자료구조 실습 보고서

[제 10주] : 큐 기본기능

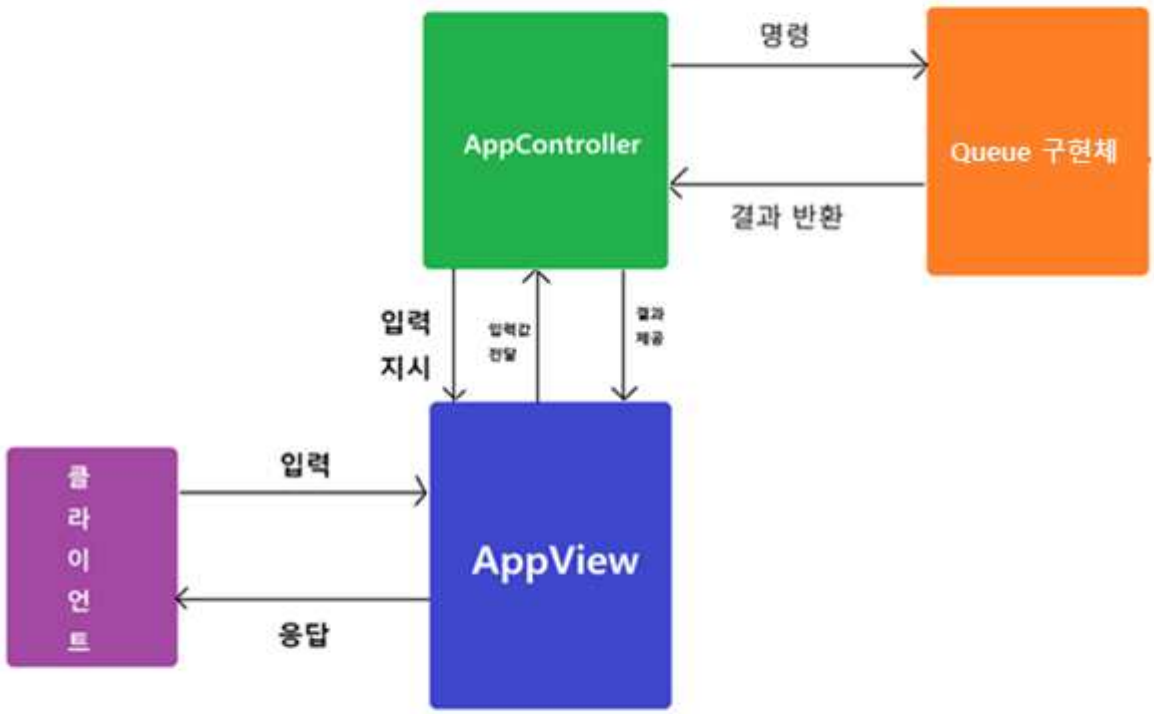
제출일: 2022-05-14(토)

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 출력 정보를 제공하며, 출력을 요청한다.

AppView

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

CircularArrayQueue, CircularLinkedQueue

큐의 기본 기능을 수행한다.

2. 함수 설명(주요 알고리즘/ 자료구조의 설명 포함)

2-1. 사용된 자료구조 & 알고리즘

사용된 자료구조는 환형 큐이다. 기본적인 Queue처럼 enQueue와 deQueue를 제공하지만, 일반적인 Array로 이루어진 Queue는 원소의 삽입과 삭제가 반복해서 이루어지다보면, 배열에 대한 낭비가 발생한다. 그러나 Array로 구현된 환영 큐는 이러한 문제를 해결해준다.

enque와 deque 수행 시 front와 rear 위치 계산법

enqueue 연산 시, 원소가 삽입되는 위치는 rear의 다음 위치이다.
즉 rearPosiotion의 길이에 1을 더한 수가, 삽입되는 원소의 인덱스가 되어야 하지만, 환형 큐이므로 rearPosiotion +1 의 값이 배열의 최대 길이를 초과할 수 있다.
따라서 이를 방지하기 위해 rearPosition + 1에다가, 배열의 최대 길이인 maxLength를 사용하여 모듈러 연산(mod, %)을 수행하여 그 결과로 반환되는 인덱스를 enqueue시의 원소의 삽입 인덱스로 한다.

dequeue 연산 시, 원소가 삭제되는 위치는 frontPosition의 다음 위치의 원소이다.
먼저 setFrontPosition 메서드를 이용하여 현재 frontPosition의 값을 한칸 이동시켜 주어야 하는데, 이때도 enqueue와 마찬가지로 frontPosition + 1에다가, 배열의 최대 길이인 maxLength를 사용하여 모듈러 연산(mod, %)을 수행하여 그 결과로 반환되는 인덱스를 원소가 삭제될 인덱스로 하며, 해당 인덱스에 해당하는 원소를 반환한다.

