알고리즘 8주차(레드블랙트리)

8주차(레드블랙트리)

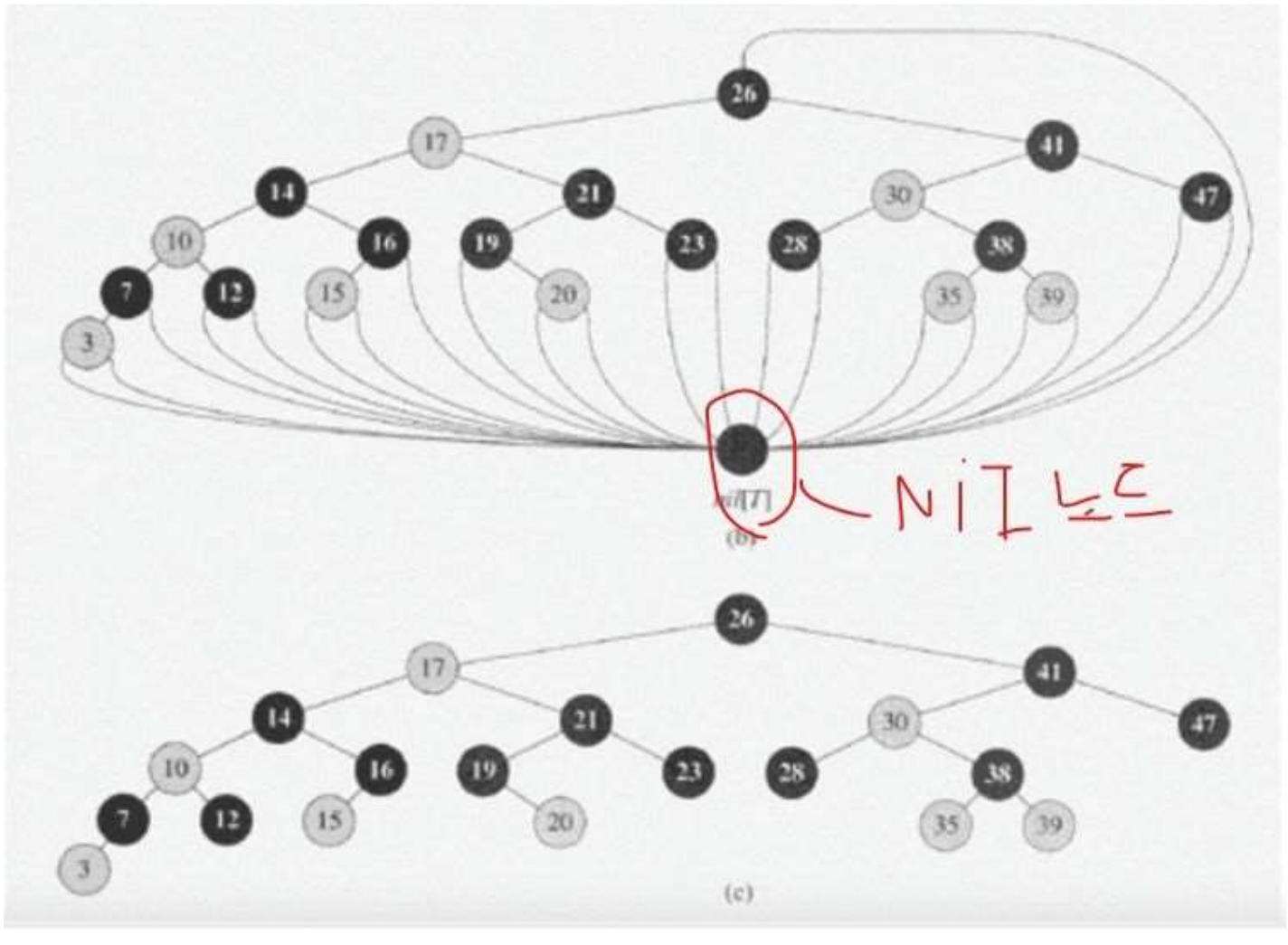
레드블랙트리 구현기

Red/Black Tree

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

레드블랙 트리란

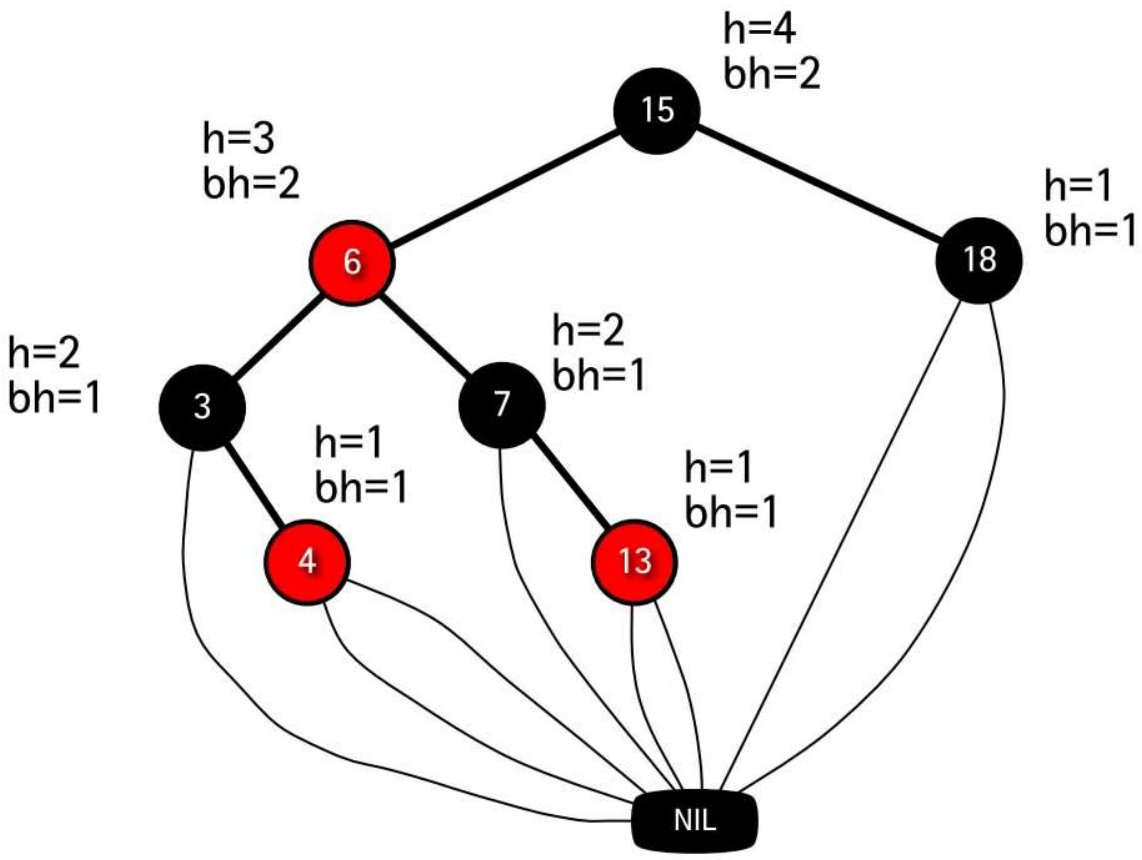
- Binary Search Tree의 일종
- **균형잡힌 트리** : 높이가 O(logn)
- Search, Insert, Delete 연산을 최악의 경우에도 O(logn)시간에 지원
  - insert, delete 알고리즘을 수정하여 트리의 균형이 잡히도록 한다.
- 각 노드는 하나의 키(Key), 왼쪽자식(Left), 오른쪽 자식(Right), 그리고 부모노드(p)의 주소를 저장
- 자식노드가 존재하지 않을 경우 NIL 노드라고 부르는 특수한 노드가 있다고 가정한다.
- 따라서 모든 **리프노드는 NIL 노드**이다.
- 노드들은 내부노드와 NIL 노드로 분류한다.
- **실제로 구현**을 할 때, NIL 노드를 포함하지 않고, 개념적인 설명을 좀 더 편하게 하기 위해서 가상적으로 NIL 노드를 사용한다.
- 개념적으로는 (b)의 그림처럼 생각하지만, 실제 구현은 (c)가 될 것이다.



정의

- 다음의 조건을 만족하는 이진탐색트리

