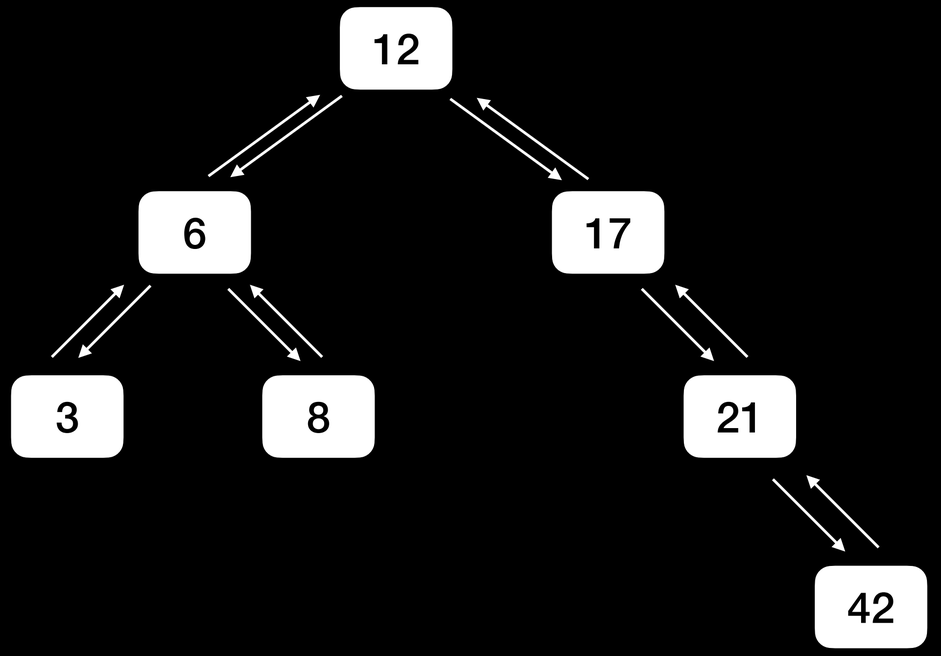
RB-tree

2022.04.23
Yongjule

Binary Search Tree

Binary Search Tree
이진 탐색 트리

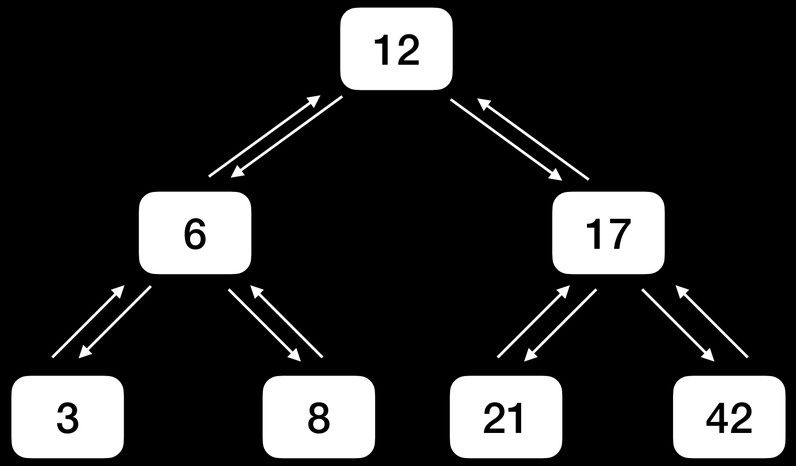


- 현재 노드보다 더 작으면 왼쪽, 크면 오른쪽으로 이동
- 다음 노드가 NULL이면 그 자리에 삽입
- 중위 순회를 통하여, 가장 작은 수부터 탐색 가능

$$= O(log_2 n * n)(= O(h * n))$$

- 하지만, 경우에 따라 $h = n$ 이 되는 한계가 있음.
- 이를 방지하기 위한 트리 - AVL-tree, RB-tree!

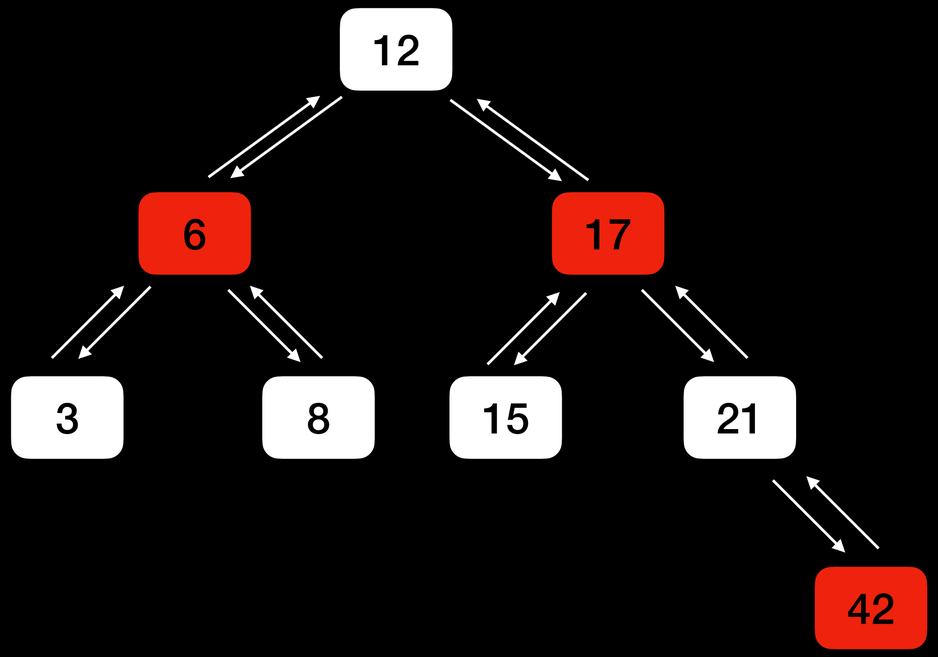
AVL Tree
AVL 트리



- 항상 완전이진트리 형태를 유지하는 트리
- 삽입 및 삭제시 balance가 깨지면, 적절한 회전을 통하여 항상 완전이진트리 형태를 유지한다.
- 항상 $h = log_2 n$ 을 유지하여 탐색 속도가 안정적임.
- 하지만 노드의 삽입/삭제 시 balance를 유지하기 위한 연산 횟수가 너무 많은 문제가 있음.

Red-Black Tree

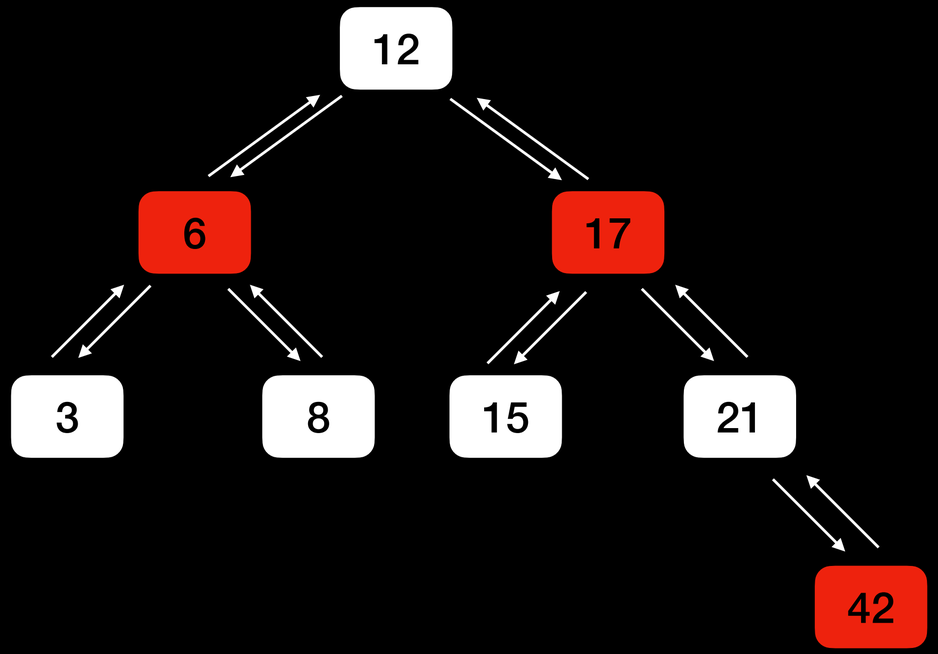
Red-Black Tree
레드 블랙 트리



- 트리에 RED, BLACK 성질을 부여하고, 적절한 규칙을 통하여 거의 균형이 맞는 (approximately balanced) 트리
- 밸런스가 너무 깨지지도 않으면서 AVL의 단점을 보완한 트리.
- 로직이 너무 복잡하다는 단점이 존재함.

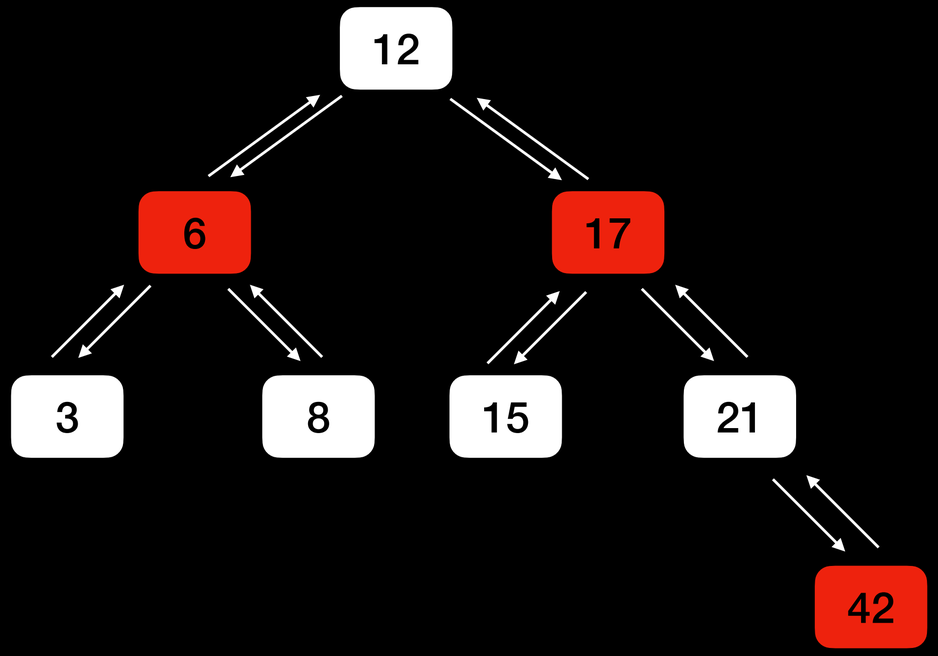
Properties of Red-Black Tree

레드 블랙 트리의 성질



1. 모든 노드는 red 이거나 black 이다.
2. root 노드는 black 이다.
3. 모든 leaf(NIL) 는 black이다.
4. 만약 한 노드가 red 이면 자식 노드(들)은 black 이다.
5. 각각 노드에서, 그 노드로부터 leaves 까지의 simple path들은 같은 수의 black 노드를 지난다.

