Data Structure Project

Project #1

담당교수 : 이 기 훈

제출일 : 2018. 10. 03

학과 : 컴퓨터정보공학부

학번 : 2017202029

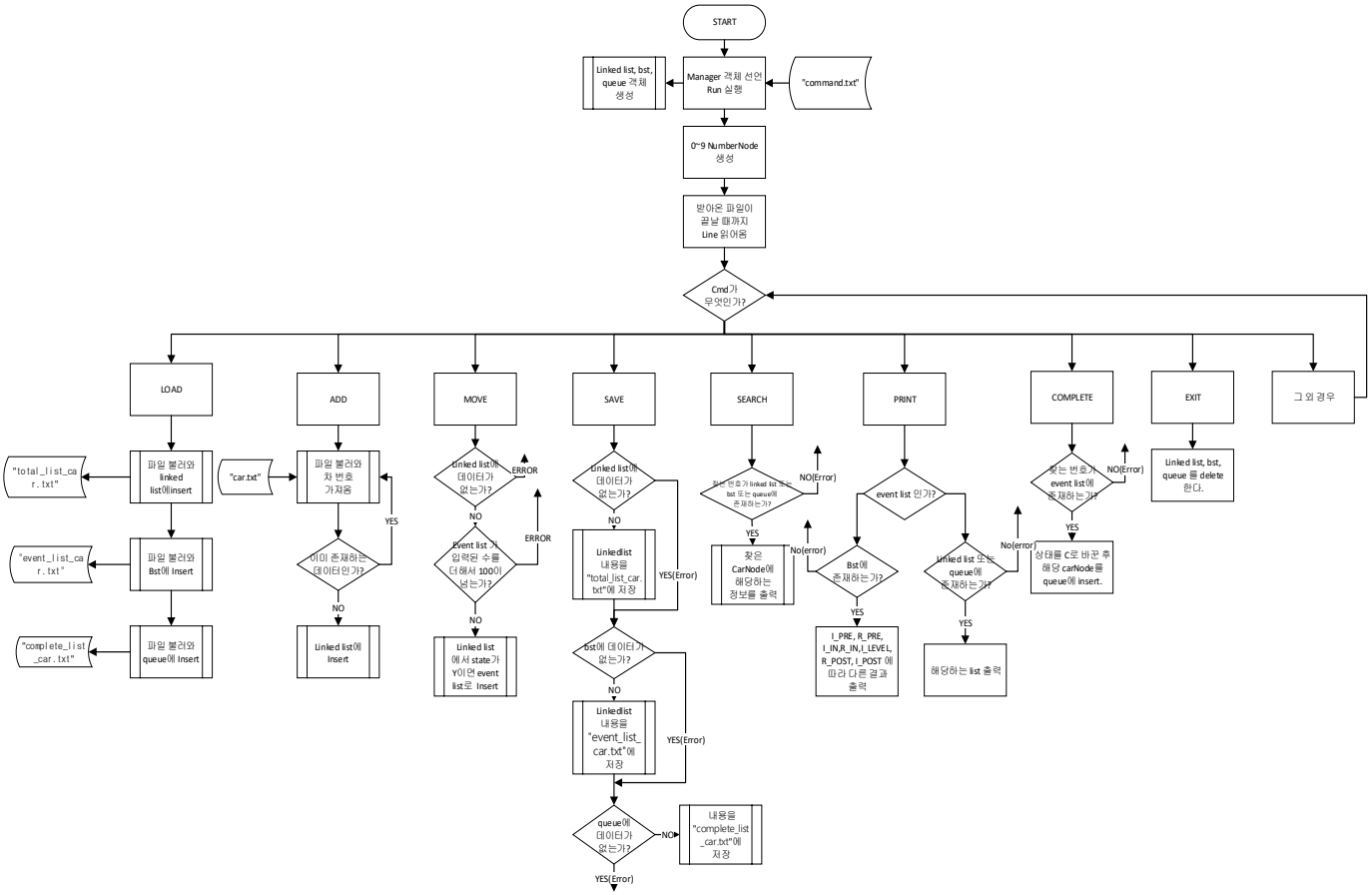
이름 : 전 효 희

1. Introduction

본 프로젝트는 linked list, bst(binary search tree), queue를 사용하여 자동차 보험 처리에 대한 데이터를 저장 및 처리하는 프로그램을 제작하는 프로젝트이다. 프로그램을 활용해 차량 정보를 입력 받아, 전체 데이터 리스와 보험처리 중 리스트, 처리 후 리스트를 제작함으로써 효율적인 데이터 정리와 처리가 가능하다.

