode와 size만 선언한다.

⇨ size는 null로, rearNode는 null로 초기화한다.

```java
@Override
public boolean enQueue(E anElement) { // anElement 큐를 rear에 추가
    LinkedNode<E> newRearNode = new LinkedNode<E>(anElement, null); // 새로운 Node생성
    if (this.isEmpty()) { // 만약 비어있다면
        newRearNode.setNext(newRearNode); // newRearNode의 다음을 newRearNode로 지정
    } else { // 비어있지 않다면
        newRearNode.setNext(this.rearNode().next()); // newRearNode의 다음을 현 rearNode의 다음 노드로 지정
        this.rearNode().setNext(newRearNode); // rearNode의 다음을 newRearNode로 지정
    }
    this.setRearNode(newRearNode); // newRearNode로 rearNode 설정
    this.setSize(this.size() + 1); // size ++
    return true;
```

rear 위치에 원소를 추가하는 enQueue 함수.

⇨ anElement를 element로 하는 newRearNode를 선언한다.

⇨ 만약 큐가 비어있다면 하나의 원소만 순환하도록 자기자신을 setNext로 지정한다.

⇨ 만약 비어있다면 newRearNode의 다음노드를 현재 rearNode로 지정하고 readNode의 다음 노드를 newRearNode로 지정한다.

⇨ readNode를 newRearNode로 바꾸어 새로 지정해주고 사이즈를 +1 한다.

```java
@Override
public E deQueue() { // front의 원소를 삭제
    E frontElement = null;
    if (!this.isEmpty()) { // 비어있지않다면
        frontElement = this.rearNode().next().element(); // front노드는 rear의 next 노드
        if (this.rearNode() == this.rearNode().next()) { // 큐에 노드가 1개라면
            this.setRearNode(null); // 해당 노드를 삭제
        } else { // 노드가 2개 이상
            this.rearNode().setNext(this.rearNode().next().next()); // front노드를 삭제
        }
        this.setSize(this.size() - 1);
    }
    return frontElement;
}
```

front 위치의 원소를 삭제하는 deQueue 함수

⇨ 삭제할 원소를 저장할 변수를 선언하고 null로 초기화한다.

⇨ 만약 큐가 비어있지 않다면 readNode의 next가 frontNode임으로 frontNode의 element를 변수에 저장한다.

⇨ 만약 큐에 노드가 하나라면, 해당노드를 null처리하여 삭제한다.

⇨ 만약 큐에 노드가 2개 이상이라면 frontNode를 삭제한다.

⇨ 사이즈는 -1하여 설정한다.

⇨ 삭제 원소를 반환

```
// 큐의 모든 원소를 Front 부터 Rear까지 출력한다
private void showAllFromFront() {
    AppView.output("[Queue] <Front> ");
    Iterator<Character> queueIterator = this.queue().iterator(); // queue()의 반복자 생성
    while (queueIterator.hasNext()) { // queueIterator의 next가 있는 동안 반복
        Character element = queueIterator.next(); // 순서대로 element에 저장
        AppView.output(element.toString() + " "); // element의 문자열 출력
    }
    AppView.outputLine("<Rear> ");
}
```

큐의 모든 원소를 front부터 rear까지 출력하는 showAllFromFront 함수.

⇨ 반복자를 이용하여 큐의 next가 있는 동안 반복한다.

⇨ element에 원소를 저장하여 차례로 출력한다.

```
// 큐의 모든 원소를 Rear 부터 Front까지 출력한다
private void showAllFromRear() {
    AppView.output("[Queue] <Rear> ");
    for (int order = this.queue().size() - 1; order >= 0; order--) {
        AppView.output(this.queue().elementAt(order).toString() + " ");
    }
    AppView.outputLine("<Front> ");
}
```

큐의 모든 원소를 rear부터 front까지 출력하는 showAllFromRear 함수.

⇨ 배열과는 다르게 연결리스트는 null이 없기 때문에 size-1만큼 걸어 나가 해당 원소를 출력한다.

```
@Override
public E elementAt(int anOrder) { // anOrder 번 째 E 반환
    LinkedNode<E> frontNode = this.rearNode().next();
    if (anOrder < 0 || anOrder > this.size() - 1) { // anOrder이 범위 밖이면
        return null; // null 반환
    } else {
        for (int order = 0; order < anOrder; order++) {
            frontNode = frontNode.next();
        }
        return frontNode.element(); // frontNode 반환
    }
}
```

⇨ 주어진 anOrder만큼 next를 하여 걸어나가 해당 원소를 반환한다.

