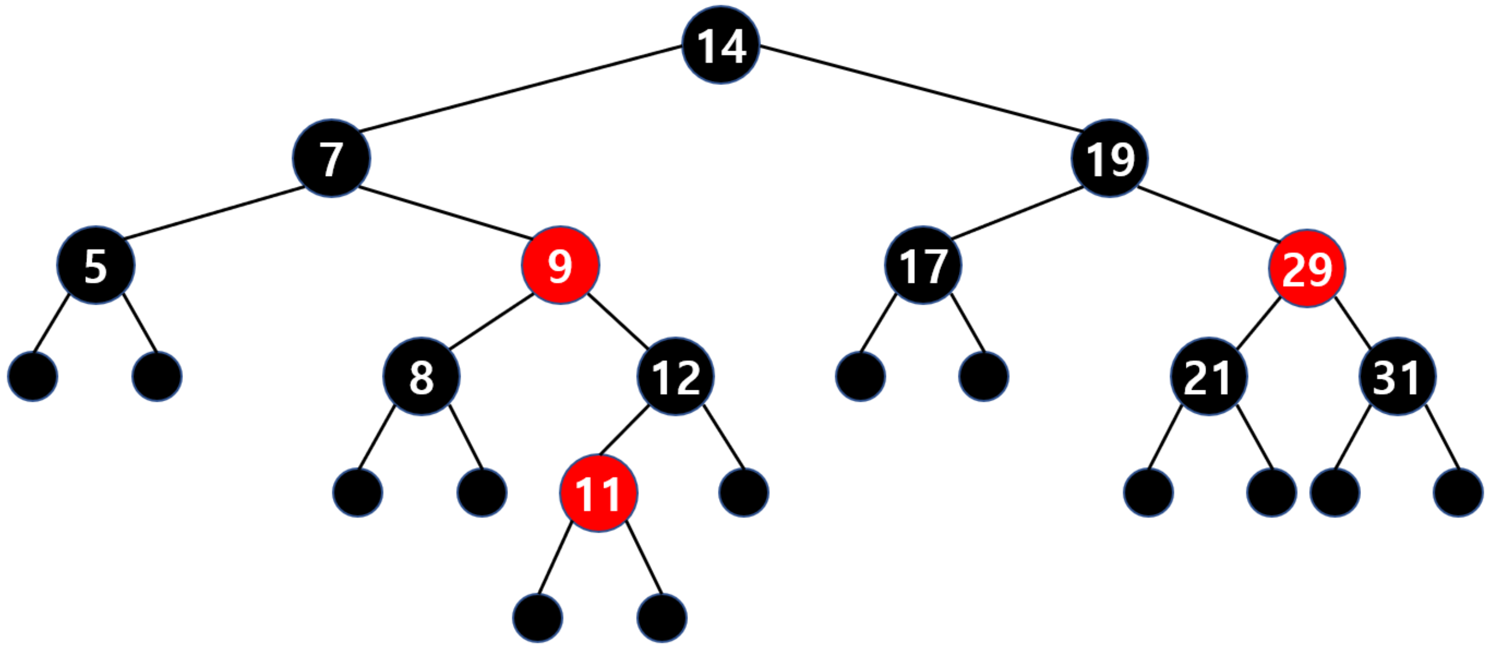
Red-Black Tree (RBT)

- 연산자들의 시간복잡도가 O(log n)이 되도록 균형 있게 만들어진 tree
- RBT는 기본적으로 BST이면서 다음과 같은 성질을 만족한다.
  - **(성질 1)** RBT의 각 노드는 Red 혹은 Black 색깔 중의 한 가지 색을 가진다.
  - **(성질 2)** NULL로 표시되는 모든 단말노드는 Black 색을 가진다.
  - **(성질 3)** 어떤 노드가 Red 색이면, 이 노드의 두 child 는 모두 Black 색이다.
  - **(성질 4)** 어떤 노드로부터 이 노드의 모든 단말노드 까지의 경로는 모두 같은 개수의 Black 노드를 가진다.
  - **(성질 5)** 루트노드는 Black이다.

root에서 leaf까지의 Black 노드 수가 같다

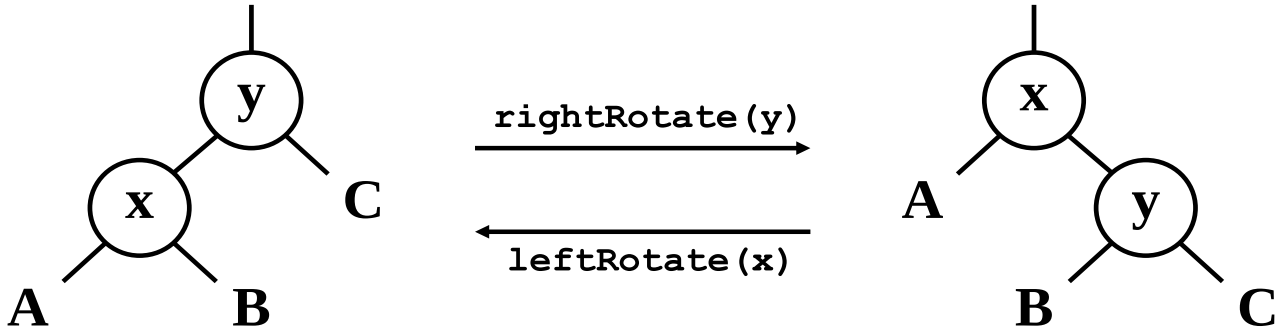
Red-Black Tree (RBT)