1. 각 노드는 red 혹은 black이고,
2. **루트노드는 black**이고,
3. 모든 **리프노드(즉, NIL 노드)는 black**이고,
4. **red노드의 자식노드들은 전부 black**이고**(즉, red노드는 연속되어 등장하지 않고),**
5. **모든 노드**에 대해서 그 노드로 부터 자손인 리프노드에 이르는 **모든 경로에는 동일한 개수의 black노드**가 존재한다.

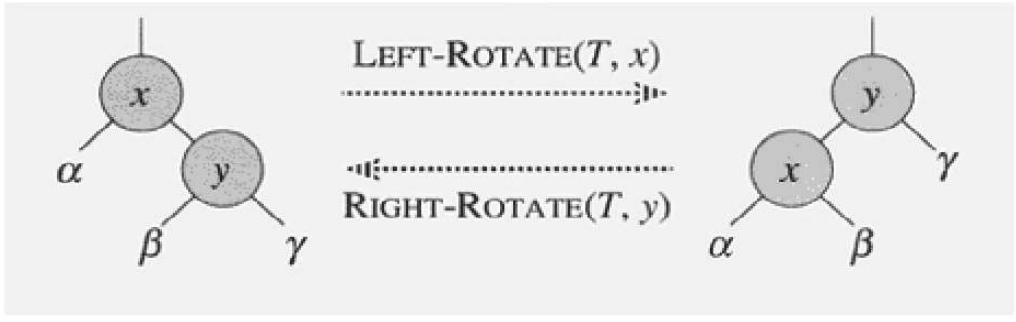


레드블랙트리의 높이

1. 노드 x의 **높이 H(x)**는 **자신으로부터 리프노드까지의 가장 긴 경로에 포함된 에지의 개수**이다
2. 노드 x의 **블랙-높이 bh(x)**는 **x로 부터 리프노드까지의 경로상의 블랙노드의 개수**이다.(노드 x 자신은 불포함)
3. 높이가 h인 노드의 **블랙-높이는 bh≥h/2**이다
    - 조건4에 의해 레드 노드는 연속될수 없으므로 당연히 경로상의 블랙노드는 절반이상이다.
4. 노드 x를 루트로 하는 임의의 부트리는 적어도 (2^bh(x)) -1(자신노드)개의 내부노드를 포함한다.
5. n개의 내부노드를 가지는 레드 블랙트리의 높이는 2log2(n+1)이하이다
    - n≥2^bh-1≥2^h/2-1이므로 여기서bh와h는 각각 로트 노드의 블랙-높이와 높이

