자료구조와 C++ 프로그래밍

이름 : 이인
학번 : 2021125050

강의시간 : 월 9:00, 수 10:30
레포트 번호 : 과제2

문제 인식 및 분석

카드를 순서대로 푸쉬 및 팝 하기위해 필요한 자료구조가 무엇일 지 먼저 생각해 보았다. 카드 간 대소를 구분하고 삽입하는 방향이 front와 rear 둘 다 가능하며, 더 빠른 삽입이 가능하도록 알고리즘을 구성하기 위해서는 컨테이너의 front와 rear가 뚫려있고, 공간의 크기가 한정적인 원형 큐를 이용하면 되겠다는 생각이 들었다. 각 카드가 가지는 모양은 static Const 변수로 따로 배열로 작성해 놓은 다음 출력할 때 이를 이용하여 출력하고, 대소 비교를 할 때 사용하면 될 것 같다고 생각했다. 특히 대소 비교를 위해 spade, dia, heart, clover의 배치 순서를 역순으로 배치하는 것이 중요했다.

본인이 생각한 형태는 Card라는 객체가 존재하고, Card 객체를 관리해주는 Dequeue라는 객체가 필요하다고 생각하여 두 객체를 이원화 하여 작성하기로 했다. 그 중 Card객체는 각자 고유한 크기가 있고, 이를 편리하게 나타내기 위해서 shape이라는 멤버 변수가 각 카드의 모양을, number라는 변수는 각 카드가 가지는 숫자를 나타내기로 했다. 카드 간 크기를 비교하는 메서드를 Card객체에 작성해 언제 어디서든 카드 간 크기 비교가 가능할 수 있도록 하였다.

Dequeue 객체가 카드 관리를 위해 필요한 기능을 생각해 보았을 때, Dequeue의 front 와 rear의 위치정보를 알 수 있어야 했다. 또 카드를 담을 공간이 필요하기 때문에 Card 타입의 배열이 필요하였고, 고정된 사이즈의 원형 큐 이기 때문에 5개의 카드만이 들어온다고 생각했을 때 큐의 size는 6이라고 판단했다. 또 큐 안에 들어온 카드의 갯수를 이용하여 front 혹은 rear, 둘 중 넣을 방향을 판단할 수 있기 때문에 카드 갯수에 대한 정보가 필요하다고 생각했고 이를 모두 Dequeue의 멤버 변수로 지정해야 한다고 판단했다.

카드를 빼고 넣을 수 있는 메소드인 push와 pop에 대해 front rear 두 방향 모두 가능하도록 큐 메소드를 구현하고, 큐가 비었을 때와 가득 찼을 때를 구분할 수 있는 메소드를 작성하여 유연하게 대체할 수 있도록 기반을 마련했다. 큐의 front rear 및 현재 저장하고있는 카드들의 상태를 표시하는 함수가 있어야 카드를 정리할 때마다 정리한 상황을 가시적으로 표시할 수 있기에 이를 선언했다.

새로 들어오는 카드가 들어갈 위치를 찾기 위해선 결국 큐 안에 있는 카드들과 새로 들어온 카드의 대소를 비교해야 하기 때문에 큐 안에 새로운 카드가 들어갈 위치를 찾는 함수를 작성하는 게 맞다고 판단하고 정의했다. 들어갈 위치를 찾는 함수, 위치를 찾았을 때 Front와 rear중 어느 방향으로 삽입해야 더욱 빠르게 삽입할 수 있는지 판단하는 함수, 판단에 따라 push 와 pop을 수행하는 함수 총 세 가지 함수로 구성하기로 생각했다.

또한 순차적으로 들어오는 5개의 카드들은 모두 다른 크기를 가지는 카드여야 하기 때문에 매번 새로운 카드를 추가할 때 마다 다른 크기를 가지는 카드를 생성할 함수의 필요성을 느껴 정의하였다.