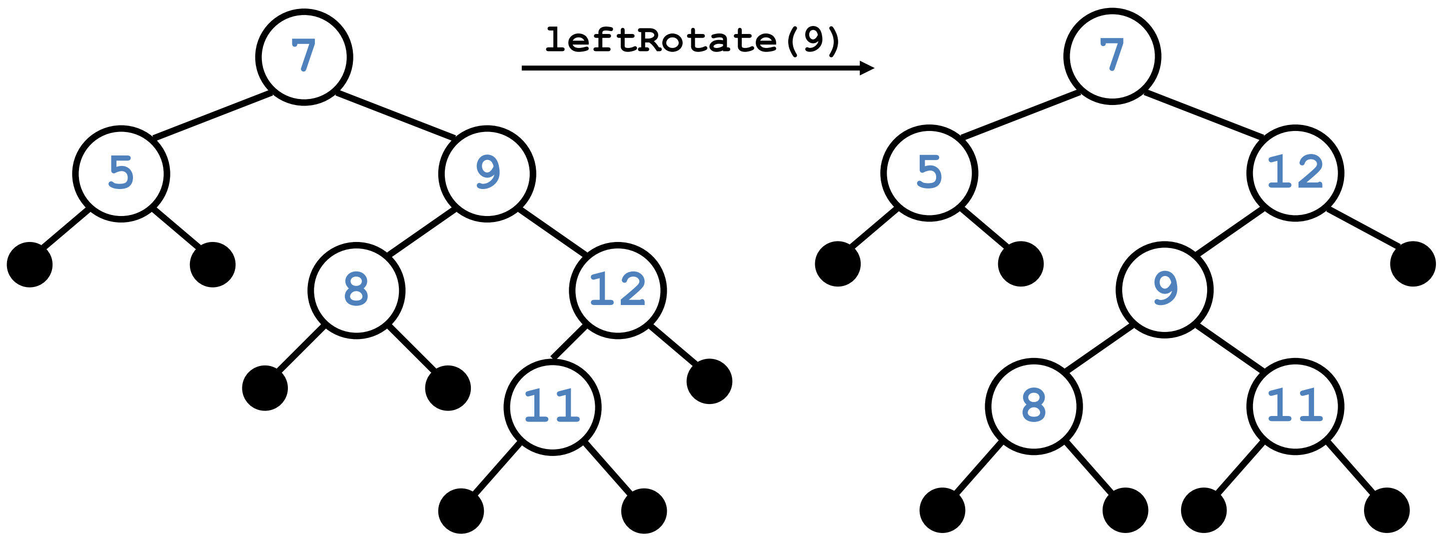
- RBT의 한 예



- 정리 1: n개의 내부노드(internal node)를 가지는 RBT의 높이(height)는 최대 2log(n+1) 이다

RBT - Rotation





RBT - Insertion

- RBT에 임의의 노드를 입력하는 과정의 스케치

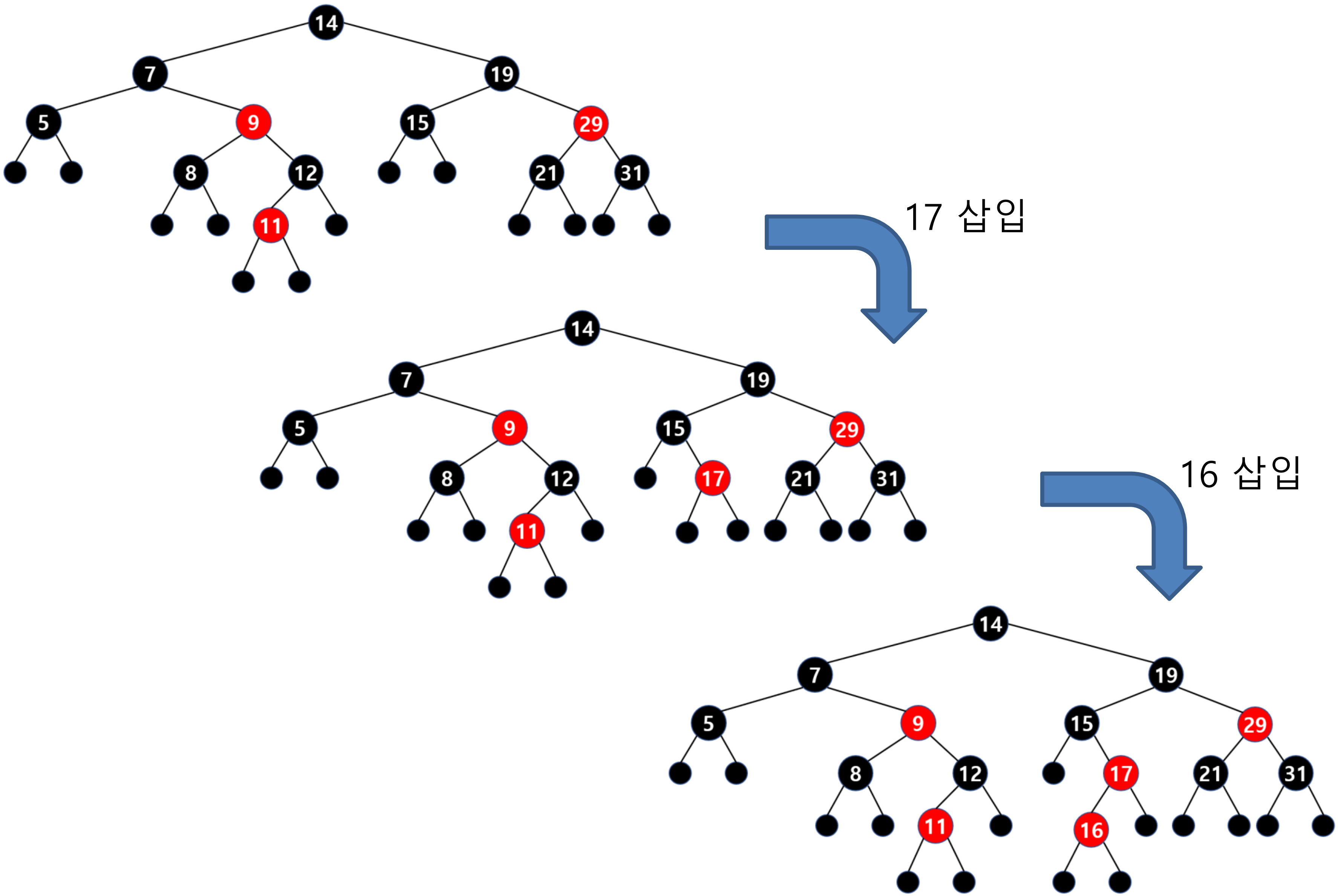
  (Step 1) x를 RBT에 삽입하고, x의 색을 red로 둠

  이 경우에는 x의 parent 의 색이 red인 경우에 RBT의 (성질 3) 을 만족하지 않을 수 있다. 그 이외의 RBT 성질은 모두 만족한다.

  (Step 2) RBT의 (성질 3)을 만족하지 않는 경우에는,

  이 성질을 만족하지 않는 상태가 되는 노드의 위치를 트리의 위쪽으로 계속 옮기고, 최종적으로 루트노드의 색이 red가 되면, 루트노드의 색을 black으로 바꾼다.

RBT - Insertion



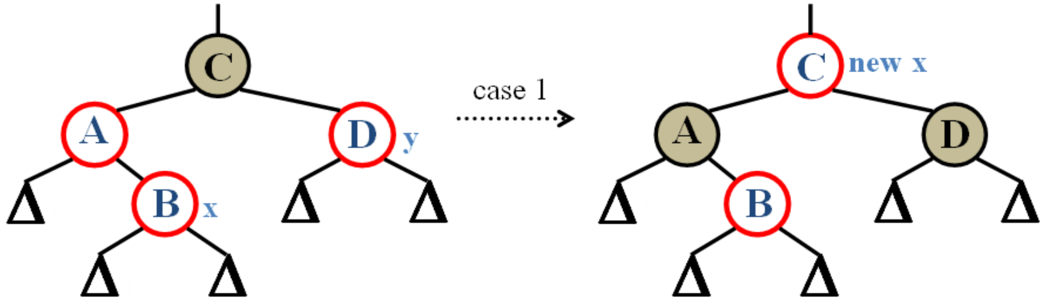
RBT - Insertion

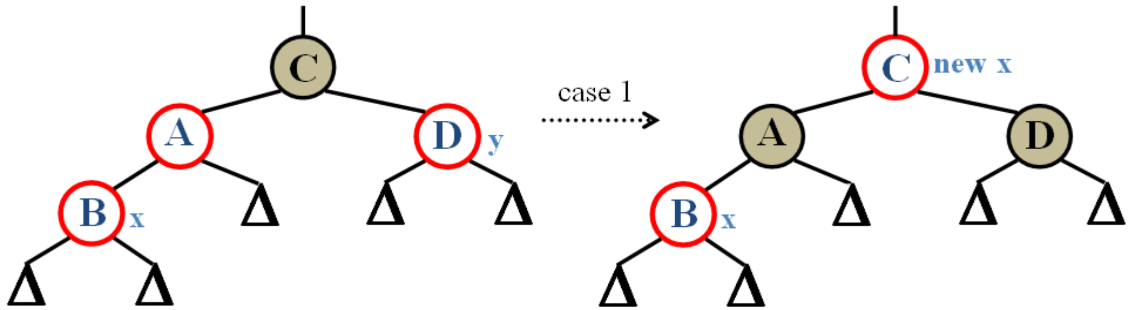
```
rbInsert(T, x)
  TreeInsert(T, x);
  x->color = RED;

  while (x!=root && x->p->color == RED)
      if (x->p == x->p->p->left)
          y = x->p->p->right;
          if (y->color == RED)
              x->p->color = BLACK;       // case 1
              y->color = BLACK;          // case 1
              x->p->p->color = RED;      // case 1
              x = x->p->p;               // case 1
          else   // y->color == BLACK
              if (x == x->p->right)
                  x = x->p;              // case 2
                  leftRotate(T, x);      // case 2
              x->p->color = BLACK;       // case 3
              x->p->p->color = RED;      // case 3
              rightRotate(T, x->p->p);   // case 3
      else    // x->p == x->p->p->right
          (same as above, but with
           "right" & "left" exchanged)

  T.root->color = BLACK;
```

- 자료에서 보인 함수에서는 x의 부모노드가 left child인 경우만을 표시
- x의 부모노드가 right child인 경우는 유사하게 구현할 수 있음
- 여기서 x와 x의 부모 노드의 색은 red임에 유의
- 위 함수에서 각 경우 1, 2, 3의 예는 다음과 같다.