Left and Right Rotation

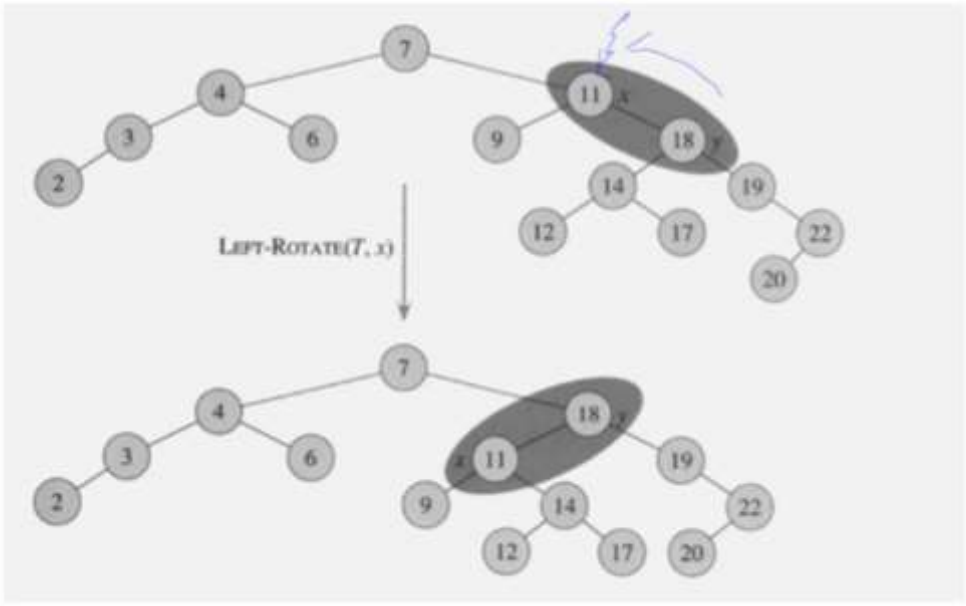
- Insert와 Delete가 모두 필요로하는 Left, Right Rotation을 알아본다.
- 한 노드를 중심으로 부분적으로 **트리의 모양을 수정하는 연산**이다.
- **시간복잡도는 O(1)**이다.
- 이진탐색트리의 특성을 유지한다.
    - Rotate를 해서 서브트리(B)가 옮겨지더라도, BST의 특성은 유지된다.
- x와 y의 자식들은 모두 서브트리이다.



수도코드(LEFT-ROTATE)

```
y = right[x] =! NIL이라고 가정 루트노드의 부모도 NIL이라고 가정 LEFT-ROTATE(T, x) //T는 트리 x는Rotation 실행할 노드 01 y <- right[x] //y에 x의 오른쪽노드주소 y로 정의 02 right[x] <- left[y] //x의 오른쪽 자식에 y의 외쪽값(베타값)을 넣는다 03 p[left[y]] <- x //각 노드마다 부모노드 주소를 저장했기때문에 필요한 연산 //y의 왼쪽값(베타)의 부모주소를x의 부모주소로 만들어라 04 p[y] <- p[x] //y의 부모를 x의 부모노드로 만들어주고 05 if p[x] = NIL[T] //현재 x노드가 루트노드라면 06 then root[T] <- y //y가 새로운 루트노드가 된다 07 else if x = left[p[x]] //x노드가 부모의 왼쪽자식이라면 08 then left[p[x]] <- y //y가 x부모의 왼쪽자식이되고 09 else right[p[x]] <- y //아니면 오른쪼자식이 된다. 10 left[y] <- x //y의 왼쪽자식은 x가된다. 11 p[x] <- y //x의 부모주소를 y로설정한다. 차례대로 수행하면 되므로, 시간복잡도는 O(1)이다.
```

Left Rotation



INSERT

```
RB-INSERT(T, z) //T는 트리z는 삽입할 노드 1 y <- nil[T] //y는 nil노드라고 설정 2 x <- root[T] //x를 루트노드로 설정 3 while x != nil[T] //x가 null일 될때까지 반복 4 do y <- x //삽입할 자리(Y)를 기억하는 부분 5 if key[z] < key[x] //삽입할 자리를 찾는 조건문 6 then x <- left[x] //현재노드의 왼쪽자식 주소를 넣는다. 7 else x <- right[x] //현재노드의 오른쪽자식 주소를 넣는다. 8 p[z] <- y // z의 부모노드값에 Y값을 넣는다. 9 if y = nil[T] //y가 nil노드라면 10 then root[T] <- z //T는 빈트리이며 z는 루트노드가 된다. 11 else if key[z] < key[y] //빈트리가 아니면 z값을 y값 크기따라 왼쪽자식 오른쪽자식이된다. 12 then left[y] <- z 13 else right[y] <- z //여기까지는 이진검색트리삽입과 유사 14 left[z] <- nil[T] //새롭게 리프노드가 된 z의 자식들은 nul노드가 된다 15 right[z] <- nil[T] 16 color[z] <- RED //새롭게 삽입된 z노드는 레드노드로 정한다. 17 RB-INSERT-FIXUP(T, z) //새로운 레드노드 추가로 되어 레드블랙트리의 조건에 맞게 조정한다. //연속된 레드노드가 있을때 또는 새롭게 추가된 노드가 루트노드가될때 TREE-INSERT(T, z)//이진검색트리의 삽입 01 y <- NULL 02 x <- root[T] 03 while x!= NULL 04 do y <- x 05 if key[z] < key[x] 06 then x <- left[x] 07 else x <- right[x] 08 p[z] <- y 09 if y = NULL 10 then root[T] <- z 11 else if key[z] < key[y] 12 then left[y] <- z 13 else 14 then right[y] <- z
```

RB-INSERT-FIXUP(2번,4번조건을 조심해야한다.)

- 레드블랙트리의 규칙이 위반될 가능성이 있는 조건들(레드블랙트리의 5가지 조건)
  - 모든 노드는 red 혹은 black이다.
    - **위반 가능성 없음**
  - **루트노드는 black이다.**
    - 새로운 노드 z가 루트 노드라면 규칙 위반, 아니라면 위반 가능성 없음
    - 새로운 노드 z가 루트 노드라면 간단하게 노드를 블랙으로 바꾸는 작업을 해주면 된다.
  - 모든 리프노드(즉, NIL 노드)는 black이다.
    - 새로운 노드의 자식들을 NIL노드로 설정하므로 **위반 가능성 없음**
  - **red노드의 자식노드들은 전부 black이다.(즉, red노드는 연속되어 등장하지 않고)**
    - **위반 될 가능성이 있다**. 부모 노드가 원래 RED였다면, RED - RED 가 되므로 위반이 발생한다.
    - 가장 큰 문제가 되는 것이 RED - RED 조건을 위반 하는 것이다.
  - 모든 노드에 대해서 그 노드로 부터 자손인 리프노드에 이르는 모든 경로에는 동일한 개수의 black노드가 존재한다.
    - 새로운 노드를 RED로 추가 했으므로, **위반 가능성 없음**

Loop Invariant (루프를 돌면서 변하지 않고 유지되는 조건) :