2. 문제 해결, 알고리즘 및 코드 분석

```
#include <iostream> // 입출력을 위한 라이브러리 추가
#include <cstdlib> // 난수 생성을 위한 라이브러리 삽입
#include <ctime> // 반복적인 난수 생성시 동일 값 생성을 막기 위핸 시간 라이브러리 삽입
```

```
static const char* SHAPES[] = { "C", "H","D" , "S" }; // 카드의 모양에 따른 대소 구분 및 차별화를, 특성있는 변수이기 때문에 전역 객체로 선언
using namespace std; // std 라이브러리의 공간을 표준으로 사용
```

```
class Card{ // Card 클래스 선언 및 정의
private:
        int shape = -1; // 카드 모양
        int number = -1; // 카드 넘버 둘다 -1인 이유는 데큐에서 빈 객체를 표현할 때 -1인 경우 비었다는 것을 상정하기 위함.
        friend class Dequeue; //Dequeue 클래스에서 Card의 멤버 변수에 직접 접근하고 메소드를 활용할 수 있도록 하기 위함.
public:
        Card() { // 기본 생성자
        };
```

```
        Card(int shape, int number){ // 랜덤값을 이용해 카드 객체를 생성할 때 활용하기 위한 생성자
                this->shape = shape; // Private 멤버 변수에 직접적으로 접근하기 때문에 포인터 접근 연산자인 -> 사용
                this->number = number; // 마찬가지
        }

        static const bool compareCard(const Card& card1, const Card& card2){   // card1 > card2  "true" card2 > card1 "false" 어디서든 메서드를 사용할 수 있게 static const 메서드로 선언 및 정의
                if (card1.shape != card2.shape) { // 모양이 최우선순위로 비교됨, 모양이 다를 때와 같을 때를 나눔
                        if (card1.shape > card2.shape) return true; //  S-D-H-C 순서대로 크기 비교
                        else return false;
                }
                else {
                        if (card1.number > card2.number) return true; // 넘버도 큰 순서대로 비교
                        else return false;
                }
        }

        friend ostream& operator<<(ostream& os, Card& card){ // 출력 연산자 오버로딩, shape과 number 출력
```

```
        os << SHAPES[card.shape] << card.number;
        return os;
    }
};
```

함수 구현에 필요한 기본 라이브러리인 입출력 라이브러리, 시간 라이브러리를 삽입한다. spade diamond heart clover 순서대로 큰 카드라고 가정한다면, 배열의 인덱스 기준에서 보았을 때 마지막에 오는 원소가 가장 높은 인덱스를 가지므로 후순위로 각 모양들을 배열하였으며, 자주 사용하고 특별한 의미를 가지는 문자이기 때문에 전역 const 변수로 선언하여 언제 어디서든 사용할 수 있도록 하였다.
Dequeue 객체에 들어갈 Card 객체를 선언 및 정의한다. 멤버변수 shpae와 number는 기본값을 -1로 둠으로써 아무런 인자도 받지 못했을 때 비어있다는 것을 상정한다. friend class로 dequeue를 지정하여 데큐에서 카드의 멤버 변수에 직접 접근할 상황이 생겼을 때 유동적으로 접근할 수 있도록 하였다. 기본 생성자 오버로딩을 통해 인자가 주어진 상황에 대해서만 진짜 Card 객체를 생성할 수 있도록 한다.
this 포인터 지정을 통해 멤버변수에 값을 할당할 수 있도록 하였다. compareCard 함수는 두 카드가 주어졌을 때 첫 번째 카드가 크다면 true를 두 번째 카드가 크다면 false를 반환하도록 구현하였다. 데큐 클래스에서 구현해도 상관없지만, 카드의 멤버변수에 직접 접근할 때 편의성을 가져가기 위해서 카드 클래스 안에 정의하였다. 전역 const 함수로 선언하여 멤버 함수로서 역할을 다하는 것이 아닌 일반 함수처럼 기능하도록 하여 유동적으로 해당 함수를 사용할 수 있도록 했다. 카드의 shape를 최우선으로 비교하고 만약 shape이 같다면 number를 비교하도록 구현하였다. 각 변수가 같은 값을 가지는 경우는 상정하지 않았다. 왜냐하면initialize 함수를 사용하여 같은 크기를 가지는 카드가 들어오는 경우를 사전에 차단했기 때문이다.
출력 연산자 오버로딩을 이용해 SHAPE변수의 인덱스로 card.shape의 값을 이용하고 card.number의 값을 이용하여 모양에 따른 문자와 숫자를 출력하도록 하였다.

```
class Dequeue // Dequeue 클래스  선언  및  정의
{
private:
        Card* queue = new Card[6]; //  카드를  담을  배열  생성
        int front = 0;      // 앞
```

```
        int rear = 0;       // 뒤
        int cardCount = 0;      // 들어온 원소 갯수
        int size = 6;       // 크기

public:
        Dequeue(){ // 기본 생성자
                for (int i = 0; i < size; i++){
                        queue[i] = Card(); // 카드를 담을 배열을 초기화
                }
        }

        bool is_full() { // 큐가 가득 찼으면 true 아니면 false 를 반환
                if ((rear + 1) % size == front){ // rear는 계속 추가를 하기 때문에 배열의 size를 초과할 수 있음, size로 나눈 나머지를 rear 와 front로 설정하면 항상 절대적인 index를 나타내게 됨
                        cout << " full." << endl;
                        return true;
                }
                return false;
        }

        bool is_empty() { // 큐에 원소가 하나도 없으면 front와rear가 같기 때문에 true , 아니면 False
                if (front == rear)
```

```
            {
                cout << "Empty." << endl;
                return true;
            }
            return false;
        }
```

