**REPORT**



|  |  |
| --- | --- |
| **과목명** | 알고리즘 |
| **과제명** | 환자 관리 프로그램 |
| **학번** | 12161658 |
| **이름** | 조재민 |
| **이메일** | jm\_company@naver.com |
| **제출일자** | 2021/05/19 |

목차:

1. 개요
2. 필요한 자료구조 및 기능
3. 기능별 알고리즘 명세
4. 인터페이스 및 사용법
5. 평가 및 개선 방향
6. 설계 후기
7. 개요

이번 알고리즘 1차 과제는 레드블랙트리 (red-black-tree)를 구현하여 환자들의 정보를 효율적으로 관리하는 프로그램 설계하는 것입니다. 효율적인 자료 관리를 위해 binary search tree를 이용하는데, 이때 binary search tree가 편향되면 검색이 최악의 경우 에 이르게 됩니다. 이를 해결하기 위한 방법으로 레드블랙트리를 사용하며, 이를 이번 과제에서 구현하였습니다.

여러 가지 질의 (Query)에 맞춰 Data (환자정보와 진료기록 등)을 레드블랙트리 구조에 맞게 설계하였습니다. 레드블랙트리의 Insert 과정을 이용하여 환자에게 부여된 고유 번호 (기준 키, 유일함)을 기준으로 tree를 구성하였으며, 레드블랙트리의 4가지 Property를 위반하지 않도록 Fix Tree (Restructuring & Recoloring)를 구성하였습니다. 질의는 I (= 신규 환자 등록), F (= 환자 검색), A (= 추가 진료, 즉 수정), E (= 병명 검색)이 있습니다.

개발 언어는 C++ 을 이용 하였으며, Mac OS를 사용하며 IDE는 JetBrain 사의 Clion을 이용했습니다.

1. 필요한 자료구조 및 기능

C++에서 입출력을 위한 iostream 헤더 파일과, 문자열 처리를 위한 string, 그리고 환자 진료기록 데이터 저장에 vector를 이용하였습니다.

입출력 (입력과 결과)를 처리하는 main 함수와 Patient (환자) 클래스, 레드블랙트리의 각 노드인 Node 클래스, 그리고 Red Black Tree 클래스로 구성되어있습니다.  
텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

Patient Class (좌)와 Tree의 노드 (우) Class

환자는 각 고유한 정보인 code를 갖고 있으며, String 형식으로 name과 연락처 contact를 갖고 있습니다. 주소는 X, Y 좌표를 pair 형식으로 갖고 있으며 진료 기록은 vector에 역시 pair 형식으로 <병명, 진료비>를 갖고 있습니다.

Node class는 트리의 각 노드를 의미하며 이는 환자 객체를 갖고 있으면서 해당 노드가 검정색 노드인지 여부와 부모, 왼쪽, 그리고 오른쪽 자식 노드를 포인터로 갖고 있습니다.

또한 레드블랙트리는 루트 노드의 포인터와, 질의 중 최근 진단 횟수를 검색할 때 쓰이는 diagnose\_count (int) 변수를 갖고 있습니다.

1. 기능별 알고리즘 명세

텍스트이(가) 표시된 사진

자동 생성된 설명

첫 번째로, 질의 입력과 결과 출력을 위한 main 함수입니다. Test case의 개수를 입력받아 while 문으로 test case의 개수만큼 질의를 반복합니다. If 문으로 질의 I, F, A, E에 맞게 나누어 주었으며, I의 경우 선언한 레드블랙트리 (rbt)에 새로운 환자를 삽입하고 결과 (성공 시 깊이와 함께 0, 실패 시 존재하는 노드의 깊이와 1)을 반환합니다. F 질의는 환자의 코드를 이용하여 레드블랙트리 내부에 존재하는 환자 정보를 가져오는 것입니다. Red Black Tree class 내부에 구현된 findPatient 함수를 사용하여 patient code를 넘겨 환자 정보를 가져옵니다. A 질의는 트리 내에 존재하는 환자를 찾아 해당 환자의 최근 진찰 기록에 새로운 정보를 입력하는 과정입니다. addTreatment 함수를 레드블랙트리 내에 구현했으며, 해당 함수는 환자 검색 (findPatient) 함수를 사용하여 존재하는 환자에 한에서 새로운 진찰 기록을 입력합니다. 마지막으로 E는 입력한 병명을 진찰한 환자의 수를 출력하는 것입니다. 역시 레드블랙트리 내에 구현한 countDI 함수를 호출하여 처리하였습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Red Black Tree 클래스 내부에 새로운 노드 (환자)를 입력하는 insertNode 함수 입니다. 새로운 환자의 입력에 호출되는 함수이기 때문에 입력되는 parameter는 환자 코드, 환자 이름, 주소, x, y좌표, 병명 그리고 진찰비를 입력 받습니다. 이 함수의 동작은 크게 2가지로 나뉠 수 있습니다.