2-2. 주요 함수

CircularArrayQueue<E>

```
/**
     * 큐의 사이즈를 반환한다
     * @return 큐의 사이즈
     */
    @Override
    public int size() {
        if(this.rearPosition() >= this.frontPosition()){
            return this.rearPosition() - this.frontPosition();
        }

        return this.rearPosition() +this.maxLength() - this.frontPosition();
    }

    @Override
    public boolean isFull() {
        return this.frontPosition()==
                ( (this.rearPosition()+1) % this.maxLength() );
    }
    @Override
    public boolean isEmpty() {
        return this.frontPosition()==this.rearPosition();
    }

    /**
     * 큐의 front 즉, 가장 먼저 들어온 원소를 반환한다.
     * @return 가장 먼저 들어온 원소를 반환한다.
     */
    @Override
    public E front() {
        if (this.isEmpty()) {
            return null;
        }
        return this.elements()[this.frontPosition()+1];
    }
    /**
     * 큐의 rear 즉, 가장 마지막에 들어온 원소를 반환한다.
     * @return  가장 마지막에 들어온 원소를 반환한다.
     */
    @Override
    public E rear() {
        if (this.isEmpty()) {
            return null;
        }
        return this.elements()[this.rearPosition()];
    }

    /**
     * 큐에 원소를 삽입한다.
     * @param element 삽입할 원소
     * @return 성공여부
     */
    @Override
    public boolean enQueue(E anElement) {
        if (this.isFull()) {
            return false;
        }
        else {
            this.setRearPosition((this.rearPosition()+1)%this.maxLength());
            this.elements()[this.rearPosition()] = anElement;
            return true;
        }
    }
```

```
/**
 * 큐에서 원소를 삭제한다
 * @return 성공여부
 */
@Override
public E deQueue() {
    E frontElement = null;

    if (! this.isEmpty()) {
        this.setFrontPosition((this.frontPosition() + 1) % this.maxLength());
        frontElement = this.elements()[this.frontPosition()];
        this.elements()[this.frontPosition()] = null;
    }
    return frontElement;
}

@Override
public void clear(){
    this.setFrontPosition(0);
    this.setRearPosition(0);
    for(int i=0;i<this.maxLength();i++){
        this.elements()[i] = null;
    }
}

@Override
public E elementAt(int anOrder){
    if(this.isEmpty()){
        return null;
    }
    else{
        return this.elements()[(this.frontPosition()+1+anOrder)%this.maxLength()];
    }
}

public Iterator<E> iterator(){
    return new CircularArrayQueueIterator();
}

private class CircularArrayQueueIterator implements Iterator<E>{
    private int _nextOrder;

    private int nextOrder(){
        return this._nextOrder;
    }
    private void setNextOrder(int newNextOrder){
        this._nextOrder = newNextOrder;
    }
    private CircularArrayQueueIterator(){
        this.setNextOrder(0);
    }

    @Override
    public boolean hasNext() {
        return this.nextOrder() < CircularArrayQueue.this.size();
    }

    @Override
    public E next() {
        E nextElement = null;
        if (this.hasNext()) {
            nextElement = (E) CircularArrayQueue.this.elementAt(this.nextOrder());
            this.setNextOrder(this.nextOrder() + 1);
        }
        return nextElement;
    }
}
```