- z는 red 노드
- 오직 하나의 위반만이 존재한다.
  - 조건 2 : z가 루트노드이면서 red이거나, 또는
  - 조건 4 : z와 그 부모 p[z]가 둘 다 red 이거나.
  - **조건 4를 해결하기 위해 부모노드를 타고 올라가다 보면, 또다른 RED - RED 위반이 있을 수 있다. 최악의 경우 루트노드까지 타고올라가게 되면, 조건 2를 위반 한 것이므로 루트노드를 블랙으로 바꿔주고 종료하면 된다.**

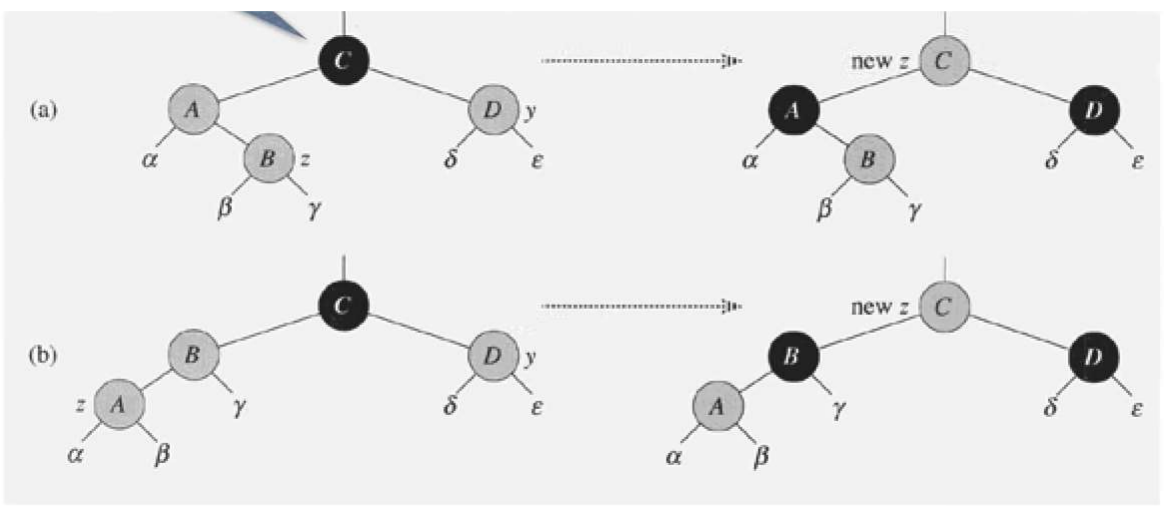
종료조건 :

- 부모노드 p[z]가 black이되면 종료한다. 조건2가 위반일 경우 z를 블랙으로 바꿔주고 종료한다.

RED_RED위반 6가지 케이스

Case 1, 2, 3(부모가 할아버지의 왼쪽자식)

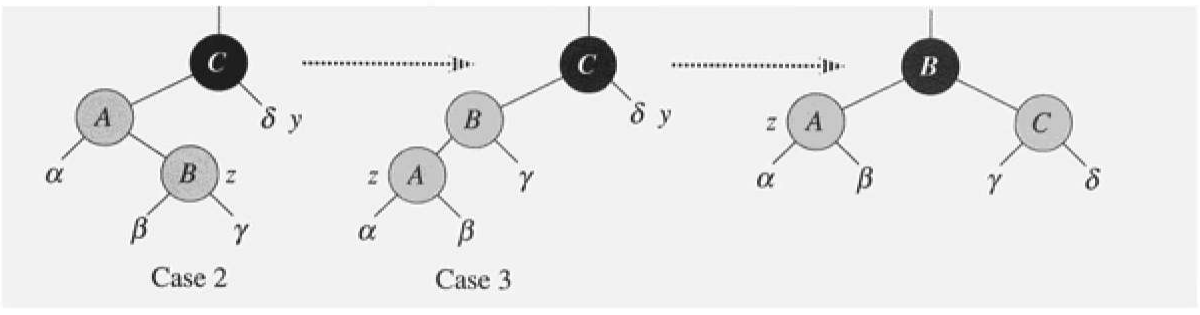
Case 1:**새로운 노드 z의 부모가 RED이면서, 부모의 형제 노드 D가 RED인 경우이다.**



(a) - 오른쪽 자식, (b) - 왼쪽 자식

- 이 경우 부모노드와 부모노드의 형제노드를 **BLACK으로 바꾸고, 할아버지 노드를 RED로 바꾼다.**
- 그리고 나서, 할아버지 노드 C를 새로운 노드 z로 설정하여 위로 타고 **올라가면서 레드 블랙 트리의 조건을 위반하는지를 검사한다.**
- Case 1의 경우 문제가 완전히 해결되지는 않았다. z가 두칸 위로 올라가서 그 위치에서 부터 다시 해결해야 한다.

Case 2, 3 **:z의 부모의 형제가 BLACK인 경우 실제로 BLACK 노드일 수도 있고NIL노드 일 수도 있다**



- Case 2 : z가 오른쪽 자식인 경우
  - p[z]에 대해서 left-rotation한 후 원래 p[z]를 z로 만든다.
  - Case 3으로 이동
- Case 3 : z가 왼쪽 자식인 경우
  - p[p[z]]에 대해서 right-rotation을 한다.
  - p[z]를 BLACK, p[p[z]]를 RED로 바꾼다.

Case 1, 2, 3 정리

- Case 1의 문제를 해결하고 나면 문제는 종료되지 않는다. Case 2로 갈수도 있고, Case 3로 갈수도 있으며, 다시 Case 1의 문제가 반복해서 발생할 수 있다. 최악의 경우 루트노드까지 올라가서 레드블랙트리의 조건 2 루트 노드가 레드인 경우를 블랙으로 변경하면서 종료하게 된다.
- Case 2는 발생하면, Case 3을 거쳐서 해결하면 종료된다.
- Case 3의 경우 z의 할아버지 노드를 기준으로 right-rotate하면 문제가 해결되고, 종료 된다.
- Case 1, 2, 3의 경우에 해당하는 것이고, Case 1에서 Case 4, 5, 6으로 넘어갈 수도 있다.