RBT - Insertion

- (Case 1) x의 uncle(parent 의 형제노드)의 색이 red인 경우
  - 아래 그림처럼 부모, 삼촌, 조부모의 색을 바꾸고 조부모를 새로 삽입된 x처럼 취급하고 계속 처리한다.

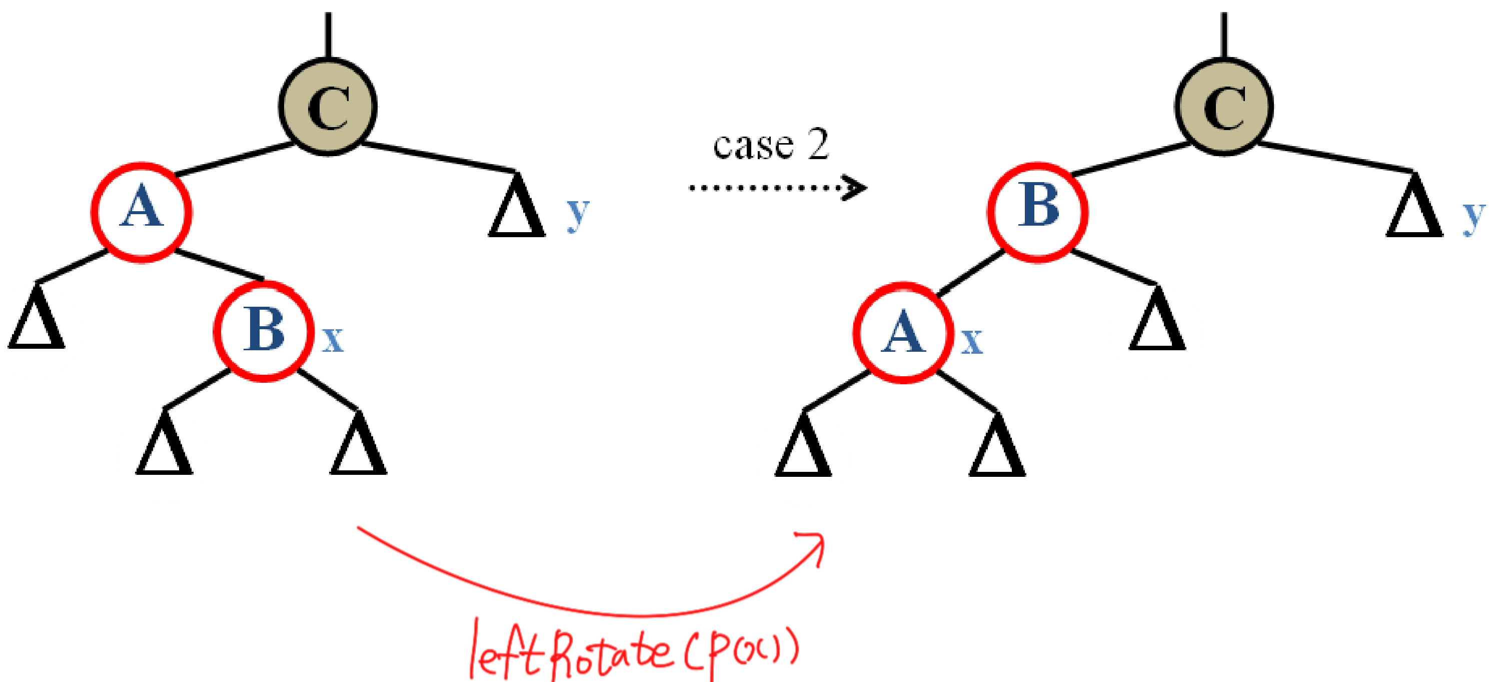




RBT - Insertion

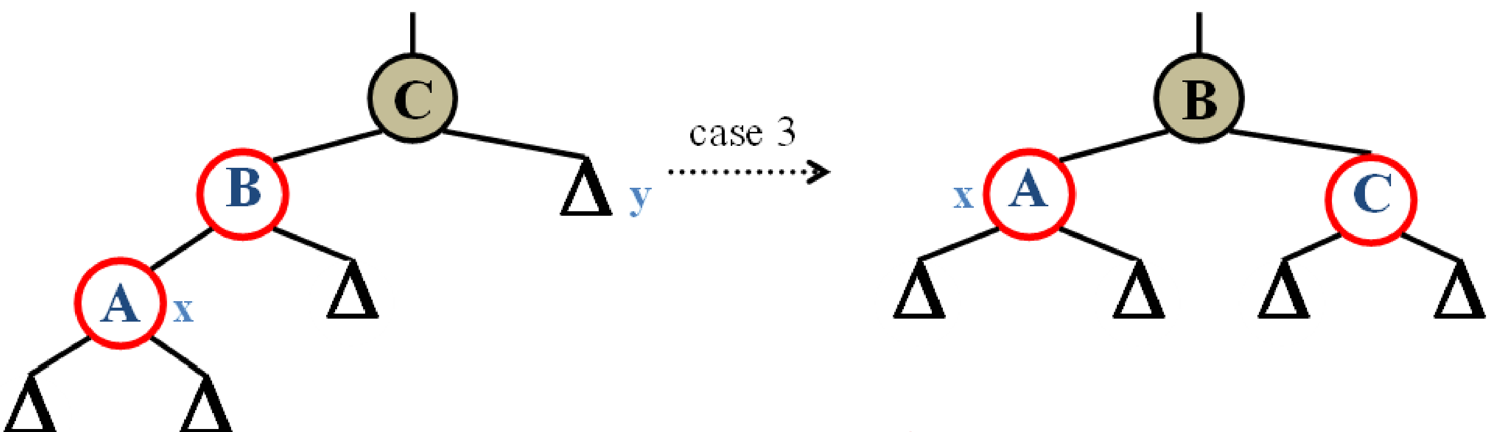
- (Case 2) x의 uncle (parent의 형제노드)의 색이 black이면서, x가 parent의 오른쪽 child인 경우

  x의 부모노드에 대하여 leftRotate() 연산 후 (Case 3) 에서 처리하게 한다.