카드 객체를 저장하고 관리하며 비교하는 객체인 Dequeue를 선언 및 정의하였다. Card를 저장할 배열을 new키워드와 포인터를 통해 선언하였고 첫 카드의 앞 자리를 나타내는 front와 마지막 원소를 나타내는 rear 변수를 지정하였다. 카드를 저장할 공간이 가득 찬 경우를 front와 rear로 나타내기 위해 하나의 빈칸이 필요하므로 size는 6으로 지정하였다. 기본 생성자로 큐의 각 자리에 빈 자리를 나타내는 카드 객체를 넣어 초기화 하였다. 큐가 가득찼을 때 true를 반환하는 is_full 함수와 완전히 비었을 때 true를 반환하는 is_empty함수를 만들고 full의 경우에는 rear가 빈칸을 가리키고 있어야 가득 찬 걸로 인식하는데 원래 rear는 가장 마지막 원소를 나타내므로 가장 마지막 원소의 다음 칸이 여유 분의 칸이면(빈칸) 가득 찼다는 의미가 성립한다. 또한 모두 비었을 경우 rear와 front는 기본 값에 의해 둘다 0을 가리키게 되므로 둘의 값이 같을 때 빈 큐라고 설명할 수 있다.

```
void print_dequeue_status(){ // 큐의 전반적인 상태를 출력
        cout << "front : " << front << endl;
        cout << "rear : " << rear << endl;
        cout << "size : " << size << endl;
        cout << "status" << endl;
        int next_index = front;
        cout << "[ ";
        while (next_index != rear){ // front+1과 rear가 같은 위치를 가르키면 순차적 탐색이 끝난 것
```

```
            next_index = (next_index + 1) % size;
            cout << queue[next_index]  << ",";
        }
        cout << "]" << endl;
}
```

```
Card get_front() {// front는  가장  첫번째  원소  앞을  가르키기  때문에 front+1 출력
        return queue[(front + 1) % size];
}
```

```
Card get_rear(){// rear 카드  반환
        return queue[rear];
}
```

```
int get_count() const { // 삽입된  카드의  갯수  반환
        return cardCount;
}
```

