**REPORT**



|  |  |
| --- | --- |
| **과목명** | 알고리즘(001) |
| **과제명** | 알고리즘 설계과제 1 |
| **학과** | 컴퓨터공학과 |
| **학번** | 12171645 |
| **이름** | 심규범 |
| **이메일** | koreanshim@naver.com |
| **제출일자** | 2021.11.14 |

1. **개요**

* **설계 목적**: 앱스토어에 앱들을 쉽게 등록하고, 검색, 업데이트, 가격변화를 주는 등 어플리케이션들을 효율적으로 관리하는 프로그램을 red-black tree를 이용하여 설계하는 것이 목적이다.
* 애플리케이션 정보 형식

1. **ID**: 기준키, 유일성, 정수.
2. **이름**: 공백 없는 문자열.
3. **용량**: 정수.
4. **가격**: 정수.

* 요구사항

1. 애플리케이션 **등록**: 애플리케이션의 정보를 입력 받아 red-black tree에 노드를 삽입하고, 해당 노드의 깊이를 출력한다. 이미 존재한다면 해당 노드의 깊이를 출력만 한다.
2. 애플리케이션 **검색**: 입력된 애플리케이션의 id를 탐색하여 존재하면 해당 애플리케이션의 정보를 출력한다. 만약 검색한 id를 보유하는 애플리케이션이 존재하지 않을 시 “NULL’’을 출력한다.
3. 애플리케이션 **업데이트**: 입력된 애플리케이션의 id를 탐색하여 존재하면, 입력받은 정보들을 존재하는 애플리케이션의 정보들에 업데이트한 후, 해당 어플리케이션 노드의 깊이를 출력한다. 만약 검색한 id를 보유하는 애플리케이션이 존재하지 않으면 “NULL” 출력한다.
4. 애플리케이션 **할인**: 입력한 시작범위와 끝범위 내의 id를 가진 애플리케이션들을 모두 탐색하여 각 애플리케이션의 할인가격을 적용한다. 출력은 없다.

* 입출력 제한사항

1. 질의는 최대 100000개이다.
2. 전체 질의에 대해 제한시간 2초 이내에 수행해야한다.
3. 제시한 입출력 형식대로 표준입출력을 사용하여 처리한다.
4. 문제에서 설명되지 않은 예외 처리할 질의는 입력되지 않는다.

* 개발환경: C, C++ 언어 사용 가능. 본인은 **Visual Studio 2019, C++**로 개발하였음.

1. **필요한 자료구조 및 기능**

* **Linked list**: 데이터들을 담고 있는 노드들을 연결시키는 자료구조이다. Red-black tree에 필요한 red와 black 유무와 애플리케이션의 정보들을 담기 위해 필요하다. 또한 Binary search tree에서 부모와 자식 관계를 설정할 때 사용된다. 마지막으로 자료를 자주 입력하고 삭제할 때 유용하다.
* **Binary Search Tree**: binary search와 linked list를 결합한 자료구조이다. 이진탐색의 효율적인 탐색 능력을 유지하면서 잦은 자료 입력과 삭제에 유용하다. 왼쪽 노드는 해당 노드보다 작은 값(id), 오른쪽 노드는 해당 노드보다 큰 값(id)를 가져야한다.

1. **기능별 알고리즘 명세**

* **void inorderPrint (Node\*)**: 중위순회로 노드를 출력하는 메소드. 노드들이 제대로 입력되는지 확인하는 용도로 사용한다. 중우순회의 복잡도는 O(n), 최악의 경우 모든 노드 n개를 다 탐색해야 하기 때문이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **int getDepth (int)**: 입력받은 id의 노드의 높이를 구하기 위한 메소드. Root부터 출발한 노드가 자신과 입력받은 id의 대소에 따라 이동해 내려가면서 depth counter을 증가시켜가는 방식이다. 노드의 id와 입력 받은 id 값이 같아지면 depth를 반환한다. Red-black tree는 균형 잡힌 트리이며, 한번 실행마다 입력받은 id와 비교하기에 시간 복잡도는 O(log n)이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **Node\* findNode (int)**: 입력받은 id와 동일한 노드가 이미 존재하는지 안하는지 찾고, 만약 있으면 해당 노드 반환, 없으면 NULL반환하는 메소드이다. Root부터 출발한 노드가 자신과 입력받은 id의 대소에 따라 이동해 내려가면서 노드의 id와 입력받은 id가 같으면 해당 노드를 반환하는 방식이다. Red-black tree는 균형 잡힌 트리이며, 한번 실행마다 입력받은 id와 비교하기에 시간 복잡도는 O(log n)이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **bool getUncleColor (Node\*)**: 해당 노드의 삼촌 노드의 색깔을 반환하는 메소드이다. 해당노드의 부모노드의 부모노드인 조부모노드와 부모노드의 상관관계를 이용하여 삼촌노드가 조부모노드 기준 어느 쪽에 있는지 판별하고, 삼촌노드의 색깔(red or black)을 반환하는 메소드이다. 조부모노드가 루트노드인 경우가 최악의 수행시간을 가지므로 시간복잡도는 O(log n)이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **void doubleRed (Node\*)**: 입력한 노드가 double red를 발생시키는지 확인하고, 조건에 따라 restructuring과 recoloring을 진행하는 메소드이다.