RBT - Insertion

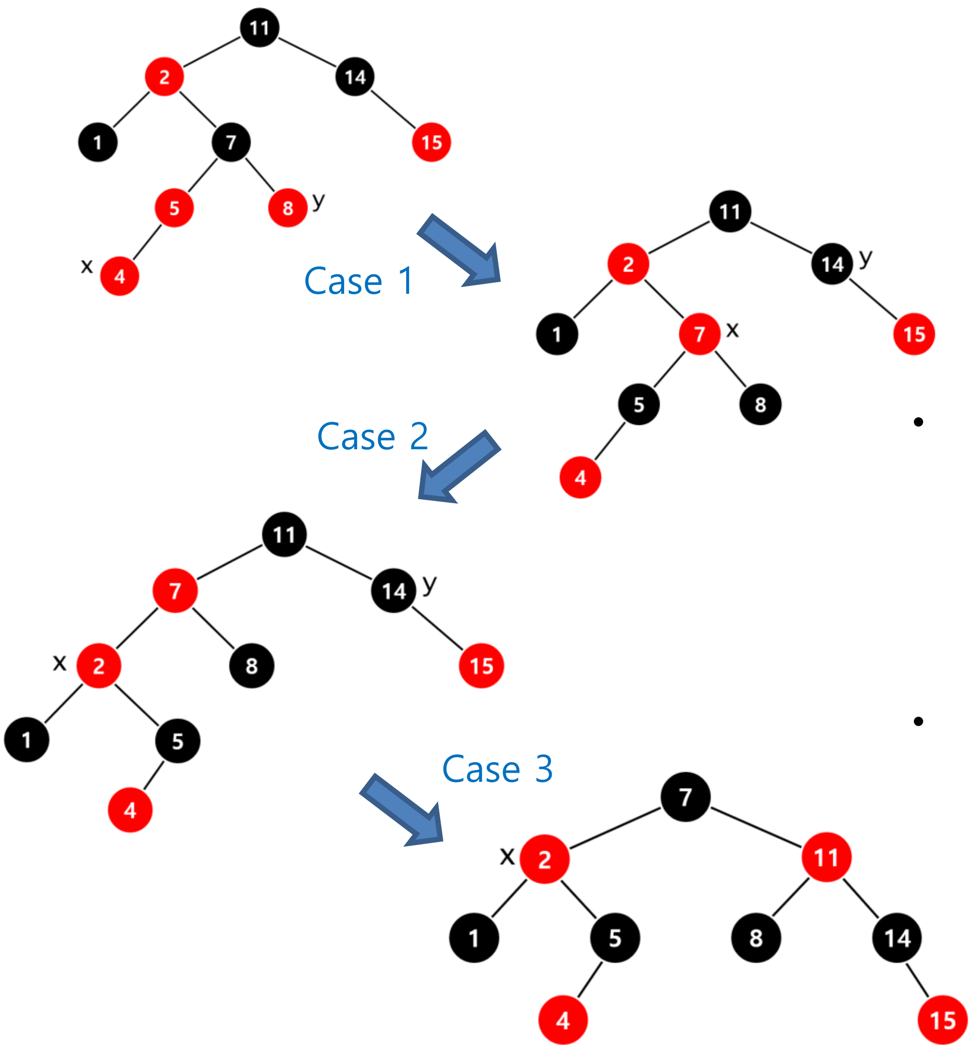
- (Case 3) x의 uncle(parent 의 형제노드)의 색이 black이면서, x가 parent의 왼쪽child 인 경우
  - x의 조부모노드에서 rightRotate() 연산 후, 노드의 색을 바꾼다.
  - (Case 3)를 수행한 이후엔 BST의 모든 성질을 만족한다.



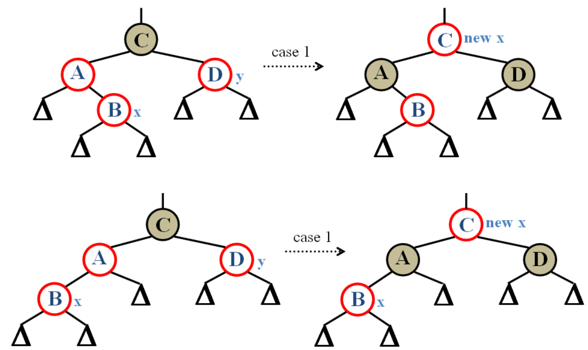
rightRotate(p(p(x))

Insertion Examples

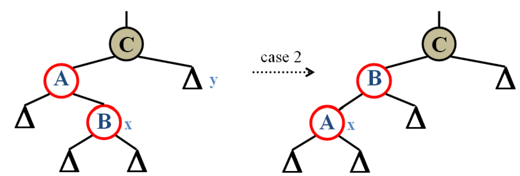
Key가 4인 노드 삽입



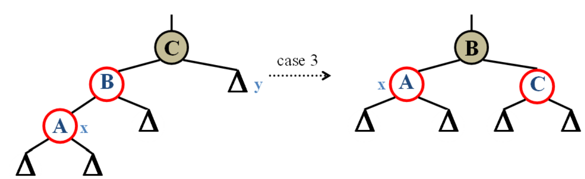
- Case 1: x의 uncle(삼촌)이 Red
  - x의 부모, 삼촌, 조부모의 색 변경
  - 조부모를 새로 삽입된 x처럼 취급하고 계속 처리



- Case 2: x의 삼촌=Black, 그리고 x = R(P(x))
  - LeftRotate(P(x))
  - Case 3로 넘김

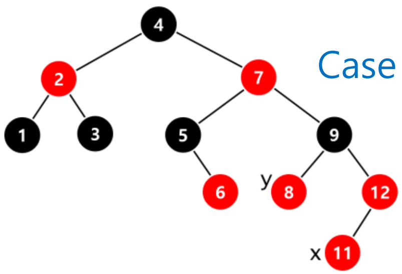


- Case 3: x의 삼촌=Black, 그리고 x = L(P(x))
  - P(x) ← Black
  - P(P(x)) ← Red
  - RightRotate(P(P(x)))

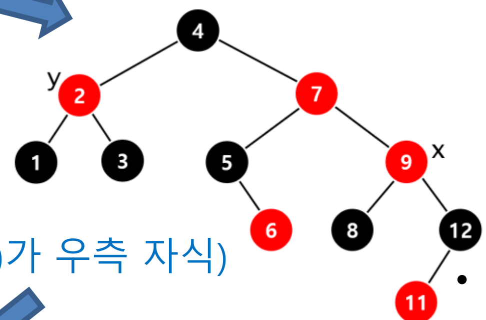


Insertion Examples

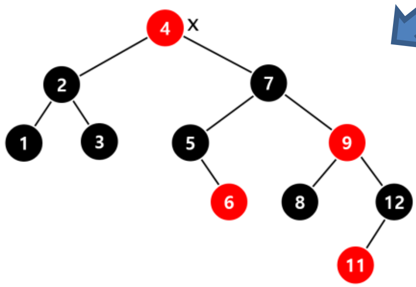
Key가 11인 노드 삽입



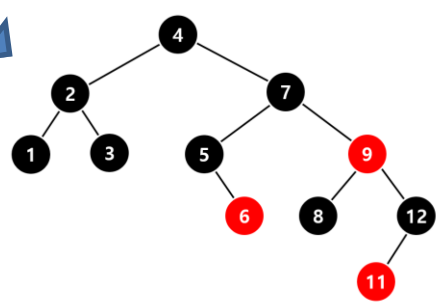
Case 1 (단, 부모의 P(x)가 우측 자식)



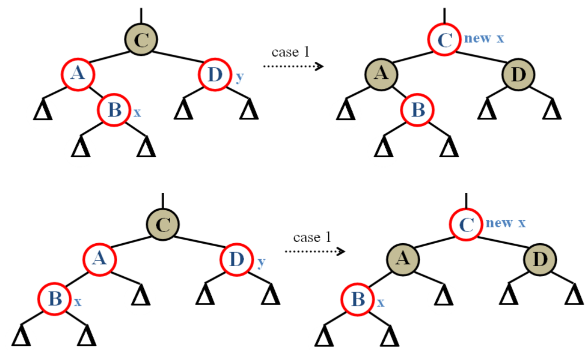
Case 1 (단, 부모의 P(x)가 우측 자식)



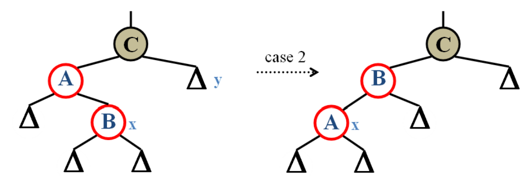
Root가 Red인 경우



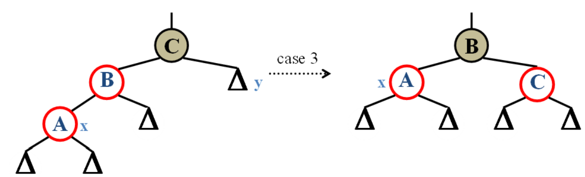
- Case 1: x의 uncle(삼촌)이 Red
  - x의 부모, 삼촌, 조부모의 색 변경
  - 조부모를 새로 삽입된 x처럼 취급하고 계속 처리



- Case 2: x의 삼촌=Black, 그리고 x = R(P(x))
  - LeftRotate(P(x))
  - Case 3로 넘김



- Case 3: x의 삼촌=Black, 그리고 x = L(P(x))
  - P(x) ← Black
  - P(P(x)) ← Red
  - RightRotate(P(P(x)))