프로그램은 데이터 입력(LOAD, ADD)과 탐색(SEARCH), 출력(PRINT)이 가능하며, 그 외의 데이터 처리(MOVE, COMPLETE)가 가능하다. 그 외 각각의 기능의 수행에서 문제가 발생할 경우, 해당하는 오류 코드를 출력할 수 있도록 예외를 처리한다.

2. Flow Chart



- 프로그램이 시작하면 manager 객체를 하나 생성하고 "command.txt"를 인자로 넣어준다. manager의 run을 시작함과 동시에 numbernode 0~9를 생성한다.
- 받아온 파일이 끝날 때까지 한 줄 씩 읽어 cmd변수에 저장한다. cmd의 값에 따라 기능

으로 이동한다.

- cmd가 LOAD일 경우 "total_list_car.txt", "event_list_car.txt", "complete_list_car.txt" 파일을 불러와 아직 데이터가 없을 경우 해당하는 자료구조에 insert한다.
- cmd가 ADD일 경우 "car.txt" 파일을 불러와 linked list에 한 줄 씩 읽어 데이터를 입력한다. 만약 존재하는 차 번호일 경우 제외하고 입력을 진행한다.
- cmd가 MOVE일 경우 개수를 입력 받아 Linked list에 충분한 데이터가 있고 event list에 100개 이하의 데이터가 존재하게 될 경우 linked list의 데이터를 bst로 옮긴다.
- cmd가 SAVE일 경우 Linked list, bst, queue가 존재할 경우 존재하는 자료구조에 대해 데이터들을 "total_list_car.txt", "event_list_car.txt", "complete_list_car.txt"에 저장한다.
- cmd가 SEARCH일 경우 찾는 번호가 각 자료구조에 존재할 경우, 해당하는 데이터를 찾아 출력한다.
- cmd가 PRINT일 경우, 파라미터가 EVENT_LIST이면 (BST에 데이터가 존재할 때,) R_PRE, I_PRE 등 출력형식에 해당하는 출력을 진행한다. 그 외일 경우 linked list 또는 queue 전체를 출력한다.
- cmd가 COMPLETE일 경우, complete를 진행할 번호가 bst에 존재하면 해당 데이터를 찾아 queue에 insert하고 상태를 'C' 로 바꿔준다.
- cmd가 EXIT일 경우, 모든 동적할당 데이터를 해제해주고 프로그램을 종료한다.
- cmd가 그 외일 경우 에러를 출력하고 다음 라인을 실행한다.
- 각각의 기능에 실패하는 경우 해당하는 에러코드를 출력한다.

3. Algorithm

- Linked list
  - ◆ Insert

    : linked list는 새로운 데이터가 들어오는 대로 이전 노드 뒤에 덧붙이는 형태였기 때문에, 반복문을 사용해 다음 데이터가 없는 노드까지 이동한 후 새로운 노드를 붙여주는 방식으로 구현했다.

  - ◆ Search

    : search 또한 linked list의 head 부터 데이터의 끝까지 이동하면서 찾는 번호와 동일한 번호가 있는지 탐색하는 방식으로 구현하였다.

◆ Print

: 헤드가 비어 있지 않을 때, 처음 노드부터 끝 노드까지 이동하면서 노드의 번호, 차주, 상태를 출력한다.

◆ Save

: 세이브 역시 헤드가 비어 있지 않을 때(정보가 존재할 때), 처음 노드부터 끝 노드로 이동하면서 노드의 정보를 텍스트 파일에 저장한다.

◆ Delete

: delete는 두 과정을 거쳐 진행했다. 먼저, 입력 받은 차 번호에 해당하는 차 노드를 찾는 과정을 거쳤다. 찾은 차가 만약 헤드일 경우 찾은 차의 next 노드를 헤드로 해준다. 만약 찾은 차가 헤드가 아닐 경우 찾은 차의 앞 노드의 next 노드를 찾은 차의 next노드로 설정해준 후 찾은 노드를 분리해낸다.

