알고리즘

Red Black Tree

|  |  |
| --- | --- |
| **15조** | |
| 이름 | 학번 |
| **김지우** | **2016154010** |
| **남주연** | **2017156012** |
| **이길형** | **2017150048** |
| **제출 날짜: 2019.12.15** | |

목차

1. 레드-블랙 트리의 개념 3

1.1 레드-블랙 트리의 특성 3

1.2 레드-블랙 트리의 5가지 규칙 4

2. 레드-블랙 트리의 회전 연산 4

2.1 회전 연산 4

2.2 소스코드 분석 5

2.2.1 우회전 연산(right\_rotate) 5

2.2.2 좌회전 연산(left\_rotate) 5

3. 레드-블랙 트리의 삽입 연산 6

3.1 삽입 연산 6

3.2 소스코드 분석 7

3.2.1 노드 생성 함수(new\_node) 7

3.2.2 삽입 연산(insert\_node) 8

3.2.3 더블 레드 교정 함수(checkDoubleRed) 9

4. 레드-블랙 트리의 삭제 연산 10

4.1 삭제 연산 10

4.2 소스코드 분석 13

4.2.1 삭제 기능 함수(Delete node) 13

4.2.2 재정렬 함수(Fix Up) 15

5. 트리 구조 출력 18

5.1 소스코드 분석 18

6. 레드-블랙 정리 20

6.1 시간 복잡도 20

6.2 결론 21

# 레드-블랙 트리의 개념

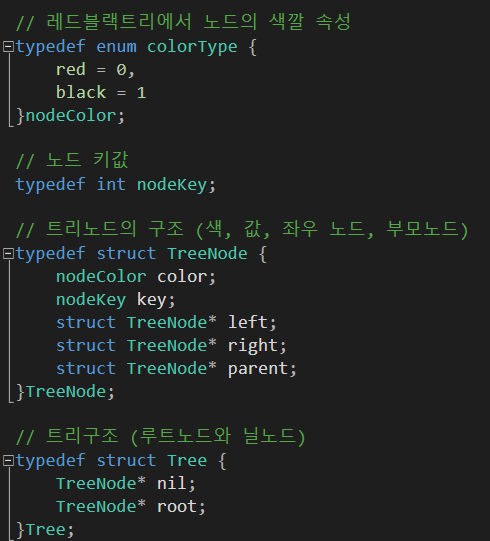
## 레드-블랙 트리의 특성

그리기이(가) 표시된 사진

자동 생성된 설명

[그림 1.1] 레드-블랙 트리의 구조

레드-블랙 트리는 이진 탐색 트리로서 기존의 이진 탐색의 노드 속성에 색깔이라는 속성을 추가한다. 각 노드는 왼쪽자식 노드의 주소, 오른쪽자식 노드의 주소, 부모 노드의 주소, 색깔 속성, 하나의 키 값 이렇게 총 5개의 정보를 가지고 있다. 자식 노드가 존재하지 않을 경우 NIL 노드라고 부르는 특수한 노드가 있다고 가정하고, 따라서 모든 리프 노드는 NIL 노드이다.



[그림 1.2] 레드-블랙 트리의 노드 구조

이런 특성을 가진 레드-블랙 트리는 삽입, 삭제 그리고 탐색의 연산 후에도 트리의 균형이 일정해지게 되며 해당 알고리즘의 시간 복잡도가 최악의 경우에도 O(을 가지게 된다.

## 레드-블랙 트리의 5가지 규칙

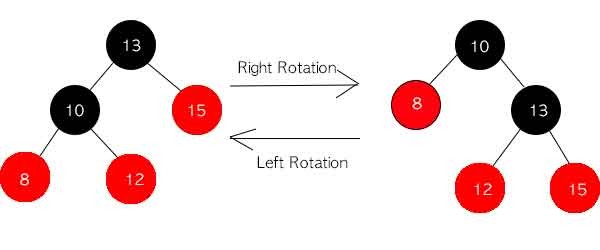
1. 모든 노드는 블랙이거나 레드이다.
2. 루트 노드는 반드시 블랙이다.
3. 모든 리프(NIL)는 블랙이다.
4. 레드 노드의 양쪽 자식 노드는 언제나 블랙이다.  
   (즉, 레드 노드가 연달아 나올 수 없다.)
5. 어떤 노드로부터 시작되어 리프 노드에 도달하는 모든 경로에는 리프 노드를 제외하면 모두 같은 개수의 블랙 노드가 있다.

레드-블랙 트리는 특별한 규칙을 가지기 때문에 삽입과 삭제를 하는 과정에서 규칙이 깨지는 상황이 발생한다. 규칙이 어긋나지 않도록 삽입과 삭제 연산 과정 후 규칙대로 레드-블랙 트리의 특성을 유지할 수 있게 추가적인 연산과정이 필요하다.

# 레드-블랙 트리의 회전 연산

## 회전 연산

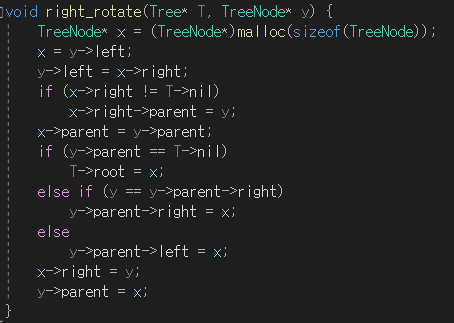
레드-블랙 트리는 삽입이나 삭제연산을 했을 경우 레드-블랙 트리의 규칙들이 깨질 수 있는데 그 규칙들을 지키도록 수정하기위해서 회전 연산을 사용한다. 경우에 따라 우회전과 좌회전 연산 2가지를 사용하며 이 둘의 구조는 완벽하게 대칭된다.