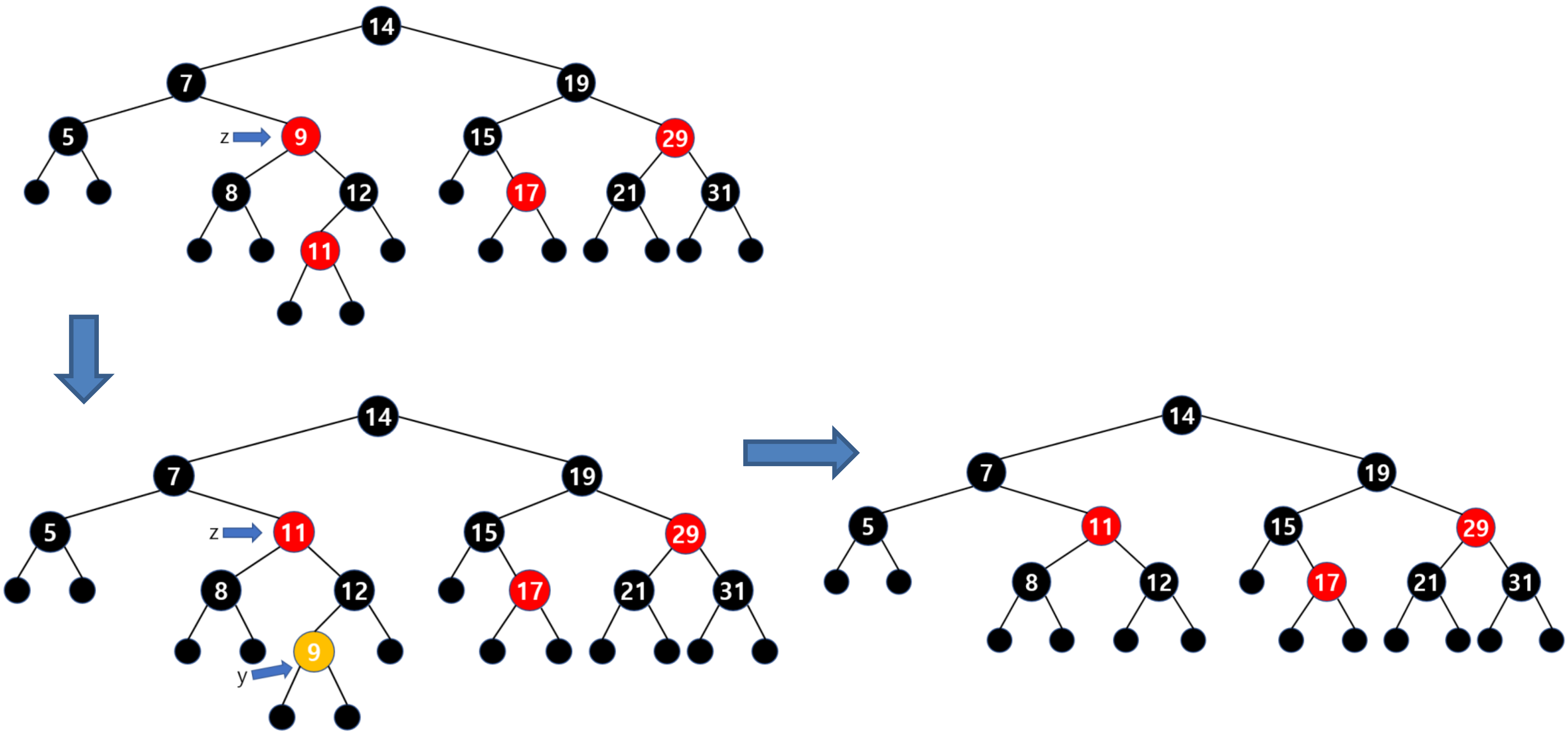
RBT - Insertion

- insert 연산의 case 2, 3에서 각 노드의 색 변화는 상수 번 변하며, 노드의 회전 연산은 최대 2번
- 그러나, case 1에서는 while 루프를 통하여 이중의 red 색을 가지는 노드를 계속 루트노드까지 올리게 된다.
- 따라서, insert 연산의 수행시간은 O(lg n)

```
if( x의 부모가 red) {
    case 1 //uncle 이 red
        x = p(p(x))
    case 2 //uncle 이 black & x가 오른쪽 자식
        x = p(x)
        leftRotate(x)
    case 3 //uncle이 black & x가 왼쪽 자식
        rightRotate(p(p(x)))
}
```

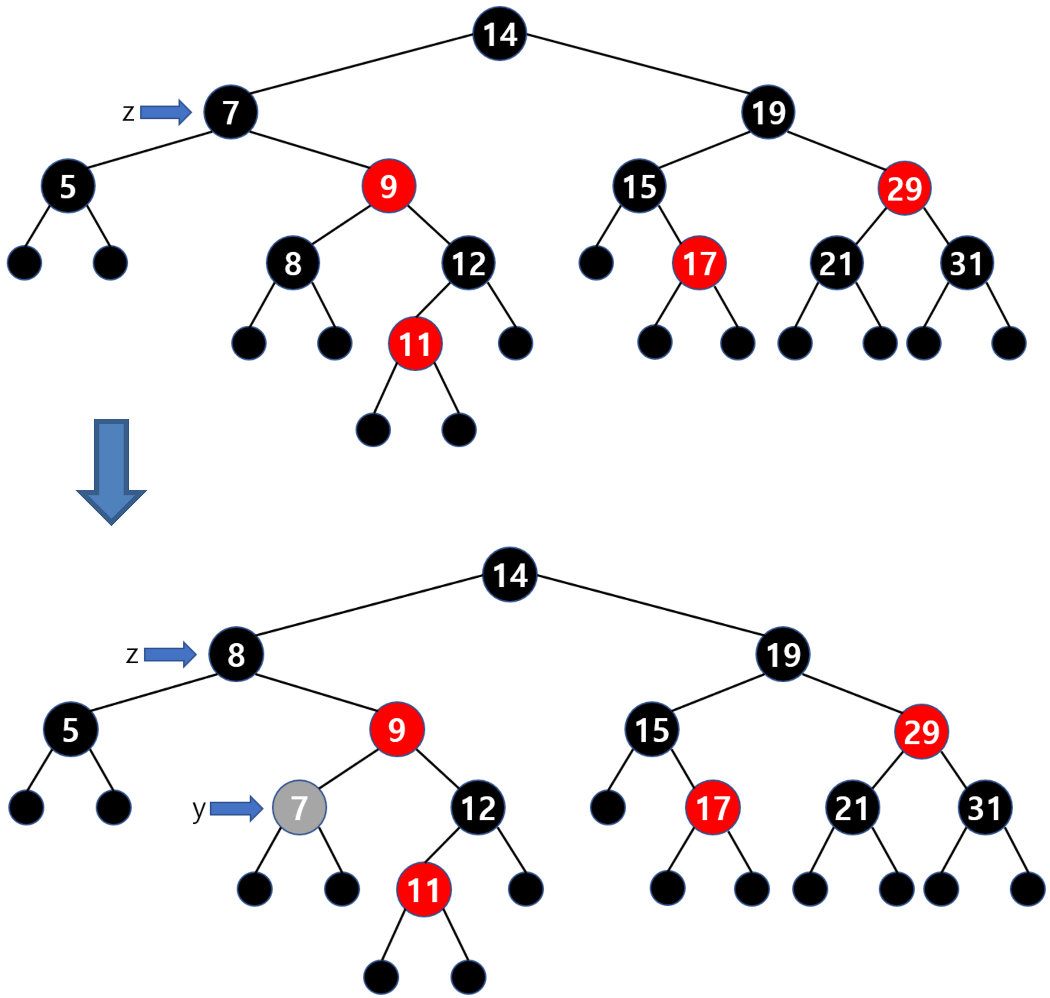
RBT - Deletion

- RBT에서 노드 z를 삭제하는 과정의 스케치



RBT - Deletion

- RBT에서 노드 z를 삭제하는 과정의 스케치



RBT - Deletion

- RBT에서 노드 z를 삭제하는 과정의 스케치

  (Step 1) BST에서 노드 z를 제거하는 방법과 동일하게 노드 z를 제거한다.