3) 종합 설명서

➔ 프로그램 실행 순서대로 설명해보자.

```
public class _DS10_1_201702039_오명주 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AppController appController = new AppController() ;
        // AppController가 실질적인 main class 이다
        appController.run() ;
        //여기 main() 에서는 앱 실행이 시작되도록 해주는 일이 전부이다
```

main에서 AppController 의 객체를 생성하여 run 한다. 프로그램을 실행한다.

```
public void run() {
    AppView.outputLine("<<< 큐 기능 확인 프로그램을 시작합니다 >>>");
    AppView.outputLine("");
    char input = this.inputChar(); // input 입력받기
    while (input != '!') { // !는 종료
        this.countInputChars();
        if ((Character.isAlphabetic(input))) { // 만약 입력 문자가 알파벳이면
            this.addToQueue(Character.valueOf(input)); // Queue에 삽입
        } else if (Character.isDigit(input)) { // 만약 입력 문자가 숫자면
            this.removeN(Character.getNumericValue(input)); // 입력문자만큼 삭제
        } else if (input == '-') { // - 는 1개 삭제
            this.removeOne();
        } else if (input == '#') { // # 은 크기 출력
            this.showQueueSize();
        } else if (input == '/') { // / 는 front부터 rear 순서 출력
            this.showAllFromFront();
        } else if (input == '\\') { // \ 는 rear부터 front 순서 출력
            this.showAllFromRear();
        } else if (input == '<') { // < 는 front원소 출력
            this.showFrontElement();
        } else if (input == '>') { // > 는 rear원소 출력
            this.showRearElement();
        } else { // 그 외에
            AppView.outputLine("[Ignored] 의미 없는 문자가 입력되었습니다.");
            this.countIgnoredChars();
        }
        input = this.inputChar();
    }
    this.quitQueueProcessing();
    this.showStatistics();
    AppView.outputLine("");
    AppView.outputLine("<<< 큐 기능 확인 프로그램을 종료합니다 >>>");
}
```

⇨ AppController의 run 함수에서는 프로그램을 실행한다.

⇨ !가 입력되면 종료된다.

⇨ 입력문자가 문자/알파벳이면 addToQueue 함수를 통해 큐에 삽입한다.

⇨ 입력문자가 숫자면 removeN을 실행하여 입력문자만큼 삭제한다.

⇨ 각각의 입력문자에 맞는 처리를 해준다.

⇨ 그외 문자는 의미없는 문자로 count한다.

⇨ '!'가 입력되면 통계를 출력하고 프로그램을 종료한다.

```
while (count < numberOfCharsToBeRemoved && (!this.queue().isEmpty())) { // queue가 비거나 count가 인자보다 커지면 탈출
    Character removedChar = this.queue().deQueue();
    if (removedChar == null) {
        AppView.outputLine("(오류) 큐에서 삭제하는 동안에 오류가 발생하였습니다.");
    } else {
        AppView.outputLine("[DeQs] 삭제한 원소는 '" + removedChar + "'입니다.");
        count++;
    }
}
if (count < numberOfCharsToBeRemoved) {
    AppView.outputLine("[DeQs.Empty] 큐에 더 이상 삭제할 원소가 없습니다.");
```

입력 받은 numberOfCharsToBeRemoved 정수만큼 && 큐가 empty가 아닌 동안 removeN을 반복한다.

만약 큐가 empty가 되어서 반복문을 탈출했을 때 입력받은 정수보다 count가 작으면 삭제할 원소가 없다는 출력문을 출력한다.

2. 프로그램 장단점 / 특이점 분석

➔ Array를 이용한 순환 원형 큐 장점

- 배열로 구현하여 인덱스를 통한 원소 접근이 용이하다. 삭제 시, front 인덱스를 통하여 가능하고 삽입 시 rear 인덱스를 통해 쉽게 가능하다.
- Iterator을 통해 모든 원소를 차례로 출력할 수 있어 편리하였다.
- 일반 배열이라면 맨 앞 원소 삭제 시 뒤 모든 배열을 한칸씩 앞으로 옮겨야 하는 불편함이 있지만 순환 원형 배열이기 때문에 frontPosition 인덱스만 옮기면 되어서 편리하였다.