**[그림 2.1] 우회전 연산과 좌회전 연산의 예시**

## 소스코드 분석

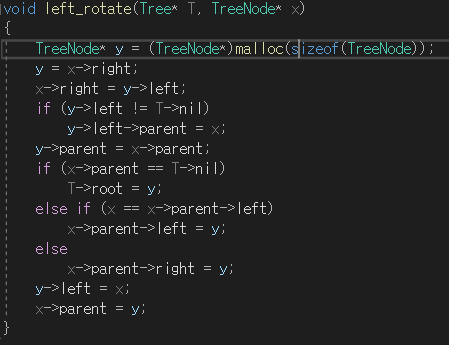
### 우회전 연산(right\_rotate)



**[그림 2.2] 우회전 연산 함수**

1. 임시로 사용할 트리노드 x를 생성하고 y의 왼쪽 자식을 저장한다.
2. y의 왼쪽 자식에 x의 오른쪽 자식을 저장한다. 여기서 만약 x의 오른쪽이 nil노드가 아니라면 x의 오른쪽의 부모를 y로 변경한다. 그 후 x의 부모를 y의 부모로 변경한다.
3. y의 부모가 nil노드였다면 x는 루트가 된다. 아니라면 y가 부모의 왼쪽 자식이었는지 혹은 오른쪽 자식이었는지에 따라 x를 y의 부모의 왼쪽 혹은 오른쪽의 자식이 된다.
4. X의 오른쪽을 y로, y의 부모를 x로 변경하고 함수를 종료한다.

### 좌회전 연산(left\_rotate)



**[그림 2.3] 좌회전 연산 함수**

2.2.1의 우회전 연산과 대칭되므로 좌, 우만 바꾸면 동일하므로 자세한 내용은 생략.

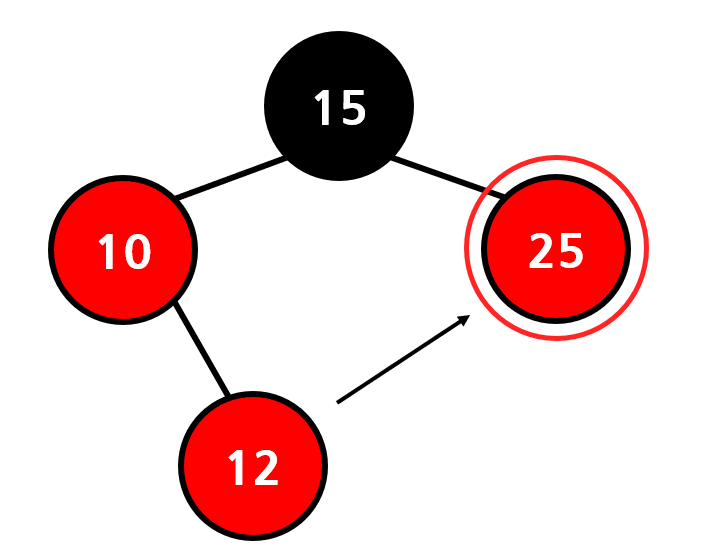
# 레드-블랙 트리의 삽입 연산

## 삽입 연산

레드-블랙 트리는 기본적으로 기존의 이진 탐색 트리와 동일하게 삽입 연산을 시행하지만 삽입되는 노드는 모두 레드로 시작하기에 언젠가 ‘레드 노드의 양쪽 자식 노드는 언제나 블랙이다.’라는 규칙에 어긋나게 되어있다. 이때의 더블 레드 현상을 교정해서 트리의 밸런스를 맞추어 나가는 것이 레드-블랙 트리의 삽입연산에서의 중점이다.

더블 레드 현상이 발생했을 때 부모의 형제 노드의 색깔에 따라 경우의 수를 나누어 교정작업이 진행된다. 총 6개의 경우의 수가 발생하고 그중 1,2,3은 부모가 부모의 부모의 왼쪽 자식일 때, 4,5,6은 부모가 부모의 부모의 오른쪽 자식일 때에 해당한다. 따라서 회전 연산과 마찬가지로 대칭구조를 이루고 있다.

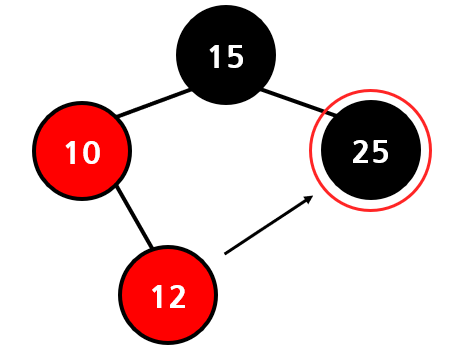
* Case 1: 부모의 형제 노드가 레드일 때



**[그림 3.1] 부모의 형제 노드가 레드일 때**

1. 부모와 부모의 형제를 블랙으로 칠한다.
2. 부모의 부모를 레드로 칠한다.

* Case 2: 부모의 형제가 블랙이고 내가 부모의 오른쪽 자식일 때



**[그림 3.2] 부모의 형제가 블랙이고 내가 부모의 오른쪽 자식일 때**

1. 부모를 기준으로 좌회전 연산을 하여 나, 부모, 부모의 부모를 일직선상에 두게 한다.
2. 부모를 검정으로, 부모의 부모를 빨강으로 칠한 후 부모의 부모를 기준으로 우회전 연산을 한다.

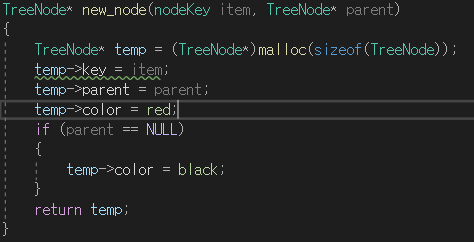
* Case 3: 부모의 형제가 블랙이고 내가 부모의 왼쪽 자식일 때

1. 나, 부모, 부모의 부모가 이미 일직선상에 둬져 있으므로 부모를 검정으로 부모의 부모를 빨강으로 칠한 후 부모의 부모를 기준으로 우회전 연산을 한다.