  - 이때, 실제로 제거된 노드는 z 자체이거나 (z가 단말노드이거나, 한 개의 child 만을 가지는 경우) 혹은 z의 successor 노드이다. 실제로 제거된 노드를 y라 하자.

  (Step 2) 위에서 제거된 노드 y의 색이 red인 경우에는 RBT의 모든 성질을 만족하므로, 그냥 종료한다. 노드 y의 색이 black인 경우에는 RBT의 (성질 4)를 만족하지 못하므로, RBT의 노드를 회전시켜 RBT의 모든 성질이 만족되도록 tree 구조를 변경한다.

RBT - Deletion

```
rbDelete(T, z)
        if (z->left == NULL or z->right == NULL)
            y = z;  // z has 0 or 1 child
        else
            y = TreeSuccessor(z);  // z has 2 children

        // now, y has 0 or 1 child, set x as one child of y
        if (y->left != NULL)
            x = y->left;
        else
            x = y->right;

        if (x != NULL)  // delete y
            x->p = y->p;

        if (y->p == NULL)
            T.root = x;
        else if (y == y->p->left)
            y->p->left = x;
        else
            y->p->right = x;

        if (y != z)
            z->key = y->key;

        if (y->color) == BLACK)
            rbDeleteFixup(T, x);

        return y;
```

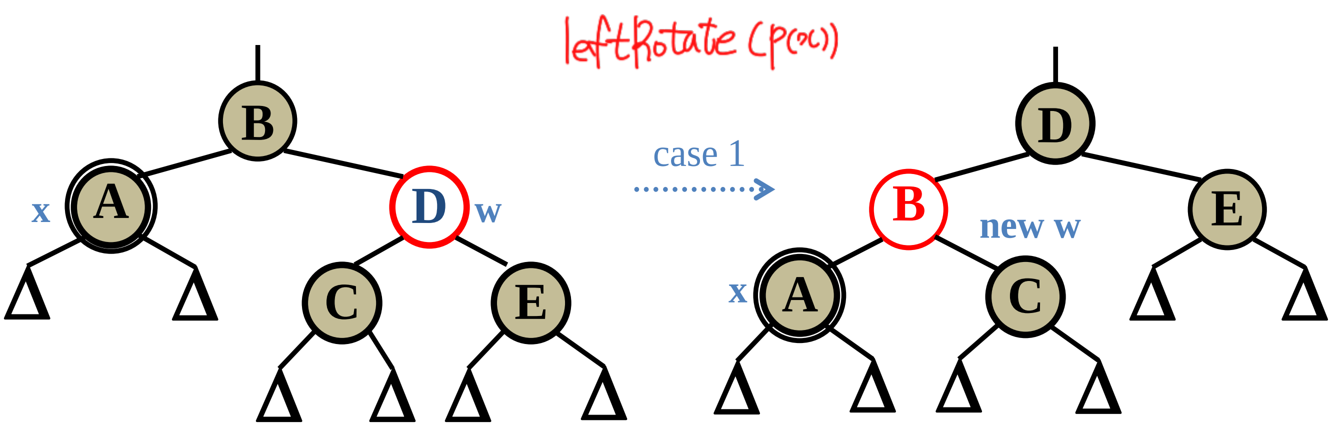
```
rbDeleteFixup(T, x)
  while (x != T.root && x->color == BLACK)
      if (x == x->p->left)
          w = x->p->right
          if (w->color == RED)
                    w->color = BLACK;        // case 1
              x->p->color = RED;             // case 1
              leftRotate(T,x->p);            // case 1
              w = x->p->right;               // case 1
          if (w->left->color == BLACK &&
                         w->right->color == BLACK)
              w->color == RED;               // case 2
              x = x->p;                      // case 2
          else if w->right->color == BLACK)
              w->left->color = BLACK;        // case 3
              w->color = RED;                // case 3
              rigthRotate(S, x);             // case 3
              w = x->p->right;               // case 3
          w->color = x->p->color;            // case 4
          x->p->color = BLACK;               // case 4
          w->right->color = BLACK;           // case 4
              leftRotate(T, x->p);           // case 4
              x->T.root;
      else    // x == x->p->right
          (same as above, but with
           "right" & "left" exchanged)

  x->color = BLACK;
```

RBT - Deletion

- 함수 rbDeleteFixup(T,x)에서 x는 실제로 삭제된 노드 y의 child
  - y가 하나의 자식만 가질 경우 x는 y의 유일한 child
  - y의 두 자식이 모두 terminal인 경우 x는 NULL
- y가 삭제되고, 노드 x가 노드 y의 위치를 차지함
- 이 경우에 제거된 노드 y의 black 색을 노드 x에 이전시켜서 문제를 해결한다.
- 노드 x의 색이 red인 경우에는 색을 black으로 바꾸면 문제가 쉽게 해결되어 그냥 종료하면 된다.
- 그러나, 노드 x의 색이 black 인 경우에는 x는 y의 black 색을 넘겨받아 이중의 black 색을 가진다고 가정하고, 여분의 black을 노드 x부터 root 사이의 경로에 존재하는 red 색의 노드로 이전시켜서 이 노드를 black 색으로 바꾸어 RBT의 (성질 4)를 만족하게 한다.
- 함수에서는 노드 x를 x 부모노드의 left child로 가정한다. x가 부모노드의 right child 인 경우에는 유사한 방법으로 처리할 수 있다.

RBT - Deletion

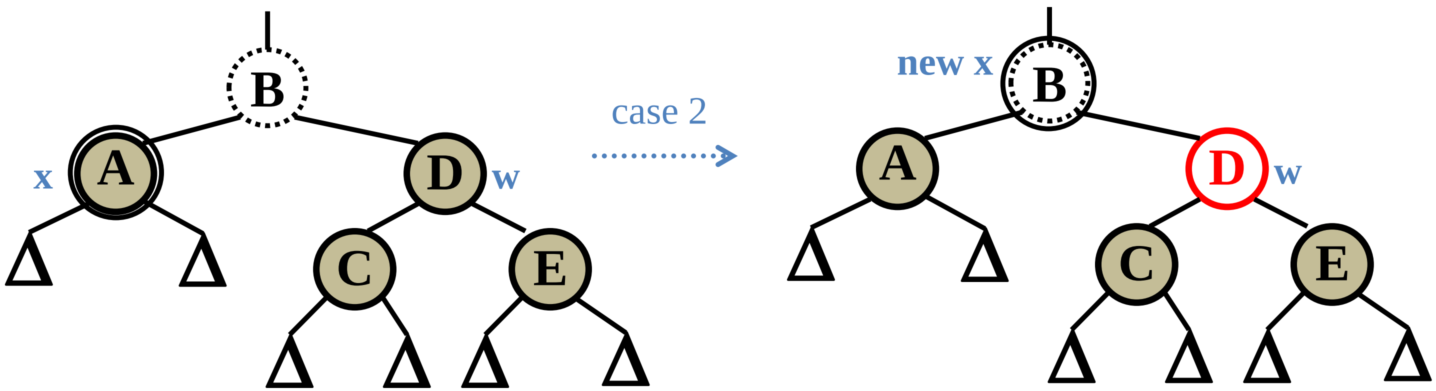
- (Case 1) x 의 형제노드 w의 색이 red인 경우 (x의 부모는 반드시 black 이다)
  - w를 black으로, x의 부모를 red로 바꾼 다음, leftRotate(p(x)) 수행한 후 w를 다시 x의 형제 노드가 되도록 한다.
  - 이렇게 RBT의 구조를 바꾼 다음에, Case 1을 Case 2, 3, 4 에 적용시킨다. 아래 그림에서 노드 x를 이중 원으로 표시한 것은 x가 여분의 black 을 가지고 있음을 나타낸다.