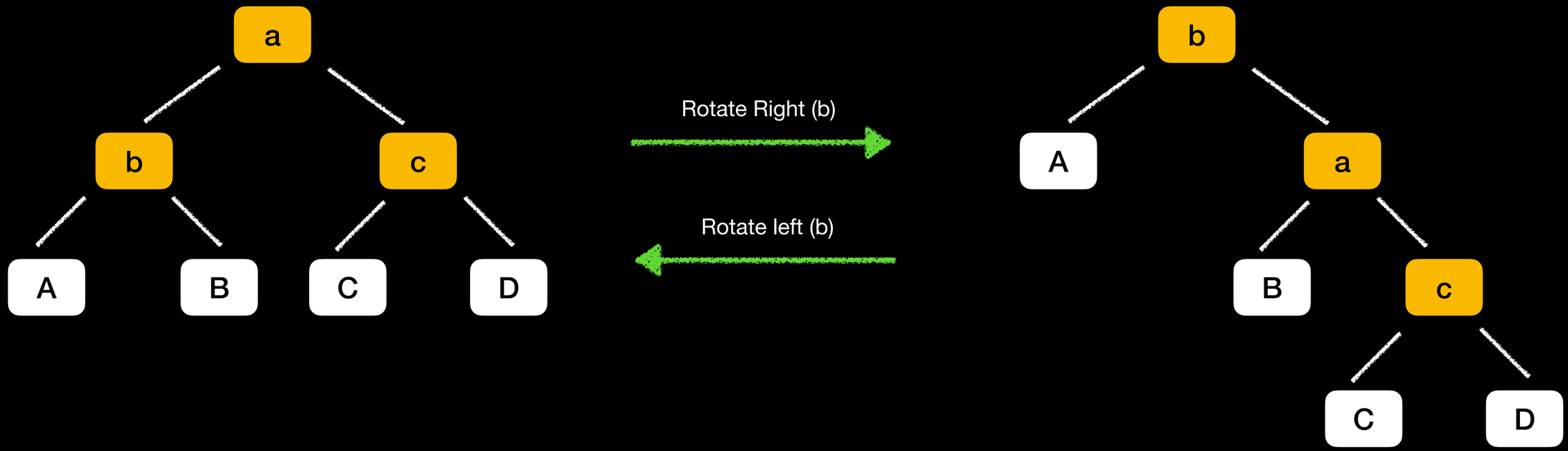
Properties of Red-Black Tree
레드 블랙 트리의 성질



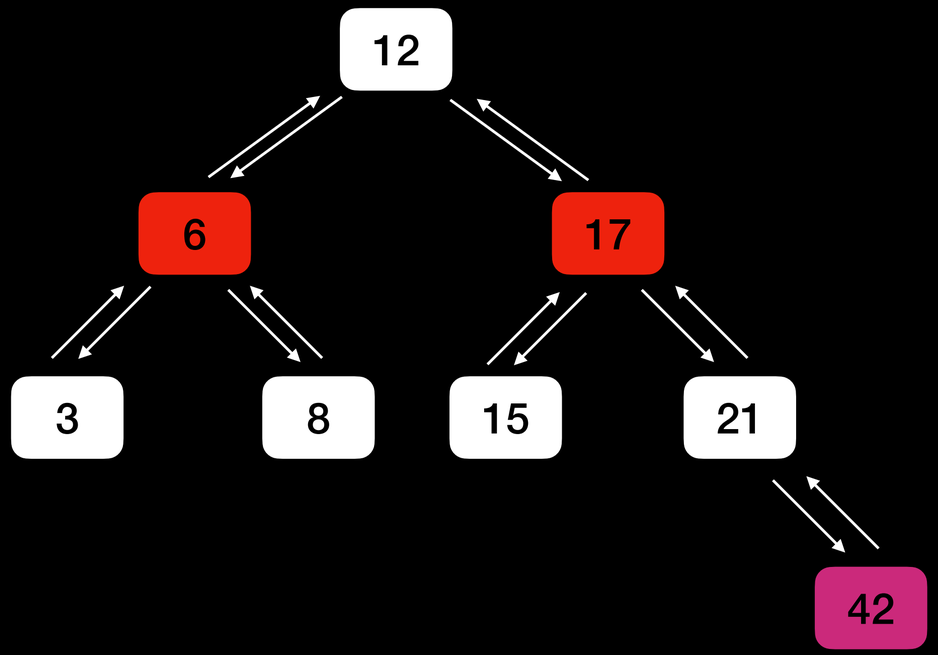
- 레드블랙 트리의 조건을 만족한다면,
  - $h \leq 2log_2(n+1)$이 항상 성립한다.(증명)
  - $h_{max} - h_{min} \leq (black\ height)$ 가 항상 성립한다.
- 삽입 / 삭제 후 적절한 rotation 및 color 설정을 통하여 RB-tree 성질 유지!

Rotation of Binary Tree

이진 트리의 회전



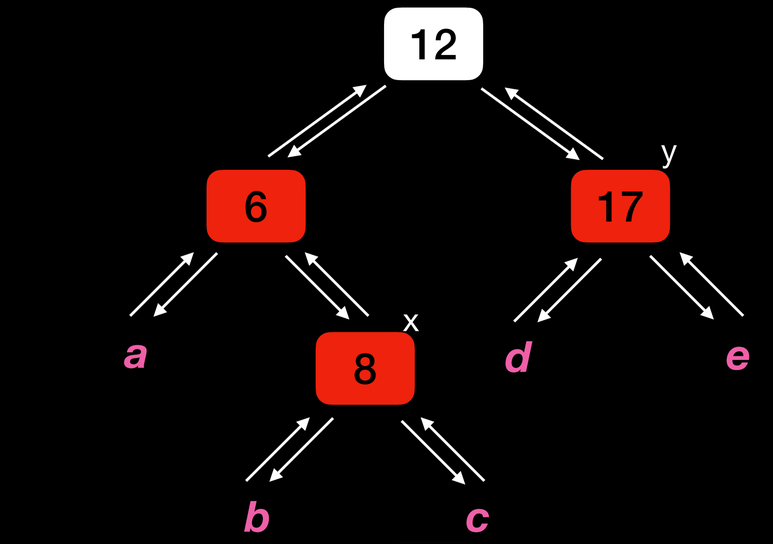
Insertion of Red-Black Tree
레드 블랙 트리의 삽입

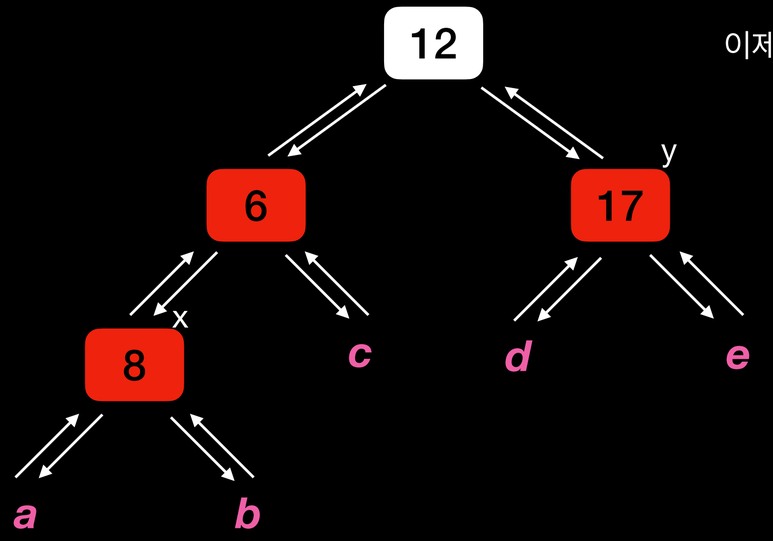


- Binary-Tree 규칙에 따라 적절한 leaf 위치에 새로운 노드를 삽입하고, 삽입한 노드는 RED로 설정한다.
- 부모 노드가 BLACK이면, RB-tree의 성질이 깨지지 않기 때문에, 삽입만 하고 종료된다.
- 부모 노드가 RED인 경우, RED-RED형태로 RB-tree의 성질이 깨지기 때문에, 적절한 회전과 색깔 변환을 통해 RB-tree의 성질을 맞춰주어야 한다.
- 이 과정에서 부모 노드의 색깔이 변경될 수 있는데, 이는 조부모까지 영향을 미치므로 uncle 노드의 색깔을 잘 맞춰주어야 한다.

