보고서

제목 : Red-Black Tree 구현 - 환자 관리 프로그램 설계

목차

프로젝트 개요

설계 목적

Self-Balancing Binary Search Tree인 Red Black Tree 자료구조를 C++ 언어로 구현하고, 이를 이용하여 환자 관리 프로그램을 설계한다.

요구사항

1) 프로그래밍 언어는 C 또는 C++로 제한한다.
2) Red-Black Tree는 직접 구현하고, Red-Black Tree와 직접적인 관련이 없는 자료구조는 STL 사용 가능하다.
3) 프로그램 실행 후 입력되는 질의에 따라 지정된 기능을 수행한다.
4) 질의는 최대 100,000개가 입력된다.
5) 전체 질의에 대해 2초의 제한시간 이내에 수행되어야 한다
6) 제시한 입출력 형식대로 표준 입출력을 사용하여 처리한다.
7) 질의의 종류는 '신규 가입', '환자 검색', '추가 진료', 유행병 조사'가 있으며, 다른 질의는 입력되지 않는다.

개발 환경

- OS: Windows 10 Pro. 버전 2004(OS 빌드 19041.928)
- IDE: CLion 20.3.2
- g++ 버전: g++ (GCC) 10.2.0 (cygwin64)

필요한 자료구조 및 기능

필요한 자료구조

필요한 자료구조로는 Linked List와 Self-Balancing Binary Search Tree인 Red Black Tree가 있다. Linked List는 각 환자의 진료 기록을 관리하는데 이용한다. 그리고 Red Black Tree는 환자 정보를 관리하는데 사용하고 다음의 property들을 만족한다.

1) root property: 루트 노드의 컬러는 블랙이다.

2) leaf property: 리프 노드의 컬러는 블랙이다.

3) Internal property: 컬러가 레드인 노드의 자식 노드의 컬러는 블랙이다.

4) dept property: 모든 리프 노드들의 black dept는 같다.

두 자료구조 모두 STL을 이용하지 않고 직접 구현하였다. 그리고 레드 블랙 트리의 노드 구조체와 환자 정보를 저장하는 구조체, 진료 기록을 저장하는 구조체인 'treeNode', 'pinfo', 'record' 구조체를 정의하여 사용하였다. 각 구조체에 대한 자세한 명세는 다음과 같다.

```
// 진료 기록
struct record {
    string disease;  // 병명
    int cost = 0;  // 진료비
    record* next = nullptr;  // 이전 진료 기록 포인터
};

// 환자 정보
struct pinfo {
    int pid = 0;  // 환자 번호
    string name;  // 환자 이름
    int tel = 0;  // 연락처
    int ax = 0;  // 주소 x좌표
    int ay = 0;  // 주소 y좌표
    record* records = nullptr;  // 진료 기록 연결 리스트 주소
};

// 트리 노드
struct treeNode {
    pinfo* patient = nullptr;  // 환자 정보 포인터
    int dept = 0;  // 노드 깊이
    int color = BLACK;  // 노드 컬러(블랙/레드)
    treeNode* parent= nullptr;  // 부모 노드 주소
    treeNode* left = nullptr;  // left child 주소
    treeNode* right = nullptr;  // right child 주소
};
```

기능

1) 신규 가입: 새로운 환자 정보를 입력 받아 레드 블랙 트리 노드에 저장하고, 저장된 노드

의 깊이와 '1'을 출력한다. 만약 입력 받은 환자 번호가 이미 존재한다면, 신규 가입은 거절되고 해당 노드의 깊이와 '0'을 출력한다.

2) 환자 검색: 환자 번호를 입력 받은 뒤, 레드 블랙 트리의 해당 환자를 탐색한다. 만약 환자 정보가 존재하면 환자의 정보를 출력한다. 존재하지 않으면, "Not found"를 출력한다.