- BST

◆ Insert

: 노드 p를 루트부터 시작하여 찾는 번호와 노드가 가진 번호가 같을 때까지 부모와 p를 저장하고 바꿔 내려간다. 루트가 있다면 찾은 노드를 부모 노드의 자식으로 연결하고, 루트가 없다면 입력된 노드를 루트로 정한다.

◆ Delete

: 먼저 루트부터 탐색하며 내려가 입력 받은 번호의 노드와 그 부모 노드를 찾는다. 이후 네 가지 경우에 따라 다르게 처리한다.

1) 찾은 노드가 leaf 노드 일 경우

: 찾은 노드 p의 부모가 null이면 p가 루트이므로 루트를 지운다. 또는 p가 부모의 왼쪽 자식일 경우 부모의 왼쪽을 null로, 오른쪽 자식일 경우 오른쪽을 null로 바꾼다.

2) 찾은 노드에게 왼쪽자식만 있을 경우

: 찾은 노드 p의 부모가 null이면 p가 루트이므로 루트를 p의 왼쪽 자식으로 한다. p가 부모의 왼쪽 자식일 경우 p의 자식을 부모의 왼쪽 자식에 연결 후 지운다. 반대의 경우 오른쪽자식을 부모의 오른쪽에 연결한다.

3) 찾은 노드에게 오른쪽 자식만 있을 경우

: 찾은 노드 p의 부모가 null이면 p가 루트이므로 루트를 p의 오른쪽 자식으로 한다. p가 부모의 왼쪽 자식일 경우 p의 자식을 부모의 왼쪽 자식에 연결 후 지운다. 반대의 경우 오른쪽자식을 부모의 오른쪽에 연결한다.

4) 찾은 노드에게 양쪽 자식이 모두 있을 경우

: 찾은 노드 p의 왼쪽 자식 중 가장 큰 값을 찾기 위해 왼쪽 서브루트에서 가장 오른쪽에 있는 값을 찾는다. 찾은 가장 큰 노드가 자식이 있으면 큰 노드의 부모와 자식을 연결하고, 아니면 널을 부모에 연결한다.

왼쪽에서 가장 큰 노드와 p의 데이터를 바꾼 후, 바꾼 자식을 제거한다.

◆ Search

: 노드 curr을 루트부터 시작하여 찾는 값이 노드보다 작으면 왼쪽 자식으로, 크면 오른쪽 자식으로 이동한다. 값이 같아지면 curr을 반환하고 값을 찾지 못하면 null을 반환한다.

◆ Print

: 문자열을 입력 받아 조건과 같은 경우에 다음의 함수들을 실행한다.

1) R_PRE

: 재귀적으로 preoreder 순서대로 출력한다. 입력 받은 노드를 먼저 방문하고, 노드의 왼쪽을 함수에 다시 넣어 재귀 함수를 사용한다. 이후 왼쪽을 모두 방문(출력)한 후 오른쪽을 함수에 넣어 재귀함수를 사용해 preorder를 구현한다.

2) R_POST

: 재귀적으로 postorder를 출력한다. 먼저 왼쪽 노드를 재귀함수에 넣어 탐색 후 오른쪽 노드를 재귀함수에 넣어 탐색한다. 이후 입력 노드를 가장 마지막에 방문(출력)한다.

3) R_IN

: 재귀적으로 inorder를 출력한다. 먼저 왼쪽 노드를 재귀함수에 넣어 탐색 후 현재 노드를 방문(출력)한다. 이후 오른쪽 노드를 다시 재귀함수에 넣어 탐색하는 방식으로 구현한다.

4) I_PRE

Stack에 입력 받은 노드를 넣고, 노드 방문(출력) 후 노드를 pop한다. 만약 temp에게 오른쪽 자식이 있다면 오른쪽 자식을 stack에 넣고 그 다음 왼쪽 자식이 있다면 왼쪽자식을 stack에 넣는 과정을 반복하면 preorder를 구현할 수 있다.

5) I_IN

노드가 null이 될 때까지 stack에 노드를 넣어가며 노드를 왼쪽으로 이동한다. 이후 가장 위의 노드부터 출력하고 노드를 오른쪽으로 이동한 후 위를 반복한다. Node가 null이면서 stack이 비었다면 반복을 멈춘다.