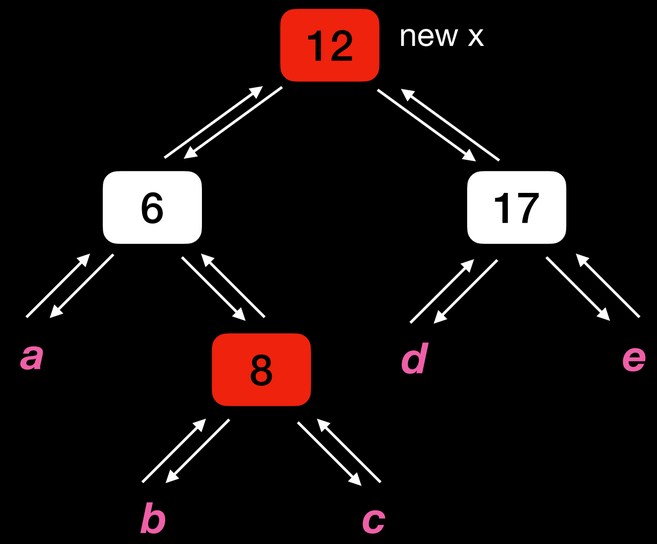
Insertion of Red-Black Tree

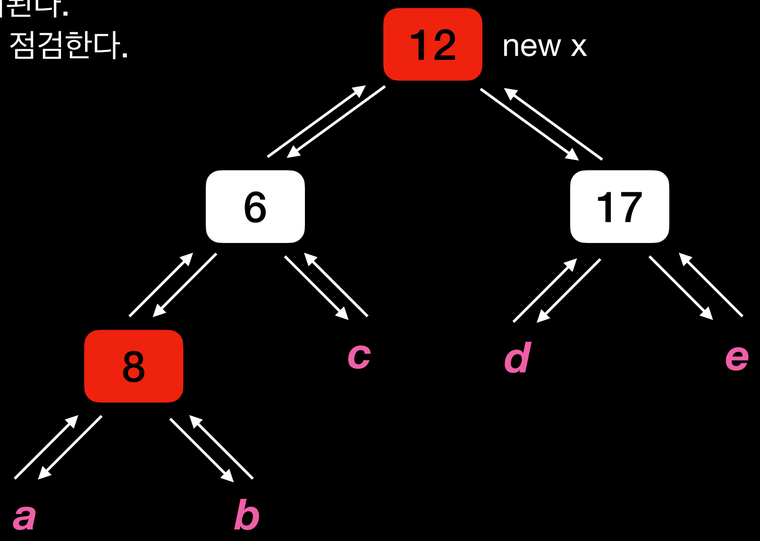
case 1: x의 uncle y가 RED





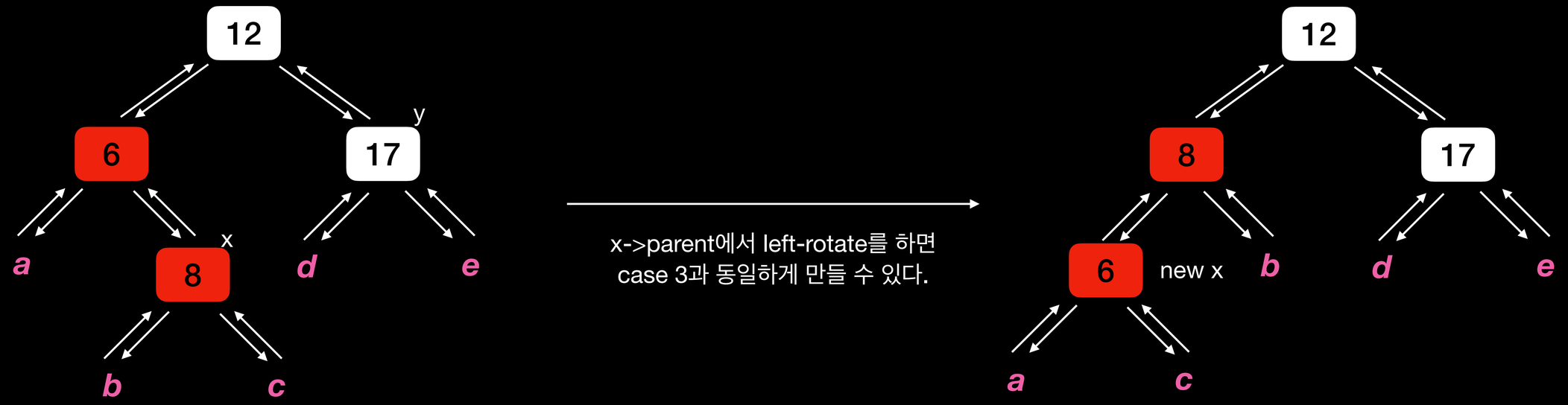
적절히 색깔을 바꾸면 sub-tree내의 규칙은 잘 유지된다.
이제 x->parent->parent를 새로운 x로 하여 rb-tree를 점검한다.





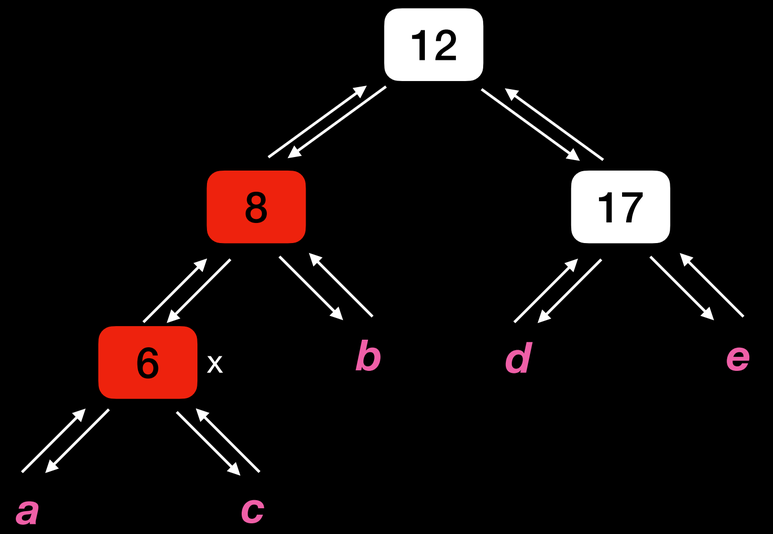
Insertion of Red-Black Tree

case 2: x의 uncle y가 BLACK이고 x가 right child

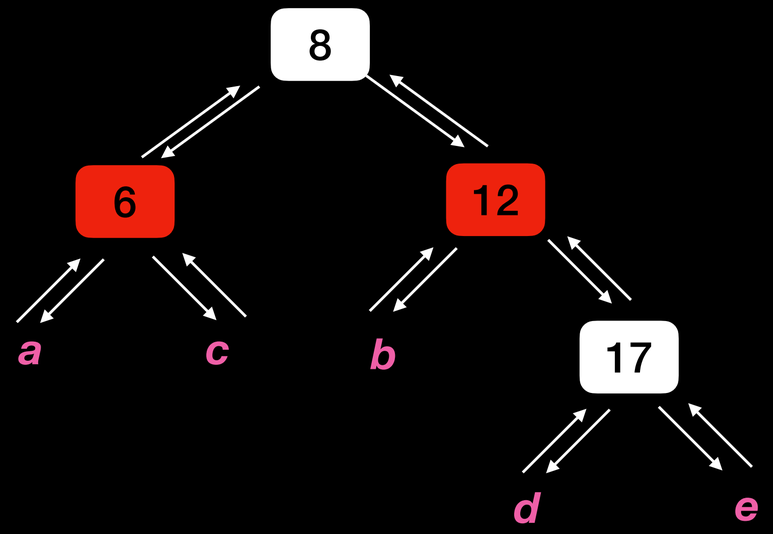


Insertion of Red-Black Tree

case 3: x의 uncle y가 BLACK이고 x가 left child



x->parent->parent의 color를 red로 바꾸고
x->parent의 color를 black로 바꾼 뒤
x->parent에서 right-rotate를 하면 된다.

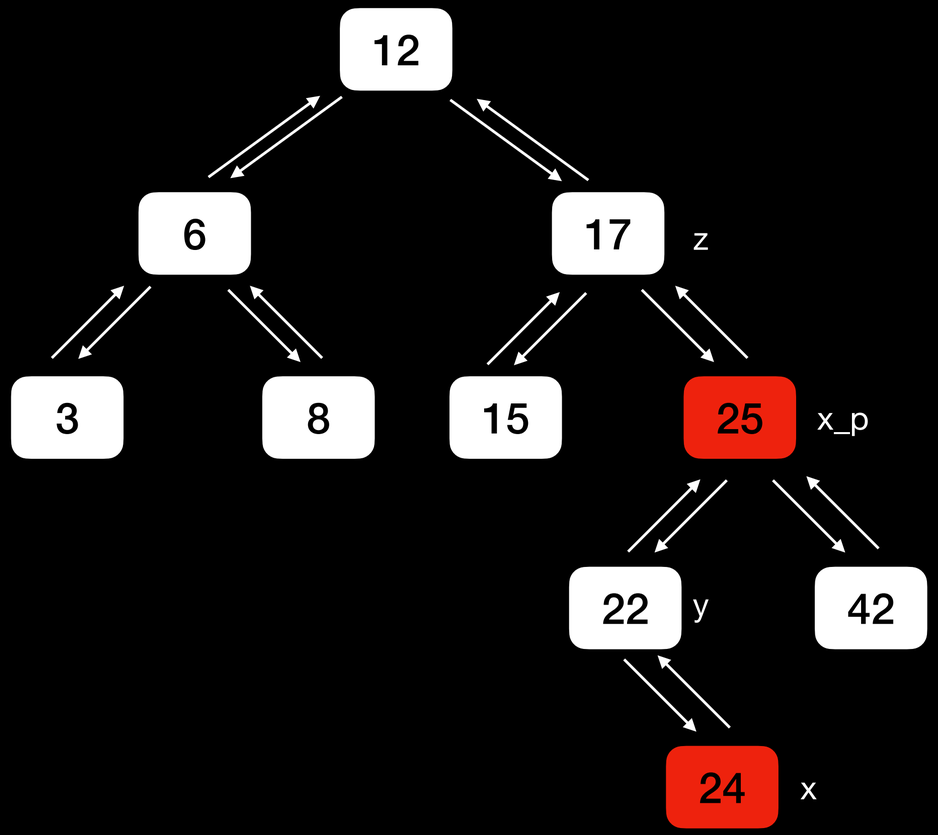


Insertion of Red-Black Tree
레드 블랙 트리의 삽입

- x의 parent가 x->parent->parent의 right인 경우 case 1, 2, 3과 대칭인 구조라서, rotation만 반대로 해주면 된다.
- 이 모든 과정을 x가 루트이거나, x->parent의 color가 BLACK일때까지 반복하면 삽입 과정이 종료된다.