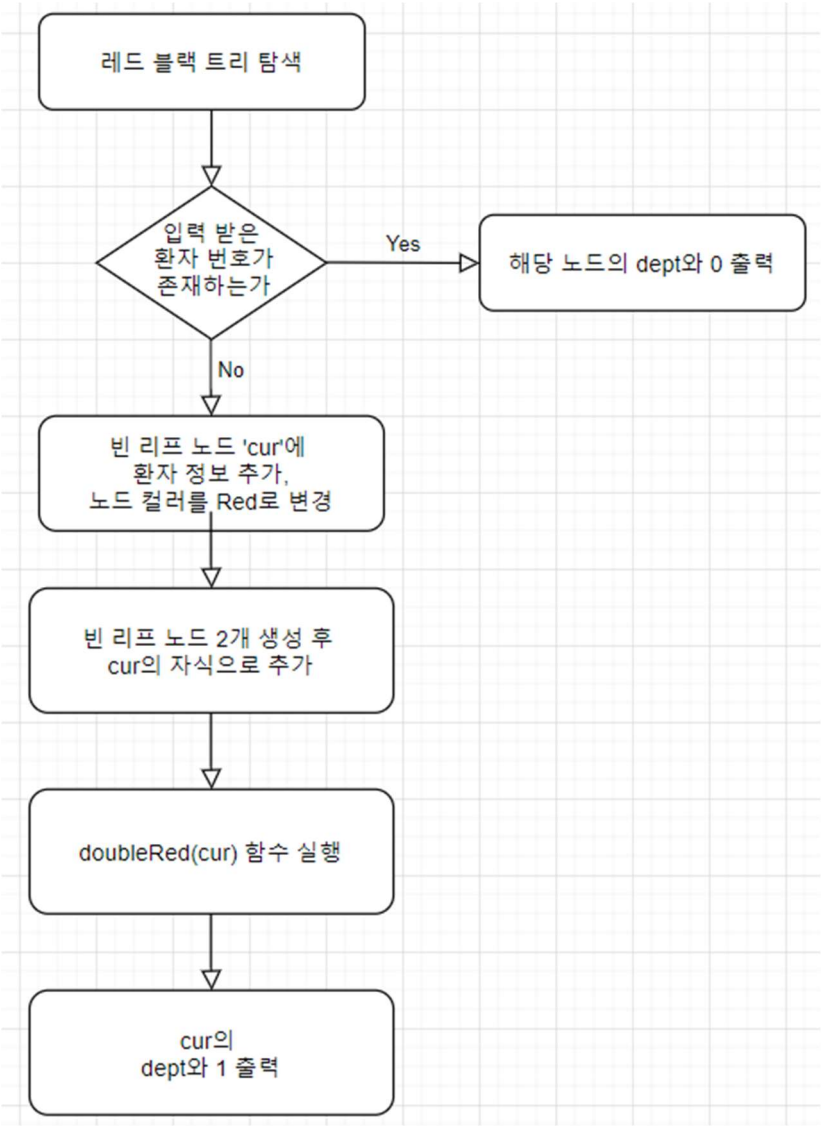
3) 추가 진료: 환자 번호와 진료 정보를 입력 받은 뒤, 레드 블랙 트리에서 해당 환자를 탐색한다. 환자 정보가 존재할 경우 입력 받은 진료 내용을 진료기록에 추가하고, 환자에 대한 노드의 깊이를 출력한다. 만약 환자가 존재하지 않는 경우, "Not found"를 출력한다.

4) 유행병 조사: 병명을 입력 받은 뒤, 트리에 저장된 모든 환자들의 마지막으로 진단받은 병명을 조사한다. 만약 마지막으로 진단받은 병명이 입력으로 주어진 병명과 동일하면 집계한다. 모든 노드를 탐색한 후 집계된 수를 출력하여 유행성의 정도를 파악한다. 다만, 편의상 골절 같은 전염성이 없는 병명도 유행병으로 간주한다.