* Case 4, 5, 6: Case 1, 2, 3과 좌우 대칭

## 소스코드 분석

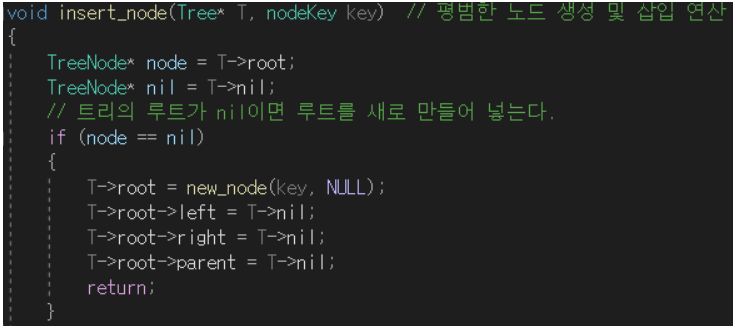
### 노드 생성 함수(new\_node)



**[그림 3.3] 노드 생성 함수**

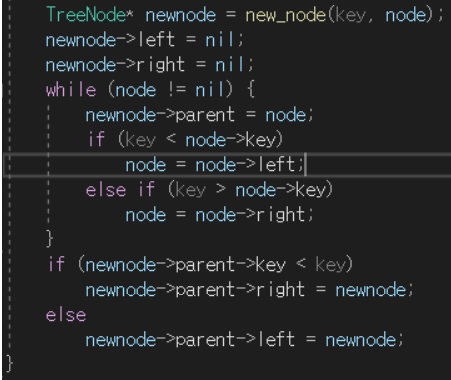
1. 키 값과 부모 노드의 주소를 매개 변수로 받는다.
2. 메모리를 할당하고 키 값, 부모 노드, 색을 초기화 한다.
3. 부모가 NULL로 되어 있다면 현재 생성할 노드가 루트 노드가 될 것이기 때문에 색깔을 검정으로 칠하고 이 노드를 반환한다.

### 삽입 연산(insert\_node)



**[그림 3.4] 삽입 함수 (1)**

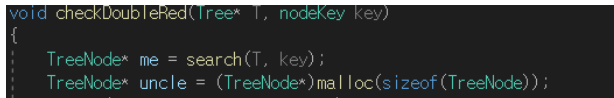
1. 트리와 삽입할 키 값을 매개변수로 받는다.
2. 트리의 루트 노드를 node에, 닐 노드를 nil에 담아놓는다.
3. node와 nil이 동일할 경우 아직 트리가 비어 있는 상태이므로 new\_node함수를 통해 새로운 노드를 생성하여 루트에 넣어주고 좌, 우, 부모 노드를 nil로 설정해주고 함수를 종료시킨다.



**[그림 3.5] 삽입 함수 (2)**

1. 그렇지 않을 경우 new\_node함수로 새로운 노드를 생성하고 왼쪽, 오른쪽 자식을 nil노드로 초기화한다.
2. 키 값을 비교하며 올바른 위치를 찾은 후에 부모의 노드와 키 값을 비교하여 부모의 왼쪽 자식인지 오른쪽 자식인지를 결정한다.

### 더블 레드 교정 함수(checkDoubleRed)



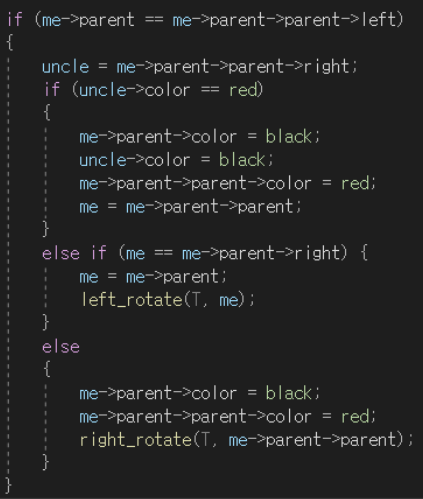
**[그림 3.6] 더블 레드 교정 함수 (1)**

1. 트리와 직전에 삽입 연산을 한 키 값을 매개 변수로 받고 삽입한 노드를 탐색하여 me에 저장한다.
2. 부모의 형제 노드를 저장할 uncle노드를 생성하여 메모리를 할당한다.



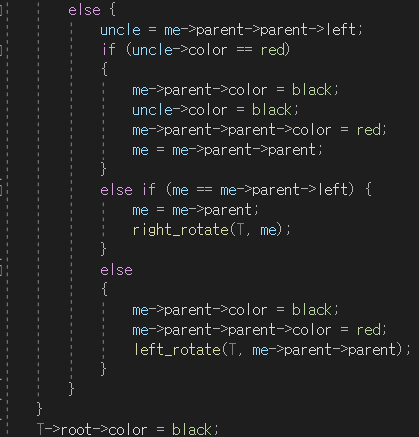
**[그림 3.7] 더블 레드 교정 함수 (2)**

1. 부모의 색깔이 레드가 아닐 때까지 무한 반복으로 교정작업을 실행한다.



**[그림 3.8] 더블 레드 교정 함수 (3)**

1. 부모가 부모의 부모의 왼쪽일 때 uncle에 부모의 부모의 오른쪽을 저장한다.
2. Uncle이 레드일 경우 부모와 uncle을 블랙으로, 부모의 부모를 레드로 칠한다. (Recoloring)
3. Recoloring의 경우 다시 더블 레드가 발생할 수 있기 때문에 부모의 부모로 me를 변경하여 다음 반복문에서 더블 레드를 체크하게 한다.
4. me가 부모의 오른쪽 자식일 경우 me를 부모로 변경하고 좌회전 연산을 실행한다.
5. me가 부모의 왼쪽 자식일 경우 부모의 색깔을 블랙으로 부모의 부모를 레드로 칠하고 부모의 부모를 기준으로 우회전 연산을 실행한다.



**[그림 3.9] 더블 레드 교정 함수 (4)**

1. ④, ⑤, ⑥, ⑦, ⑧에서 left와 right만 서로 바꾸어 주면 반대쪽의 동일한 연산을 한다.
2. Recoloring 연산을 하다 보면 root가 레드로 바뀌었을 수 있으므로 루트를 블랙으로 수정한다.

# 레드-블랙 트리의 삭제 연산

## 삭제 연산

레드-블랙 트리는 기존의 이진 탐색 트리보다 복잡한 삭제 연산을 필요로 한다. 노드를 삭제하고 다시 규칙대로 레드-블랙 트리의 속성을 유지할 수 있도록 하는 안정화 과정을 필요로 한다. 우선 레드-블랙 트리의 삭제 연산의 시작은 다양한 경우로 상황을 나누어 삭제 연산을 진행한다.

레드-블랙 트리에서 삭제 연산에서 가장 중요한 점은 노드를 삭제하고 나서, 무너진 레드-블랙 트리의 규칙을 복원하는 것이다.