큐의 전반적인 상황을 가시적으로 표현하기 위해 print dequeue 함수를 정의하여 front rear size 배열의 원소들을 표시할 수 있도록 했다. 이 때 front의 값에 직접적인 영향을 주지 않고 순차적인 탐색만 해야하기 때문에 새로운 변수에 front의 값을 할당하여 새로운 변수의 값이 rear에 도달할 때 까지의 모든 카드를 출력한다.
가장 앞에 있는(왼쪽에 있는) 카드에 대한 정보를 반환하는 get front함수와 가장 뒤에 있는(오른쪽에 있는) 카드에 대한 정보를 반환하는 get rear함수를 정의하였다. 큐에 몇 장의 카드가 들어있는 지 알기 위해 카드의 갯수를 반환하는 함수 get count를 정의하였다.

```
void push_front(Card& data) { // 가장 완쪽 원소의 앞을 가르키기 때문에 Front에 그대로 삽입
        if (is_full()) return;

        queue[front] = data;
        front -= 1;
        if (front < 0){ // 프론트는 음수가 될 수 없으므로 절대적 Front를 반환하기 위해 size를 더함
                front += size;
        }
        cardCount += 1;
}

void push_rear(Card& data){
        if (is_full()) return;

        rear = (rear + 1) % size; // rear 가 Size를 초과할 수 없으므로 절대적 size를 반환하기 위해 size로 나눔
        queue[rear] = data;

        cardCount += 1;
}

Card pop_front(){ //원하는 자리에 카드를 삽입하기 위해서 pop 한 카드를 리턴해야 함
```

```
        Card result = queue[(front + 1) % size];
        if (is_empty()){ // 비었을 때는 빈 카드를 리턴하여 빈 자리임을 알림
                return Card();
        }
        front = (front + 1) % size;
        cardCount -= 1;
        return result;
}
```

```
Card pop_rear(){ // 마찬가지로 Pop 한 카드의 정보를 가지고 있기 위해 pop 한 카드를 리턴함
        Card result = queue[rear];
        if (is_empty()){
                return Card();
        }
        rear -= 1;
        if (rear < 0){
                rear += size;
        }
        cardCount -= 1;
        return result;
}
```

푸쉬,팝/프론트, 리어의 모든 경우의 수는 4가지 이고 각각에 대한 함수를 정의하였다. 그중 pop의 경우에는 내가 pop한 카드에 대한 정보를 계속 가지고 있어야 이후 푸쉬를 하던 버리던 선택할 수 있기 때문에 팝한 카드의 정보를 반환하도록 함수를 구성하였고, push의 경우는 단순히 해당 자리에 카드를 삽입하도록 구성했다. front에 푸쉬를 할 때 프론트 값이 음수가 될 수 없으므로 원형큐에서 절대적인 front 위치를 가리키도록 하기 위해 음수가 되었을 때 마다 +size를 하도록 했다. rear에 푸쉬를 할 때 rear는 원형큐의 기본 사이즈를 초과할 수 없으므로 %size를 통해 절대적인 rear위치를 나타낼 수 있도록 하였다. pop front의 경우도 front가 size초과하는 것을 방지하기위해 %size로 절대적 front를 나타낼 수 있도록 하였고, pop rear도 rear가 음수가 되는 것을 방지하기 위해, 음수가 될 경우 절대적인 rear위치를 나타낼 수 있도록 +size를 하였다.

```
void sortCard(Card& targetCard){ // 카드 간 대소 비교를 통해 들어갈 자리 찾아 삽입 실행하는 함수
        int tmpF = (front+1)%size; // 실제 front에 영향을 주지 않으면서 큐 전체를 순차적으로 탐색하기 위해 임시 변수를 만듬
        int tmpR = rear; // tmpF 와 같은 역할의 rear
        int countF = 0; // front에서 탐색했을 때 몇번 pop push 해야하는지 알려주는 변수
        int countR = 0; // rear에서 탐색했을 때 몇번 Pop push 해야하는지 알려주는 변수
```

```
        while(tmpF != rear+1){ // rear를 넘어가게 되면 큐의 모든 카드를 탐색한 것이 된다.
                if (Card::compareCard(targetCard, queue[tmpF])){
                        tmpF = (tmpF+1) % size; // 삽입할 카드가 더 크면 true이기 때문에 다음 카드를 탐색해야 함
                        countF++;
                }
                else if(!Card::compareCard(targetCard, queue[tmpF])) break; // 삽입할 카드가 더 작을 경우에는 비교 대상군의 카드 위치를 반환한다.
```

```
        }

        while (tmpR!= front%size){ // rear부터 역순으로 탐색
                if (Card::compareCard(targetCard, queue[tmpR])) {
                        tmpR = (tmpR-1) ;// 삽입할 카드가 더 크면 true이기 때문에 다음 카드를 탐색해야 함, 역순이기 때문에 -1
                        if(tmpR < 0) tmpR = tmpR + size; // 음수가 되면 안되기 때문에 절대적 위치를 가리키기 위함
                        countR++;
                }
                else if(!Card::compareCard(targetCard, queue[tmpR])) break; // 삽입할 카드가 더 작을 경우에는 비교 대상군의 카드 위치를 반환한다.
        }

        if(tmpF == rear+1) { // 넣을 카드보다 더 큰 카드가 없을 때
                push_rear(targetCard);
        }
        else if( tmpR == front%size) {// rear에서부터 넣을 카드보다 큰 카드가 없을 때
                push_front(targetCard);
        }
        else{ // 이 외의 모든 경우에 대해
                countR += 1; // rear의 경우 한 번 더 팝 푸쉬를 진행해야 자기 자리로 옴
                if(countR<countF){ // rear에서 탐색한 숫자가 더 짧을 때
```

```
                    for(int i = 0; i<countR; i++){
                        Card tmp =pop_rear(); // 변수에 pop한걸 저장해야 해당 카드 정보를 가져와 push 가능
                        push_front(tmp);
                        print_dequeue_status();
                    }
                    push_rear(targetCard); // 삽입할 카드 push
                    print_dequeue_status();
                    for(int i = 0; i<countR; i++){
                        Card tmp =pop_front();
                        push_rear(tmp);
                        print_dequeue_status();
                    }
                }
                else if( countF<=countR){ // front에서 탐색한 숫자가 더 클 때
                    for(int i = 0; i<countF; i++){
                        Card tmp =pop_front();// 변수에 pop한걸 저장해야 해당 카드 정보를 가져와 push 가능
                        push_rear(tmp);
                        print_dequeue_status();
                    }
                    push_front(targetCard);// 삽입할 카드 push
                    print_dequeue_status();
```

```
                    for(int i = 0; i<countF; i++){
                        Card tmp =pop_rear();
                        push_front(tmp);
                        print_dequeue_status();
                    }
                }
            }
}
```