* 1. 새로운 환자 입력 (트리안의 위치 검색, ) (이미 존재하는 환자가 발견되면 그 즉시 해당 노드의 깊이와 입력 거절 코드를 반환).
  2. 트리를 재 구성 (fixTree)함수를 호출

루트부터 patient code의 값에 따라 적절한 위치까지 의 시간 복잡도를 이용하여 내려갑니다. 적절한 위치를 찾았다면 그 부모 노드와 이어주며 트리 안에 새로운 노드 입력이 완료됩니다. 이때 새로운 입력이 트리 재구성이 필요없는 경우(첫 입력이거나 부모가 루트였을 경우)를 제외하고는 트리 재구성 함수를 호출하여 레드블랙트리 4원칙을 만족하도록 합니다. 그 후 새로 삽입된 노드의 깊이와 정상입력처리 코드인 1을 반환합니다.

이 메소드는 점근적으로 의 시간복잡도를 갖습니다. 새로운 노드의 정보를 입력하는 시간과 새로운 노드가 적절한 위치를 찾기 위해 루트부터 리프노드까지 의 시간복잡도를 활용합니다. 그 후 fixTree를 호출하는데 이 함수는 의 시간복잡도를 갖습니다. 종합적으로 제가 설계한 레드블랙트리에 삽입하는 과정은 의 시간 복잡도를 갖습니다.

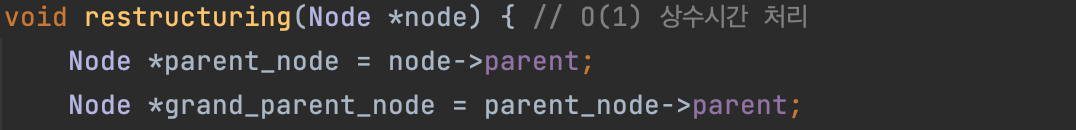
텍스트이(가) 표시된 사진

자동 생성된 설명

InsertNode 함수에서 레드블랙트리 4가지 property를 만족하도록 트리를 변형시키기 위해 호출되는 함수 입니다. 새롭게 삽입된 노드를 입력받아, 해당 노드부터 최악의 경우 root 노드까지 Recoloring & Restructuring을 진행합니다.

레드블랙트리의 경우 삼촌노드 (uncle\_node)의 색에 따라 recoloring (삼촌노드가 빨간색) 또는 restructuring (삼촌노드가 검정색)을 진행합니다. Recoloring의 경우 현재 새로 삽입된 노드 (node)와 부모노드, 그리고 삼촌노드의 색을 바꿔줍니다 , 그 후 현재 위치의 노드가 조상 노드로 올라간 뒤 계속해서 트리에 변화가 있어야 하는지 while문을 통해 검색합니다. 이때의 조건은 해당 노드가 루트가 아니면서 자신의 부모노드가 검정색이 아니라면 계속해서 트리 수정 과정을 수행하도록 합니다. 만약 삼촌노드가 검정색 노드라면 restructuring 함수를 호출하여 트리를 수정하고, root를 검정색으로 바꿔주고 종료합니다 (이는 예외처리입니다. 루트가 빨간색으로 Restructuring 과정이 된 다음 끝날 수 있기때문입니다).