삭제된 노드가 레드인 경우 규칙을 맞추기 위한 추가 연산을 필요로 하지 않는다. 삭제 연산 시 규칙이 무너지는 경우는 레드 노드가 연속적으로 부모 자식 관계인 경우인데(더블 레드) 레드 노드가 삭제되는 것으로는 레드-블랙 트리의 규칙을 벗어나지 않기 때문이다.

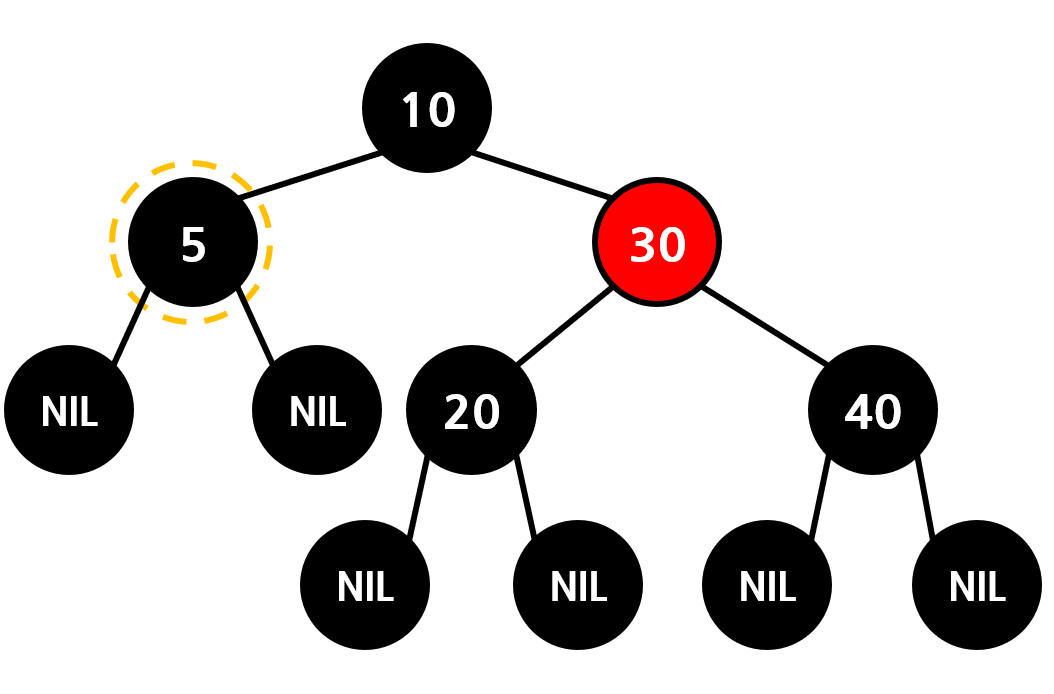
삭제된 노드가 블랙인 경우에는 적절한 추가 연산을 하여 레드-블랙 트리의 규칙을 벗어나지 않도록 한다.

* Default Case

삭제가 일어나면 반드시 시행되는 경우이다. 삭제된 노드가 블랙인 경우, 그 자리를 대체하는 노드의 색깔 속성을 블랙으로 변경한다. 하지만 이것 만으로 삭제 연산이 끝나지 않는다. 만약 자리를 대체하는 노드가 블랙 속성의 노드라면, 원래 블랙 노드를 다시 블랙으로 칠하는 경우가 발생하게 되는데, 이를 이중 블랙 노드라고 부른다. 지금부터 가정하는 상황은 부모의 왼쪽 자식을 기준으로 가정하고 있다. 하지만 부모의 오른쪽 자식의 경우 현 상황과 대칭되게 연산하고 조건을 맞추면 되므로 실제로는 총 경우는 8가지의 경우이다.

* 이중 블랙 노드 처리

**Case 1: 이중 블랙 노드의 형제 노드가 레드인 경우**

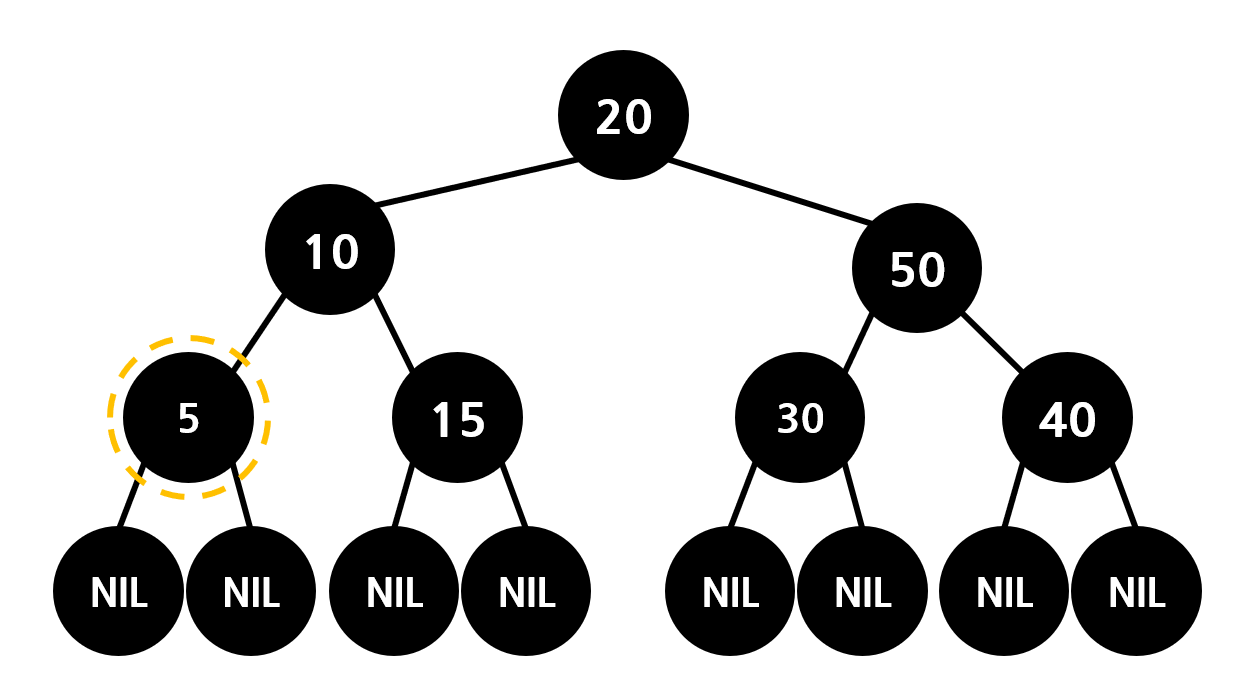


[그림 4.1] Case 1: 이중 블랙 노드의 형제 노드가 레드인 경우

1. 형제 노드를 블랙으로 칠한다.
2. 부모 노드를 레드로 칠한다.
3. 부모 노드를 기준으로 좌회전 연산을 한다.