6) I_POST

먼저 한 stack(s1)에 입력 받은 노드를 넣는다.

이후 s1에 있던 노드를 다른 stack (s2)로 옮기고 s1의 가장 위 노드에게 왼쪽자식이 있다면 왼쪽자식을 넣고, 그 다음 오른쪽자식이 있다면 오른쪽자식을 stack(s1)에 넣는 과정을 반복한다. 만약 s1이 비었으면 옮겨진 s2를 위부터 차례로 출력한다.

7) I_LEVEL

Queue에 입력 받은 노드를 넣는다.

Queue에 가장 앞 노드를 출력하고 지운다. 그 다음 만약 queue의 가장 앞 노드에게 왼쪽 자식이 있다면 왼쪽 자식을 queue에 넣고, 이후 오른쪽 자식이 있다면 오른쪽 자식을 queue에 넣는다. 이를 반복한다.

◆ Save

저장 후 파일이 load 될 때 skewed되는 것을 막기 위해 preorder 순으로 저장한다. I_PRE와 동일하게 반복적으로 노드의 출력을 하되 세이브 파일에 출력한다.

- Queue

◆ Push

처음 들어온 노드를 헤드로 하여 노드의 끝까지 이동한 후, 노드의 next가 null일 때 노드의 next로 들어온 노드를 이어준다.

◆ Pop

헤드가 가장 먼저 들어온 데이터이므로 헤드를 현재 헤드의 next 노드로 바꿔 준 후 헤드였던 노드를 삭제한다.

◆ Search

: queue의 head 부터 데이터의 끝까지 이동하면서 찾는 번호와 동일한 번호가 있으면 해당 노드를 반환하고, 없으면 null을 반환한다.

◆ Print

: 헤드가 비어 있지 않을 때, 처음 노드부터 끝 노드까지 이동하면서 노드의 번호, 차주, 상태를 출력한다.

◆ Save

: 헤드가 비어 있지 않을 때, 첫 노드부터 끝노드(null)이 나올 때까지 이동하면서 차 번호, 차주, 상태를 출력한다.

4. Result Screen

1. 파일이 없는 채로 최초실행

Command.txt

```
LOAD
ADD
MOVE 10
COMPLETE 3654
SEARCH 3654
PRINT TOTAL_LIST
PRINT EVENT_LIST I_PRE
PRINT EVENT_LIST R_PRE
PRINT EVENT_LIST I_IN
PRINT EVENT_LIST R_IN
PRINT EVENT_LIST I_POST
PRINT EVENT_LIST R_POST
PRINT EVENT_LIST I_LEVEL
SAVE
EXIT
```

- Car.txt

3654	조만재	Y
6873	김태환	Y
8925	김홍지	N
7931	김호준	N
8695	신희민	Y
5703	전선경	Y
2754	권주영	N
8971	박채원	N
9705	조은별	Y
8014	남궁민	Y
3041	이지은	Y
5084	유한욱	N
8605	정현우	Y
7635	한성민	N
6053	이한솔	N
2549	전예은	N
2763	전보은	Y
9048	이미연	N
4301	김은지	Y
7382	백현은	N
9138	남윤창	Y
5028	김성훈	N
9314	이지연	N
2748	김종훈	N
4623	박지현	Y

Log.txt

1) LOAD, ADD, MOVE, COMPLETE, SEARCH

```
======== ERROR ========
100
=======================

======== ADD ========
Success
=======================

======== MOVE ========
Success
=======================

======== COMPLETE ========
Success
=======================

======== SEARCH ========
3654 조만재 C
=======================
```

- Load 할 데이터가 없음으로 error(100) 출력
- Add 성공
- Move 성공
- Complete 성공
- Complete한 대상의 search : 대상의 상태가 C로 바뀐 것을 확인 할 수 있다.