Case 4, 5, 6(부모가 할아버지의 오른쪽자식)

- Case 1, 2, 3은 p[z]가 p[p[z]]의 왼쪽 자식인 경우들
- Case 4, 5, 6은 p[z]가 p[p[z]]의 오른쪽 자식인 경우들
  - Case 1, 2, 3과 대칭적이므로 생략

수도코드(RB-INSERT-FIXUP)

▶ RB-INSERT-FIXUP

INSERT의 시간복잡도

- BST에서의 INSERT : O(logn)
- RB-INSERT-FIXUP
  - Case 1, 4에 해당할 경우 z가 2레벨 상승한다.
  - Case 2, 3, 5, 6에 해당할 경우 O(1)
  - 따라서, 트리의 높이에 비례하는 시간복잡도를 가진다.
- 즉, INSERT의 시간복잡도는 O(logn)

DELETE

```
x는 삭제할 노드의 남은자식,y는 삭제할 노드자리에 들어갈 노드 RB-DELETE(T, z) //T는 트리 z는 삭제할 노드 01 if left[z] = nil[T] or right[z] = nil[T]//삭제할 노드의자식이 한명이하라면 02 then y <- z //y에 삭제할 노드을 저장한다. 03 else y <- TREE-SUCCESSOR(z) //두명의 자식이 있다면 그다음으로 큰 자손을 y에 저장한다. 04 if left[y] != nil[T] //y는 왼쪽자식만 가지거나 자식이 없는데 why=>successor이기때문에 //만약 왼쪽자식있다면 05 then x <- left[y] //x에 y의 왼쪽자식을 넣는다. 06 else x <- right[y] //아니면 x에 nil값을 넣는다. 07 p[x] <- p[y] //y의 부모주소를 x의 부모주소설정한다. 08 if p[y] = nil[T] //만약, y의 부모가 NULL이라면(즉 삭제할 노드가 트리의 root라면) 09 then root[T] <- x //x를 트리의 root로 설정한다.(삭제할 노드의 다음번쨰 큰수가 root가 된다.) 10 else if y = left[p[y]] //아니면 y가 자신의 부모의 왼쪽자식이면 11 then left[p[y]] <- x //자신의 부모의 왼쪽자식의자리 즉 자신의 자리를 successor노드(x)한테 증여한다. 12 else right[p[y]] <- x //왼쪽이 아니라 오른쪽자식이면 똑같이 자신의자리를 successor한테 증여한다. 13 if y != z //삭제하려는 노드 대신 Successor를 삭제한 경우 y와 z가 다르다. //따라서 Successor y노드의 데이터를 카피해주는 작업이 필요하다. 14 then key[z] <- key[y] 15 copy y's satellite data into z //14~15은 y의 데이터들을 z노드로 copy해주는 작업 //(그말은 즉 노드z는 삭제됐다.) //여기까지는 bts의 삭제작업과 유사하다 아래부터의 부분은 레드블랙특성을 위해서 추가된 부분이다. 16 if color[y] = BLACK //삭제한 노드가 블랙노드이면 레드블랙노드 규칙 변화는 경우가 발생한다. 17 then RB-DELETE-FIXUP(T, x) //레드블랙트리복원작업 18 return y //없어도 되는것
```

RB-DELETE-FIXUP(T, x)

x가 NIL 노드일 수도 있다. 그리고 x가 red일 경우 쉽게 해결할 수 있다. 이 두가지를 기억한다. 실제로 DELETE에서 문제는 x가 BLACK인 경우다.

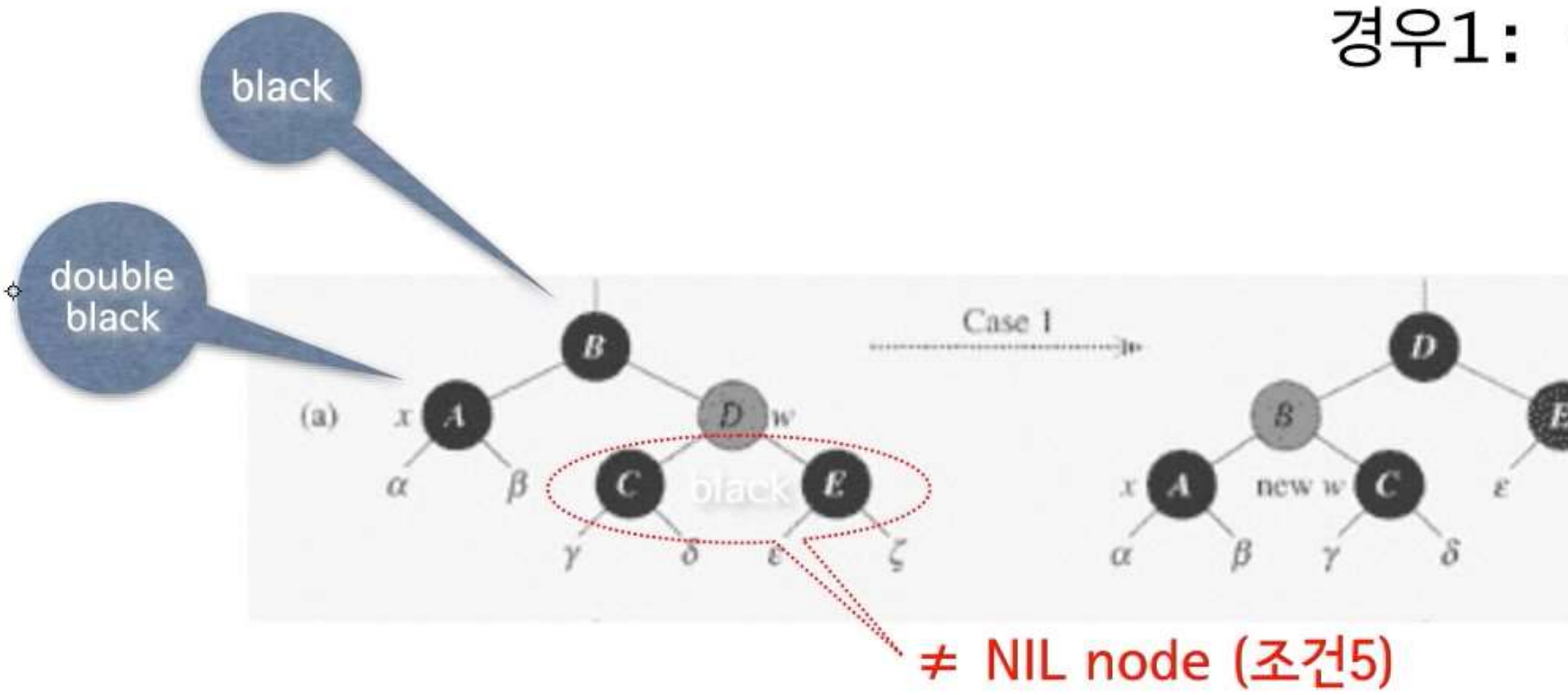
- 위반 될 수 있는 규칙 정리
  - 1번)각 노드는 red 혹은 black이다.
    - 문제 없음
  - 2번)루트노드는 black이다.
    - y가 루트였고, x가 red인 경우 위반된다. 하지만, 심각한 문제는 아니다.
  - 3번)모든 리프노드(즉, NIL 노드)는 black이다.
    - 문제 없음
  - 4번)red노드의 자식노드들은 전부 black이다.(즉, red노드는 연속되어 등장하지 않고)
    - p[y]와 x가 모두 red일 경우 위반
    - x가 레드인 경우 red를 black으로 바꾸어 주면 되기 때문에 심각한 문제는 아니다.
  - 5번)모든 노드에 대해서 그 노드로 부터 자손인 리프노드에 이르는 모든 경로에는 동일한 개수의 black노드가 존재한다.
    - 원래 y를 포함했던 모든 경로는 이제 black노드가 하나 부족하다.
      - 노드 x에 "extra black"을 부여해서 일단 조건5를 만족시킨다. 색을 두개 가지고 있게 하는 임시 방법이다.
      - 그렇게 되면 노드 x는 "double black" 혹은 "red & black"이 된다. 앞으로 해결해야하는 문제가 이 것을 블랙노드로 바꾸는 것이다.