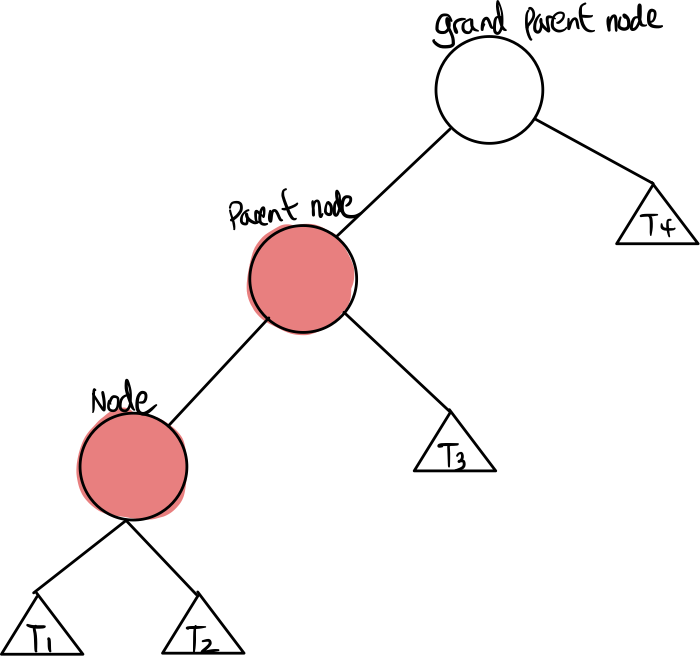
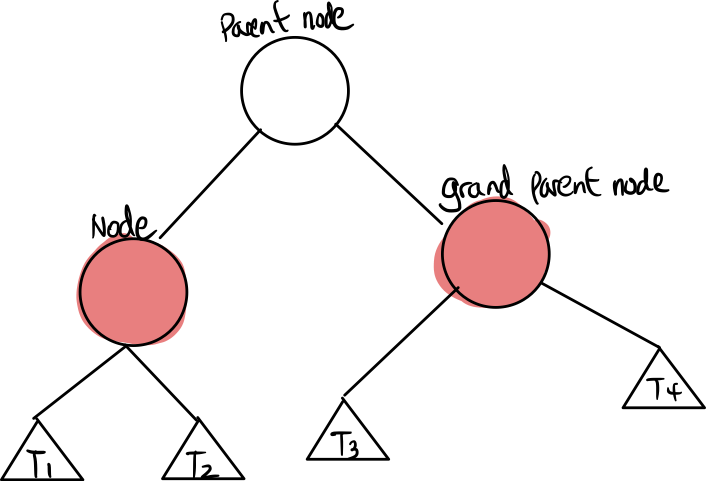
점근적으로 fixTree 메소드는 최악의 경우 새로 삽입된 node가 리프노드에서부터 루트노드까지 올라갈 수 있습니다. 즉 tree의 높이만큼의 cost가 수행되는데, 이는 의 시간이 소요됩니다.



트리의 수정 사항 중 restructuring 과정에 사용되는 메소드 입니다. 이 메소드에서는 입력값으로 들어온 새로 입력된 노드 (node)와 그 노드의 부모노드, 그리고 조상 노드를 주로 이용합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

첫 번째 수정가능성 입니다. Restructuring은 4가지 가능성으로 나누었습니다. 첫 번째는 내 부모 노드가 조상 노드의 왼쪽 자식이며, 내가 부모 노드의 왼쪽 자식인 경우입니다. 이 경우 나의 부모 노드가 가장 위로 올라가고 부모의 오른쪽 자식이 조상노드, 조상노드의 왼쪽자식이 부모의 오른쪽 자식이 됩니다.  🡪 

이론에 맞도록 하나하나의 노드 관계를 순서에 맞게 새롭게 연결해 주며, 마지막으로 색칠 해주며 종료됩니다. 이때 중요한 점은, 레드블랙트리에서 리프노드를 저는 NULL로 갖고 있기 때문에 연결과정에서 NULL을 참조하지 않도록 주의해야 했습니다. 이 부분은 if 문으로 존재할 경우 새로운 연결을 이어주고 존재하지 않으면 넘어가도록 설계하였습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

두 번째 부모 노드가 조상 노드의 왼쪽 자식이며, 내가 부모 노드의 오른쪽 자식일 경우의 restructuring 과정 입니다. 이 경우 본인 노드 (node)가 조상 노드 (grand\_parent\_node), 부모 노드 (parent\_node)의 부모 노드로 올라가며, 본인 노드의 왼쪽 자식이 부모 노드의 오른쪽 자식, 오른쪽 자식이 조상 노드의 왼쪽 자식이 됩니다.