➔ Array를 이용한 순환 원형 큐 단점

- MAX_CAPACITY를 미리 지정하여 최대 크기에 제한이 있다.
- 모든 원소 출력 시 front-rear 출력은 Iterator을 통해 쉽게 할 수 있었지만 rear-front 역순 출력은 배열에 null이 존재하기 때문에 frontPosition이 rearPosition보다 앞에 있는 경우와 뒤에 있는 경우를 나누어 생각하여야 하기 때문에 코드 구현이 길어졌다.

➔ LinkedNode를 이용한 순환 원형 큐 장점

- 연결 리스트를 이용하여 구현하였기 때문에 크기에 대한 제한이 없다.
- 배열에 비해 큐의 모든 원소를 출력하는게 쉽다. elementAt 함수를 이용하여 size-1 ~ 0까지 한번에 역순으로 출력이 가능하다.
- 순환 원형이기 때문에 rearNode의 next가 frontNode라고 정의할 수 있기때문에 따로 frontNode를 정의하지 않아도 되었다.

➔ LinkedNode를 이용한 순환 원형 큐 단점

- 배열에 비해 LinkedNode 클래스에 대한 정의가 필요하였다.

3. 실행 결과 분석

(1) 입력과 출력

1) ArrayCircularQueue

```
<terminated> _DS10_1_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 5. 11. 오후 10:52:21)
<<< 큐 기능 확인 프로그램을 시작합니다 >>>

? 문자를 입력하시오 : a
[EnQ] 삽입된 원소는 'a'입니다.
? 문자를 입력하시오 : b
[EnQ] 삽입된 원소는 'b'입니다.
? 문자를 입력하시오 : c
[EnQ] 삽입된 원소는 'c'입니다.
? 문자를 입력하시오 : d
[EnQ] 삽입된 원소는 'd'입니다.
? 문자를 입력하시오 : e
[EnQ] 삽입된 원소는 'e'입니다.
? 문자를 입력하시오 : #
[Size] 큐의 원소의 개수는 5개 입니다.
? 문자를 입력하시오 : /
[Queue] <Front> a b c d e <Rear>
? 문자를 입력하시오 : 2
[DeQs] 삭제한 원소는 'a'입니다.
[DeQs] 삭제한 원소는 'b'입니다.
? 문자를 입력하시오 : \
[Queue] <Rear> e d c <Front>
? 문자를 입력하시오 : <
[Front] 큐의 맨 앞 원소는 c 입니다.
? 문자를 입력하시오 : >
[Rear] 큐의 맨 뒤 원소는 e 입니다.
? 문자를 입력하시오 : !
```

```
<큐를 비우고 사용을 종료합니다>
[Queue] <Front> c d e <Rear>
[DeQs] 삭제한 원소는 'c'입니다.
[DeQs] 삭제한 원소는 'd'입니다.
[DeQs] 삭제한 원소는 'e'입니다.

<큐 사용 통계>
- 입력된 문자는 11개 입니다.
- 정상처리된 문자는 11개 입니다.
- 무시된 문자는 0개 입니다.
- 삽입된 문자는 5개 입니다.

<<< 큐 기능 확인 프로그램을 종료합니다 >>>
```

2) LinkedCircularyQueue

```
<terminated> _DS10_2_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 5. 11. 오후 10:54:35)
<<< 큐 기능 확인 프로그램을 시작합니다 >>>

? 문자를 입력하시오 : a
[EnQ] 삽입된 원소는 'a'입니다.
? 문자를 입력하시오 : b
[EnQ] 삽입된 원소는 'b'입니다.
? 문자를 입력하시오 : c
[EnQ] 삽입된 원소는 'c'입니다.
? 문자를 입력하시오 : d
[EnQ] 삽입된 원소는 'd'입니다.
? 문자를 입력하시오 : -
[DeQ] 삭제된 원소는 'a'입니다.
? 문자를 입력하시오 : <
[Front] 큐의 맨 앞 원소는 b 입니다.
? 문자를 입력하시오 : /
[Queue] <Front> b c d <Rear>
? 문자를 입력하시오 : >
[Rear] 큐의 맨 뒤 원소는 d 입니다.
? 문자를 입력하시오 : \
[Queue] <Rear> d c b <Front>
? 문자를 입력하시오 : 3
[DeQs] 삭제한 원소는 'b'입니다.
[DeQs] 삭제한 원소는 'c'입니다.
[DeQs] 삭제한 원소는 'd'입니다.
? 문자를 입력하시오 : !

<큐를 비우고 사용을 종료합니다>
[Queue] <Front> <Rear>
[DeQs] 삭제할 원소의 개수가 0개입니다.

<큐 사용 통계>
- 입력된 문자는 10개 입니다.
- 정상처리된 문자는 10개 입니다.
- 무시된 문자는 0개 입니다.
- 삽입된 문자는 4개 입니다.

<<< 큐 기능 확인 프로그램을 종료합니다 >>>
```