**Case 2: 이중 블랙 노드의 형제 노드가 블랙인 경우**

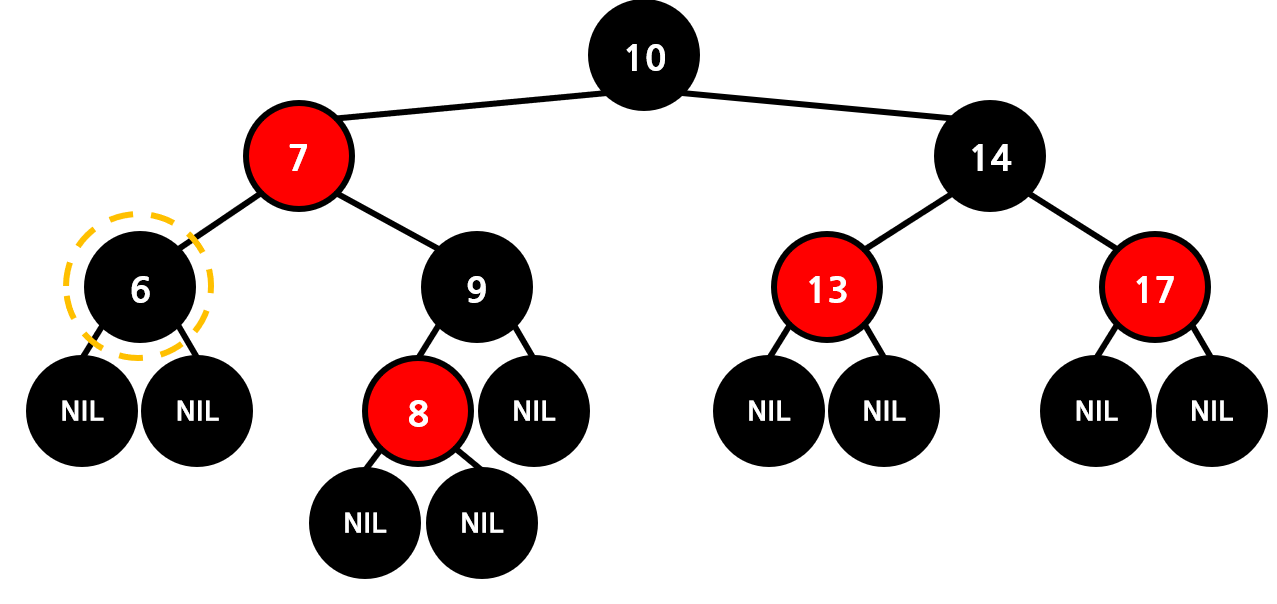
**Case 2-1: 형제 노드의 자식이 모두 블랙인 경우**

****

[그림 4.2] Case 2-1: 형제 노드의 자식이 모두 블랙인 경우

1. 형제 노드만 레드로 만든다.
2. 이중 블랙 노드의 블랙을 부모 노드의 색깔 속성에 전달한다.

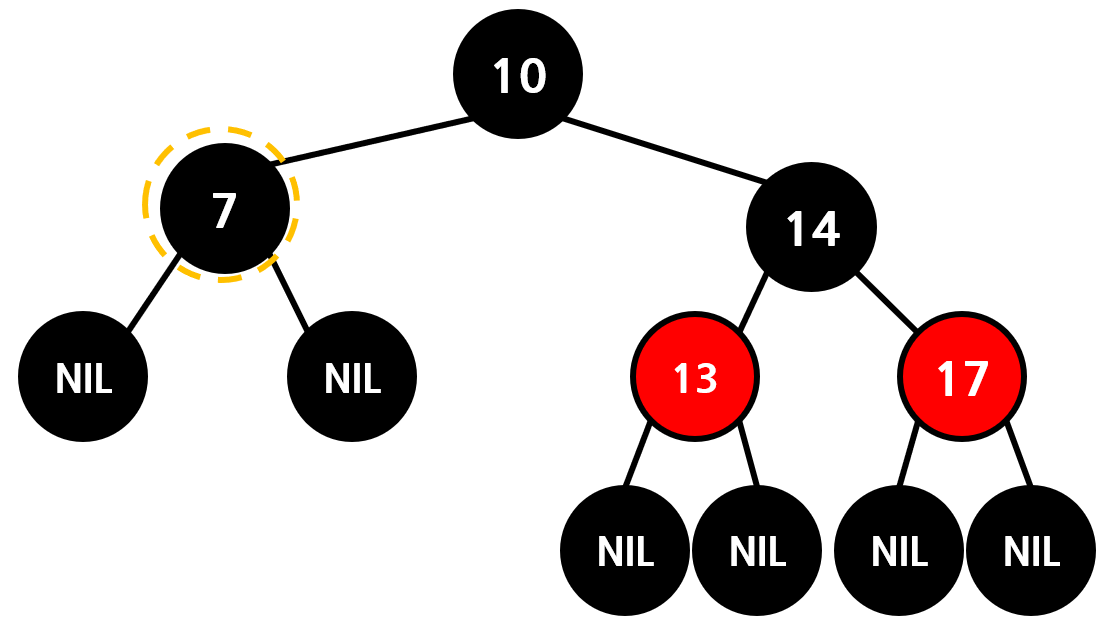
**Case 2-2:** **형제 노드의 왼쪽 자식이 레드인 경우**

****

[그림 4.3] 형제 노드의 왼쪽 자식이 레드인 경우

1. 형제 노드를 레드로 변경한다
2. 형제 노드의 왼쪽 자식 블랙으로 변경한다.
3. 형제 노드 기준으로 좌회전한다.