RED_RED위반 8가지 케이스(5~8까지는 삽입과 똑같이 대치되는 부분이 있어 생략)

Case 1 - x의 형제노드 w가 RED인 경우

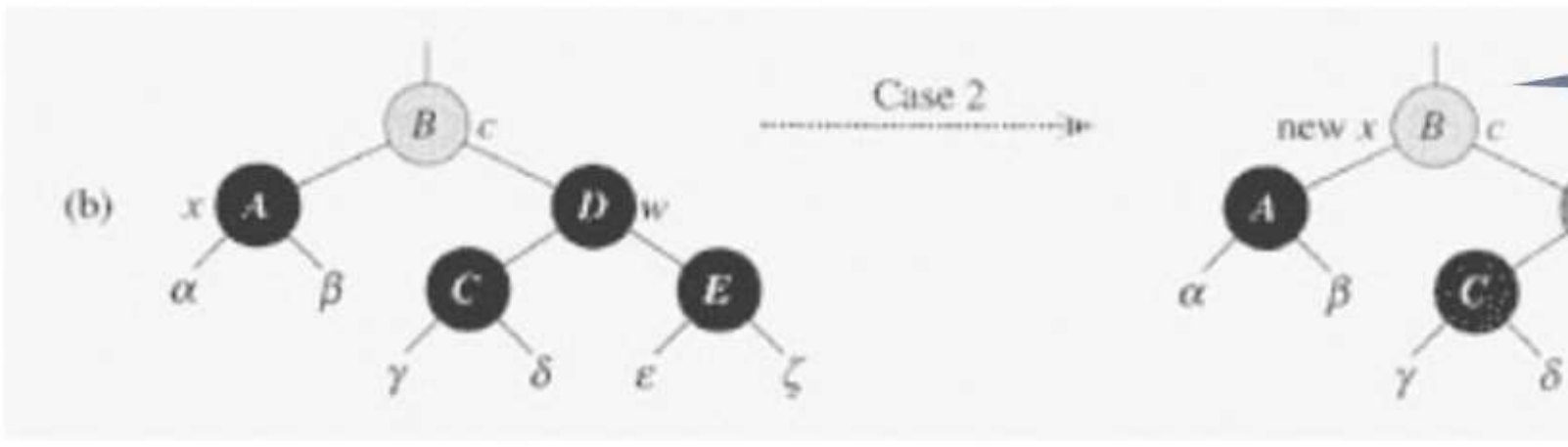
- Case 1,2,3,4의 공통 조건에 해당하며, 케이스 1인 경우는
- x의 형제노드 w가 RED인 경우이다. 이 경우는 자식노드가 NIL일 수 없고, BLACK 노드이다.
- 이 상황에서 문제를 해결하기 위해,
- w를 BLACK으로 p[x]를 RED로 바꾼 뒤
- p[x]에 대해서 left-rotation을 적용한다. x는 여전히 double-black 노드를 가지고 있다.
- x의 새로운 형제노드는 원래 w의 자식노드이다.
- 따라서, 새로운 w노드는 black노드이다. 이 경우 Case 2, 3, 4로 넘어가게 된다.

- 정리를 하면, Case1의 경우 x의 부모 노드에 대해서 left-rotation을 적용하면, 새로운 w노드가 red가 아닌 black이 되는 상황이라서 Case 2, 3, 4로 넘어가게 된다.