벡터그래픽이(가) 표시된 사진

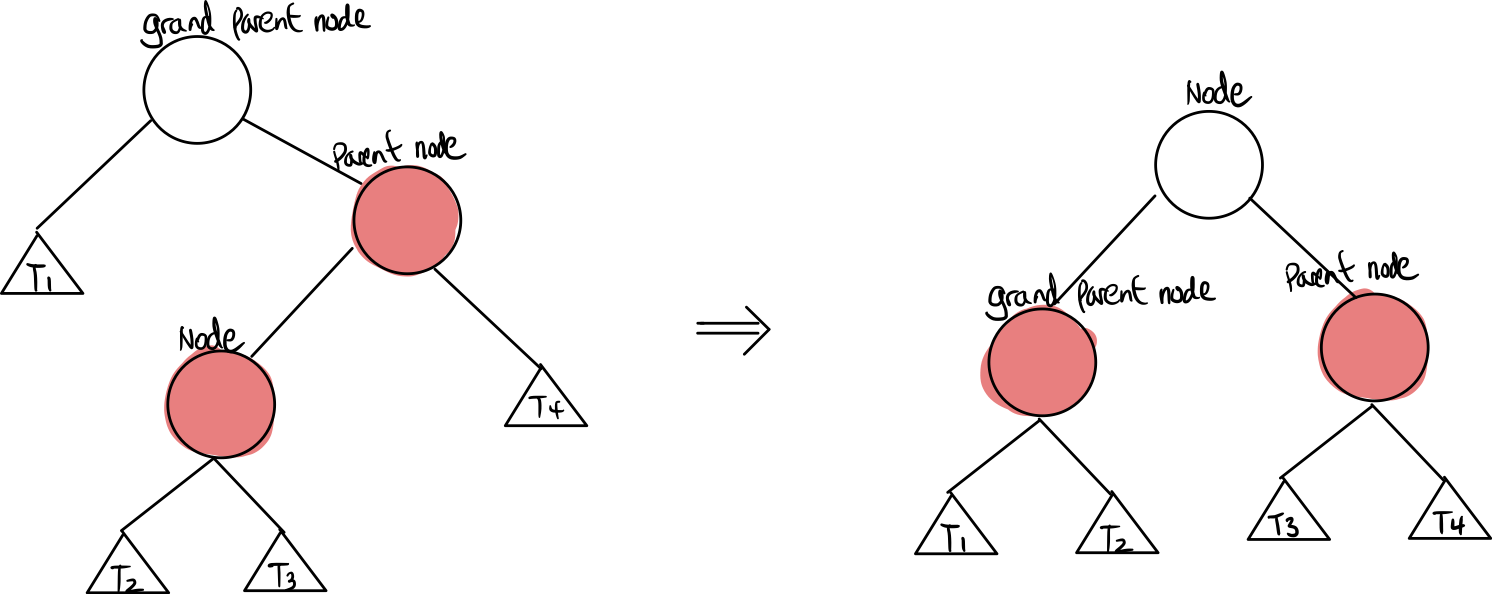
자동 생성된 설명

이때 역시 존재하지 않는 노드 (NULL)을 참조하여 실행 오류가 나지 않도록 여러 if문 처리를 해 주었습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