2) PRINT

- Total_list

```
======== PRINT ========
8925    김홍지  N
7931    김호준  N
2754    권주영  N
8971    박채원  N
5084    유한욱  N
7635    한성민  N
6053    이한솔  N
2549    전예은  N
9048    이미연  N
7382    백현은  N
9138    남윤창  Y
5028    김성훈  N
9314    이지연  N
2748    김종훈  N
4623    박지현  Y
=======================
```

: total list에 남은 목록

- I_PRE, R_PRE

```
======== PRINT ========
3041    이지은  Y
2763    전보은  Y
5703    전선경  Y
4301    김은지  Y
6873    김태환  Y
9705    조은별  Y
8695    신희민  Y
8014    남궁민  Y
8605    정현우  Y
=======================
```

```
======== PRINT ========
3041    이지은  Y
2763    전보은  Y
5703    전선경  Y
4301    김은지  Y
6873    김태환  Y
9705    조은별  Y
8695    신희민  Y
8014    남궁민  Y
8605    정현우  Y
=======================
```

: event list에 남은 목록이 preorder로 출력됨

(Recursive와 Iterative preorder의 결과가 동일한 것을 볼 수 있다.)

- I_IN, R_IN

```
======== PRINT ========
2763    전보은   Y
3041    이지은   Y
4301    김은지   Y
5703    전선경   Y
6873    김태환   Y
8014    남궁민   Y
8605    정현우   Y
8695    신희민   Y
9705    조은별   Y
=======================

======== PRINT ========
2763    전보은   Y
3041    이지은   Y
4301    김은지   Y
5703    전선경   Y
6873    김태환   Y
8014    남궁민   Y
8605    정현우   Y
8695    신희민   Y
9705    조은별   Y
=======================
```

: event list에 남은 목록이 inorder로 출력됨

(Recursive와 Iterative inorder의 결과가 동일한 것을 볼 수 있다.)

- I_POST, R_POST

```
======== PRINT ========
2763    전보은   Y
4301    김은지   Y
6873    김태환   Y
5703    전선경   Y
3041    이지은   Y
8605    정현우   Y
8014    남궁민   Y
8695    신희민   Y
9705    조은별   Y
=======================

======== PRINT ========
2763    전보은   Y
4301    김은지   Y
6873    김태환   Y
5703    전선경   Y
3041    이지은   Y
8605    정현우   Y
8014    남궁민   Y
8695    신희민   Y
9705    조은별   Y
=======================
```

: event list에 남은 목록이 postorder로 출력됨

(Recursive와 Iterative postorder의 결과가 동일한 것을 볼 수 있다.)

- I_LEVEL

```
======== PRINT ========
3041    이지은   Y
9705    조은별   Y
5703    전선경   Y
8695    신희민   Y
8014    남궁민   Y
8605    정현우   Y
2763    전보은   Y
4301    김은지   Y
6873    김태환   Y
=======================
```

: event list에 남은 목록이 inorder로 출력됨

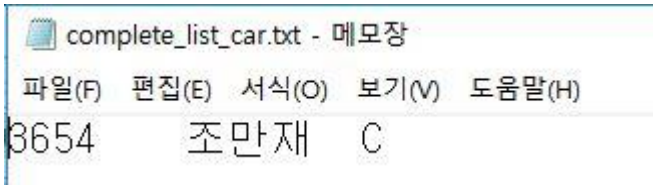
3) SAVE, EXIT

```
======== SAVE ========
Success
=======================

======== EXIT ========
Success
=======================
```

: save, exit 성공 결과

SAVE 결과로 저장된 total_list_car.txt, event_list_car.txt



Complete_list_car.txt

2. List 파일이 있는 채로 실행

   - Car.txt / command.txt

9178	서예지	Y
8126	차성우	Y
9254	한준희	Y
2635	김준석	N
7954	전성원	N
8063	홍현기	Y
9720	김준동	Y
8495	신진한	N
2879	김윤기	N
8547	심재현	Y
1506	고성협	Y
1497	송민준	Y
2361	문요한	N
6123	김효빈	Y
6538	권혁신	N
8326	강예정	N
7965	배득현	N
8534	신호준	Y
3629	김도엽	N
9743	김기선	Y
4629	주재홍	N
1068	김지웅	Y
3165	이명진	N
8432	나승학	N
6725	이지훈	N
4860	김재준	Y
8694	김주형	Y
4236	이상효	N
1467	서진석	Y
6251	박현	N

```
LOAD
ADD
MOVE 10
SEARCH 3654
PRINT EVENT_LIST I_IN
SAVE
EXIT
```