CircularArrayQueue<E>

```
public int size() {
        return this._size;
    }

    public boolean isEmpty() {
        return this.rearNode() == null;
    }

    public boolean isFull() {
        return false;
    }
    /**
     * 큐의 front 즉, 가장 먼저 들어온 원소를 반환한다.
     * @return 가장 먼저 들어온 원소를 반환한다.
     */
    public E front() {
        if (this.isEmpty()) {
            return null;
        }
        return this.rearNode().next().element();
    }
    /**
     * 큐의 rear 즉, 가장 마지막에 들어온 원소를 반환한다.
     * @return  가장 마지막에 들어온 원소를 반환한다.
     */
    @Override
    public E rear() {
        if (this.isEmpty()) {
            return null;
        }
        return this.rearNode().element();
    }
    /**
     * 큐에 원소를 삽입한다.
     * @param anElement 삽입할 원소
     * @return 성공여부
     */
    public boolean enQueue(E anElement) {

        LinkedNode<E> newRearNode = new LinkedNode<E>(anElement, null);
        if (this.isEmpty()) {
            newRearNode.setNext(newRearNode);
        }
        else {
            newRearNode.setNext(this.rearNode().next());
            this.rearNode().setNext(newRearNode);

        }
        this.setRearNode(newRearNode);
        this.setSize(this.size() + 1);
        return true;
    }
```

```
/**
 * 큐에서 원소를 삭제한다
 * @return 성공여부
 */
public E deQueue() {
    if (this.isEmpty()) {
        return null;
    }
    E element = this.rearNode().next().element();
    this.setRearNode(this.rearNode().next());
    this.setSize(this.size() - 1);
    if (this.size() == 0) {
        this.setRearNode(null);
    }
    return element;
}

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

public E elementAt(int anOrder) {
    if (this.isEmpty()) {
        return null;
    }
    LinkedNode<E> node = this.rearNode().next();
    for (int i = 0; i < anOrder; i++) {
        node = node.next();
    }
    return node.element();
}
```

```
@Override
public Iterator<E> iterator() {
    return new CircularlyLinkedQueueIterator<E>();
}

private class CircularlyLinkedQueueIterator<E> implements Iterator<E> {
    private LinkedNode<E> _nextNode;
    private int _count;

    private LinkedNode<E> nextNode() {
        return this._nextNode;
    }

    private void setNextNode(LinkedNode<E> newNextNode) {
        this._nextNode = newNextNode;
    }

    private int count() {
        return this._count;
    }

    private void setCount(int newCount) {
        this._count = newCount;
    }

    private CircularlyLinkedQueueIterator() {
        this.setNextNode((LinkedNode<E>) CircularlyLinkedQueue.this.rearNode());
        this.setCount(CircularlyLinkedQueue.this.size());
    }

    @Override
    public boolean hasNext() {
        return this.count() > 0;
    }

    @Override
    public E next() {
        if (this.hasNext()) {
            this.setNextNode(this.nextNode().next());
            E nextElement = this.nextNode().element();
            this.setCount(this.count() - 1);
            return nextElement;
        } else {
            return null;
        }
    }
}
```

3. 종합 설명

해당 프로그램은 문자에 대하여 Queue의 기본 연산들을 수행하며, 이러한 결과를 종합하여 보여주는 프로그램이다.
앱을 실행하기 위해서는 '_DS10_S1_201902708_신동훈' 혹은 '_DS10_S2_201902708_신동훈' 이라는 이름의 클래스에 속한 main 메서드를 실행시켜주면 된다.

프로그램 장단점/특이점

1. 프로그램의 장점

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

2. 프로그램의 단점

굳이 프로그램을 두 개로 나눴을 이유가 있나 싶다. 두 프로그램 모두 동일한 Queue 인터페이스를 가지고 있으며, 따라서 다형성을 활용하여 이를 AppController에서 구현체를 선택하는 코드 한 줄만 바꾼다면 하나의 프로그램으로만으로도 충분해 보인다. 혹은 사용자에게 무엇을 사용할 것인지 정하게 하여 사용자가 원하는 구현체를 사용하게끔 구현할 수도 있었을 것이다.

CircularyLinkedQueue의 경우에는 LinkedNode를 사용하여 구현하였기에 사실상 용량에 제한이 없으나 필드 _size의 자료형이 int이므로 용량이 int의 최댓값으로 제한되었다. 실제로 int의 최댓값을 넘어 장애가 발생한 사건도 있었으며, 이를 방지하기 위해서는 BigInteger 혹은 size의 자료형을 Long으로 바꿔, Long.MAX_VALUE를 사용하게끔 할 수도 있을 것이다.

