자료구조와 C++ 프로그래밍

이름 : 이인
학번 : 2021125050

강의시간 : 월 9:00, 수 10:30
레포트 번호 : 과제4

1.문제 인식 및 분석

결국 구현해야 하는 부분은 포인터로 연결된 노드간 데이터 값 비교를 통해 각 노드들이 위치한 순서를 옳바르게 설정해야 하는 것이다. 링크드 리스트 구조를 감안해 생각한다면 각 노드의 위치 뿐만아니라 각 노드가 가리키는 노드 또한 고려해야할 부분인 것이다.
카드 정렬을 링크드 리스트로 구현하기 위해 필요한 자료 구조가 무엇인지 생각해 보았다. 비교할 카드는 타겟 카드와 타겟 카드의 다음에 오는 카드밖에 없으므로 단일 연결리스트로 구현하는 것이 맞다고 판단하였다. 또한 링크드 리스트 전반을 탐색할 기능을 생각해 보니 전체 리스트에 대한 이터레이터를 메소드로 구현하는 방법이 적합하다고 생각했다. 전체적인 객체 설계에 대해서 생각 해 보았을 때 카드를 추상화 하여 카드의 모양이나 숫자를 가지고 있는 객체인 Card 객체와, Card인스턴스들을 연결리스트로 구현하기 위한 기반 노드 객체인 Node 객체, Node들을 탐색하며 정렬 및 출력 기능을 수행하기 위한 Chain객체로 세 개로 분할하여 설계하는 것이 맞다고 판단했다.
main.cpp에서 전체적인 코드의 기능을 한눈에 살펴볼 수 있도록 하기 위해, 카드를 추가하고 정렬하는 전반적인 동작을 담당하는 기능을 하는 함수를 만들어 메인을 간결화 하고 전반적인 구조를 파악할 수 있도록 하는 것이 좋겠다고 판단했다. 각기 다른 크기를 가진 카드를 생성하기 위해 rand함수를 사용하기로 결정했다. 카드의 모양과 숫자에 대한 정보를 담되, 정렬 시 비교를 편하게 하기 위한 새로운 멤버 변수를 하나 추가하기로 결정해 이후 정렬 시 하나의 변수만으로 비교하여 쉽게 자리를 변경할 수 있도록 한다.

2.해결 방안 구상 및 코드 분석

카드 모양을 매번 값에 따라 문자열에 대응시켜 나타내는 건 너무 비효율 적인 작업이라 판단하여 정적 변수를 사용해 미리 SDHC에 대한 정보를 전역으로 선언해놓고 필요할 때 마다 가져다 쓰는 방식으로 사용하기로 했다. 링크드 리스트이고 단일 연결이기 때문에 삽입되는 방향은 신경쓰지 않기로 했다. 그래서 front rear 구분없이 무조건 front방향에서만 삽입하는 걸 가정하고 push 함수 하나만 구현하기로 하였다.

```
class Card { // 기본적인 데이터를 구성하는 카드 객체
public:
	int number; //Card의 숫자
	char shape; //Card의 모양 S,D,H,C
	int shapingNum; //Card의 shape를 결정 해 주는 숫자
	int id; //카드간 크기를 비교할 때 하나의 숫자로 쉽게 비교하기 위함

	Card() {// 기본 생성자,
		int number = 0;
		char shape = NULL;
		int shapingNum = 0;
		int id = 0;
	}

	Card(int _number, int _shapingNum) {// constructor overloading , number와 shapingNum을 받아 새로운 카드를 생성함
		number = _number;
		shapingNum = _shapingNum;
		id = ((_shapingNum - 1) * 13) + (_number); //고유한 식을 통해 각 카드마다 기존 카드 룰에 맞는 크기를 할당
```

```
            setForm();
        }
        ~Card() =default;
```

```
        void setForm(){//shapeNum 값에  따라서 Card 객체의 shape 결정하는  함수
            this->shape = SHAPES[shapingNum]; // string은  인덱싱 가능하기  때문에 SHAPE를  정적  변수로  선언하고  인덱싱으로  고유  문자  할당
        }
};
```

노드의 데이터 부분을 구성할 카드 객체를 정의한다. 기본적으로 카드가 가지는 멤버변수는 number, shape, shapingNumber, id 이다. number shape는 각각 모양과 숫자를 나타내는 변수고, shapingNumber는 shpae를 결정하기 위한 멤버 변수이다. setForm 함수를 통해 SHAPE전역 변수에 접근하여 인덱싱을 통해 SDHC중 하나를 할당한다. 랜덤 값을 받아 각기 다른 값을 가지는 카드객체를 생성하기 위해 construtor 오버로딩을 구현하였다.

```
template<class T> // 템플릿  클래스  선언
class Node {
        friend class Chain<T>; // Chain 클래스에서 Node클래스  사용을 위한 friend 선언
private:
```

```
        T data; // Card객체가 담길 멤버변수, 템플릿이기에 T타입으로 선언
        Node<T>* link; // 다음 node를 가르킬 포인터 변수
};
```