[예외 처리]

```
? 문자를 입력하시오 : f
[EnQ] 삽입된 원소는 'f'입니다.
? 문자를 입력하시오 : d
[EnQ.Full] 큐가 꽉 차서 더 이상 넣을 수가 없습니다.
? 문자를 입력하시오 : d
[EnQ.Full] 큐가 꽉 차서 더 이상 넣을 수가 없습니다.
? 문자를 입력하시오 : w
[EnQ.Full] 큐가 꽉 차서 더 이상 넣을 수가 없습니다.
```

➔ ArrayCircularQueue에 MAX_CAPACITY 이상 넣은 경우

➔ 의미 없는 문자를 입력한 경우

(2) 결과 분석 (자신의 논리적 평가, 기타 느낀 점)

- 지난 과제들에서 구현한 LinkedNode, Iterator, Interface 를 이용한 자료구조, 등을 이용하여 쉽게 구현할 수 있었다.
- Stack보다는 Queue가 더 효율적인 자료구조라는 생각이 들었다. 마트 계산대, 프로세스 관리 등 일상생활에서 쓰이는 경우가 더 많다.

➔ 생각해 볼 점

1. 구현에 있어서 리스트와 배열의 장단점

	장점	단점
Array	- 인덱스를 통한 각각의 원소에 대한 접근이 편리하다.	- 최대 크기에 대한 제한이 존재한다. - 순서대로 출력할 때 rear - front 순으로 있다면 출력이 길어진다. - 인덱스 예외와 널 포인터 예외가 생길 수 있다. - 배열이 삭제되면 공간이 남아 메모리가 낭비된다.
LinkedList	- 크기에 대한 제한이 없다. - 메모리 재사용이 편리하다. - rear의 next가 front 로 정의해둬서 front에 대한 접근이 쉬웠다.	- 각각의 원소에 대한 접근이 배열에 비해 반복적이다. (head부터 걸어 나감)

2. Queue를 Interface로 선언하는 이유는?

- 다른 방식으로 Queue를 구현하는데 용이하다. Array와 LinkedList를 통한 Queue를 구현할 때 Queue 인터페이스는 그대로 두고 자료구조만 바꾸어 implement를 이용하여 구현하였다.

4. 소스코드

```
import java.util.Scanner;

public class AppView {

    private static Scanner scanner = new Scanner(System.in);

    // 생성자
    public AppView() {

    }

    // 출력 관련 함수
    // 한줄을 출력하는 함수 (한줄이 띄워지지않는다)
    public static void output(String message) {
        System.out.print(message); // 입력받은 message를 출력한다
    }

    // 한줄을 출력하는 함수 (한줄이 띄워진다)
    public static void outputLine(String message) {
        System.out.println(message); // 입력받은 message를 출력한다
    }

    // 입력 관련 함수
    public static char inputChar() {
        String line = AppView.scanner.nextLine().trim();
        while(line.equals("")) {
            line = AppView.scanner.nextLine().trim();
        }
        return line.charAt(0);
    }
}
```

[AppView]

```
public interface Iterator<T> {
    public boolean hasNext();
    public T next();
}
```

[Iterator]

```
public interface Queue<E> {
        public int size();
        public boolean isFull();
        public boolean isEmpty();

        public E front();
        public E rear();

        public boolean enQueue(E anElement);
        public E deQueue();

        public void clear();

        public E elementAt(int anOrder);
        public Iterator<E> iterator();
        public int frontPosition();
        public int rearPosition();
    }
```

[Queue]