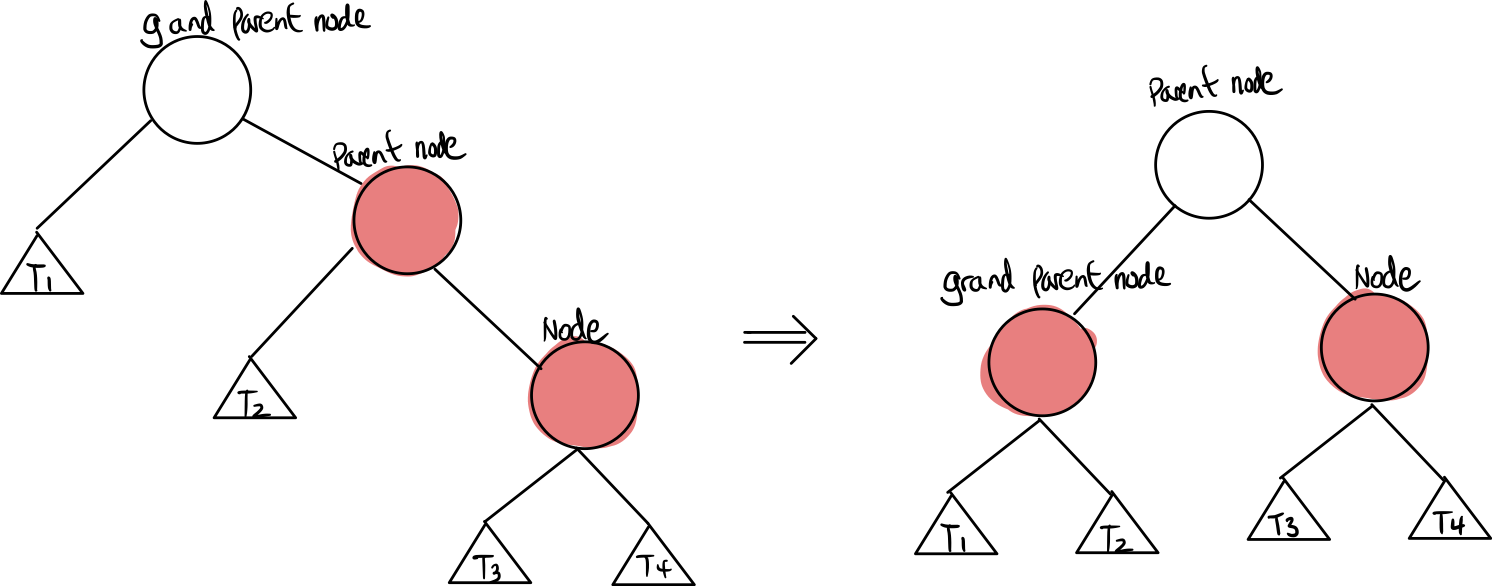
세 번째는 부모 노드가 조상의 오른쪽 노드이며, 내가 부모 노드의 왼쪽 자식일 경우의 Restructuring 과정 입니다. 이 경우는 두 번째 수정 과정이던 조상의 왼쪽 자식이 부모자식, 그리고 부모의 오른쪽 자식이 본인 노드인 경우와 같은 로직입니다.



텍스트이(가) 표시된 사진

자동 생성된 설명

마지막으로, 부모 노드 (parent\_node)가 조상 노드 (grand\_parent\_node)의 오른쪽 자식이며, 본인 노드 (node)가 부모 노드의 오른쪽 자식인 경우입니다. 이 경우에는 부모 노드의 왼쪽 자식이 조상 노드로, 오른쪽 자식이 본인 노드가 되며, 부모 노드의 왼쪽 자식은 조상 노드의 오른쪽 자식으로 이동됩니다.



조상 노드 (grand\_parent\_node)와 부모 노드 (parent\_node), 그리고 본인 노드 (node)의 위치에 따른 4가지 다른 방법의 restructuring 방식으로 나누어서 구현 하였습니다. 굉장히 복잡한 코드 이지만 점근적으로 이 과정은 모두 시간에 수행이 가능합니다. 노드간 연결을 새롭게 해주고 각 노드의 색칠을 다시해주는 작업이 주된 목적입니다.

*텍스트이(가) 표시된 사진

자동 생성된 설명*

트리 내부에서 환자 코드에 따른 올바른 환자 노드를 찾는 findPatient 함수와 질의 ‘A’, 기존 환자에 추가 진료를 삽입하는 addTreatment 함수입니다.

findPatient 함수는 트리의 root 부터 patient의 code 값을 비교해가며 올바른 노드를 찾고, 그 노드의 depth를 계산해서 Pair 형식으로 값을 리턴합니다. Leaf 까지 도달 하였지만 올바른 노드를 찾지 못한경우 -1과 null을 리턴합니다. 최악의 경우는 트리의 가장 깊은 리프까지 찾아가는 것 이기 때문에 Tree height 만큼의 cost가 소요됩니다. 즉 점근적으로 이 소요됩니다.