링크드 리스트로 card를 관리하기 위해 탬플릿 클래스로 Node를 선언하고 Chain에서 해당 Node를 전반적으로 관리한다. Node객체는 Data와 다음 Node를 가리켜야 하기 때문에 다음 노드에 대한 포인터 변수 link를 선언한다.

```
template<class T> //Node<T>를 받을 탬플릿 클래스 선언
class Chain {
private:
        Node<T>* front; // linkedList의 헤드가 되는 노드 front에 대한 주소 변수 선언
public:
        Chain() { // constructor 선언
                front = 0; // front노드의 디폴트 값
        }
        ~Chain() =default; //  destructor 선언
```

```
        void printCard() { //  카드의 모양과 숫자 출력
                Node<T>* current = front; // 맨 첫번째 자리의 노드에 대해 변경없이 탐색하기 위해 첫 노드에 대한 포인터 복사.
```

```
            while (current != 0) { // 맨 끝 노드에 도달할 때 까지
                cout << current->data.shape // 카드 출력
                    << current->data.number
                    << " ";
                current = current->link; // 다음 노드 탐색을 위해 링크에 접근
            }
            cout << endl;
        };
```

정상적인 링크드리스트 탐색을 위해선 탐색 시작 및 끝의 기준이 되는 노드가 필요하고 첫번째 후킹을 가능하게 하는 노드가 필요하다 때문에 이 역할을 담당하도록 할 수 있는 변수 front노드를 만들어 이후 추가로 카드 노드가 생성 되었을 때 링크되도록 한다. Chain constructor를 통해 front의 기본 값을 0으로 설정해 둠으로써 탐색의 시작 혹은 끝을 인식할 수 있도록 정의한다. printCard 함수는 front 노드에 대해 직접적인 수정없이 전체 링크 리스트를 탐색할 수 있도록 새로운 current 포인트 변수를 만들어 front의 디폴트 값인 0이 나타날 때 까지 해당 노드의 데이터를 출력하도록 정의한다. 특히 current = current->link 구문을 통해 노드의 링크를 끝까지 접근할 수 있도록 한다.

```
void order() { // 삽입된 카드 노드 들을 크기에 맞게 정렬
        Node<T>* current = front; // 맨 첫번째 자리의 노드에 대해 변경 없이 탐색하기 위해 첫 노드에 대한 포인터 복사
        Node<T>* pre = 0; // 정렬되는 카드가 카드와 카드 사이에 있는 경우를 고려하기 위한 변수 선언
```

```
        while (current->link != 0) { // 맨 끝 노드에 도달할 때 까지
            if (current->data.id > current->link->data.id) { // 기준 노드가 다음 노드보다 크다면
                Node<T>* temp; // 노드 정보 손실을 막기 위한 임시 노드 포인터 변수 선언
                temp = current->link; // 타겟 노드의 링크 정보 저장
                current->link = temp->link; // 타겟 노드와 다음 노드의 위치를 바꾸기 때문에 다음노드의 링크를 타겟 노드 링크에 할당
                temp->link = current; // 다음 노드는 타겟노드의 위치에 도달, 다음 노드가 가리키는건 타겟 노드가 되어야 하기 때문
```

```
                if (current != front) { // 자리가 바뀌는 노드가 노드와 노드 사이에 있을 때
                    pre->link = temp; // current와 temp(뒤노드)가 자리가 바뀌었기 때문에 current를 가리키던 pre는 temp를 가리켜야 함
                }
```

```
                if (current == front) { // 자리가 바뀌는 두 노드를 제외한 카드가 없는 경우
                    front = temp;
                }
```

```
                pre = temp; //
            }
            else { // 기준노드 < 다음노드  순서로  크기가  있는  경우
                break;
            }
            cout << "--Swapping--";
            printCard(); // 자리  변환  이후  전체  리스트  출력
        }
```

order함수는 링크드 리스트의 카드 데이터를 바탕으로 순서를 정렬해주는 함수이다. 핵심 로직은 스왑하는 카드가 카드와 카드 사이에 있는지 혹은 front에 위치하는 지다. 어짜피 삽입 방향은 front이기 때문에 넣은 카드가 세 개 이상인 경우에는 만약 스왑이 일어나는 경우 target 카드 기준 앞 뒤 카드 총 세 개의 노드가 영향을 받게 된다. 외냐하면 previous 카드는 target카드를 가리키게 되고 target카드는 next카드를 가리키기 때문이다. 또한 order함수가 전체 노트를 탐색할 수 있도록 이터레이터 확장 함수 형태로 구현한다. 스왑하는 노드들에 대해 data와 link모두 적절하게 바꾸어줘야 정상적인 node 스왑을 할 수 있다.

```
void push(T& o) {
        Node<T>* tmpNode = new Node<T>; //새로운  노드  객체  생성
        tmpNode->data = o; // initalize함수에서  받은  카드  데이터를  할당
```

```
        tmpNode->link = front;  //front는  노드  삽입을  위한  명목상의
헤드  노드, 디폴트  값으로 0을  가짐, 새로  추가할  노드  객체의  링크에
front정보  할당
        front = tmpNode; // 기존의 front 자리에  새로  추가한  노드  할당
        cout << "New Card : " //새로  추가한  노드  카드에  대한  정보  출
력
                << tmpNode->data.shape
                << tmpNode->data.number
                << endl;

        cout << "push : ";
        printCard(); // 푸쉬  함수  수행이후  전체  링크드  리스트  출력
        order(); // 순서  정렬  시행
}
```