**Case 2-3: 형제 노드의 오른쪽 자식이 레드인 경우**

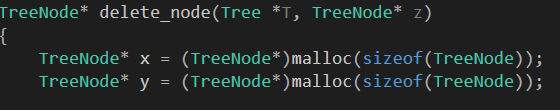


[그림 4.4] Case 2-3: 형제 노드의 오른쪽 자식이 레드인 경우

1. 부모 노드의 색을 형제 노드에게 넘긴다.
2. 부모 노드와 형제 노드의 오른쪽 자식을 블랙으로 칠한다.
3. 부모 노드 기준으로 좌회전을 한다.

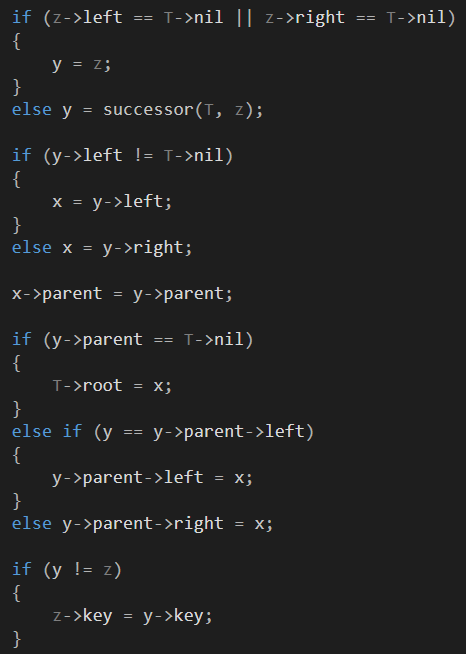
## 소스코드 분석

### 삭제 기능 함수(Delete node)



[그림 4.5] delete\_node 함수(1)

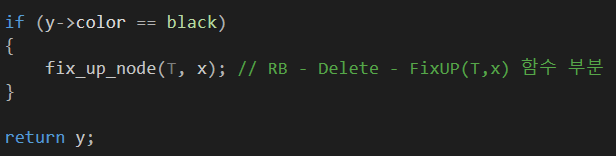
1. delete\_node함수는 Tree의 root와 nil정보를 가지고 있는 Tree 구조체와 삭제할 대상인 TreeNode구조체 z의 주소를 매개변수로 입력 받는다. 그리고 연산 과정에서 임시로 필요한 TreeNode x,y를 생성한다.



[그림 4.6] delete\_node 함수(2)

1. 삭제할 노드z의 양쪽 자식이 모두 nil(NULL)이 아니라면 즉, 자식이 없다면 노드z를 임시 노드 y에 저장한다.
2. 만약 삭제할 노드z가 자식을 가지고 있다면, 이진 탐색 트리처럼 자기 자신을 대체할 노드를 찾는 successor( )함수를 이용하여 삭제 대상 노드의 오른쪽 서브 트리 중 최소값을 찾아 노드 y에 저장한다. 만약 삭제 대상 노드의 오른쪽 자식 노드가 없다면 왼쪽 자식 노드를 반환한다.
3. 위의 과정을 거치고 나면 y 노드는 자식이 하나이거나 없다. successor는 자식이 두 개 일 수가 없다. y노드의 왼쪽 자식이 nil(NULL)이 아니라면 y의 왼쪽 자식 노드를 노드x에 저장한다.   
   y노드의 왼쪽 자식이 nil(NULL)이라면 노드y의 오른쪽 자식 노드를 노드x에 저장한다.
4. y노드의 부모 노드를 x노드의 부모 노드로 설정한다. 만약 이 때 y노드의 부모 노드가 nil(NULL)이라면 노드 x를 트리의 root로 설정한다. 만약 y노드의 부모 노드가 nil(NULL)이 아니고 y노드가 부모 노드의 왼쪽 자식이면 x노드를 y노드의 부모 노드의 왼쪽 자식으로 설정한다. 반대로 y노드의 부모 노드가 nil(NULL)이 아니고 y노드가 부모 노드의 오른쪽 자식이라면 x노드를 y노드의 부모 노드의 오른쪽 자식으로 설정한다.
5. y노드의 key 값을 z노드의 key 값으로 설정한다.

지금까지 수행한 작업은 이진 탐색 트리의 삭제 연산 과정이다. 하지만 레드-블랙 트리의 경우 노드를 삭제하고나서 레드-블랙 트리의 규칙을 유지해야 하기 때문에 추가적으로 필요연산을 수행한다. 다만 삭제하는 노드가 레드인 경우는 삭제하더라도 규칙을 유지하는데 문제가 없기 때문에 삭제하는 노드가 블랙이라면 fix-up( )함수를 호출하여 레드-블랙 트리의 속성을 유지할 수 있도록 한다.



[그림 4.7] 삭제하는 노드y의 색이 블랙인 경우

### 재정렬 함수(Fix Up)

레드-블랙 트리에서 실제로 삭제한 노드는 y노드이다. 삭제한 노드 y의 자식인 x를 넘겨주면서 Delete Fixup을 하게 된다.

**Case 1,2,3,4:** **노드 x가 부모의 왼쪽 자식 노드인 경우**

x는 double black노드이거나, NIL 노드 일수도 있다.

x의 형제 노드인 w노드는 반드시 존재하고, NIL일 수는 없다.

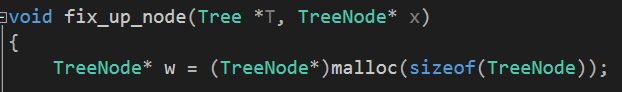
x는 자신의 부모의 왼쪽 자식이다.

**Case 5,6,7,8: 노드 x가 부모의 오른쪽 자식 노드인 경우**

x는 double black노드이거나, NIL 노드 일수도 있다.

x의 형제 노드인 w노드는 반드시 존재하고, NIL일 수는 없다.

x는 자신의 부모의 오른쪽 자식이다.



[그림 4.8] fix\_up\_node 함수(1)

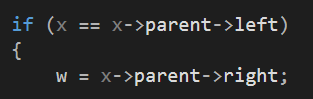
만약 x가 루트 노드이거나, x가 레드라면 While문을 빠져나가서 x를 블랙으로 만들어주고 종료하면 된다.