  - 처리된 후 w의 색은 반드시 black. 이제 w의 색에 따라 처리함

RBT - Deletion

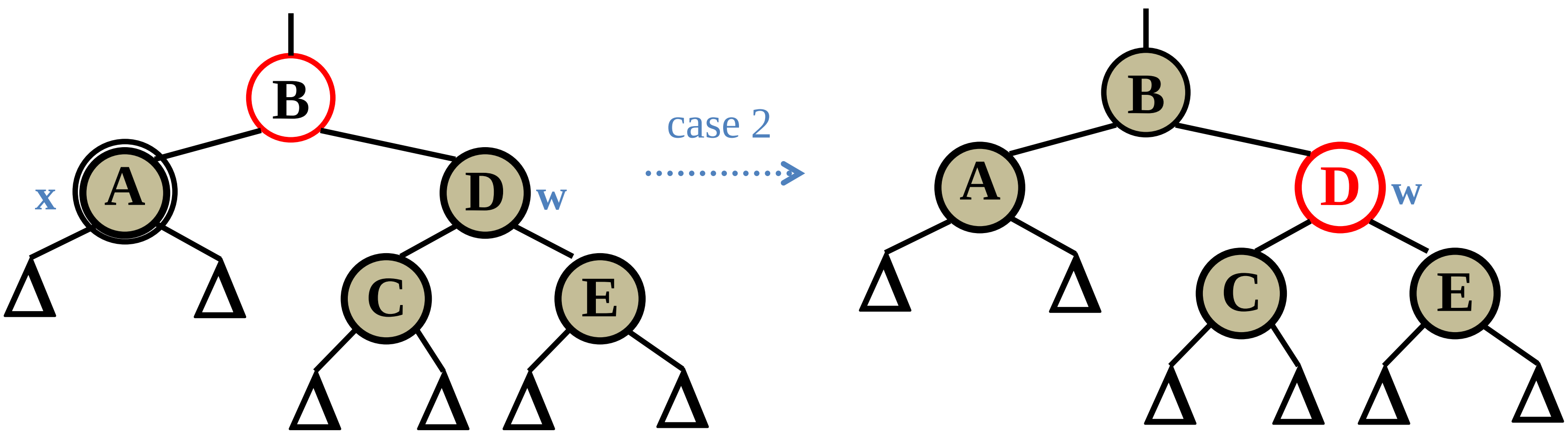
- (Case 2) (w는 black) w의 left, right child 모두 black인 경우
  - w의 색을 red로 바꾸고, x가 가지고 있던 여분의 black을 x의 부모노드로 옮긴다.



  - 이 경우에는 x의 부모노드의 색에 따라 두 가지 경우가 발생한다

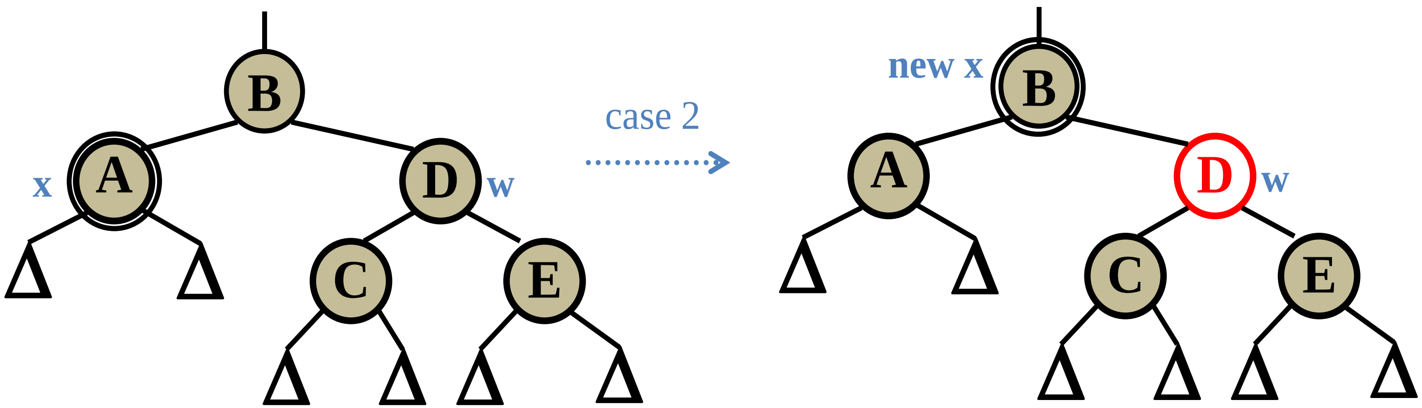
RBT - Deletion

- (Case 2-1) (w는 black) x의 부모노드의 색이 red인 경우
  - x의 부모노드를 red에서 x로부터 전달받은 black으로 바꾸고, rbDeleteFixup() 루틴을 종료한다.



RBT - Deletion

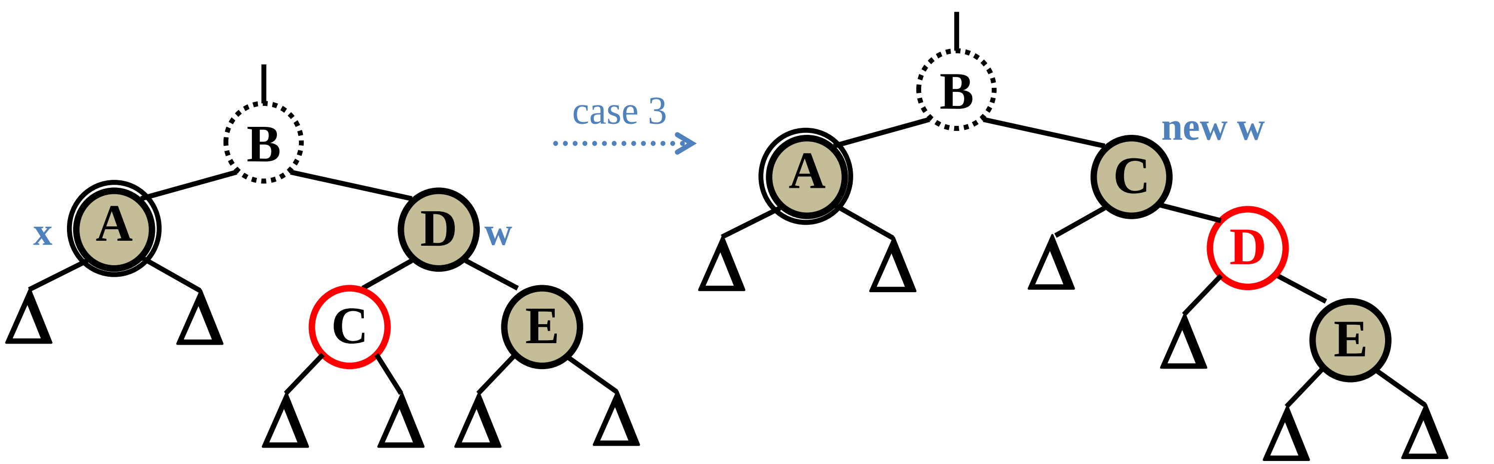
- (Case 2-2) (w는 black) x의 부모노드의 색이 black 인 경우
  - x의 부모노드가 x로부터 black으로 전달받아 여분의 black을 가지게 된다.
  - 이제 이 부모노드를 x로 정하고, 계속 loop를 수행.



  - 나머지 경우인 Case 3, 4에서는 w의 색이 black이며, w의 자식노드 중에서 적어도 한 개의 자식노드가 red 인 경우를 처리한다.