addTreatment 함수는 올바른 환자 노드를 찾은 뒤 해당 노드의 질병 기록인 treatRecord 벡터에 추가적인 값을 입력하는 것 입니다. findPatient 함수를 호출하여 올바른 환자 노드를 가져온 뒤, 해당 환자 노드가 null이 아니라면 추가해 줍니다. 이 역시 의 시간이 소요됩니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

노드의 깊이를 찾는 함수입니다. 루트노드부터 최악의 경우 가장 깊은 리프 노드까지 내려가며 위치를 찾습니다. 해당 노드의 위치를 찾으면 깊이를 리턴하고, 만일 해당 노드가 존재하지 않으면 -1을 리턴합니다. 이 함수는 Insert Node시에 fixTree로 인해 새로 입력된 노드의 위치가 계속해서 바뀔 수 있기에 마지막 노드의 깊이를 출력할때 사용됩니다. 최악의 경우 트리의 높이만큼의 탐색이 필요하기 때문에 점근적으로 의 시간이 소요됩니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

마지막으로 ‘E’의 질의에 사용되는 countDI와 countByPreorder 메소드입니다. 이는 주어진 질병명을 가장 마지막으로 진료받은 환자들의 수를 리턴하는 용도입니다. countDI 함수에 해당 질병명을 입력하여 호출하면, 트리안에 선언되어있는 diagnose\_count 변수를 이용하여 개수를 리턴합니다. 값은 트리를 전위순회 하며 각 노드의 마지막 진료 기록이 입력된 병명과 같다면 diagnose\_count 값을 증가시킵니다. 메인 함수에서는 Tree의 private 변수인 root를 참조할 수 없었기에 두 가지 함수로 나누어서 트리 내부에서 전위 순회 함수를 호출하도록 설계하였습니다.

트리를 전위순회하여 모든 노드에 대한 마지막 진료 기록을 검사하기 때문에 이진 트리의 전위 순회 시간복잡도와 동일한 의 시간 복잡도를 보여줍니다.

레드블랙트리를 구성하며 새로운 노드의 삽입은 올바른 위치를 찾는데 , 트리 수정에 그리고 마지막 출력 값인 노드의 깊이를 찾는데 의 시간 종합적으로 의 시간을 소요합니다. 또한 환자 검색 (질의 ‘F’)와 진료 추가 (질의 ‘A’)는 의 시간에 가능하며, 마지막으로 질의 ‘E’인 병명에 따른 최근 진료 환자 수 검색이 가장 느린 의 시간복잡도를 갖고 있도록 설계하였습니다. 이론적으로 구현된 레드블랙트리의 가장 Optimal한 시간복잡도와 같은 복잡도로 구현을 성공 하였으며, 질의 E의 경우 모든 노드를 탐색해야 하기 때문에 트리 순회를 하여 의 시간 복잡도보다 빠른 방법을 찾지 못하였습니다.

4. 인터페이스 및 사용법

콘솔을 이용하여 입력과 결과를 출력합니다. 텍스트이(가) 표시된 사진

자동 생성된 설명

첫 번째 입력으로 Test Case의 개수를 입력합니다. 그다음 원하는 질의 ‘I’, ‘F’, ‘A’, ‘E’와 함께 해당 질의에 맞는 내용을 입력해줍니다. 각 질의에 맞는 결과가 바로 출력됩니다.

예제로 주어진 테스트 케이스에 대한 결과 출력 창입니다. 한 번에 입력을 모두 완료하였기에, 각 줄에 맞는 결과가 이후 순차적으로 표시됩니다. 주어진 정답 출력과 일치하는 것을 보여줍니다.텍스트이(가) 표시된 사진

자동 생성된 설명

예제로 주어진 테스트 케이스에 대한 결과 출력 창 입니다. 한 번에 입력을 보두 완료 하였기에, 각 줄에 맞는 결과가 이후 순차적으로 표시됩니다. 주어진 정답 출력과 일치하는것을 보여줍니다.