➔ Array

```
private Queue<Character> _queue;
private int _inputChars; // 입력된 문자의 개수
private int _addedChars; // 삽입된 문자의 개수
private int _ignoredChars; // 무시된 문자의 개수

// Getters/Setters
private Queue<Character> queue() {
    return this._queue;
}

private void setQueue(Queue<Character> newQueue) {
    this._queue = newQueue;
}

private int inputChars() {
    return this._inputChars;
}

private void setInputChars(int newInputChars) {
    this._inputChars = newInputChars;
}

private int addedChars() {
    return this._addedChars;
}

private void setAddedChars(int newAddedChars) {
    this._addedChars = newAddedChars;
}

private int ignoredChars() {
    return this._ignoredChars;
}

private void setIgnoredChars(int newIgnoredChars) {
    this._ignoredChars = newIgnoredChars;
}

// 생성자
public AppController() {
    this.setQueue(new CircularArrayQueue<Character>(AppController.QUEUE_CAPACITY));
    this.setInputChars(0);
    this.setAddedChars(0);
    this.setIgnoredChars(0);
}

private void countInputChars() { // 입력문자 횟수계산
    this.setInputChars(this.inputChars() + 1);
}

private void countIgnoredChars() { // 무시문자 횟수계산
    this.setIgnoredChars(this.ignoredChars() + 1);
}

private void countAddedChar() { // 추가문자 횟수계산
    this.setAddedChars(this.addedChars() + 1);
}

// 큐의 모든 원소를 Front 부터 Rear까지 출력한다
private void showAllFromFront() {
    AppView.output("[Queue] <Front> ");
    Iterator<Character> queueIterator = this.queue().iterator(); // queue()의 반복자 생성
    while (queueIterator.hasNext()) { // queueIterator의 next가 있는 동안 반복
        Character element = queueIterator.next(); // 순서대로 element에 저장
        AppView.output(element.toString() + " "); // element의 문자열 출력
    }
    AppView.outputLine("<Rear> ");
}

// 큐의 모든 원소를 Rear 부터 Front까지 출력한다
private void showAllFromRear() {
    AppView.output("[Queue] <Rear> ");
    if (this.queue().rearPosition() >= this.queue().frontPosition()) { // front - rear 순으로 있는 경우
        for (int order = this.queue().rearPosition(); order > this.queue().frontPosition(); order--) { // rearPosition부터 frontPosition까지
            AppView.output(this.queue().elementAt(order).toString() + " "); // 원소를 출력한다
        }
    } else { // rear - front 순으로 있는 경우
        for (int order = this.queue().rearPosition(); order >= 0; order--) { // 0 - rearPosition 를 먼저 출력
            AppView.output(this.queue().elementAt(order).toString() + " ");
        }
        for (int order = AppController.QUEUE_CAPACITY; order > this.queue().frontPosition(); order--) { // front - MAX 까지 출력
            AppView.output(this.queue().elementAt(order).toString() + " ");
        }
    }
```

```java
    private int _maxLength; // capacity + 1
    private int _frontPosition;
    private int _rearPosition;
    private E[] _elements;

    // 생성자
    @SuppressWarnings("unchecked")
    public CircularArrayQueue(int givenCapacity) {
        this.setMaxLength(givenCapacity + 1);
        this.setFrontPosition(0);
        this.setRearPosition(0);
        this.setElements((E[]) new Object[this.maxLength()]);
    }

    public CircularArrayQueue() {
        this(CircularArrayQueue.DEFAULT_CAPACITY);
    }

    // Getters/Setters
    private int maxLength() {
        return this._maxLength;
    }

    private void setMaxLength(int newMaxlength) {
        this._maxLength = newMaxlength;