- log.txt

```
======== LOAD ========
Success
======================

======== ADD ========
Success
======================

======== MOVE ========
Success
======================

======== SEARCH ========
3654 조만재 C
======================

======== PRINT ========
1497    송민준   Y
1506    고성협   Y
2763    전보은   Y
3041    이지은   Y
4301    김은지   Y
4623    박지현   Y
5703    전선경   Y
6873    김태환   Y
8014    남궁민   Y
8063    홍현기   Y
8126    차성우   Y
8547    심재현   Y
8605    정현우   Y
8695    신희민   Y
9138    남윤창   Y
9178    서예지   Y
9254    한준희   Y
9705    조은별   Y
9720    김준동   Y
======================

======== SAVE ========
Success
======================

======== EXIT ========
Success
======================
```

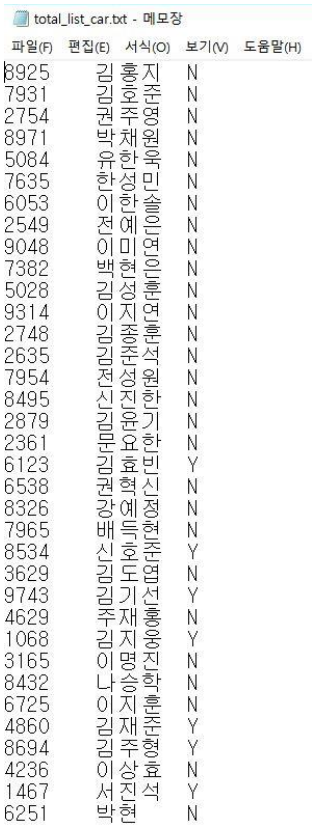
1) LOAD: 3개의 파일이 모두 존재하고 자료구조에 구성에 성공함

2) ADD, MOVE 성공 출력

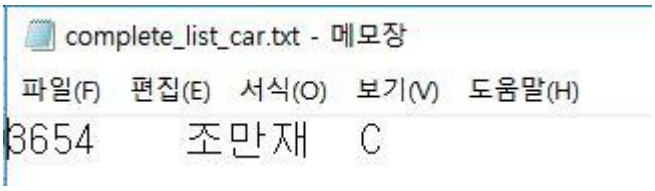
3) SEARCH: command에서 complete를 진행하지 않았으나, 이전 complete_list_car.txt에서 load한 내용이 옮겨져 C를 가진 3654가 검색됨.

4) PRINT: 기존 event_list_car.txt에서 load한 내용에 덧붙여 move한 내용까지 총 20개의 event list가 출력된다.

5) SAVE, EXIT 성공



ADD 와 MOVE로 목록이 증가한 total_list_car.txt, event_list_car.txt



이전과 동일한 complete_list_car.txt

5. Question

아래 설명하고 소스코드 예를 보일 것

- 가상 함수가 무엇인가?

```cpp
#include <iostream>

using namespace std;

class Animal {
public:
    void cry() { cout << "동물의 울음소리" << endl; }

};

class Human : public Animal {
public:
    void cry() { cout << "사람은 엉엉 울어요" << endl; }
};

int main() {
    Animal A, *pA;
    Human H;

    pA = &A;
    pA->cry();
    pA = &H;
    pA->cry();
    return 0;
}
```

```
동물의 울음소리
동물의 울음소리
계속하려면 아무 키나 누르십시오 . . .
```

- 객체지향 프로그래밍에서 부모 클래스(Animal)를 상속받은 자식 클래스(Human)가 같은 이름의 함수를 오버라이딩 하면 컴파일 또는 실행 중에 바인드 된다.
- 만약, 위 코드와 같이 진행하면 Animal 포인터 형인 pA에 Animal인 A의 주소를 담거나 Human 인 H를 담거나 상관없이, cry 함수의 결과는 똑같이 Animal의 cry가 나온다.
- C++의 경우, 컴파일 시에 실제 객체가 아닌 선언된 포인터 변수의 자료형을 기준으로 하여 멤버 함수를 호출한다. 따라서 실제 포인터가 가리키는 Human이 아닌 포인터 변수로 선언된 Animal의 함수를 출력하는 것이다.

```cpp
#include <iostream>

using namespace std;

class Animal {
public:
    virtual void cry() { cout << "동물의 울음소리" << endl; }

};

class Human : public Animal {
public:
    virtual void cry() { cout << "사람은 엉엉 울어요" << endl; }
};

int main() {
    Animal A, *pA;
    Human H;

    pA = &A;
    pA->cry();
    pA = &H;
    pA->cry();
    return 0;
}
```

```
동물의 울음소리
사람은 엉엉 울어요
계속하려면 아무 키나 누르십시오 . . .
```