1. 평가 및 개선 방향
2. 구현한 알고리즘의 장점
   1. 비교적 빠른 수행시간 (이론적으로 Optimal)으로 구현하는데 성공 하였습니다. 새로운 환자 정보 등록 (insert)이 시간에 처리가 가능하며, 모든 질의를 가장 Optimal한 시간복잡도로 구현하였습니다. 종합적인 과제 문제의 시간 복잡도는 Test Case (T)까지 고려하여 시간에 처리가 가능합니다.
   2. 코드가 최대한 직관적으로 작성하였습니다. 노드가 환자가 되는게 아닌, 환자 객체를 만들 수 있도록 환자 클래스를 별도로 생성하여 환자의 정보는 환자 객체가 고유하게 갖고 있을 수 있도록 하였습니다. 또한 트리에 새로운 노드가 삽입되며 발생하는 레드블랙트리의 4가지 특성을 유지하기위한 fixTree 과정 역시 최대한 직관적이고, 이론에서 벗어나지 않는 방식으로 구현되었습니다.
   3. 예외 처리가 모두 잘 짜여져 있습니다. C++로 구현한 알고리즘이기에 많은 포인터 참조를 하면서 잘 못 사용될 경우 프로그램 전체가 다운될 수 있습니다. 제 프로그램은 예상 가능한 nullptr에 대한 예외처리를 모두 해 주었습니다.
3. 구현한 알고리즘의 단점
   1. 노드의 깊이를 계산하기 위해, 노드 객체가 각각의 Depth 변수를 갖고있고, 트리가 수정됨에 따라 새롭게 자신의 Depth를 가지며 깊이 결과를 이 아닌 에 가능하도록 설계하고 싶었습니다. 하지만 특정 트리가 수정됨에 따라 자식 노드 모두의 깊이가 달리지는 걸 설계하기 어려웠습니다.
   2. Red-black Tree의 External Property는 모든 Leaf node는 Black node 이어야 한다 입니다. 이는 리프노트에 추가적으로 자식 (검정색 비어있는 노드)이 있다는 가정을 전제로 하지만, 제가 구현한 레드블랙트리는 null로 갖고있습니다. 그렇기 때문에 fix tree 과정에서 null 여부를 확인하고 수행 과정에서 잘못된 null 참조를 피하기 위한 과정이 매우 길게 있습니다.
4. 향후 개선 방향
   1. 환자의 진료 기록을 vector<>에 담아서 저장합니다. 즉 제가 만든 환자 기록 프로그램은 해당 환자의 모든 진료기록을 갖고 있습니다. 만약 최근 기록만을 필요로 하는 질의 (E)만이 요구된다면 벡터 형식으로 저장하여 메모리 낭비를 하지 않고, <병명, 가격> 형식의 pair 변수에 값을 바꿔주는 것이 효율적일 것입니다.
   2. Depth를 시간에 구하는 방법을 찾아 수정한다면 보다 효율적일 것 입니다. 세그먼트 트리의 Lazy Propagation (느린 갱신) 방식을 적용하기 위해 노력하였으나 실패하였습니다. 트리 안의 노드가 각자의 깊이를 안다면 이 아닌 시간에 깊이 리턴이 가능합니다.
5. 설계 후기

이번 ‘환자 관리 프로그램 설계’ 과제를 진행하며, 이론적으로만 배운 레드블랙 트리를 직접 구현하여 더 자세히 배울 수 있었습니다. 또한, 알고리즘을 좋아하고 이러한 구현에 평소에 자신 있었기에 많은 수행 착오를 거치는 과정이 오히려 즐거웠습니다. 스스로 클래스를 구현하고 포인터를 참조해가며 설계해본 경험이 많지 않았는데 이번 과제를 성공적으로 설계하며 더욱 자신감이 생겼습니다.

이론에서 배운 레드블랙트리의 구성과 규칙을 제대로 이해하였기에 스스로 그림을 그리고 여러 테스트케이스를 만들어 보며 ‘이론대로’ 구현을 해보니 성공적으로 구현할 수 있었습니다. 최대한 효율적인 코딩을 하기 위해 노력하였고, 이론적으로 더 빠른 알고리즘을 생각해 볼 좋은 기회였습니다.