    }

    private E[] elements() {
        return this._elements;
    }

    public void setElements(E[] newElements) {
        this._elements = newElements;
    }

    public int capacity() {
        return (this.maxLength() - 1);
    }

    public int frontPosition() {
        return this._frontPosition;
    }

    private void setFrontPosition(int newFrontPosition) {

        this._frontPosition = newFrontPosition;
    }

    public int rearPosition() {
        return this._rearPosition;
    }

    private void setRearPosition(int newRearPosition) {
        this._rearPosition = newRearPosition;
        if (newRearPosition == maxLength()) {
            _rearPosition = 0;
        }
    }

    @Override
    public int size() {
        if (this.rearPosition() >= this.frontPosition()) { // frontPosition보가 rearPosition이 더 크면
            return (this.rearPosition() - this.frontPosition()); // 둘 사이의 차를 구한다
        } else { // maxLength를 넘어온 경우
            return (this.rearPosition() - this.frontPosition() + this.maxLength()); // maxLength를 더해주면 된다.
        }
    }

    @Override
    public boolean isFull() { // Queue가 가득 찼는지 확인
        int nextRearPosition = (this.rearPosition() + 1) % this.maxLength(); // 다음 삽입 위치
        return (nextRearPosition == this.frontPosition()); // 다음 삽입 위치가 맨 앞과 같으면 가득 찬 것
    }

    @Override
    public boolean isEmpty() { // Queue가 비어있는지 확인
        return (this.frontPosition() == this.rearPosition());
    }

    @Override
    public E front() { // front 원소 반환
        E frontElement = null;
        if (!this.isEmpty()) {
            frontElement = this.elements()[this.frontPosition() + 1];
        }
        return frontElement;
    }

    @Override
    public E rear() { // rear 원소 반환
        E rearElement = null;
        if (!this.isEmpty()) {
            rearElement = this.elements()[this.rearPosition()];
        }
        return rearElement;
    }

    @Override
    public boolean enQueue(E anElement) { // 큐 rear 위치에 원소 추가
        if (this.isFull()) { // 큐가 가득 찼다면
            return false; // false를 반환
        } else {
            if (this.isEmpty()) { // 만약 큐가 비어있을 경우
                this.elements()[this.rearPosition()] = anElement; // rear 현재 위치에 원소를 추가
                this.setFrontPosition((this.frontPosition() + this.maxLength() - 1) % this.maxLength()); // front를 하나 전으로 밀고 rear를 현재 위치에 삽입
                return true;
            } // 만약 큐에 원소가 있다면
            this.setRearPosition((this.rearPosition() + 1) % this.maxLength()); // rear을 다음 위치로 설정
            this.elements()[this.rearPosition()] = anElement; // rear위치에 원소를 추가
            return true;
        }
    }

    @Override
    public E deQueue() { // 큐 front 위치 원소 삭제
        E frontElement = null; // 비어있다면 null을 반환
        if (!this.isEmpty()) { // 비어있지않다면
            this.setFrontPosition((this.frontPosition() + 1) % this.maxLength()); // front를 다음 위치로 설정
            frontElement = this.elements()[this.frontPosition()]; // frontPosition의 원소를 변수에 저장
            this.elements()[this.frontPosition()] = null; // 삭제
        }
        return frontElement; // 삭제된 원소 반환
    }

    @Override
    public void clear() {
        this.setFrontPosition(0); // Front = 0으로
        this.setRearPosition(0); // Rear = 0으로
        for (int i = 0; i < this.maxLength(); i++) {
```