입력된 노드가 루트노드일 때, 입력된 노드가 black일 때(루트일 경우 밖에 없음), 입력된 노드의 부모노드가 black 일 때는 double red가 발생할 수 없으므로 해당 메소드에서 빠져나온다.

getUncleColor 메소드를 이용하여 입력된 노드의 삼촌노드의 색깔을 판별한 후, 삼촌노드 색깔이 black일 경우 resturcturing을 진행한 후, doubleRed 메소드에서 빠져나온다. 하지만 삼촌노드의 색깔이 red 일 경우 recoloring을 진행한 후, recoloring이 진행한 후 recolor된 노드의 조부모 노드를 다시 doubleRed 메소드에 입력하여 다른 double red가 발생했는지 재귀적으로 확인한다. 매 입력마다 발생하므로 O(1)의 복잡도를 가진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **void reStructure (Node\*)**: 4가지의 경우에 맞게 restructuring을 하는 메소드이다. 해당 노드의 부모노드, 조부모노드의 연결관계에 따라 서로의 자식노드와 부모노드를 새로 설정하여 restructuring을 한 뒤, 새로운 조부모노드와 기존의 증조부모 노드를 적절히 부모-자식 관계를 맺어준다. **Ex)** 1번 케이스(깊이 0-2-1): 조부모노드의 오른 자식을 본인노드의 왼 자식으로 설정. 만약 본인 노드의 왼 자식이 존재한다면 왼자식의 부모는 조부모노드로 설정. 부모노드의 왼자식을 본인노드의 오른자식으로 설정. 만약 본인노드의 오른자식이 존재한다면 오른자식의 부모는 부모노드로 설정. 본인노드의 왼자식을 조부모노드로, 오른자식을 부모노드로 설정하고, 조부모노드의 부모를 본인노드로, 부모노드의 부모를 본인노드로 설정하면 restructering이 진행되었다. 만약 증조부노드가 존재한다면, 증조부노드와 본인노드의 id값을 비교하여 증조부노드의 자식을 본인노드, 본인노드의 부모를 증조부노드로 설정한다. 증조부 노드가 존재하지 않는다면 본인노드를 루트노드로 설정하고, 본인노드의 부모를 nullptr로 처리하여 부모를 없앤다. 마지막으로 위치에 맞게 red/black을 재설정한다. O(log n)의 복잡도를 가진다.

2번 케이스: (깊이 1-2-0), 3번 케이스: (깊이 2-1-0), 4번 케이스: (깊이 0-1-2)

텍스트이(가) 표시된 사진

자동 생성된 설명

* **Node\* reColor (Node\*)**: 해당노드의 조부모노드를 red, 조부모의 왼쪽과 오른쪽 자식노드들을 black으로 설정한다. 만약 조부모노드가 루트노드라면 조부모노드를 black으로 설정한다. 그리고 recoloring 때문에 추가적으로 발생할 수도 있는 double red를 검사하기 위해서 조부모노드를 반환한다. 조부모노드와 그 두 자식노드에 대해서만 작동하므로 O(1)의 복잡도를 가진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **void inorderDiscount (Node\*)**: 중위순회 방식으로 해당노드의 가격(price)를 업데이트 한다. 재귀적으로, 왼쪽의 tree먼저 중위적으로 순회하고, disCount 메소드를 통해 해당 노드의 가격을 수정하고, 오른쪽의 tree를 중위적으로 순회한다. 중위순회 구조이므로 O(log n)의 복잡도를 가진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **void disCount (Node\*)**: 해당노드의 id값이 할인 대상의 범위 안에 들어간다면, 해당 노드의 가격 값을 수정한다. 해당노드의 가격만 수정하면 되므로 복잡도는 O(1)이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **void Insert (int, string, int, int)**: 입력 받은 정보들을 red인 새로운 노드에 입력하여서 조건에 맞게 linked list에 입력한다.