Delete of Red-Black Tree

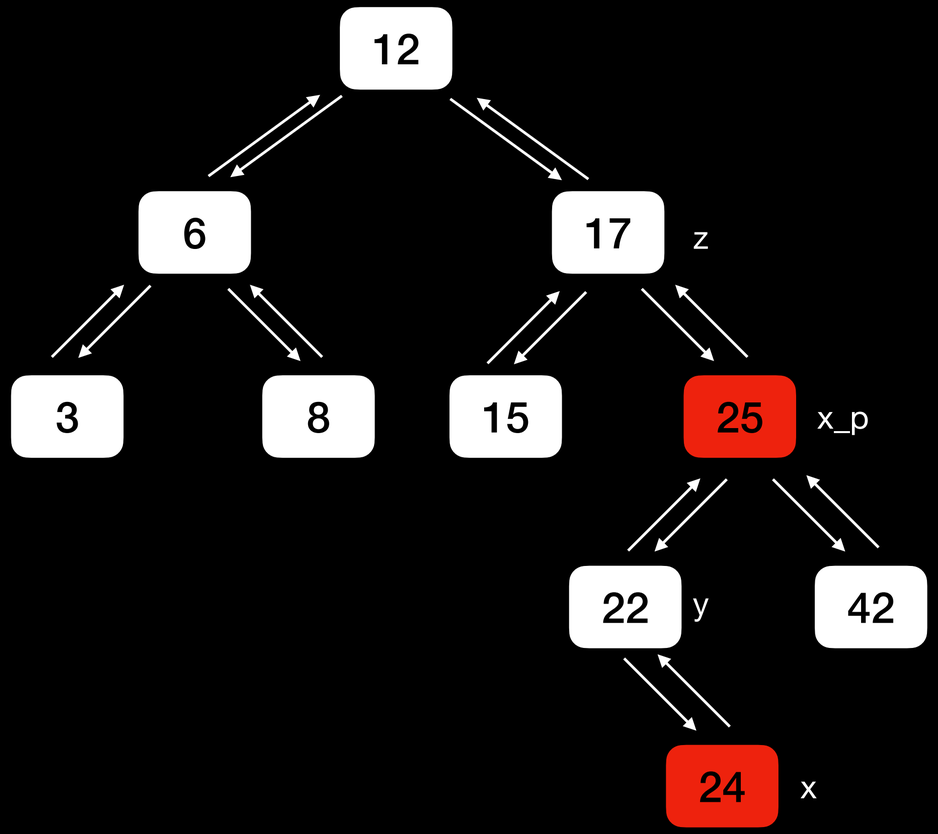
레드 블랙 트리의 삭제

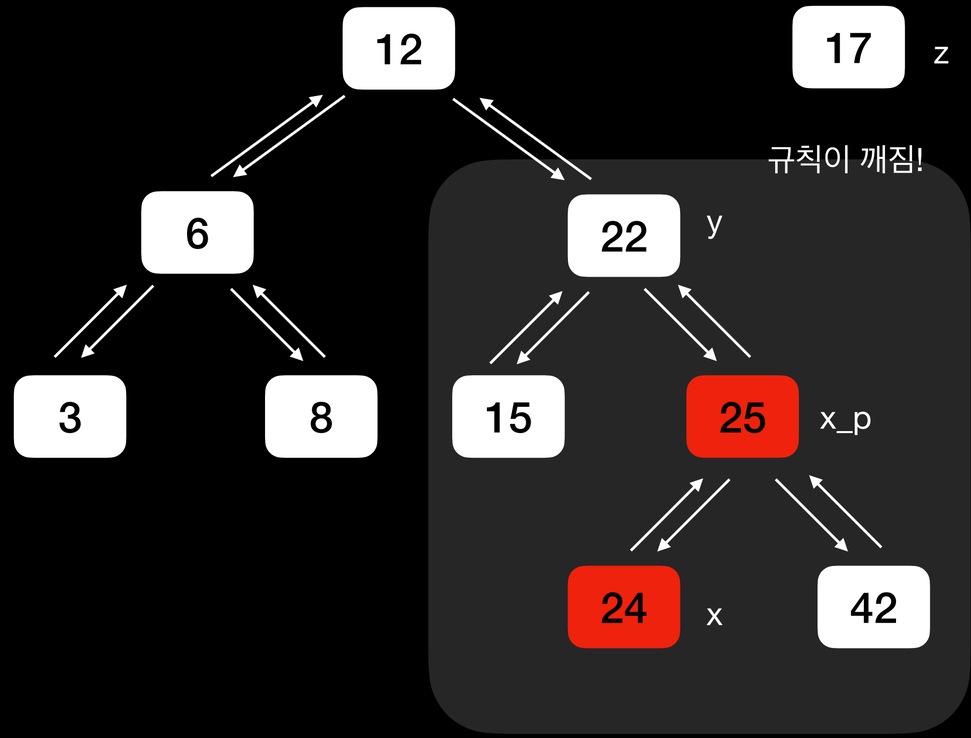


- z의 자식 노드가 2개라면, z의 successor인 y가 z의 위치에 오면 된다.
- 삭제할 노드 = z, z의 successor = y, y->right = x 라 하자. (y->left == NULL)
- y가 z자리에 올 수 있도록 잘 연결 해주고, 비어있는 y자리는 x가 대체할 수 있도록 연결해줌.
- y를 z에 연결 후 y와 z의 색깔을 swap하면 삭제된 자리에선 RB-tree의 규칙이 유지됨.
- 만약 y가 RED면 fixup 과정이 필요없다.