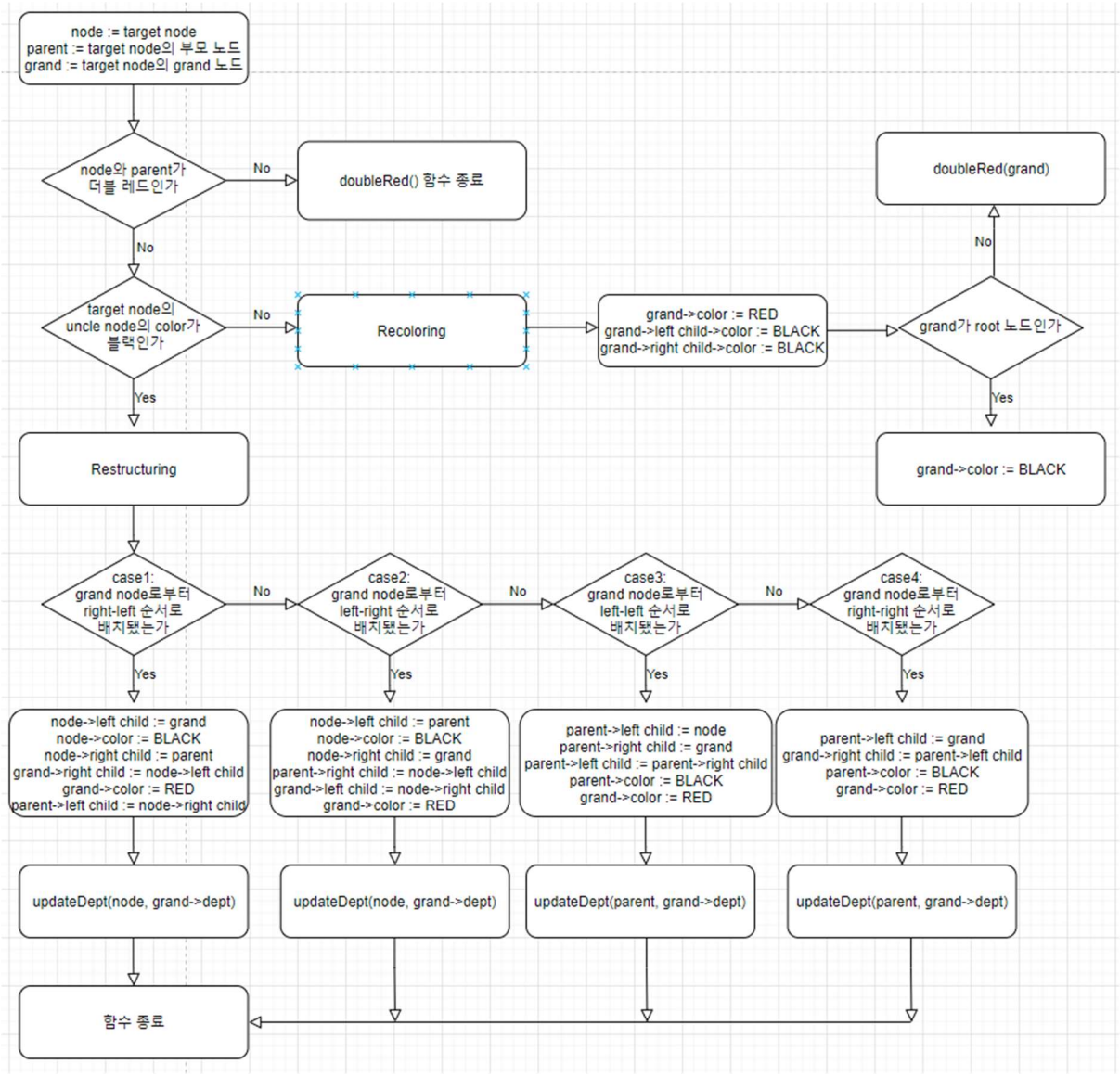
기능별 알고리즘 명세

1) 신규 가입

신규 가입 질의가 입력된 경우, pinfo 구조체와 record 구조체를 동적 할당한 후, 환자 정보를 저장한 뒤 'insertPatient()'함수에 인자로 넘긴다. 'insertPatient()'함수는 다음의 flow chart를 가진다.