Case 2 - w는 BLACK, w의 자식들도 BLACK인 경우

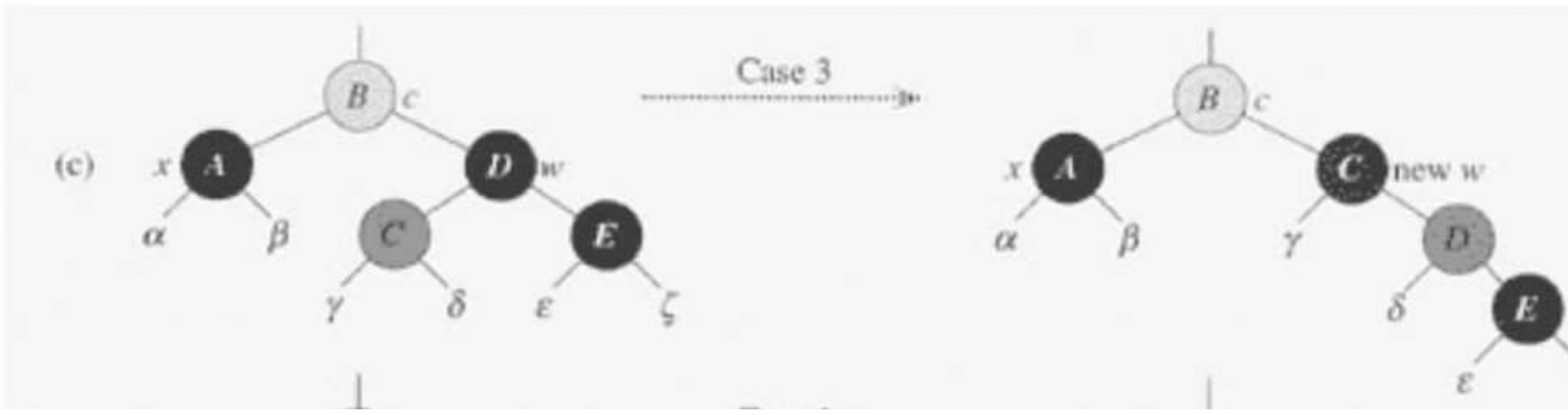
- case 2, 3, 4의 경우 x의 형제노드 w가 BLACK인 경우이다. 이 중에서 w의 자식들도 모두 BLACK인 경우가 Case 2에 해당한다.
- 이 경우, x와 w가 모두 블랙이므로 부모인 B노드는 RED 일수도 있고, BLACK 일수도 있다.
- 현재 x는 double black 노드이고, w는 black 노드이다.
- 이 상황에서 x와 w로부터 black을 하나씩 뺏어서, 부모노드에게 준다.
- 결과적으로 x는 extra black이 하나 없어졌으므로 BLACK노드가 됐고, w는 black을 뺏겼으므로 RED노드가 된다.
- 다음으로 p[x]에게 extra-black 노드를 준다. 이렇게 하면, 트리의 위에서 부터 내려오면서 유지하던 BLACK노드의 갯수가 유지 된다.
- 만약 p[x]가 RED였다면, 위에서 설명했던 것처럼 red & black을 가지고 있는 노드를 BLACK노드로 만들고 끝내면 되고, p[x]가 BLACK이었다면 p[x]를 새로운 x로 해서 계속한다.
- 만약 Case1에서 Case2에 도달한 경우면 p[x]는 red였고, 따라서 새로운 x는 red & black이 되어서 종료된다.
- 하지만 Case2로 바로 온 경우에 p[x]가 원래 BLACK이었다면, p[x]가 double black이 되므로 반복해서 문제를 해결해야 할 수도 있다.(뒤에서 설명) 다만, extra-black이 한 level 올라갔다.



Case 3 - w는 BLACK, w의 왼쪽자식이 RED

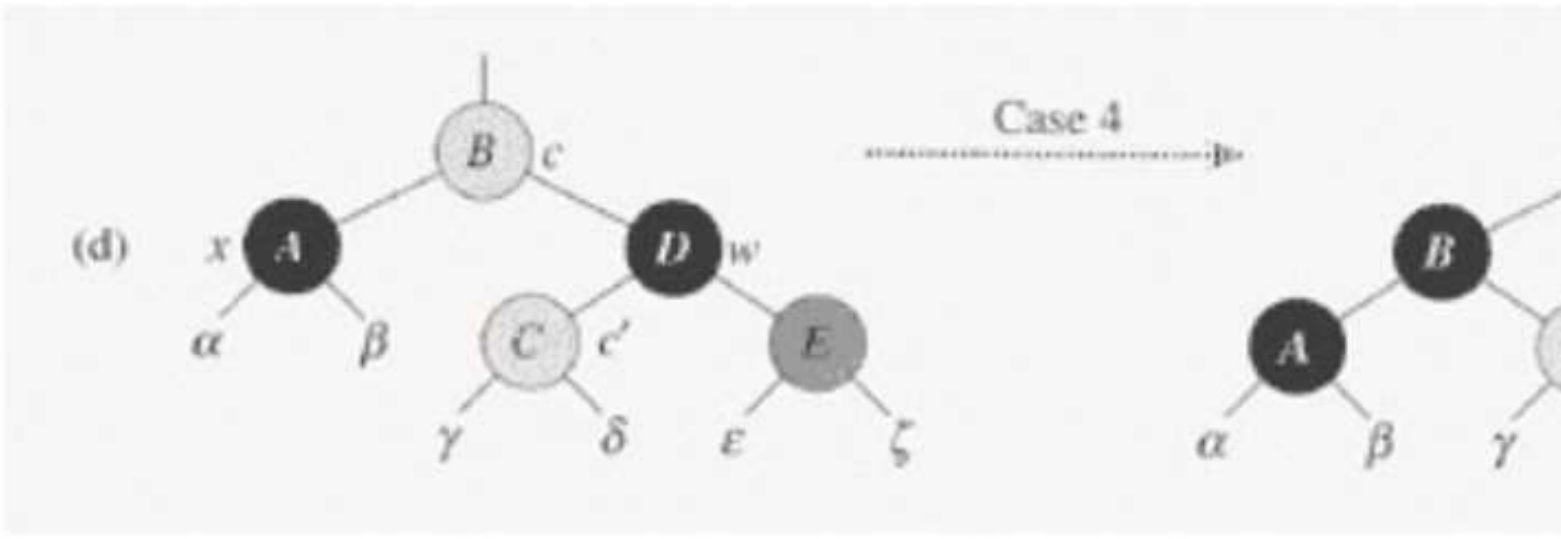
- w를 RED로, w의 왼쪽자식노드를 BLACK으로 바꾼다.