비어있는 linked list인 경우 루트노드에 새로운 노드를 입력한 후, 루트노드를 black으로 설정한다. 마지막으로 0 출력한 후 linked list의 크기를 1 증가시킨다. 비어있는 list가 아니고, 새로운 노드와 동일한 정보를 가진 노드가 이미 존재한다면 이미 존재하는 노드의 깊이를 출력한다. 중복되는 노드가 없다면 입력할 노드의 id와 트리 내부의 노드들의 id값을 비교해가면서 노드를 추가한다. 새로운 노드가 추가되면 새로운 노드에 대해doubleRed메소드를 통해서 double red가 발생하는지 검사한다. 그 후 해당 노드의 깊이를 출력한 후 linked list의 크기를 1 증가시킨다. Id값들의 대소를 비교하므로 트리의 높이보다 더 많은 시간을 필요로 하지 않기에 O(log n)의 복잡도를 가진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **void Search (int)**: findNode메소드에 입력받은 id를 입력해 해당 id를 가진 노드가 linked list에 존재하는 지 확인한다. 만약 존재하지 않으면(NULL) “NULL”을 출력하고, 이미 존재하는 이미 존재하는 노드라면 해당 노드의 id, 이름, 크기, 가격을 출력한다. 해당 노드의 존재 유무에 대한 메소드이기에 O(1)의 복잡도를 가진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **void Update (int, string, int, int)**: 입력받은 정보들 중 id를 가진 노드가 있는지 확인한다. 만약 없으면(NULL) “NULL”을 출력한다. 이미 존재하는 노드라면, 입력받은 정보들(수정될 이름, 크기, 가격)을 입력해서 해당 노드의 정보를 업데이트한다. 그 후 해당 노드의 깊이를 출력한다. 마찬가지로 O(1)의 복잡도를 가진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* **void Discount (int, int, int)**: 입력받은 할인할 구간 정보(시작 위치와 끝 위치)를 RedBlackTree 클래스의 멤버변수에 입력한다. 할인율에 대한 멤버변수인 discountRate는 입력받은 할인율에 따라 계산해서 double로 casting을 한 후에 입력한다. 그 후 inorderDiscount 멤버변수로 루트노드부터 중위순회로 구간내의 노드들의 가격을 할인(업데이트)한다. 입력 받은 정보들을 멤버변수에 넣는 작업이므로 O(1)의 시간복잡도를 가진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. **인터페이스 및 사용법**

* 텍스트이(가) 표시된 사진

  자동 생성된 설명텍스트이(가) 표시된 사진

  자동 생성된 설명인터페이스 및 실행화면
* 사용법: 입력할 질의의 횟수를 입력한 후, 그 횟수동안 질의 형식에 맞게 값들을 입력한다. 각 질의에 맞는 메소드들이 작동한다.

1. **평가 및 개선 방향**

* **구현한 알고리즘 장점**: 함수 및 메소드들을 사용하여 가독성이 좋으며, 메소드들을 여기저기 사용하기 편리하다. 또한 메소드들을 사용함으로 디버깅 할 때 어디서 문제가 생기는지 찾기 편하며, 문제 생긴 메소드만 수정하면 되므로 수정도 편리하다.

Binary Search Tree 구조이므로 정보의 저장과 노드의 입력, 삭제, 수정이 빠르고 편리하다

* **구현한 알고리즘 단점**: reStructure() 메소드의 케이스 분류에 따라 조건이 다르므로 그 부분의 가독성이 매우 떨어지며, 케이스마다 부모-자식 관계를 잘못 설정하는 실수를 범하기가 쉽다.
* 향후 개선 방향: reStructure()의 가독성을 높이고 잠재적인 실수를 방지하기 위해 해당 메소드의 알고리즘을 변경해보아야 겠다는 생각이 들었다.