[그림 4.9] fix\_up\_node 함수(2)

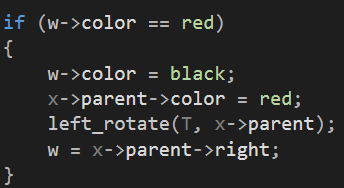
case 1,2,3,4 와 case 5,6,7,8은 대칭적으로 수행하면 되므로, case 1,2,3,4만 다루도록 한다.

노드 x의 형제 노드인 w를 저장한다. 노드x가 부모 노드의 왼쪽 자식이므로, w는 부모 노드의 오른쪽 자식 노드가 된다.

****

[그림 4.10] case 1,2,3,4: 노드 x가 부모의 왼쪽 자식 노드인 경우

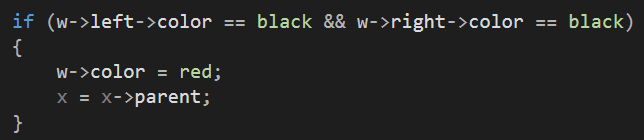
**case 1: 노드x의 형제 노드(w)가 레드인 경우**

****

[그림 4.11] 이중 블랙 노드의 형제 노드(w)가 레드인 경우

1. 형제 노드 w의 색을 블랙으로 바꾼다.
2. 노드 x의 부모 노드의 색을 레드로 바꾼다.
3. x노드의 부모 노드를 기준으로 좌회전한다.

**case 2: 노드x의 형제 노드(w)가 블랙이고, 형제 노드(w) 자식이 모두 블랙인 경우**

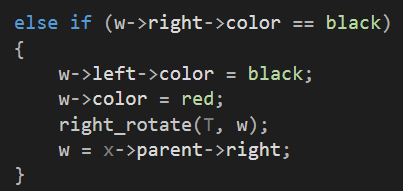


[그림 4.12] 노드x의 형제 노드(w)가 블랙이고, 형제 노드(w) 자식이 모두 블랙인 경우

노드w와 노드x로부터 black을 하나씩 빼앗아 부모 노드에게 전달한다. 그렇게 하기 위해서 w는 RED노드가 되고, x노드의 부모 노드를 x로 만들어 준다. x노드의 부모 노드가 레드였다면 다시 While문을 돌지 않고 x노드를 레드로 만든 뒤에 종료하면 되고, x노드의 부모 노드가 블랙이었다면 x를 x노드의 부모 노드로 놓고 더블-블랙 노드가 된 노드x를 다시 반복해서 처리해주면 된다.

1. 형제 노드w의 색을 레드로 바꾼다.
2. x노드의 부모 노드를 x로 설정한다.

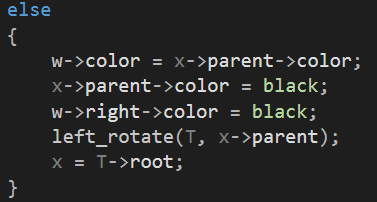
**case 3: 노드x의 형제 노드(w)가 블랙이고, 형제 노드(w) 왼쪽 자식이 레드인 경우**



[그림 4.13] case 3: 노드x의 형제 노드(w)가 블랙이고, 형제 노드(w) 왼쪽 자식이 레드인 경우

1. w를 RED로, w의 왼쪽자식 노드를 BLACK으로 바꾼다.
2. w에 대해서 우회전 연산을 적용한다.
3. x노드의 새로운 형제 노드w는 오른쪽자식 노드는 레드이다

**case 4: 노드x의 형제 노드(w)가 블랙이고, 형제 노드(w) 오른쪽 자식이 레드인 경우**

****

**[그림 4.14] case 4: 노드x의 형제 노드(w)가 블랙이고, 형제 노드(w) 오른쪽 자식이 레드인 경우**

1. 형제 노드w의 색을 x노드의 부모 노드의 색으로 바꾼다.
2. x노드의 부모 노드의 색을 블랙으로 바꾸고, 형제 노드w의 오른쪽자식 노드의 색을 블랙으로 바꾼다.
3. x노드의 부모 노드를 기준으로 좌회전을 한다.

case 5,6,7,8은 위의 case 1,2,3,4와 대칭되기 때문에 대칭적으로 시행하면 해결된다. 총 8가지의 케이스로 분류하고 그에 알맞게 재정렬을 하여 레드-블랙 트리의 규칙을 위반하지 않게 유지하며 삭제 연산을 시행한다.

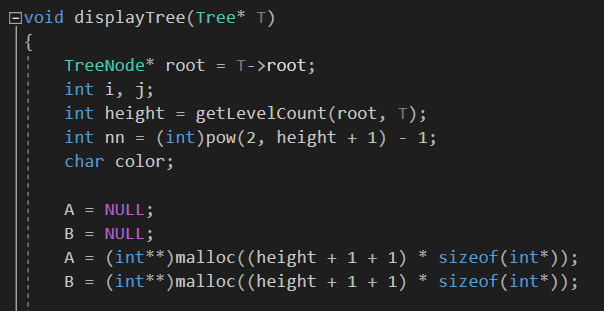


[그림 4.15] 8가지 케이스 분류 후 마지막 x의 색을 블랙으로 바꿈

총 8가지의 케이스의 경우 알맞은 연산을 시행하고 마지막으로 x노드의 색깔을 블랙으로 바꾸며 재정렬 연산을 마무리한다

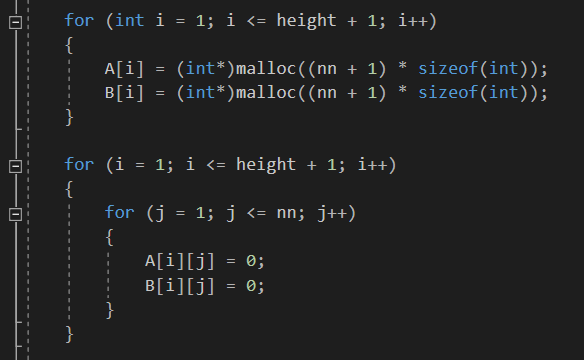
# 트리 구조 출력

## 소스코드 분석



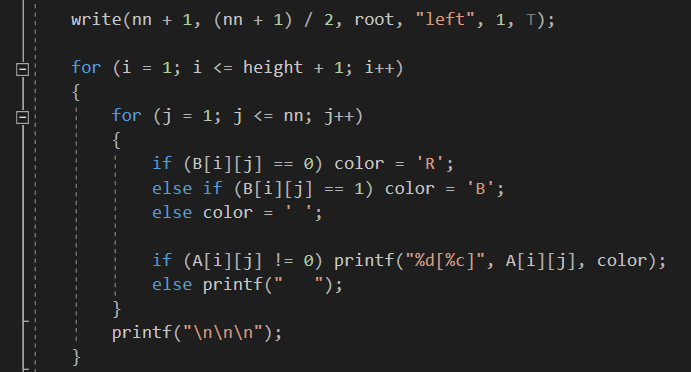
[그림 5.1] displayTree 함수(1)

1. 트리를 매개변수로 받고 트리의 루트 노드를 root에 받는다.
2. 변수에 노드의 최대 높이, 최대 노드의 개수, 색 정보를 저장한다.
3. A와 B를 초기화하고 메모리를 할당한다.