sortCard함수는 큐 안에 카드가 하나 이상 있을 때 삽입할 카드가 들어가야할 위치를 찾아 front rear중 짧은 방향을 찾아 삽입을 진행하는 함수이다. tmpF와 tmpR는 실제 front rear에 영향을 주지 않고, 순차적으로 큐를 탐색하기 위해 설정한 변수이며, 만약 삽입하려는 함수가 제일 클 때 삽입할 위치의 index를 의미하기도 한다. countF 와 countR은 각각 front와 rear에서 삽입할 때 몇번 pop 과 push를 해야하는 지를 측정하는 변수이다. tmpF != rear+1인 이유는 rear는 마지막 원소를 가르키기 때문에 마지막 원소까지 검사를 하려면 tmpF가 rear도 검사를 해야 하기 때문에 +1을 하였다. tmpR != front%size 프론트는 리어와 다르게 맨 앞 원소의 앞쪽을 가르키기 때문에 추가로 1을 더할 필요가 없다. 각각의 while문은 countF,R을 구성하고 탐색 도중 삽입할 카드보다 더 큰 카드를 발견하면 탐색을 중단하게 된다. 이 때 삽입할 카드보다 더 큰 카드가 없는 경우가 존재할 수 있는데, front에서 탐색한 경우와 rear에서 탐색한 경우를 각각 구분하여 push rear, push front를 진행한다. 이 외의 모든 경우(카드들 사이에 새로운 카드를 삽입하는 경우)에는 countR+1을 해준다. 왜냐하면 rear에서 삽입이 이루어 지면 삽입이 될 카드는 자신보다 큰 카드의 왼쪽에 삽입해야 하기 때문이다. 이후 countF이 더 큰 경우 뒷쪽에서 pop및 push를 실시한다. countR이 더 큰 경우 앞쪽에서 pop 및 push를 실시한다. 각 pop 및 push에 따라 print_dequeue 함수를 실행하여 삽입이 진행되는 과정을 가시적으로 나타내도록 한다.

```
void sortDequeue(Card& targetCard) { // 저장된 카드 수에 따라 각기 다른 카드 삽입 함수 실행
        if (get_count() == 0) { // 카드가 하나도 없을 때
```

```
            push_rear(targetCard);
            print_dequeue_status();
            return;
        } else if (get_count() == 1) { // 카드가 하나 일 때
            if (Card::compareCard(targetCard, queue[rear]))
push_rear(targetCard); // 뒤 또는 앞에 삽입
            else push_front(targetCard);
            print_dequeue_status();
            return;
        } else if (get_count() > 1) { // 2~5개의 카드면
            cout << "두개 이상" << endl;
            this->sortCard(targetCard); // 삽입하려는 카드보다 큰 카드의 위치를 찾
음
        }

};
```