Delete of Red-Black Tree

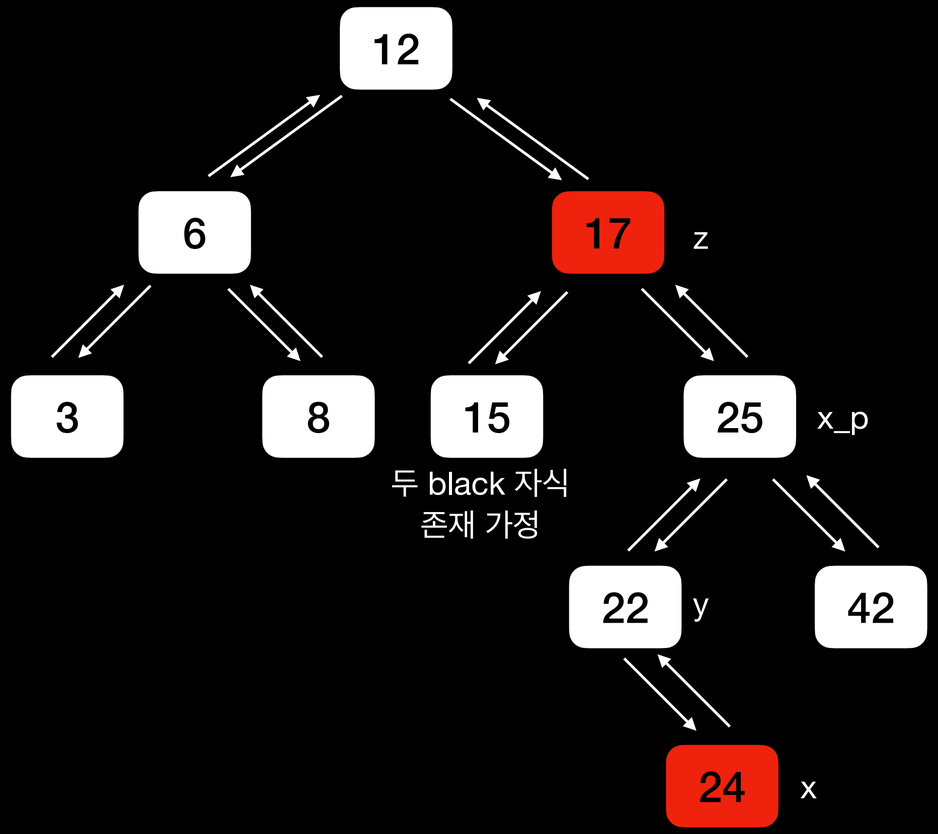
레드 블랙 트리의 삭제

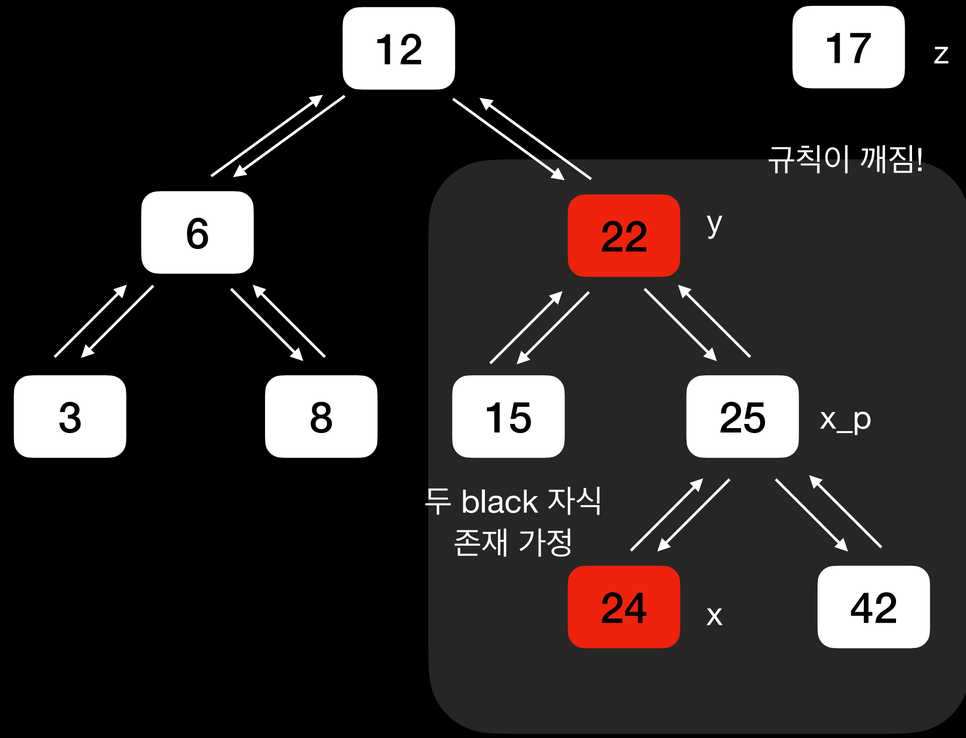




Delete of Red-Black Tree

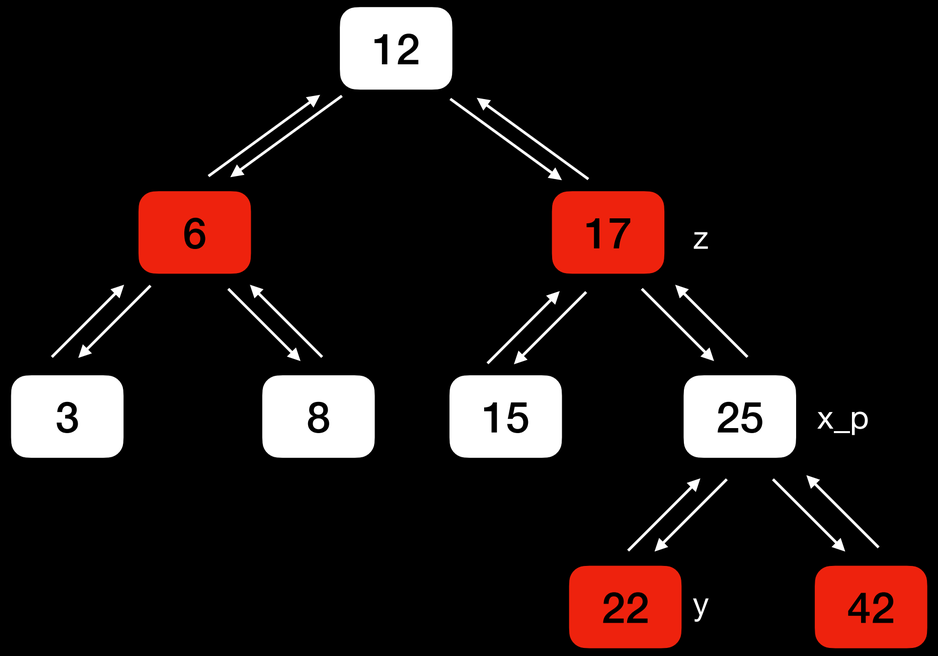
레드 블랙 트리의 삭제



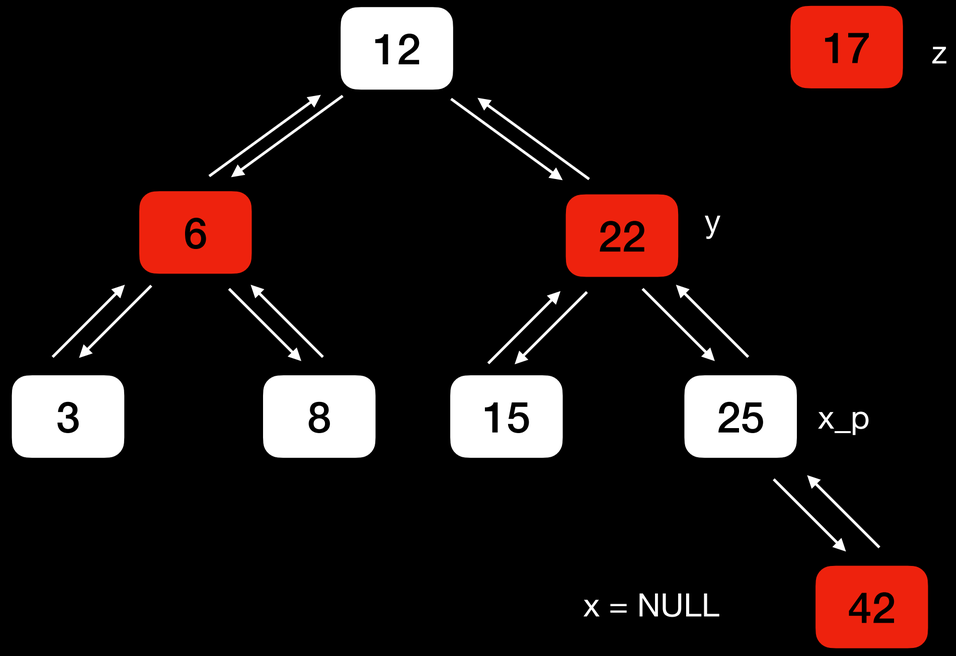


Delete of Red-Black Tree

레드 블랙 트리의 삭제

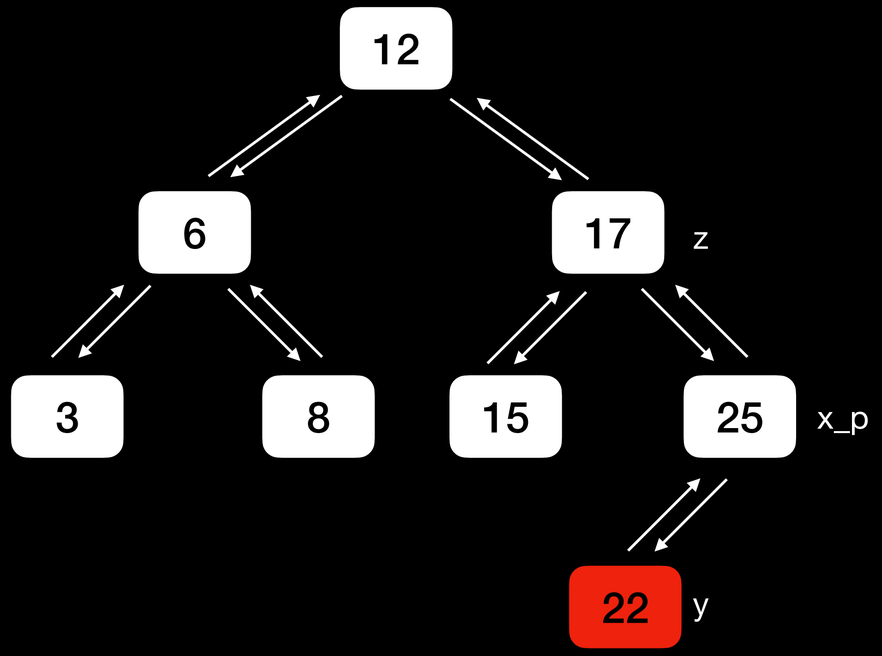


x = NULL

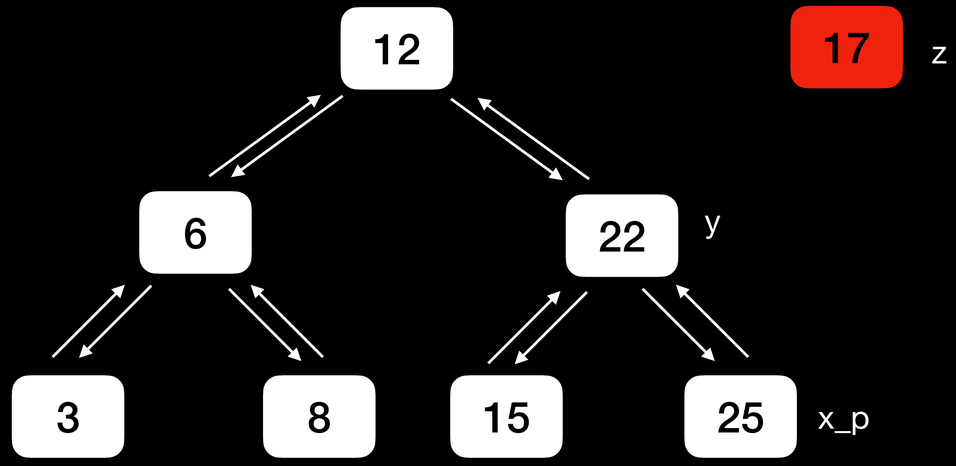


Delete of Red-Black Tree

레드 블랙 트리의 삭제



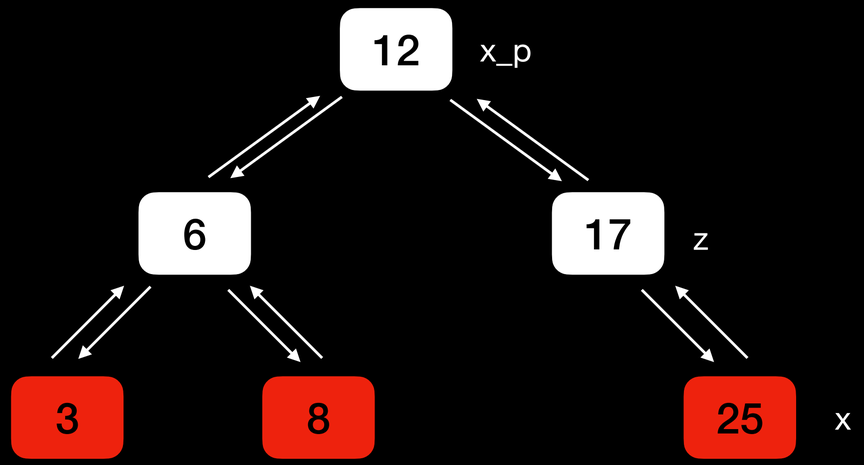
x = NULL



x = NULL

Delete of Red-Black Tree

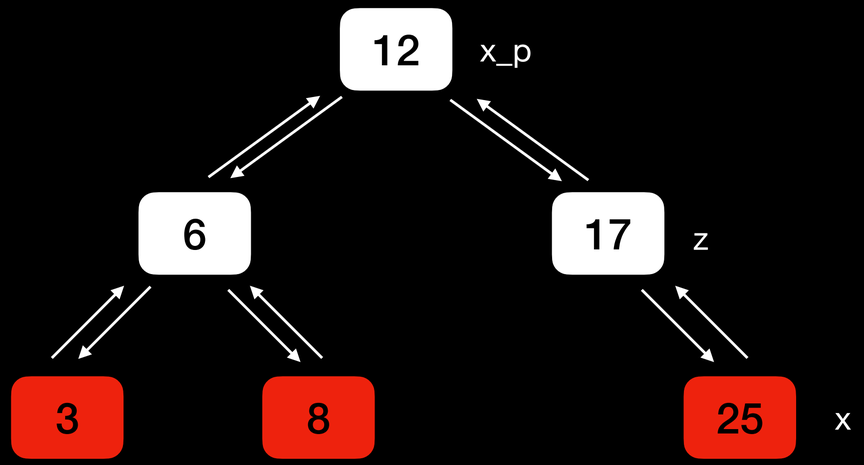
레드 블랙 트리의 삭제

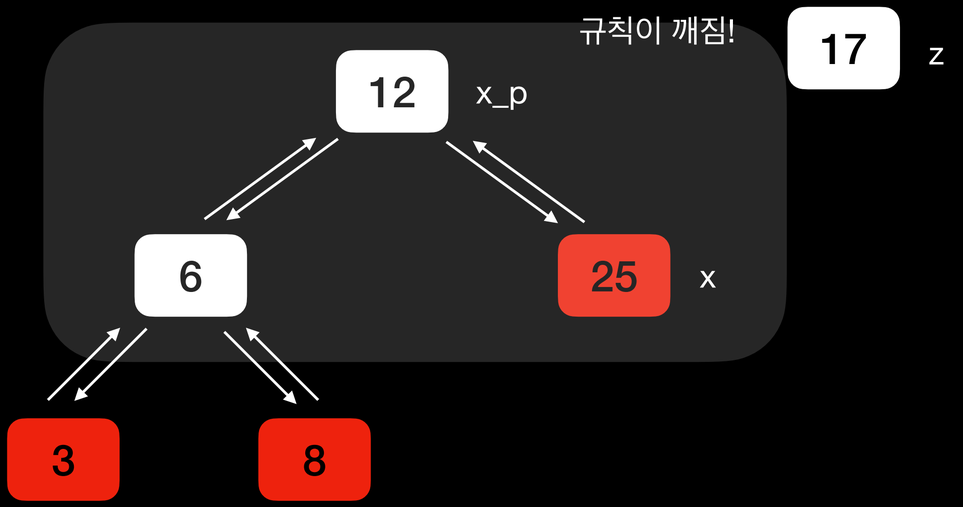


- z의 자식 노드가 1개 이하라면, 그냥 그 자식노드를 z의 위치로 옮겨주면 된다.
- z (== y)의 NULL이 아닌 자식을 x라 하고 (둘 다 NULL이면 NULL) x를 z위치에 잘 연결해주면 된다.
- 이때, leftmost나 rightmost값이 변할 수 있으므로 헤더에 잘 연결해준다.
- z의 두 자식 모두 NULL이면 그냥 삭제해주면 된다.
- z(== y)의 색깔이 RED이면 fixup 과정이 필요없다.

