Writeup

20140475 임범혁

I did the project 3 alone.
I'll use 3 tokens.

A. Description the algorithm of each operation.

A-1. **Binary Search Tree**

1. printall, printrecur : 트리가 짜여져 있는 구조를 보기 위해 printrecur에서 recursion을 이용해 inorder 순서로 모든 엔트리들을 방문하여 데이터들을 프린트하였다.

2. find, findRecursion : key값이 k인 엔트리의 개수를 알기 위해 findRecursion에서 recursion을 이용해 inorder 순서로 모든 엔트리들을 방문하여 key값이 k인 엔트리의 개수를 세주었다.

3. insert : key값이 k인 엔트리를 만들어 트리에 넣어줄텐데 우선 트리가 비어있다면 엔트리를 트리의 루트로 지정해준다. BST의 구조에 맞게 while문을 이용하여 알맞는 위치를 찾은 다음 엔트리를 생성하여 트리에 넣어준다.

4. remove : 우선 전체적으로 구조를, 삭제하고 싶은 key를 가진 엔트리가 루트인 경우와 루트가 아닌 경우, 2개로 나누었다. 엔트리가 루트인 경우에는 루트를 새로 지정해주어야 한다. 두 부분에서 세부적인 구조는 같다. 총 4가지의 경우로 나누어 주었는데 삭제하고 싶은 엔트리가 external인 경우, 왼쪽 자식이 없는 경우, 오른쪽 자식이 없는 경우와 양쪽 자식이 둘다 있는 경우이다. 마지막의 경우에 삭제할 엔트리의 위치를 오른쪽 서브트리에서 key값이 가장 작은 엔트리로 대체해주었다.

5. countRange, countRecursion : find 함수에서와 마찬가지로 countRecursion 함수에서 recursion을 이용해 inorder 순서로 모든 엔트리를 방문하여 key값이 k1, k2 사이에 있는 엔트리의 개수를 세주었다.

A-2. **Red-Black Tree**

1. insert : 우선 BST에서 정의해주었던 insert 함수를 이용하여 k의 key값을 가지는 엔트리를 삽입해주었고, RBT의 조건이 맞는지 확인해주었다. BST::insert를 이용해 넣었던 엔트리의 위치를 먼저 while문을 이용하여 얻었다. (코드와 맞게 R이라 칭하겠다.) 우선 R의 색이 RED인지 확인해준 후, 4가지의 경우로 나누었다. R이 루트인 경우, R의 부모 엔트리가 루트인 경우(이 경우 부모 엔트리의 색은 BLAK일 것이다.), 부모 엔트리의 색이 RED일 경우, 그 외의 경우로 나누었다. 부모 엔트리의 색이 RED인 경우, 삼촌 엔트리의 색에 따라 restructioning 혹은 recoloring 작업이 이루어지게 하였다.

2. restructioning : 크게 4가지로 분류하였는데, R이 부모 엔트리보다의 왼쪽 자식인지, 오른쪽 자식인지, 그리고 부모 엔트리가 조부모 엔트리의 왼쪽 자식인지, 오른쪽 자식인지를 구분해주었다. 각각의 경우 바뀌는 구조가 다르고, 세부적으로는 조부모 엔트리가 증조부모 엔트리의 왼쪽 자식인지, 오른쪽 자식인지 구분하였고, R이 왼쪽 혹은 오른쪽 자식을 가지는지 구분해주었다. 각각의 경우 바뀌는 구조가 다르다.

3. recoloring : 삼촌 엔트리의 색이 RED일 때, 삼촌 엔트리와 부모 엔트리의 색을 BLACK으로 수정하였고, 조부모 엔트리가 루트가 아닌 경우에만 색을 RED로 바꿔주었다.

4. remove : 시간이 부족해 마저 구현하지 못하였다. test를 위해 return 값은 true로 설정해두었다. 과제 제출하고, 혼자 마무리해볼 계획이다.

B. Explain how you tested the correctness of each operation and how you evaluated the performance over a series of operations.

우선 구글에서 BST, RBT 시뮬레이션 사이트를 검색해 트리를 시뮬레이션 해보며 내가 설계한 트리와 같은지 비교하였다. 비교를 위해 printall 함수를 만들어 루트인지 아닌지, 엔트리의 색, 부모 엔트리, 왼쪽 자식 엔트리, 오른쪽 자식 엔트리의 key 값을 출력하여 내가 만든 트리를 그려보며 비교하였다. 또한 tester_private.cpp 를 짜고, assert 함수를 이용하여 트리가 잘 형성되었는지 확인하였다.

C. Timing measurement results for various number of operations.

```
<<test 1>>
<<Preprocessing starts>>
Random seed: 0  # of integers: 1000
Generating 1000 integers...
<<Preprocessing ends>>

BST Insertion:  0.229 ms
BST Searching:  74.856 ms
BST Removal:    0.211 ms
RB Insertion:   0.459 ms
RB Searching:   19.758 ms
RB Removal:     0.003 ms
```

```
<<test 2>>
<<Preprocessing starts>>
Random seed: 0  # of integers: 1000
Generating 1000 integers...
<<Preprocessing ends>>

BST Insertion:  0.23 ms
BST Searching:  37.345 ms
BST Removal:    0.211 ms
RB Insertion:   0.458 ms
RB Searching:   18.901 ms
RB Removal:     0.003 ms
```

```
<<test 3>>
<<Preprocessing starts>>
Random seed: 0  # of integers: 1000
Generating 1000 integers...
<<Preprocessing ends>>

BST Insertion:  0.228 ms
BST Searching:  21.677 ms
BST Removal:    0.216 ms
RB Insertion:   0.453 ms
RB Searching:   19.832 ms
RB Removal:     0.003 ms
```

```
<<test 4>>
<<Preprocessing starts>>
Random seed: 0  # of integers: 1000
Generating 1000 integers...
<<Preprocessing ends>>

BST Insertion:  0.253 ms
BST Searching:  22.243 ms
BST Removal:    0.23 ms
RB Insertion:   0.48 ms
RB Searching:   19.581 ms
RB Removal:     0.004 ms
```

```
<<test 5>>
<<Preprocessing starts>>
Random seed: 0  # of integers: 1000
Generating 1000 integers...
<<Preprocessing ends>>

BST Insertion:  0.238 ms
BST Searching:  21.978 ms
BST Removal:    0.222 ms
RB Insertion:   0.454 ms
RB Searching:   16.366 ms
RB Removal:     0.003 ms
-----------------
```

RB Removal은 RBTree 의 remove 함수 출력을 항상 true 로 설정해놓았기 때문에 큰 의미가 없다.

D. Explain a scenario (a set of operations) where IntRBTree outperforms IntBST and explain why. if you see no performance difference, explain why there was no performance difference.

RB 는 엔트리를 삽입할 때 BST 와 다르게 restructioning 혹은 recoloring 작업을 해주어야 하기 때문에 RB Insertion 이 BST Insertion 보다 오래 걸리는 것을 확인할 수 있었다. 또한, RB 의 구조가 BST 의 구조보다 효율적이기 때문에 RB Searching 이 BST Searching 보다 적은 시간이 걸리는 것을 알 수 있었다. RB Tree 의 삭제 함수를 완성하지 못하여 비교는 못 하지만 RB 의 Removal 은 BST 보다 더 많은 작업이 필요하기 때문에 오래 걸릴 것이라 예상한다.