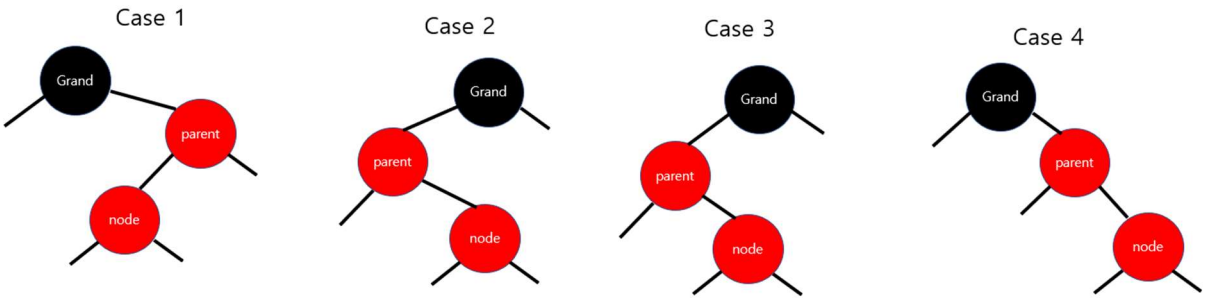
위에서 'doubleRed(treeNode* node)' 함수는 현재 노드와 부모 노드의 컬러가 모두 RED인지 확인하고, 모두 RED라면 이러한 더블 레드 상태를 제거하는 함수이다. 이 함수의 Flow chart는 다음과 같다.



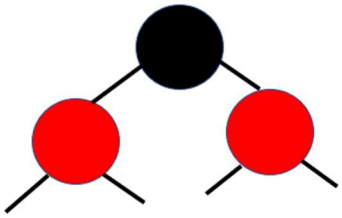
위 플로우 차트에서 우선 현재 타겟 노드가 더블 레드 상태인지 확인한다. 만약 더블 레드가 아니라면 함수를 종료하고, 더블 레드라면 타겟 노드의 uncle 노드의 컬러를 확인한다. uncle 노드의 컬러가 블랙이 아니라면 Recoloring 작업을 하고, 블랙이라면 Restructuring 작업을 한다.

Recoloring 작업은 grand 노드의 자식 노드들의 컬러를 모두 블랙으로 바꾸고 자신의 컬러는 레드로 바꾼다. 만약 이때 grand 노드가 전체 트리의 루트 노드라면 다시 컬러를 블랙으로 바꾸고 아니라면 grand 노드가 더블 레드 상태인지 확인하기 위해 doubleRed() 함수의 인자로 넣어 실행한다.

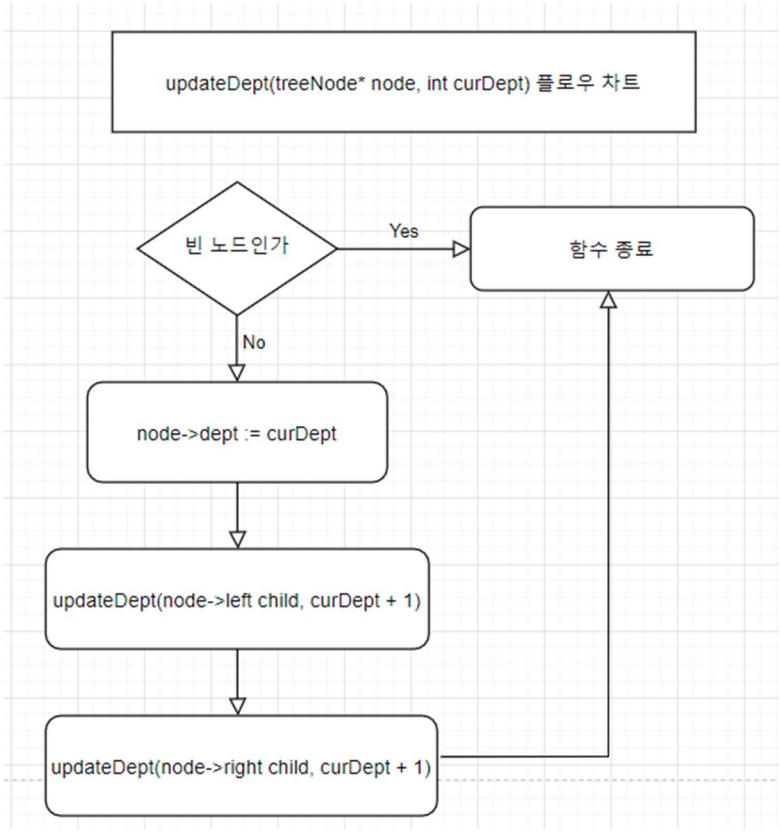
Resturcturing 작업은 현재 노드, 부모 노드, grand 노드의 구조를 바꾸는 것이다.