sortDequeue 함수는 큐에 삽입된 카드의 개수에 따라 각기 다른 카드 삽입 함수를 실행하는 함수이다. 0개 일때는 push rear, 1개 일때는 기존 카드 하나와만 비교하여 push front 혹은 push rear를 실행한다. 2개 이상일 때는 sortCard함수를 실행한다.

```
void initializeDequeue() { // 프로그램의 전반적인 모든 기능을 종합하여 실행하기 위한
함수
        Card* card_list = new Card[52]; // 카드를 섞기 위한 52칸의 card 배열 생성
        srand(time(NULL)); // rand함수 사용 시 중복을 피하기 위함
        srand((unsigned int)time(0)); // 현재 시간을 활용하여 중복 값 도출을 피함
```

```
        for (int i = 0; i < 52; i++){ // rand로 나온 값으로 카드를 만듬
                int cardShape = i % 4;
                int cardNum = (i % 13);
                card_list[i] = Card(cardShape, cardNum);
        }
        for (int i = 0; i < 100; ++i) // 만든 카드를 다시 섞어 중복을 또 방지
        {
                int id, id2;
                Card iTemp;
                id = rand() % 52; // 카드의 갯수가 총 52개, 배열의 크기도 52개이기 때문에
랜덤으로 배열의 인덱스 선택 가능
                id2 = rand() % 52;

                iTemp = card_list[id]; // 뽑힌 인덱스 끼리 카드 교환
                card_list[id] = card_list[id2];
                card_list[id2] = iTemp;
        }

        for(int i = 0; i<size-1; i++) sortDequeue(card_list[i]); // 카드를 삽입하는 함수 실
행
}
```

initializeDequeue함수는 중복되는 카드가 삽입되는 것을 피하기 위해 52장의 카드가 들어갈 배열을 만들어 rand함수를 통해 각각 삽입한다. 이때 현재 시각을 기준으로 rand가 실행되게 하도록 하여 최대한 중복되는 값을 피하도록 한다. 삽입된 52장의 카드들을 또 다시 100번 섞고 섞는 카드들을 rand함수로 index를 랜덤으로 나

오게 하여 무작위로 섞는다. 이후 index 0부터 5까지 차례대로 sortDequeue함수의 인자로 전달한다.

3. 출력 결과 분석

front : 0
rear : 1 // rear+1
size : 6
status
〔 C7,〕// push rear
front : 0
rear : 2 // rear+1
size : 6
status
〔 C7,S6,〕// push rear(s6)
두개 이상
front : 5 //(front-1)+size
rear : 2
size : 6
status
〔 C6,C7,S6,〕// push front (c6)
두개 이상
front : 4 //push front
rear : 1 //pop rear
size : 6
status
〔 S6,C6,C7,〕// H6를 삽입하기 위해 기존 카드 위치 이동 pop rear, push front
front : 4
rear : 2 //push rear (h6)
size : 6
status
〔 S6,C6,C7,H6,〕//h6 삽입
front : 5 //pop front(s6)
rear : 3 //push rear(s6)
size : 6
status

〔 C6,C7,H6,S6,〕
front : 4 // pop rear(s6)
rear : 2 // push front(s6)
size : 6
status
〔 S6,C6,C7,H6,〕// D10 삽입을 위한 원소 이동
front : 4
rear : 3 //push rear(d10)
size : 6
status
〔 S6,C6,C7,H6,D10,〕
front : 5 //pop front(s6)
rear : 4 //push rear(s6)
size : 6
status
〔 C6,C7,H6,D10,S6,〕