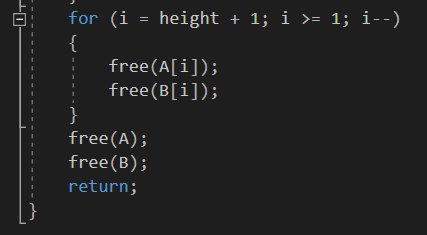
[그림 5.2] displayTree 함수(2)

1. A, B를 이차원 배열로 초기화한다.



[그림 5.3] displayTree 함수(3)

1. write함수로 A배열에 트리의 키, B 배열에 색 정보를 저장한다.
2. B 값이 0일 경우 노드의 색이 RED이므로 color 변수에 R을 저장하고, 1일 경우 노드의 색이 BLACK이므로 color 변수에 B를 저장한다.
3. A 값이 0이 아닐 경우(값이 존재할 경우) A값(트리의 키)과 color를 출력하고, 0일 경우 공백을 출력한다.
4. ⑥, ⑦을 j가 1일때부터 최대 노드의 개수에 다다를 때까지 반복한다.
5. ⑥, ⑦, ⑧을 i가 1일때부터 최대높이에 다다를 때까지 반복한다.

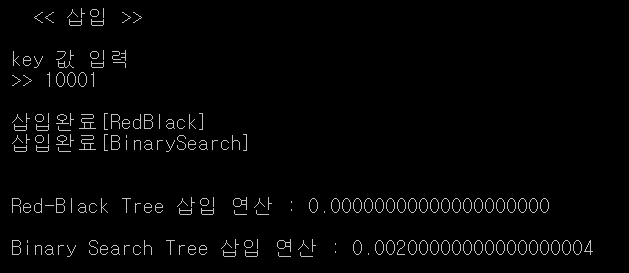


[그림 5.4] displayTree 함수(4)

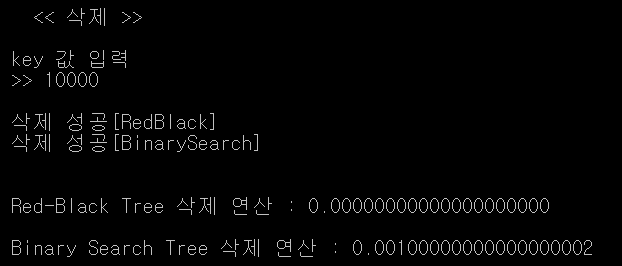
1. 트리의 정보를 넣기 위해 할당된 메모리인 A와 B를 해제한다.

# 레드-블랙 정리

## 시간 복잡도

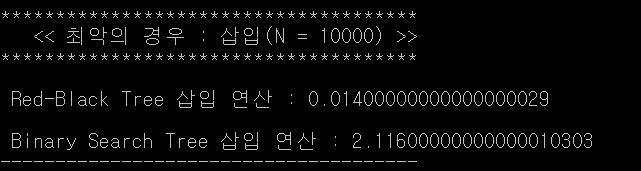


**[그림 6.1] 삽입 수행 시간**



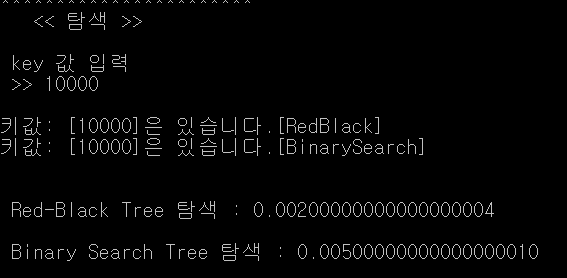
**[그림 6.2] 삭제 수행 시간**

레드-블랙 트리와 이진 탐색 트리 삽입, 삭제, 탐색의 경우 수행 시간을 통해 측정하는 시간 복잡도를 출력해 보았다. 레드-블랙 트리 삽입과 삭제의 경우에는 빨라서 측정값이 출력되지 않았으며 이진 탐색 트리의 경우에 더 많은 시간을 소요함을 확인할 수 있었다.



**[그림 6.3] 삽입 수행 시간 (1~10000)**

이전은 숫자 하나를 삽입하는 경우이고 위의 그림은 1부터 10000의 숫자를 순차적으로 삽입하는 경우의 수행시간이다. 시간 소요에 약 150배 차이가 확인된다.



**[그림 6.4] 탐색 수행 시간 (1~10000 중 10000 탐색)**

1부터 10000의 수가 순차적으로 삽입된 상태에서 10000의 숫자를 탐색한 경우의 수행 시간이다. 이 경우 값이 편향되게 들어오게 되어 이진 탐색 트리는 한쪽으로 치우친 트리가 되며 이때 트리의 높이만큼 시간이 필요하게 된다. 그리하여 탐색 시간이 약 2.5배 차이가 남을 확인할 수 있다.

## 결론

레드-블랙 트리 알고리즘은 삽입의 경우 Color Promtion과 Rotation으로, 삭제의 경우 Color Demotion과 Rotation으로 스스로 균형을 맞추는 Balance tree이다. 이로 인해 노드 수가 N개라면 높이는 항상 2log(N+1)보다 작다. 결과적으로 삽입, 삭제, 탐색의 성능은 모두 O(logN)이며 좋은 성능을 가진다. 또한 이진 탐색 트리와 수행 시간을 비교한 결과 레드-블랙 트리의 성능이 더 좋은 것으로 나타났다.