위의 4 개에 case 별로 다르게 처리하여 아래의 트리 구조로 변경한다.



트리 구조를 변경한 뒤에는 'updateDept()' 함수를 이용하여 각 노드들의 깊이를 업데이트 한다. 'updateDept()'의 플로우 차트는 다음과 같다.



위에서 현재 노드의 dept 값을 업데이트 하고 자식 노드들에 대해 'updateDept()' 함수를 재귀적으로 실행하여 각 노드들의 dept 를 업데이트한다.

종합적으로 신규가입 기능의 시간 복잡도는 다음과 같다.

우선 'insertPatient()' 함수에서 노드를 탐색하는데 O(log(N)) time 이 걸린다. Binary Search Tree 를 탐색할 때 탐색 중인 노드의 깊이가 한 층 깊어지면 탐색 범위는 반 씩 줄어드므로 최악에 경우 O(log(N)) time 이 걸리기 때문이다. 그리고 'doubleRed()' 함수에서 최악에 경우 Recoloring 을 루트 노드까지 해야 하므로 O(log(N)/2) time 이 걸린다. 그리고 'updateDept()'함수에서는 구조가 변경된 서브 트리의 모든 노드들의 깊이를 업데이트를 해야 하므로 상수 시간이 걸린다. 따라서 전체 시간 복잡도는 O(log(N))이다.

2) 환자 검색 기능

환자를 검색하는 함수인 'findPatient()'에 찾고자 하는 환자 번호를 인자로 넘겨 실행한다. 이 함수는 트리에서 환자 정보를 발견하면 해당 노드의 포인터를 리턴하고, 환자 정보가 없으면 nullptr 을 리턴한다. 'findPatient()' 함수에서 트리의 루트 노드부터 탐색한다. 만약 찾는 환자 번호를 가지는 환자 정보가 들어있는 노드를 발견하면 해당 노드를 리턴 한다. 하지만 찾지 못하고 리프 노드에 도달한 경우에는 nullptr 을 리턴한다. Binary Search Tree 에서 탐색은 깊이가 한 칸 씩 깊어질수록 탐색 범위가 반 씩 줄어들기 때문에 최악의 경우 O(lon(N)) time 의 시간 복잡도를 가진다. 따라서 환자 검색 기능의 시간 복잡도는 O(lon(N))이다.

3) 추가 진료 기능

진료를 추가할 환자 번호와 병명, 진료비를 입력 받고, record 구조체를 생성하여 병명과 진료비를 저장한다. 그 다음 'addRecord()' 함수에 환자 번호와 record 구조체 주소를 인자로 넘겨 실행한다.

addRecord() 함수에서는 findPatient() 함수를 이용하여 트리에서 환자 정보가 들어있는 노드를 탐색한다. 만약 findPatient()가 nullptr 을 리턴하면 환자 정보가 존재하지 않는 것이므로 'Not found'를 출력한다. findPatient 가 환자 정보가 들어있는 노드를 리턴한 경우, 해당 노드 환자 정보 구조체의 records 변수에 추가하고자 하는 진료 정보가 담긴 구조체 주소를 저장한다. 그리고 추가된 진료 구조체의 next 변수에는 이전 진료 기록 record 구조체의 주소를 넣어 Linked List 구조를 만든다.

추가 진료 기능의 시간 복잡도는 O(log(N))이다. findPatient() 함수의 시간 복잡도는 O(log(N)) 이고, 진료 정보를 추가하는 데에는 상수 시간이 걸리기 때문이다.