실행 결과 분석

1. 입력과 출력

(이곳에서는 ppt와 동일한 출력을 위해 AppController의 QUEUE_CAPACITY를 5로 설정했습니다.)

```
<<< 큐 기능 확인 프로그램을 시작합니다 >>>

? 문자를 입력하시오: -
[DeQ.Empty] 큐에 삭제할 원소가 없습니다.
? 문자를 입력하시오: A
[Push] 삽입된 원소는 'A' 입니다.
? 문자를 입력하시오: x
[Push] 삽입된 원소는 'x' 입니다.
? 문자를 입력하시오: h
[Push] 삽입된 원소는 'h' 입니다.
? 문자를 입력하시오: d
[Push] 삽입된 원소는 'd' 입니다.
? 문자를 입력하시오: #
[Size] 큐에는 현재 4개의 원소가 있습니다.
? 문자를 입력하시오: /
[Queue] <Front> A x h d <Rear>
? 문자를 입력하시오: \
[Queue] <Rear> d h x A <Front>
? 문자를 입력하시오: >
[Rear] 큐의 맨 뒤 원소는 'd' 입니다.
? 문자를 입력하시오: <
[Front] 큐의 맨 앞 원소는 'A' 입니다.
? 문자를 입력하시오: -
[DeQ] 삭제된 원소는 'A' 입니다.
? 문자를 입력하시오: 2
[DeQs] 삭제된 원소는 'x' 입니다
[DeQs] 삭제된 원소는 'h' 입니다
```

```
? 문자를 입력하시오: /
[Queue] <Front> d <Rear>
? 문자를 입력하시오: 2
[DeQs] 삭제된 원소는 'd' 입니다
[DeQs.Empty] 큐에 더 이상 삭제할 원소가 없습니다.
? 문자를 입력하시오: >
[Rear.Empty] 큐가 비어서 맨 뒤 원소가 존재하지 않습
? 문자를 입력하시오: <
[Front.Empty] 큐가 비어서 맨 앞 원소가 존재하지 않
? 문자를 입력하시오: B
[Push] 삽입된 원소는 'B' 입니다.
? 문자를 입력하시오: w
[Push] 삽입된 원소는 'w' 입니다.
? 문자를 입력하시오: E
[Push] 삽입된 원소는 'E' 입니다.
? 문자를 입력하시오: T
[Push] 삽입된 원소는 'T' 입니다.
? 문자를 입력하시오: m
[Push] 삽입된 원소는 'm' 입니다.
? 문자를 입력하시오: 6
[DeQs] 삭제된 원소는 'B' 입니다
[DeQs] 삭제된 원소는 'w' 입니다
[DeQs] 삭제된 원소는 'E' 입니다
[DeQs] 삭제된 원소는 'T' 입니다
[DeQs] 삭제된 원소는 'm' 입니다
[DeQs.Empty] 큐에 더 이상 삭제할 원소가 없습니다.
```

```
? 문자를 입력하시오: /
[Queue] <Front> <Rear>
? 문자를 입력하시오: !

<큐을 비우고 사용을 종료합니다>
[Queue] <Front> <Rear>
[DeQs] 삭제할 원소의 개수가 0 개 입니다.

<큐 사용 통계>
- 입력된 문자는 23 개 입니다.
- 정상 처리된 문자는 23 개 입니다.
- 무시된 문자는 0 개 입니다.
- 삽입된 문자는 9 개 입니다.

<<< 큐 기능 확인 프로그램을 종료합니다 >>>
```

```
<<< 큐 기능 확인 프로그램을 시작합니다 >>>

? 문자를 입력하시오: A
[Push] 삽입된 원소는 'A' 입니다.
? 문자를 입력하시오: x
[Push] 삽입된 원소는 'x' 입니다.
? 문자를 입력하시오: D
[Push] 삽입된 원소는 'D' 입니다.
? 문자를 입력하시오: H
[Push] 삽입된 원소는 'H' 입니다.
? 문자를 입력하시오: w
[Push] 삽입된 원소는 'w' 입니다.
? 문자를 입력하시오: k
[EnQ.Empty] 큐이 꽉 차서, 더 이상 넣을 수 없습니다.
(오류) 큐에 넣는 동안에 오류가 발생하였습니다.
? 문자를 입력하시오: $
[Ignore] 의미 없는 문자가 입력되었습니다.
? 문자를 입력하시오: /
[Queue] <Front> A x D H w <Rear>
? 문자를 입력하시오:



5
```

```
[DeQs] 삭제된 원소는 'A' 입니다
[DeQs] 삭제된 원소는 'x' 입니다
[DeQs] 삭제된 원소는 'D' 입니다
[DeQs] 삭제된 원소는 'H' 입니다
[DeQs] 삭제된 원소는 'W' 입니다
? 문자를 입력하시오: >
[Rear.Empty] 큐가 비어서 맨 뒤 원소가 존재하지 않
? 문자를 입력하시오: <
[Front.Empty] 큐가 비어서 맨 앞 원소가 존재하지 않
? 문자를 입력하시오: -
[DeQ.Empty] 큐에 삭제할 원소가 없습니다.
? 문자를 입력하시오: !

<큐을 비우고 사용을 종료합니다>
[Queue] <Front> <Rear>
[DeQs] 삭제할 원소의 개수가 0 개 입니다.

<큐 사용 통계>
- 입력된 문자는 12 개 입니다.
- 정상 처리된 문자는 11 개 입니다.
- 무시된 문자는 1 개 입니다.
- 삽입된 문자는 6 개 입니다.

<<< 큐 기능 확인 프로그램을 종료합니다 >>>
```