- w에 대해서 right-rotation을 적용한다.
- x의 새로운 형제 w는 오른자식이 RED이다. 이
- 것은 경우 4에 해당한다.



Case 4 - w는 BLACK, w의 오른쪽자식이 RED

- w의 색을 현재 p[x]의 색으로(unknown color)
- p[x]를 BLACK으로, w의 오른자식을 BLACK으로 바꾼다.
- p[x]에 대해서 left-rotation을 적용한다.
- x의 extra-black을 제거하고 종료한다.
- 이렇게 되면, double-black노드가 없어졌음에도 불구하고, 기존의 A노드를 지나는 블랙노드의 갯수가 로테이션 전과 똑같아 진다. 그리고 나머지 C와 E를 지나는 블랙노드의 갯수도 기존과 동일하게 유지된다.



RB DELETE FIXUP(수도코드)

```
RB-DELETE-FIXUP(T, x) 01 while x != root[T] and color[x] = BLACK 02 do if x = left[p[x]]
03 then w <- right[p[x]] 04 if color[w] = RED 05 then color[w] <- BLACK // Case1 06
color[p[x]] <- RED // Case1 07 LEFT-ROTATE(T, p[x]) // Case1 08 w <- right[p[x]] //
Case1 09 if color[left[w]] = BLACK and color[right[w]] = BLACK 10 then color[w] <- RED
// Case2 11 x <- p[x] // Case2 12 else if color[right[w]] = BLACK 13 then color[left[w]]
<- BLACK // Case3 14 color[w] <- RED // Case3 15 RIGHT-ROTATE(T, w) // Case3 16 w <-
right[p[x]] // Case3 17 color[w] <- color[p[x]] // Case4 18 color[p[x]] <- BLACK //
Case4 19 color[right[w]] <- BLACK // Case4 20 LEFT-ROTATE(T, p[x]) // Case4 21 x <-
root[T] // Case4 22 else (same as then clause with "right" and "left" exchanged) 23
color[x] <- BLACK
```

- 레드블랙트리에서 실제로 삭제한 노드는 y이다. 삭제한 노드 y의 자식인 x를 넘겨주면서 Delete Fixup을 하게 된다.
- 01 : 만약 x가 루트노드이거나, x가 레드노드라면 While문을 빠져나가서 x를 BLACK으로 만들어 주고 종료하면 된다.
- 02 : While문 안에서는 크게 둘로 나뉘어지게 된다. 만약 x가 x의 부모노드의 왼쪽 자식이라면 Case 1, 2, 3, 4에 해당하며 부모노드의 오른쪽 자식이라면 Case 5, 6, 7, 8에 해당한다.

- 03 : 노드 x의 형제노드인 w를 저장한다. x가 p[x]의 왼쪽 자식이므로, w는 오른쪽 자식 노드가 된다.
- 04 : 형제 노드인 w노드가 RED인 경우가 Case 1에 해당한다.
- 05 - 08 : Case1의 경우 w노드를 BLACK으로 만들고, p[x]노드를 RED로 만든 다음, p[x]를 기준으로 LEFT-ROTATE한다. 새로운 w는 p[x]의 right가 되므로 새로 저장한다. 이때, 새로운 w노드는 BLACK 이므로(p[x]가 RED로 변경됨) 다시 While문으로 들어왔을 때, Case2, 3, 4로 가게 된다.
- 09 : Case 2, 3, 4를 구분하게 된다. w의 왼쪽, 오른쪽 자식노드가 둘다 BLACK인 경우 Case2에 해당한다.
- 10 - 11 : 위에서 학습한 내용과 같이 w와 x로 부터 black을 하나씩 뺏어서 부모 노드에게 전달한다. 그렇게 하기 위해서 w는 RED노드가 되고, p[x]를 x로 만들어 준다. p[x]가 RED였다면 다시 While문을 돌지 않고 x를 RED로 만든 뒤에 종료하면 되고, p[x]가 BLACK이었다면 x를 p[x]로 놓고 double-black 노드가 된 x를 다시 반복해서 처리해주면 된다.
- 12 : w의 오른쪽 자식이 BLACK이고, 왼쪽 자식이 RED인 경우 Case 3에 해당한다.
- 13 - 16 : RIGHT-ROTATE의 대상인 두 노드의 색을 exchange 하고(w를 RED로, w의 왼쪽 자식노드를 BLACK으로 바꾸고), w를 기준으로 RIGHT-ROTATE 한다. 이렇게 되면 w는 p[x]의 새로운 오른쪽 자식노드가 되고, 그것의 색은 RED이다. 그리고 Case 4로 바로 넘어 간다.
- 17 - 20 : LEFT-ROTATE의 대상인 두 노드의 색을 exchange 하고(w의 색을 p[x]의 색으로, p[x]를 BLACK으로) w의 오른쪽 자식노드의 색을 BLACK으로 바꾼다. 그 다음 p[x]를 기준으로 LEFT-ROTATE를 수행한다.
- 21 : x라는 포인터 변수를 root[T]로 변경하여 Case 4가 끝나면 while문이 종료되도록 한다. 실제 트리에는 변화가 없다. 트리의 루트가 변한것도 아니다.
- 22 : Case 5, 6, 7, 8을 대칭적으로 처리한다.
  - x가 p[x]의 오른쪽 자식인 경우이다.
- 23 : 트리의 루트의 색을 BLACK으로 변경하는 것은 언제 해도 문제가 되지 않는다.

시간복잡도

- BST에서의 DELETE : O(logn)
- RB-DELETE-FIXUP : O(logn)
  - 가장 최악의 경우인 Case2와 6이 반복되는 경우에도 최대 트리의 높이만큼 실행된다.
- 따라서, DELETE와 FIXUP을 합쳐도 O(logn)의 시간이 된다.