4) 유행병 조사 기능

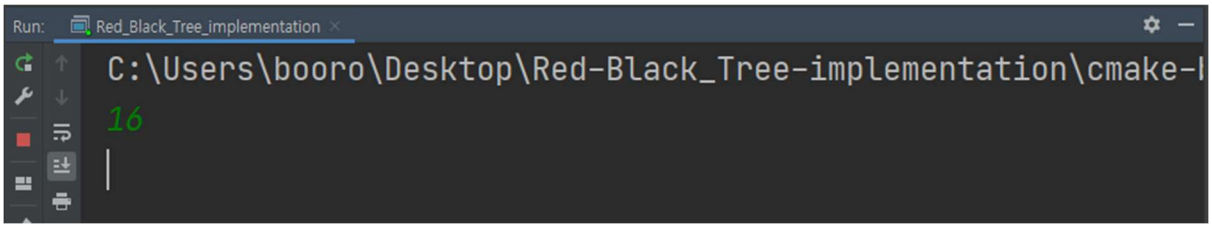
유행병 조사 기능은 조사하고자 하는 병명을 입력 받은 뒤에 'findDisease()' 함수에 루트 노드 주소와 병명을 인자로 넘긴다. findDisease() 함수는 재귀적으로 dept first traversal 을 하며 트리의 모든 노드들을 탐색한다. 그리고 각 노드에 저장된 환자 정보에서 마지막 진료 기록의 병명이 조사하고자 하는 병명과 동일하면 집계에 추가한다.

유행병 조사 기능의 시간 복잡도는 O(N)이다. 트리에 존재하는 모든 노드들을 탐색하여 유행병을 집계해야 하기 때문이다.

인터페이스 및 사용법

프로그램 인터페이스의 사용법과 실행 화면은 아래와 같다. 아래 예시에서 녹색 글은 사용자가 입력한 내용이고, 흰색 글은 프로그램이 콘솔에 출력한 내용이다.

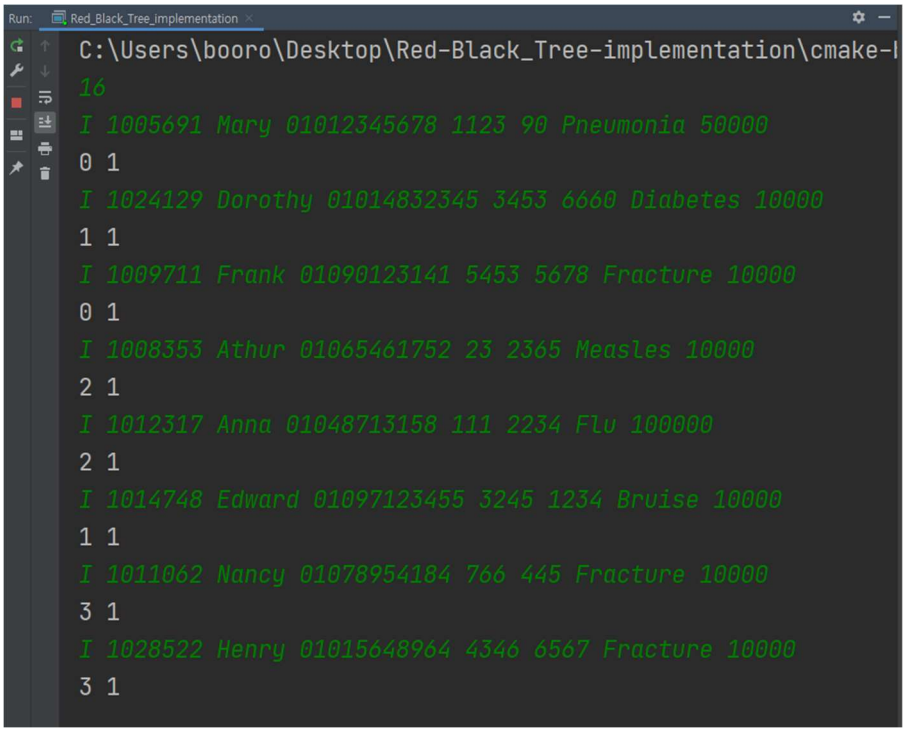
프로그램이 실행되면 먼저 다음과 같이 질의의 개수를 입력한다.