2. 결과 분석

모든 기능이 문제 없이 잘 작동한다.

생각해 볼 점에 대한 의견

1. 큐를 포함하여 일반적으로 리스트를 배열로 구현할 경우, 사용자의 관점인 원소의 순서 (order) 와 실제 구현된 배열에서의 위치 (position) 개념은 동일한 개념이 아니다. 이 둘, 즉 사용자의 order 와 구현적 관점인 position 을 구분하는 이유는?

기본적으로 배열의 시작 인덱스는 0이며, 대부분의 사용자들은 1을 시작 위치로 생각할 것이다. 즉 일반적인 배열을 사용하는 경우만 하더라도 사용자의 관점인 원소의 순서와, 실제 구현된 배열에서의 위치의 개념이 차이가 난다.
배열의 위치와 사용자의 위치를 맞추어 줄 수도 있겠지만, 그렇게 된다면 배열에서 낭비하는 부분이 발생하게 되고, 이러한 과정 속에서 성능상으로 손해를 볼 수도 있다.
예를 들어 이번 과제의 CircularArrayQueue에서, 만약 사용자의 관점에서의 원소의 순서와, 실제 배열에서의 위치를 맞추기 위해서는, dequeue 연산이 발생할 때마다, front 인덱스 이후에 존재하는 모든 원소들을 한칸씩 왼쪽으로 이동시켜주어야 하는데, 이는 n만큼의 시간복잡도가 소요된다.
즉 이러한 문제를 해결하기 위해 사용자의 관점과 구현적 관점을 구분한다.

2. 환형 배열 (circular array) 로 구현한 큐와 연결체인 (linked chain) 으로 구현한 큐의 차이점과 장단점은?

용량 관점에서는 환형 배열 큐는 배열로 구현되었기 때문에, 원소의 최대 개수에 한계가 있으나, 연결체인 큐는 저장할 수 있는 원소의 최대 크기에 대한 제약을 없앨 수 있기에 연결체인 큐가 장점을 가진다.
그러나 임의의 위치의 원소에 대한 접근 연산, 즉 elementAt 연산에서는 환영 배열로 구현한 큐는 O(1)만큼의 시간복잡도를 가지는 반면 연결체인으로 구현한 큐는, n번째 원소에 접근하기 위해 LinkedNode의 next연산을 n번만큼 반복해 주어야 하기 때문에 O(n)의 시간복잡도가 소요되므로 환형 배열 큐가 장점을 가진다.

3. Queue를 interface로 선언하면 좋은 점은?

Queue는 배열을 사용하여 구현될 수도 있고, LinkedNode를 사용하여 구현될 수도 있다.
또한 일반적인 Queue로 구현할 수도 있고, 환형 큐로 구현할 수도 있다.
즉 Queue의 역할을 수행하기 위한 여러 방법들이 존재할 수 있고, 따라서 여러 구현체가 정의될 수 있는데, 이를 Queue라는 인터페이스를 구현하게 만든다면, 코드의 최소한의 변경으로 사용자가 원하는 Queue의 구현체를 선택하여 사용할 수 있다.
즉 요구사항이 변경되었을 때, 코드의 수정을 최소한으로 발생시키게 하여, 이후 유지보수 등에 도움이 될 뿐만 아니라, 이를 API로 만든다면 이를 사용하는 사용자들은 여러 가지 Queue의 구현체들 중에서, 사용자의 상황에 사용하기에 알맞은 구현체를 직접 선택하여 사용할 수 있게 된다.