- 위와 같이 virtual 키워드를 써서 virtual function을 만들면, 가리키는 대상의 바인딩이 컴파일 시에 이루어 지지 않고, 프로그램이 실행될 때 일어나게 된다. 따라서 pA가 Human의 객체를 가리키는 것을 인식하고 Human의 cry 함수를 가져오게 된다.
- 이렇듯 실행시에 바인딩이 이뤄지게 하는 함수를 가상함수(virtual function)라고 한다.

- 순수 가상 함수가 무엇인가?

```cpp
#include <iostream>

using namespace std;

class Animal {
public:
    virtual void cry() = 0;

};

class Human : public Animal {
public:
    virtual void cry() { cout << "사람은 엉엉 울어요" << endl; }
};

int main() {
    Animal *pA;

    pA = new Human;
    pA->cry();
    return 0;
}
```

```
사람은 엉엉 울어요
계속하려면 아무 키나 누르십시오 . . .
```

- 순수가상함수는 부모 클래스에서 함수를 선언만 함으로써 자식클래스가 구현해야 할 함수를 제시만 하는 함수이다. 따라서 자식클래스에서 반드시 재정의가 필요하다.
- 가상함수와 같은 형태를 취하고 정의 대신 = 0; 을 붙여 만든다.
- 코드와 결과를 통해 순수 가상 함수는 자식 클래스에서 정의하여 사용됨을 알 수 있다.

- 추상 클래스가 무엇인가?

```cpp
#include <iostream>

using namespace std;

class Animal {
public:
    virtual void cry() = 0;

};

class Human : public Animal {
public:
    virtual void cry() { cout << "사람은 엉엉 울어요" << endl; }
};

int main() {
    Animal *pA;

    pA = new Human;
    pA->cry();
    return 0;
}
```

- 순수 가상 함수를 한 개 이상 포함한 클래스를 추상 클래스라고 한다. 그 함수가 추상적으로 제시되어 있고, 자식클래스에서 구현해야 하기 때문이다.
- 추상 클래스는 구성 요소인 순수 가상 함수가 구성되지 않았기 때문에, 객체를 따로 만들 수 없다. 그러나 예시와 같이 추상 클래스의 포인터를 선언할 수 있다.

- Polymorphism이 무엇인가?

```
5   class Animal {
6   public:
7       virtual void cry() = 0;
8   };
9
10  class Human : public Animal {
11  public:
12      virtual void cry() { cout << "사람은 엉엉 울어요" << endl; }
13  };
14  class Dog : public Animal {
15  public:
16      virtual void cry() { cout << "개는 멍멍 울어요" << endl; }
17  };
18  class Cat : public Animal {
19  public:
20      virtual void cry() { cout << "고양이는 야옹 울어요" << endl; }
21  };
22  int main() {
23      Animal *pA;
24
25      pA = new Human;
26      pA->cry();
27
28      pA = new Dog;
29      pA->cry();
30
31      pA = new Cat;
32      pA->cry();
33      return 0;
34  }
```

```
사람은 엉엉 울어요
개는 멍멍 울어요
고양이는 야옹 울어요
계속하려면 아무 키나 누르십시오 . . .
```

- Polymorphism(다형성)이란 부모클래스를 상속한 각각의 다른 자식 클래스들, 다른 객체들이 동일한 명령에서 서로 다르게 반응할 수 있는 기능을 말한다.
- 추상 클래스의 순수가상함수를 각각의 자식클래스에서 다르게 구현함으로써 만들 수 있다.
- 예시와 같이, Animal의 순수가상함수 cry를 Human, Dog, Cat에서 각각 다르게 구현하고 이를 실행하면 같은 명령이지만 객체에 따라 다른 결과를 가져오게 된다는 것을 볼 수 있다.