Delete of Red-Black Tree

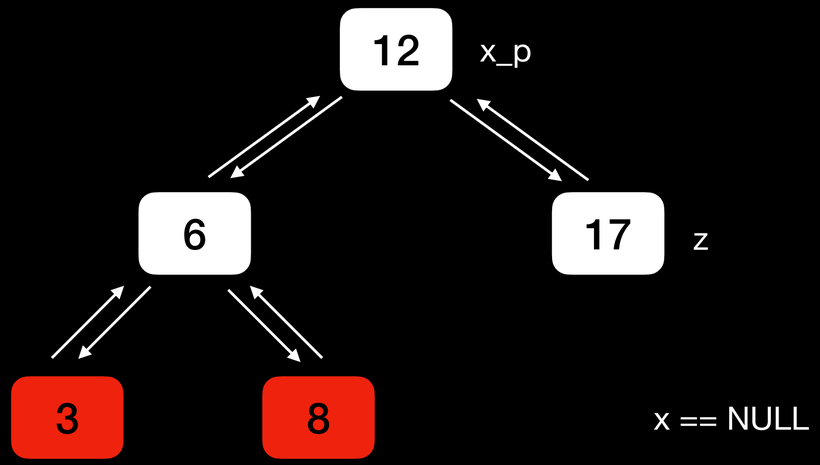
레드 블랙 트리의 삭제

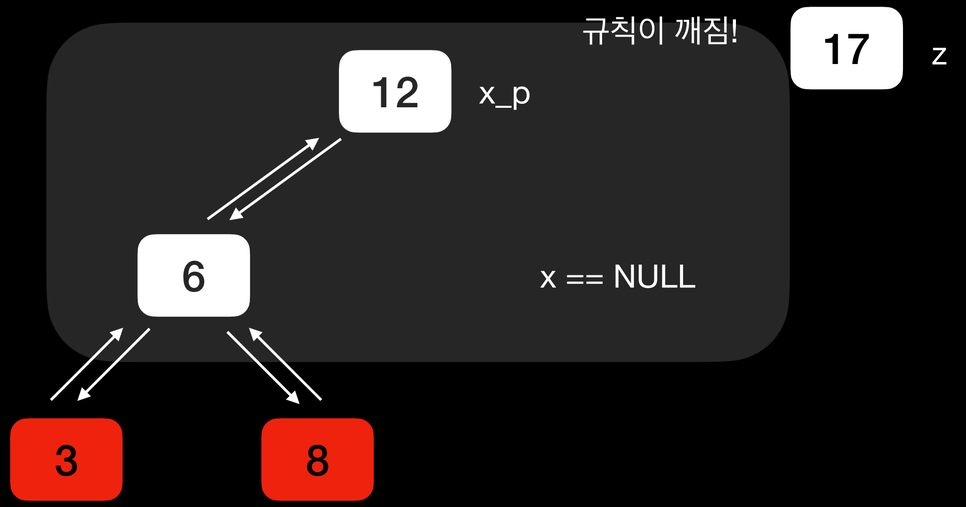




Delete of Red-Black Tree

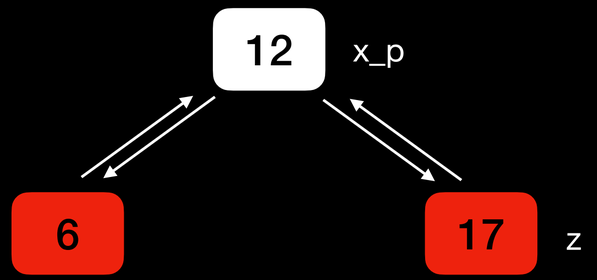
레드 블랙 트리의 삭제



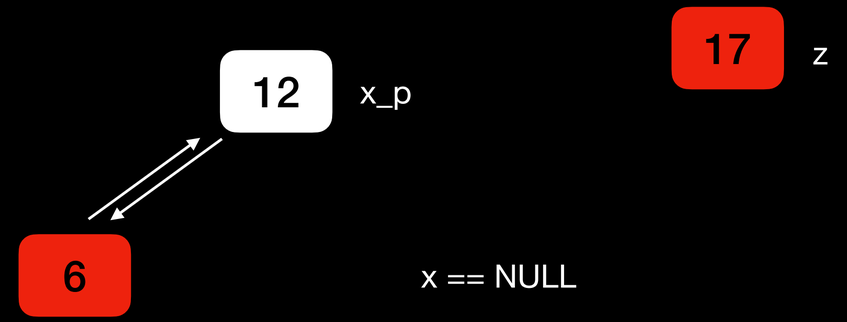


Delete of Red-Black Tree

레드 블랙 트리의 삭제



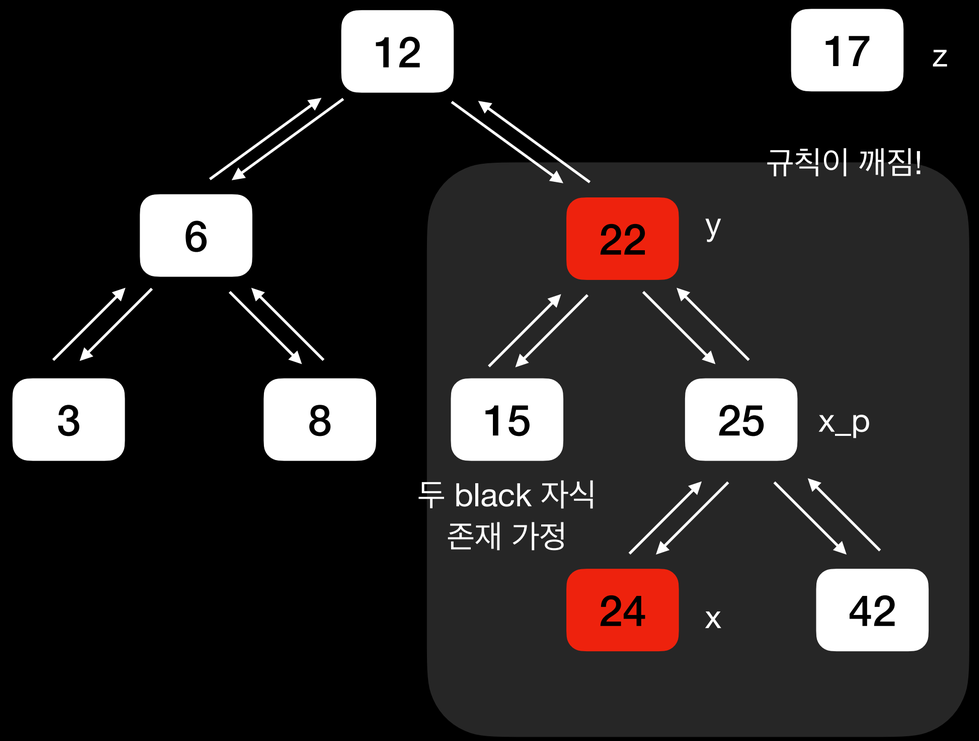
x == NULL



x == NULL

Fixup of Red-Black Tree Delete

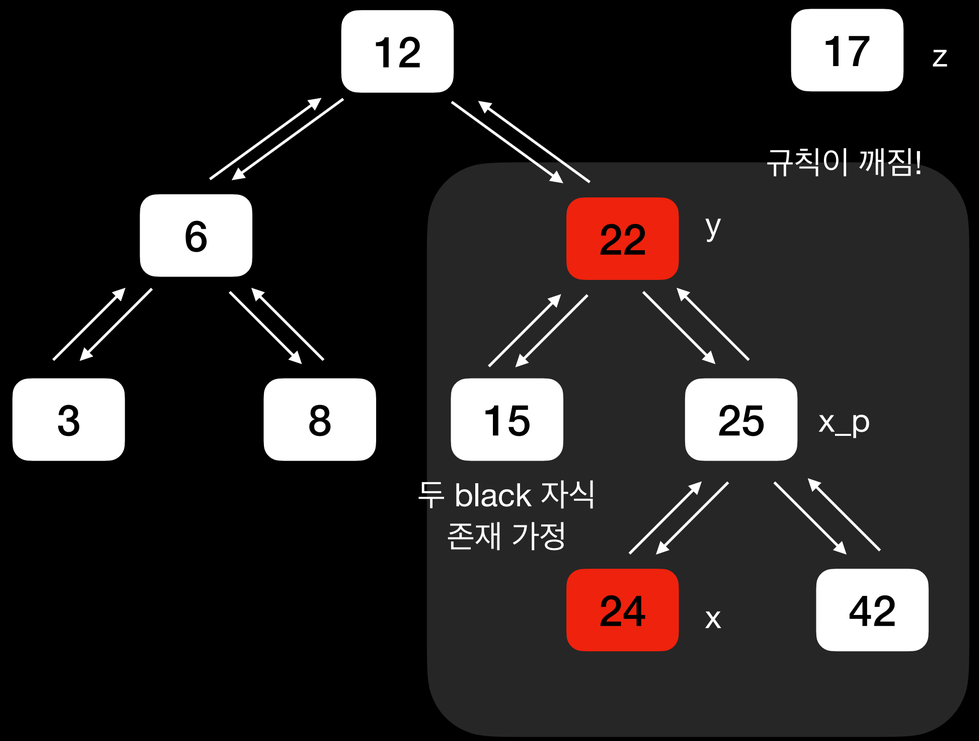
레드 블랙 트리의 삭제 후 재정렬



- y가 BLACK이었으면 규칙이 깨짐.
  1. y가 root였고, y의 RED 자식 x가 새로운 root가 된다면 규칙 위반
  2. x와 x->parent가 RED면 규칙 위반
  3. y의 빈 자리를 차지한 x를 포함하는 simple path는 BLACK이 하나 부족하므로 규칙 위반
- x는 RED와 BLACK 모두 될 수 있지만, BLACK이 하나 부족한 상태임.
- x가 BLACK이며 BLACK이 부족한 상황을 doubly-black, RED이며 BLACK이 부족한 상황을 red-and-black라고 하자.

Fixup of Red-Black Tree Delete

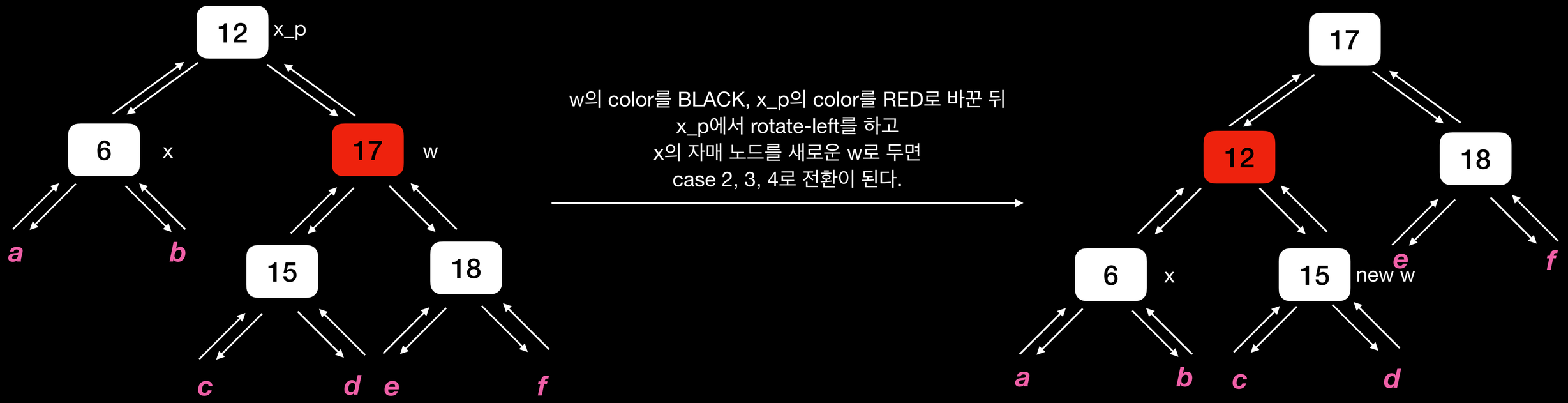
레드 블랙 트리의 삭제 후 재정렬



- red-and-black이면 x를 BLACK으로 바꿔주면 됨.
- x가 doubly-black인 동안 루프를 돌며 red-and-black이 되면 루프를 빠져나가 BLACK로 만들어주면 됨.
- 적절한 회전과 색상변경을 통하여 rb-tree 성질을 만족한다면 break하면 됨.