---

front : 0
rear : 1
size : 6
status
〔 H1,〕// push rear
front : 0
rear : 2
size : 6
status
〔 H1,S12,〕// push rear
front : 1 pop front(s12)
rear : 3 push rear(s12)
size : 6
status
〔 S12,H1,〕// H8삽입을 위한 카드 이동
front : 0
rear : 3 push front(h8)
size : 6
status

〔H8,S12,H1,〕
front : 5 pop rear(h1)
rear : 2 push front(h1)
size : 6
status
〔H1,H8,S12,〕
front : 4 pop rear(s12)
rear : 1 push front(s12)
size : 6
status
〔S12,H1,H8,〕// h11 삽입을 위한 카드 이동
front : 4
rear : 2 push rear(h11)
size : 6
status
〔S12,H1,H8,H11,〕
front : 5 pop front(s12)
rear : 3 push rear(s12)
size : 6
status
〔H1,H8,H11,S12,〕
front : 0 pop front(h1)
rear : 4 push rear(h1)
size : 6
status
〔H8,H11,S12,H1,〕// h7삽입을 위한 카드 이동
front : 5 // push front(h7)
rear : 4
size : 6
status
〔H7,H8,H11,S12,H1,〕
front : 4 pop rear(h1)
rear : 3 push front(h1)
size : 6
status
〔H1,H7,H8,H11,S12,〕

4. 배운 점, 깨달은 점, 어려웠던 점

클래스에서 다른 클래스를 다룬다는 개념이 무엇인지 정확하게 체화하게 되었습니다. 특히 원형 큐를 구현하는 과정에서 front rear 등의 위치와 자료구조의 형태를 이미지로 상상하며 코드를 짠다는 느낌이 새롭게 다가왔다. 이미지를 통해 코드를 작성할 때 코드 작성이 매끄러워 지고 비교적 예외에 대한 처리를 적게 해줄 수 있는 좋은 알고리즘을 작성할 수 있었다. priate 멤버 변수에 대한 직접적인 접근을 최대한 피하고 전체적인 클래스와 메소드 작성 이전에, 어떤 객체가 어떤 기능을 하고, 다른 객체는 해당 객체에 어떻게 작용하며 어떤 기능을 수행할 건지 구체적으로 작성하고 명세화 하는 과정을 통해 전반적인 로직 작성 시간을 크게 줄일 수 있었다. 뿐만 아니라 중복되는 함수의 작성이나 필요없는 기능을 하는 함수의 작성을 최소화 할 수 있었고 하나의 함수에는 하나의 기능만 하도록 구성하고 함수 이름을 작성하니 메소드를 활용하기도 쉬울 뿐더러 재사용의 용이성을 크게 체감할 수 있었다. 함수를 최대한 분리하고 각 함수는 하나의 반환값, 하나의 기능을 가지는 것이 중요하다는 것을 깨달았다. 전역 변수를 어떨 때 선언하고 어떤 방식으로 이용해야 하는지 구체적으로 알 수 있었다. 카드의 각 모양에 대해 크기를 비교하거나, 문자열로 출력하는 과정에서 중복되는 문자의 작성을 피함으로써, 가독성도 향상 될 뿐만 아니라 로직을 짜는 과정에서도 쓸데없는 시간 낭비를 줄일 수 있었다. 어려웠던 점은 템플릿이다. 템플릿의 개념 자체와 단독 클래스로서 템플릿을 구현하고 실행하는 것은 어렵지 않았지만, 하나의 탬플릿 클래스에서 다른 클래스 객체를 다룰 때 constructor가 제대로 작동하지 않는 부분을 해결할 수 없었다. 뿐만아니라 템플릿 멤버 함수에서 다른 클래스 객체의 메소드에 접근할 때도 오류가 발생하여 해결을 위해 구글링을 했으나 본인과 똑같은 상황에 대한 해답을 찾을 수는 없었다. 이후 추가적인 공부를 통해 템플릿이 정상적으로 작동하지 않는 이유를 찾고 템플릿 클래스로써 구현해 볼 예정이다