➔ LinkedList

```
private Queue<Character> _queue;
private int _inputChars; // 입력된 문자의 개수
private int _addedChars; // 삽입된 문자의 개수
private int _ignoredChars; // 무시된 문자의 개수

// Getters/Setters
private Queue<Character> queue() {
    return this._queue;
}

private void setQueue(Queue<Character> newQueue) {
    this._queue = newQueue;
}

private int inputChars() {
    return this._inputChars;
}

private void setInputChars(int newInputChars) {
    this._inputChars = newInputChars;
}

private int addedChars() {
    return this._addedChars;
}

private void setAddedChars(int newAddedChars) {
    this._addedChars = newAddedChars;
}

private int ignoredChars() {
    return this._ignoredChars;
}

private void setIgnoredChars(int newIgnoredChars) {
    this._ignoredChars = newIgnoredChars;
}

// 생성자
public AppController() {
    this.setQueue(new CircularlyLinkedQueue<Character>());
    this.setInputChars(0);
    this.setAddedChars(0);
    this.setIgnoredChars(0);
}

private void countInputChars() { // 입력문자 횟수계산
    this.setInputChars(this.inputChars() + 1);
}

private void countIgnoredChars() { // 무시문자 횟수계산
    this.setIgnoredChars(this.ignoredChars() + 1);
}

private void countAddedChar() { // 추가문자 횟수계산
    this.setAddedChars(this.addedChars() + 1);
}

// 큐의 모든 원소를 Front 부터 Rear까지 출력한다
private void showAllFromFront() {
    AppView.output("[Queue] <Front> ");
    Iterator<Character> queueIterator = this.queue().iterator(); // queue()의 반복자 생성
    while (queueIterator.hasNext()) { // queueIterator의 next가 있는 동안 반복
        Character element = queueIterator.next(); // 순서대로 element에 저장
        AppView.output(element.toString() + " "); // element의 문자열 출력
    }
    AppView.outputLine("<Rear> ");
}

// 큐의 모든 원소를 Rear 부터 Front까지 출력한다
private void showAllFromRear() {
    AppView.output("[Queue] <Rear> ");
    for (int order = this.queue().size() - 1; order >= 0; order--) {
        AppView.output(this.queue().elementAt(order).toString() + " ");
    }
    AppView.outputLine("<Front> ");
}

private void showFrontElement() { // Front 큐를 출력
    if (this.queue().isEmpty()) {
        AppView.outputLine("[Front.Empty] 큐가 비어서 맨 앞 원소가 존재하지 않습니다.");
    } else {
        AppView.outputLine("[Front] 큐의 맨 앞 원소는 " + this.queue().front().toString() + " 입니다.");
    }
}

private void showRearElement() { // Rear 큐를 출력
    if (this.queue().isEmpty()) {
        AppView.outputLine("[Rear.Empty] 큐가 비어서 맨 뒤 원소가 존재하지 않습니다.");
    } else {
        AppView.outputLine("[Rear] 큐의 맨 뒤 원소는 " + this.queue().rear().toString() + " 입니다.");
    }
}

private void showQueueSize() { // Queue 사이즈 출력
    AppView.outputLine("[Size] 큐의 원소의 개수는 " + this.queue().size() + "개 입니다.");
}

private void addToQueue(Character anElement) {
```

```java
    private LinkedNode<E> _rearNode;

    // 생성자
    public CircularlyLinkedQueue() {
        this.setSize(0);
        this.setRearNode(null);
    }

    // Getters/Setters
    @Override
    public int size() {
        return this._size;
    }

    private void setSize(int newSize) {
        this._size = newSize;
    }

    public LinkedNode<E> rearNode() {
        return this._rearNode;
    }

    private void setRearNode(LinkedNode<E> newNode) {
        this._rearNode = newNode;
    }

    @Override
    public boolean isFull() { // 연결체인이 가득차는 경우는 없다
        return false;
    }

    @Override
    public boolean isEmpty() { // 큐가 비어있는지 확인
        return (this.rearNode() == null);
    }

    public E front() { // 큐의 front 원소 반환
        E frontElement = null;
        if (!this.isEmpty()) {
            frontElement = this.rearNode().next().element(); // rear의 다음번째가 첫번째 큐
        }
        return frontElement;
    }

    @Override
    public E rear() { // 큐의 rear 원소 반환
        E frontElement = null;
        if (!this.isEmpty()) {
            frontElement = this.rearNode().element(); // rear번째 큐
        }
        return frontElement;
    }

    @Override
    public boolean enQueue(E anElement) { // anElement 큐를 rear에 추가
        LinkedNode<E> newRearNode = new LinkedNode<E>(anElement, null); // 새로운 Node생성
        if (this.isEmpty()) { // 만약 비어있다면
            newRearNode.setNext(newRearNode); // newRearNode의 다음을 newRearNode로 지정
        } else { // 비어있지 않다면
            newRearNode.setNext(this.rearNode().next()); // newRearNode의 다음을 현 rearNode의 다음 노드로 지정
            this.rearNode().setNext(newRearNode); // rearNode의 다음을 newRearNode로 지정
        }
        this.setRearNode(newRearNode); // newRearNode로 rearNode 설정
        this.setSize(this.size() + 1); // size ++
        return true;

    }

    @Override
    public E deQueue() { // front의 원소를 삭제
        E frontElement = null;
            frontElement = this.rearNode().next().element(); // front노드는 rear의 next 노드
```

```
    // 비공개 인스턴스 변수
    private T _element; // 값을 저장하는 리스트 원소
    private LinkedNode<T> _next; // 다음 노드를 나타냄

    // 생성자
    // 주어진 값이 없을 경우 생성되는 생성자
    public LinkedNode() {
        this.setElement(null);
        this.setNext(null);
    }

    // 주어진 값으로 생성하는 생성자
    public LinkedNode(T givenElement, LinkedNode<T> givenNext) {
        this.setElement(givenElement);
        this.setNext(givenNext);
    }

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

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

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

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

}
```

[LinkedNode]