Fixup of Red-Black Tree Delete

case 1: x의 sibling 노드 w 가 RED

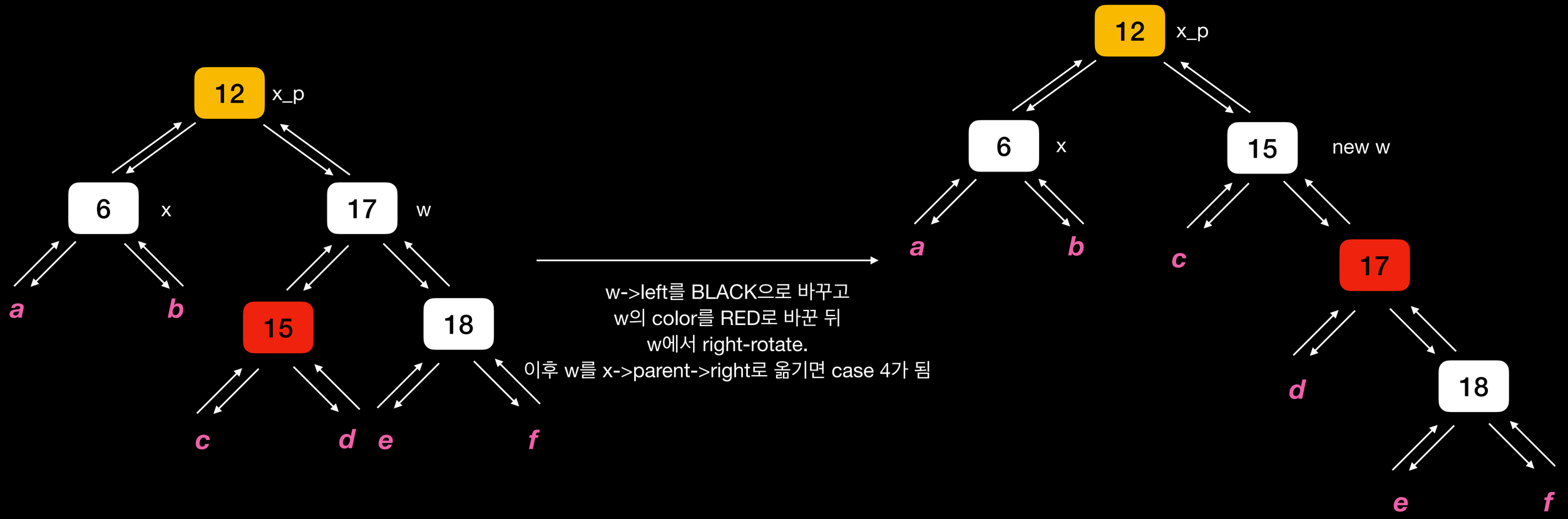


Fixup of Red-Black Tree Delete

case 2: x의 sibling 노드 w가 BLACK이고 w의 두 자식 노드가 BLACK

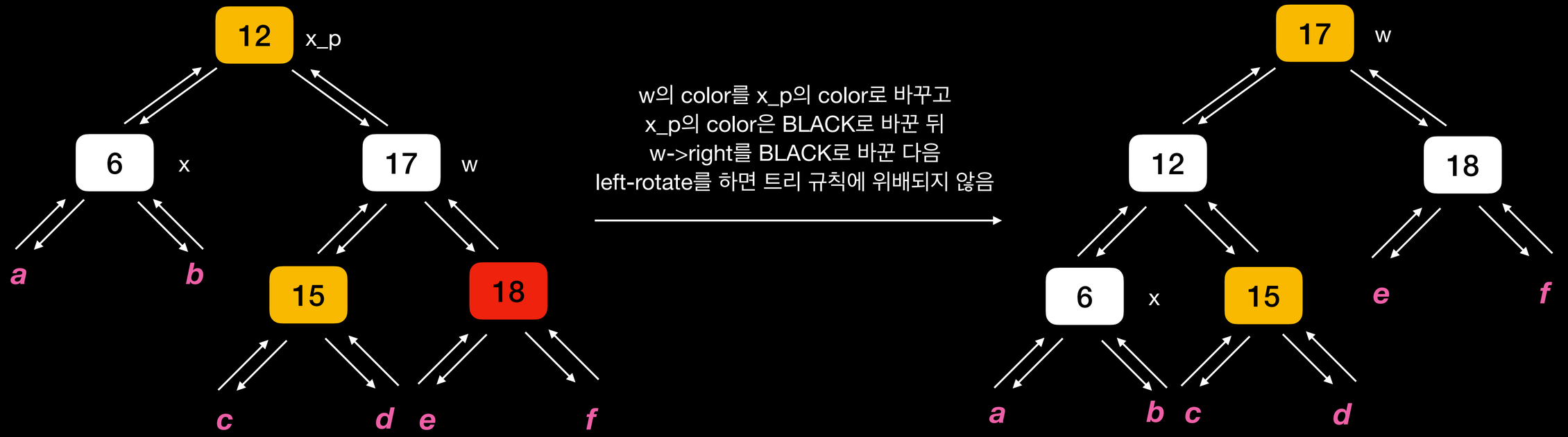
Fixup of Red-Black Tree Delete

case 3: x의 sibling 노드 w가 BLACK이고 w의 left는 RED, w의 right는 BLACK



Fixup of Red-Black Tree Delete

case 4: x의 sibling 노드 w가 BLACK이고 w의 right 노드가 RED



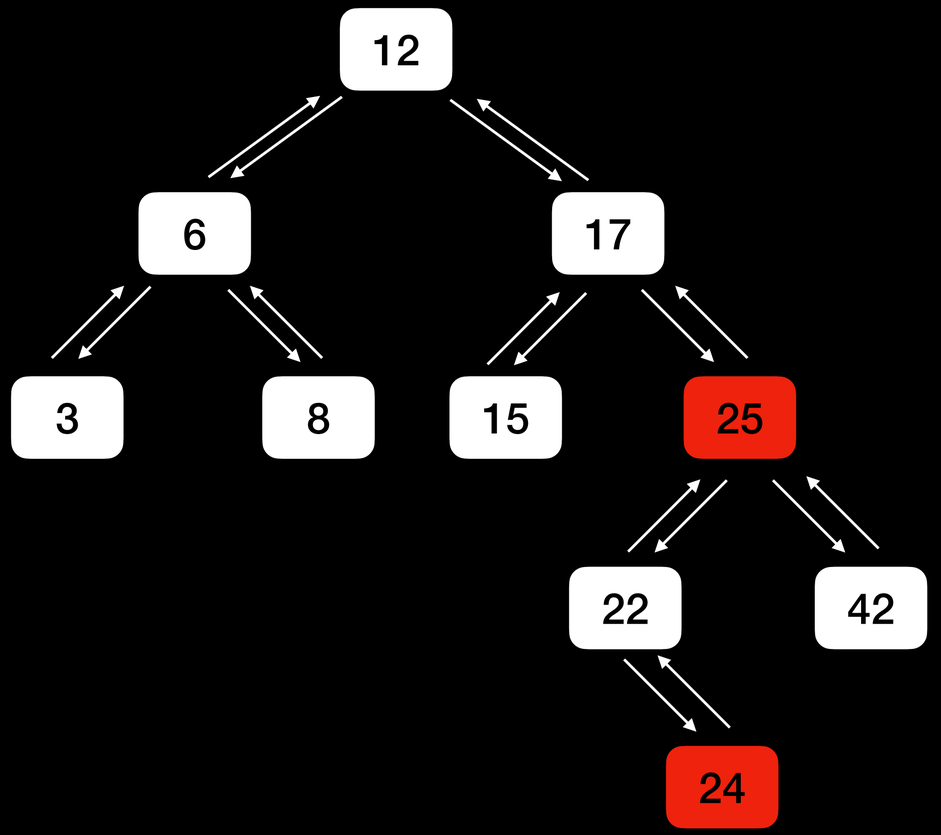
Fixup of Red-Black Tree Delete
레드 블랙 트리의 삭제

- z의 parent가 x->parent의 right인 경우 case 1, 2, 3, 4과 대칭인 구조라서, rotation만 반대로 해주면 된다.