랜덤 값을 통해 생성한 새로운 카드객체를 참조로 받아 새로운 노드를 만들고 해당 노드는 front방향에서 삽입이 되게 때문에 항상 제일 최근에 들어온 노드가 front에 위치하게 된다. 만약 처음 노드를 추가하는 경우라면 원래 Chain클래스에 존재하는 최초의 노드인 0을 디폴트 값으로 가지는 노드는 뒤로 밀려나게 되고 이는 곧 새로 추가하는 카드의 링크에 연결되는 것이다.

```
void initializeChain(){ // 연결리스트  체인  객체  초기화
```

```
        srand((unsigned int)time(NULL));// rand호출마다  다른  값이  출
력되기  위한 srand 및 time


        for(int i = 0; i<5; i++){ // 5장의  카드  삽입
                int num = (rand() % 13) + 1; // 1~13사이의  수  랜덤  생성
                int shpaingNum = (rand() % 4) + 1;// 1~4사이의  수  랜덤
생성
                Card tmpCard = Card(num, shpaingNum); //각  반복마다
임시  카드  객체  생성
                this->push(tmpCard); // 각  카드 push
        }
}
```

initialize함수를 통해 위에서 구현했던 모든 함수들을 활용해 카드를 생성하고 크기 순서대로 정렬 삽입하는 기능을 수행하게 된다. 랜덤 함수를 사용해 고유 값을 가지는 카드를 만들어내고 chain에 푸쉬하는 것을 반복하여 최종적으로 요구하는 기능을 수행하게 되는 것이다.

3.출력결과

New Card : H11
push : H11
New Card : H8
push : H8 H11
New Card : H3

push : H3 H8 H11
New Card : C9
push : C9 H3 H8 H11
--Swapping--H3 C9 H8 H11
--Swapping--H3 H8 C9 H11
--Swapping--H3 H8 H11 C9
New Card : D6
push : D6 H3 H8 H11 C9

New Card : C5
push : C5
New Card : H8
push : H8 C5
New Card : S5
push : S5 H8 C5
New Card : D7
push : D7 S5 H8 C5
--Swapping--S5 D7 H8 C5
New Card : D13
push : D13 S5 D7 H8 C5
--Swapping--S5 D13 D7 H8 C5
--Swapping--S5 D7 D13 H8 C5

New Card : S12
push : S12
New Card : D11
push : D11 S12
--Swapping--S12 D11
New Card : C10
push : C10 S12 D11

--Swapping--S12 C10 D11
--Swapping--S12 D11 C10
New Card : C8
push : C8 S12 D11 C10
--Swapping--S12 C8 D11 C10
--Swapping--S12 D11 C8 C10
New Card : H10
push : H10 S12 D11 C8 C10
--Swapping--S12 H10 D11 C8 C10
--Swapping--S12 D11 H10 C8 C10

4. 배운점, 어려웠던 점 등

저번 과제에서 제대로 알지 못했던 템플릿 클래스에 대해 구체적으로 배울 수 있었다. 왜 템플릿 클래스를 선언하여 사용하는지 커스텀 클래스를 구현하는 과정을 통해서 느낄 수 있었고 탬플릿을 사용했을 때 어떤 타입이든 유동적으로 받아 처리할 수 있다는 점이 와닿았다. cpp상 문법에서 탬플릿 클래스 인스턴스를 생성할 때 처음에는 이해 안됐던 문법 구조가 작동 과정을 이해하고 나니 납득이 가능 문법 구조였다. 또 프로그램을 구성할 때 코드를 작성하기 이전에 조금 더 효율적이고 함수의 재사용을 방지하기 위한 방법을 고민하다 보니 설계 단계에서 조금 더 힘을 실어야 겠다는 걸 느꼈다. 이때문에 객체지향 설계에 대해 알아보며 SOLID원칙에 대해 알 수 있었다. single posibility 원칙을 적용하여 클래스 구현을 해보려 했지만, 어느정도 모양만 나올 뿐 완벽하게 객체 분할을 이루진 못하였다. 다음 번에는 SOLID원칙을 최대한 더 많이 준수할 수 있는 구조를 설계 하려 노력하겠다고 생각했다. 또 함수 호출 시점에 따라 출력 결과가 달라지는 것을 디버깅 과정에서 뼈저리게 느꼈는데 알고리즘 구상 단계에서 정확히 어느 부분에서 값 할당과 함수 호출이 이루어지는 지 신경써서 설계 해야겠다고 생각했다. 특히 -> 와 . 을 연쇄적으로 사용하는 부분이 처음에는 이해가 가질 않았다. 그러나 ->와 . 의 용도 차이를 정확하게 이해하니 포인터와 일반 변수에 개념에 대한 이해가 더욱 구체화 되고 더불어 어떤 상황에 정확히 ->를 사용해야하는지 배웠다.