질의 개수를 입력한 뒤에는 원하는 기능의 질의를 조건에 맞게 입력한다.

1) 신규 가입 기능

신규 가입 기능을 원할 경우 다음과 같이 문자 'I', 환자 번호, 환자 이름, 연락처, 주소 x좌표, 주소 y좌표, 병명, 진료비 순서대로 입력한다. 각 정보는 띄어쓰기로 구분한다. 정보를 입력한 뒤에는 결과가 출력된다. 환자가 신규 가입에 성공할 경우 입력된 노드의 깊이와 '1'이 출력되고, 이미 환자 번호가 존재하여 신규 가입이 실패한 경우, 해당 환자 번호가 존재하는 노드 깊이와 '0'이 출력된다.



2) 환자 검색 기능

환자 검색 기능을 사용할 경우 문자 'F'와 환자 번호를 띄어쓰기로 구분하여 입력한다. 해당 환자가 존재할 경우, 트리에서 환자 정보가 저장된 노드의 깊이와 환자 이름, 연락처, 주소 x좌표, 주소 y좌표 순서대로 띄어쓰기로 구분하여 출력한다. 만약 존재하지 않을 경우 'Not found'를 출력한다.

```
F 1005691
1 Mary 1012345678 1123 90
F 1003200
Not found
```

3) 추가 진료 기능

추가 진료 기능은 문자 'A', 환자 번호, 병명, 진료비 순서에 띄어쓰기로 구분하여 입력한다. 트리에서 환자 정보를 찾아 진료 기록을 추가한 경우 해당 노드의 깊이를 출력하고, 트리에 환자 정보가 존재하지 않아 추가에 실패한 경우 'Not found'를 출력한다.

```
A 1008000 Pneumonia 30000
Not found
A 1011062 Pneumonia 30000
3
```

4) 유행병 조사 기능

유행병 조사 기능을 실행하기 위해서는 문자 'E', 병명 순서대로 띄어쓰기 구분하여 입력한다. 그 다음 프로그램은 트리에 저장된 모든 환자들에 대해서 마지막으로 진단 받은 병명과 입력된 병명이 같으면 집계하고, 집계된 수를 출력한다. 다만 편의상, 골절 같은 전염성이 없는 병명도 유행병으로 간주한다.

```
E Fracture
2
E Headache
0
```

평가 및 개선 방향

구현한 알고리즘의 장점

구현한 알고리즘을 이용하면 환자 정보들을 Balanced Binary Search Tree 구조로 관리한다. 따라서 환자 정보를 검색하는데 걸리는 시간은 O(log(N)) time으로, Linked List의 탐색 시간인 O(N) 보다 더 빠른 성능을 보여준다. 또한 새로운 환자 정보를 추가할 때 걸리는 시간은 O(log(N))이므로 sorted sequence 구조의 삽입 소요 시간인 O(N)보다 빠른 성능을 보여준다.

구현한 알고리즘의 단점

유행병 조사 기능의 시간 복잡도는 트리의 모든 노드를 방문하므로 O(N)이다. 다른 기능들이 O(log(N))의 시간 복잡도를 가지는 것을 고려하였을 때 상대적으로 오래 걸린다고 할 수 있다.

향후 개선 방향

진료 병들을 따로 관리하는 binary search tree 를 만들면 유행병 조사 기능의 속도를 개선할 수 있다. 여기서 트리의 각 노드들은 진료 병명의 id, 이름, 진료 횟수를 저장한다. 이러한 경우 유행병 조사 기능을 수행할 때 O(log(N))의 시간 복잡도를 가진다. 다만 이 경우, 질병을 관리하는 트리의 메모리가 추가로 필요하므로 공간 복잡도가 늘어나게 된다는 단점이 있다.