Red-Black Tree for containers

Red-Black Tree for containers

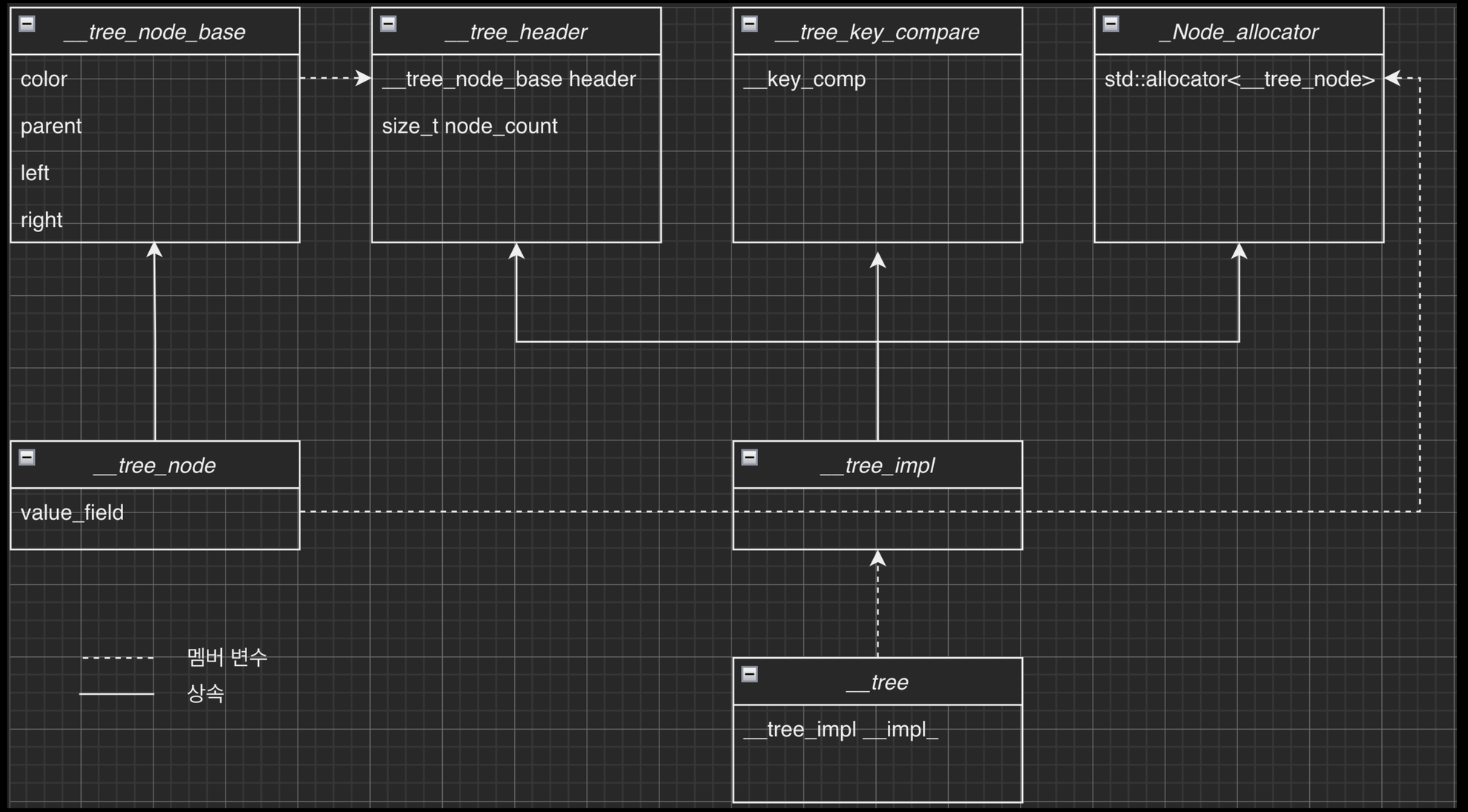
컨테이너를 위한 RB-Tree



- RB-Tree의 빠른 검색, 삽입, 삭제라는 이점을 살리면서 컨테이너의 특징들도 잘 반영 되어야 함.
- Generic한 타입을 저장할 수 있어야 하며 iterator를 통하여 빠른 순회도 가능해야함.

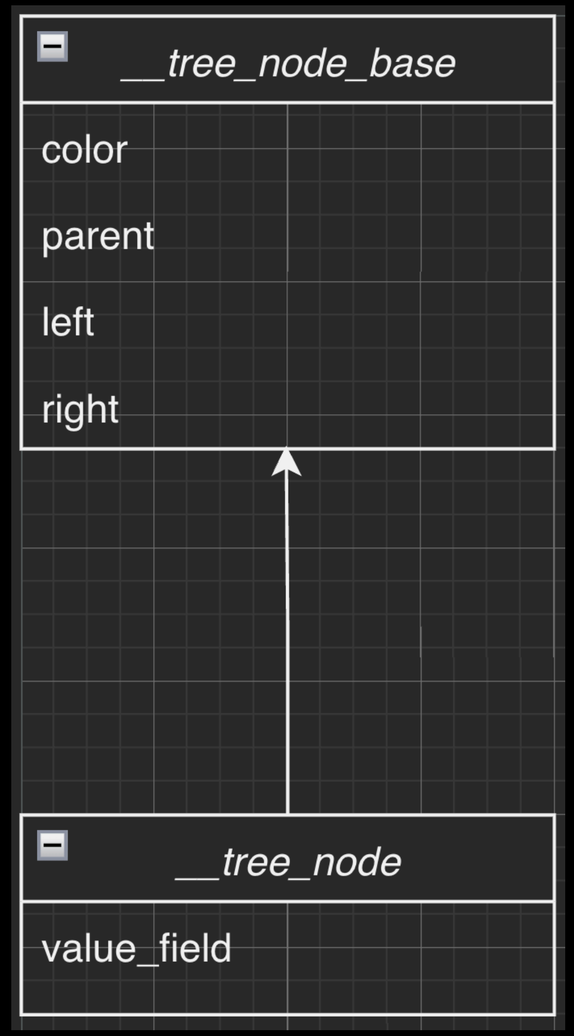
Red-Black Tree for containers

컨테이너를 위한 RB-Tree



Red-Black Tree for containers

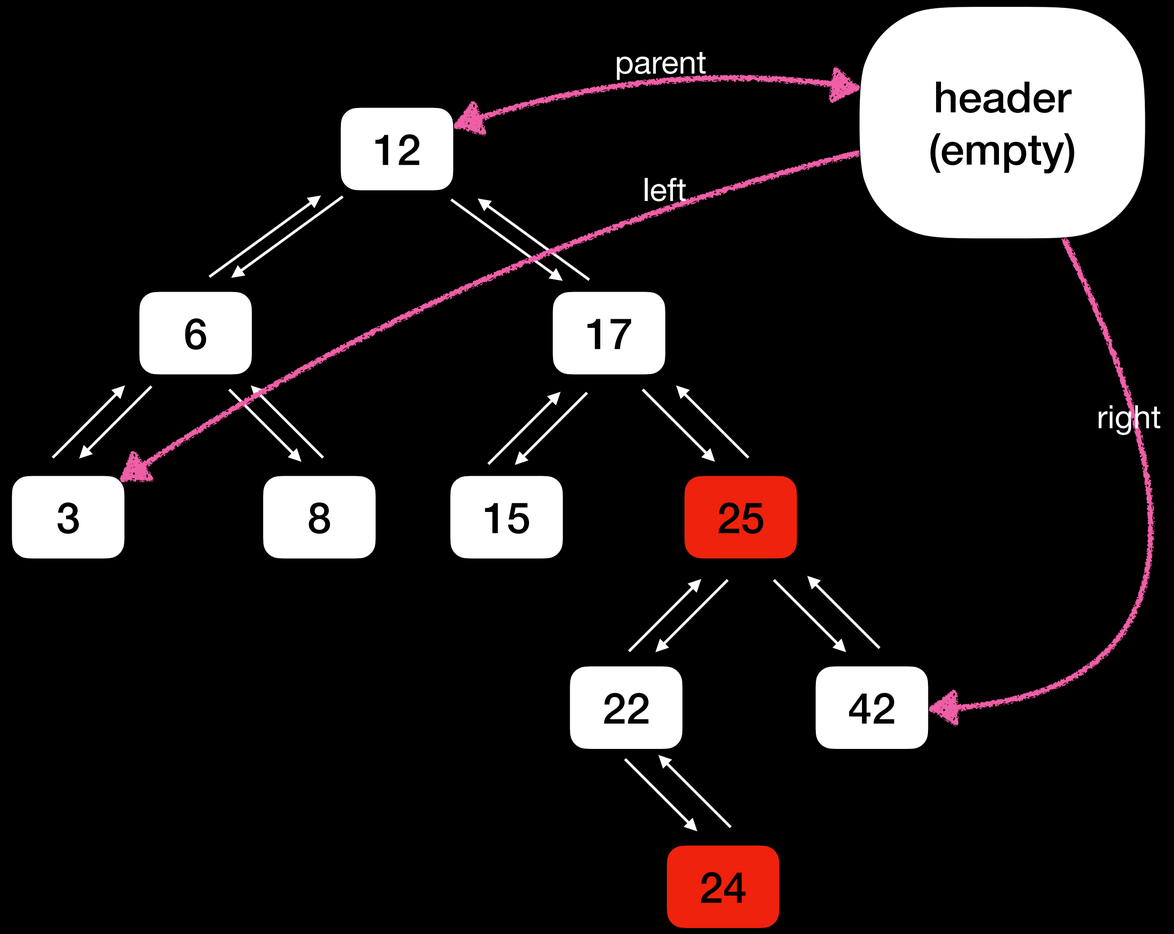
컨테이너를 위한 RB-Tree :: Node structure



- RB-Tree의 기본 구조를 형성하는 __tree_node_base를 만들어 필요한 데이터를 저장
- 컨테이너 종류에 따라 노드 안의 데이터 타입이 변하므로 __tree_node_base를 상속받는 __tree_node를 만들어 데이터를 저장

Red-Black Tree for containers

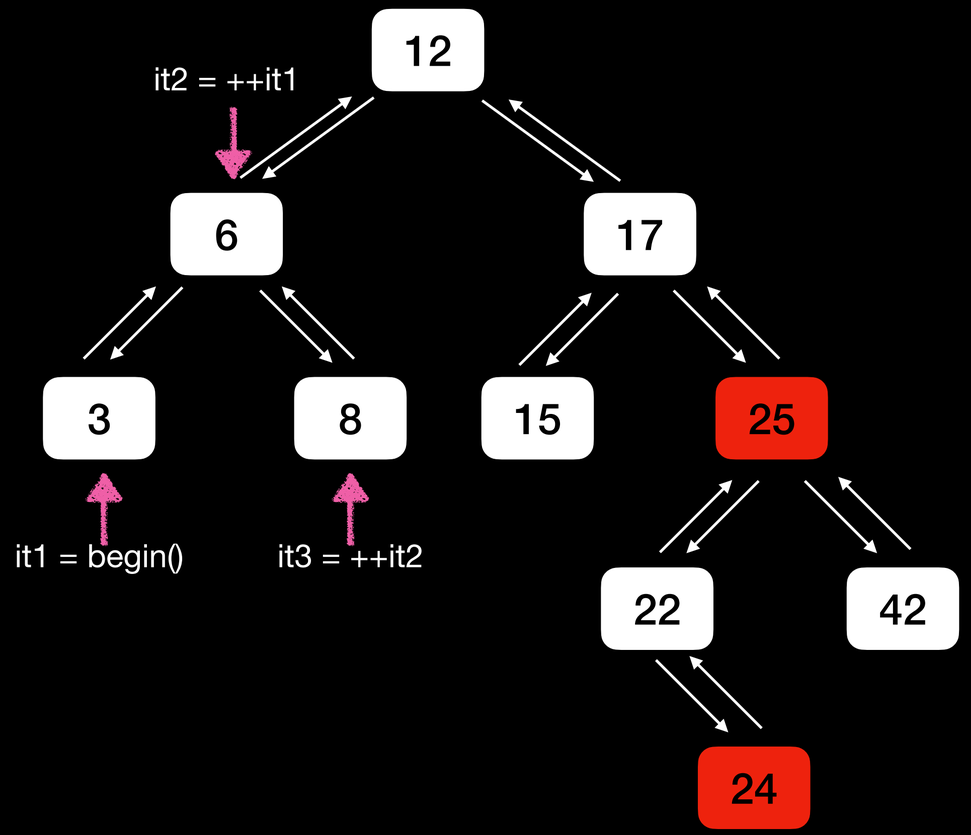
컨테이너를 위한 RB-Tree :: header



- 빠른 데이터 접근과 순회를 위하여 header라는 empty node를 만들고, parent = root, left = leftmost, right = rightmost 를 넣어줌.
- begin()은 header.left, end()는 header 를 가져오므로써 상수 시간 내에 이터레이터에 접근 가능.

Red-Black Tree for containers

컨테이너를 위한 RB-Tree :: iterator



- 트리의 iterator은 bidirectional_iterator로 increment / decrement만 지원함.
- 이때 증감은 중위순회 기준 이전 / 이후 노드를 가리키게 함.
- end()는 empty인 header 노드를 가리키게 해서 iterator 구조 완성!