- 소멸자 앞에 virtual을 왜 붙여야 하는가?

```cpp
#include <iostream>

using namespace std;

class Animal {
public:
    Animal() { ; }
    ~Animal()
    {
        cout << "Animal 소멸자" << endl;
    }
};
class Human : public Animal {
public:
    Human() { ; }
    ~Human()
    {
        cout << "Human 소멸자" << endl;
    }
};
int main()
{
    Animal *A = new Human;
    delete A;
}
```

```
Animal 소멸자
계속하려면 아무 키나 누르십시오 . . .
```

- 상속의 경우 자식클래스의 소멸자가 호출되면, 자신이 상속받는 부모 클래스의 소멸자를 호출하게 된다.
- 그러나 예시와 같이 Animal 클래스의 포인터에 Human 객체를 할당했으나, 객체의 메모리를 해제해줄 때 Human의 소멸자가 아닌 Animal의 소멸자만 작동하는 것을 볼 수 있다.
- 소멸자도 다른 함수와 마찬가지로 virtual 없이 선언할 경우 컴파일 때 바인딩이 이루어진다. 그럴 경우 역시 포인터의 자료형에 의존하여 함수를 선택하기 때문에, 부모클래스 포인터에 객체를 할당하여도 소멸자는 부모 클래스의 것으로 작동하게 된다.

```cpp
#include <iostream>

using namespace std;

class Animal {
public:
    Animal() { ; }
    virtual ~Animal()
    {
        cout << "Animal 소멸자" << endl;
    }
};
class Human : public Animal {
public:
    Human() { ; }
    virtual ~Human()
    {
        cout << "Human 소멸자" << endl;
    }
};
int main()
{
    Animal *A = new Human;
    delete A;
}
```

```
Human 소멸자
Animal 소멸자
계속하려면 아무 키나 누르십시오 . . .
```

- 위와 같이 각 소멸자에 virtual을 붙여줄 경우, 바인딩이 프로그램 실행시에 이루어지기 때문에 A가 지워질 때 이는 자식 클래스인 Human이 제거되는 것으로 인식되어 Human의 소멸자가 호출되고, 이에 따라 부모클래스의 소멸자도 연속적으로 호출되게 된다.

- 따라서 원하는 '객체'의 알맞은 delete를 위해서는 virtual 키워드가 필요하게 된다.

6. Consideration

프로그램을 구성하면서 가장 까다로웠던 점은 파일의 입력 및 동적할당 데이터의 삭제였다. 파일에서 데이터를 불러오기 위해서는 문자열을 동적 할당하여 입력 받고 삭제하는

과정을 진행해야 했는데, 메모리의 사용이 잘못되어서 인지 연속적인 중단점 트리거 문제가 발생하는 것을 볼 수 있었다. 웹 서치 결과, 이러한 문제는 문자열이 제대로 지워지지 않거나 초기화에 문제가 있었다. 이를 수정하여 매번 데이터가 입력될 때 문자열을 동적 할당하고, 다음 입력 전에 해제하는 방식을 활용하여 고칠 수 있었다.

윈도우에서는 헤더의 선언 없이도 strtok, strcpy와 같은 string 함수를 c++에서 자유롭게 사용할 수 있었다. 그러나 이를 리눅스 환경에서 컴파일 할 경우 에러가 출력되는 것을 볼 수 있었다. 따라서 각각의 string 함수가 쓰인 파일에서 헤더에 <cstring>을 include 하고 리눅스에서 에러가 없어지는 것을 볼 수 있었다.

Carbst의 delete 함수를 구현할 때, 지울 차량 노드의 양쪽자식이 있는 경우에 문제가 발생했다. 왼쪽 자식 중에서 가장 큰 노드를 찾아 지우려는 노드와 데이터를 교환하는 알고리즘으로 진행했다. 그러나 노드를 찾아 데이터를 바꿀 때, 단순히 차 노드를 가리키는 포인터(temp)에 저장하고, 이후 바꿀 자식 노드의 노드와 교체하여 저장된 데이터를 이후 옮기려 했다. 하지만 자식 노드 = 지울 노드처럼 진행하자, temp의 값이 동시에 바뀌어 자식 노드, 지울 노드, temp 모두가 같은 값을 나타내는 것을 볼 수 있었다. 이는 포인터를 잘못사용한 것으로, 내용이 따로 저장되지 않는 것이었다. 이를 고치기 위해 카 노드가 가지고 있는 변수들을 각각 tempNumber, tempState와 같이 저장한후 자식 노드와 그 변수를 교환하는 방식으로 진행했다.