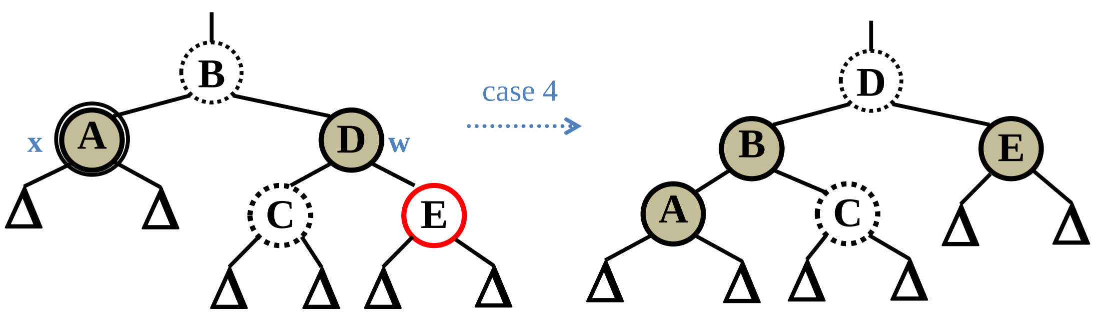
RBT - Deletion

- (Case 3) (w는 black) w의 오른쪽 child가 black인 경우. (따라서, 자동적으로 왼쪽 child는 red이다)
  - w의 색을 red로, w의 왼쪽 child색을 black으로 바꿈
  - w를 중심으로 right rotate
  - 새로운 w의 right child가 red가 되게 하여 Case 4에서 처리



RBT - Deletion

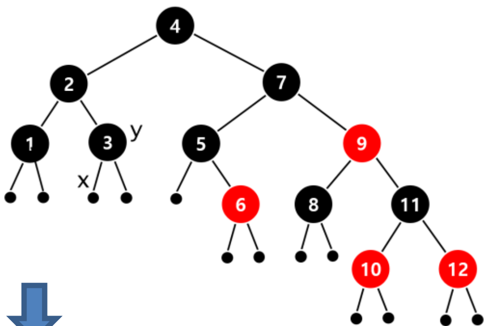
- (Case 4) (w는 black) w의 오른쪽 child가 red인 경우. (따라서, 왼쪽 child 는 red 또는 black 일 수 있다.)
  - w의 색은 x의 부모의 색으로, x부모의 색은 black으로 바꿈
  - w의 우측 자식의 색은 black으로 바꿈
  - x의 부모노드를 중심으로 왼쪽회전
  - 이러면, x가 가지고 있던 여분의 black이 위로 전달된다.

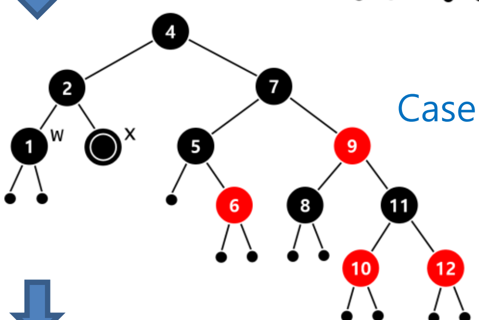


  - 이 작업을 마친 이후에는 RBT의 모든 성질을 만족하므로 바로 rbDeleteFixup() 를 종료시키게 된다.

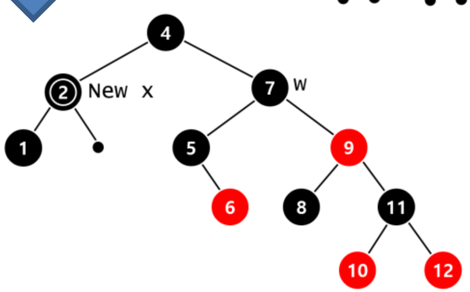
Deletion Examples

Key가 3인 노드 삭제

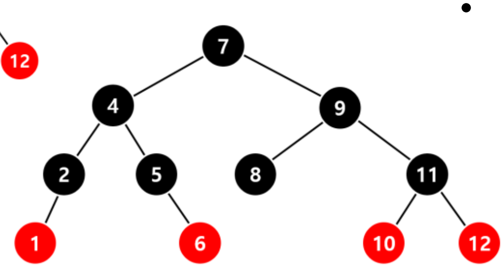




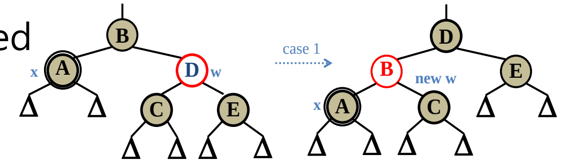
Case 2 (단, x == R(P(x)))



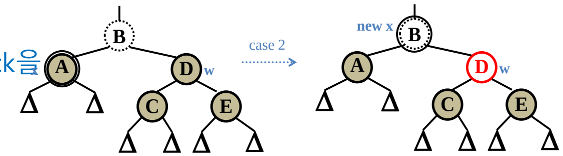
Case 4



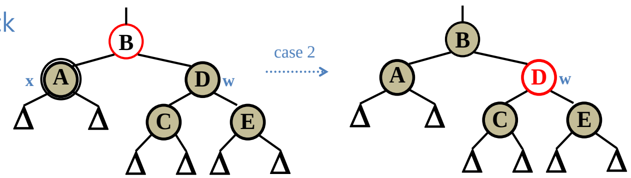
- Case 1: w == Red
  - w ← Black
  - P(x) ← Red
  - LeftRotate(p(x))



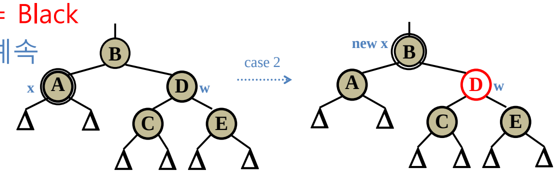
- Case 2: w의 두 자식이 모두 black
  - w ← Red
  - x의 여분의 Black을 부모에게 옮김



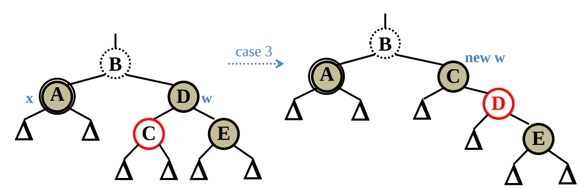
  - Case 2-1: P(x) == Red
  - P(x) ← Black
  - Done



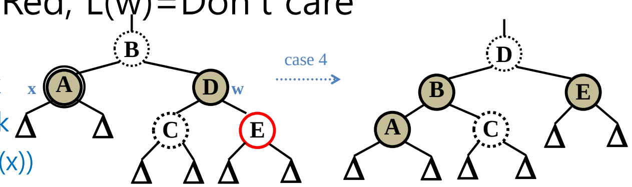
  - Case 2-2: P(x) == Black
  - P(x)를 x로 보고 계속
  - 이때 P(x)는 이중 Black을 가짐



- Case 3: R(w) == Black, L(w) == Red
  - w ← Red
  - L(w) ← Black
  - RightRotate(w)

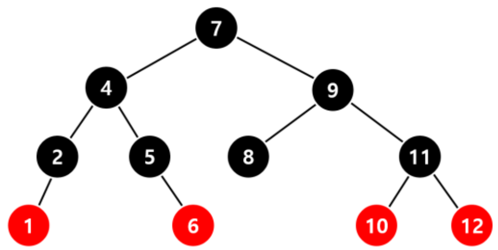


- Case 4: R(w)=Red, L(w)=Don't care
  - w ← P(x)색
  - P(x) ← Black
  - R(w) ← Black
  - LeftRotate(P(x))
  - Done

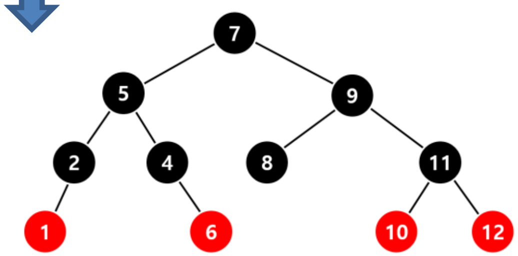


Deletion Examples

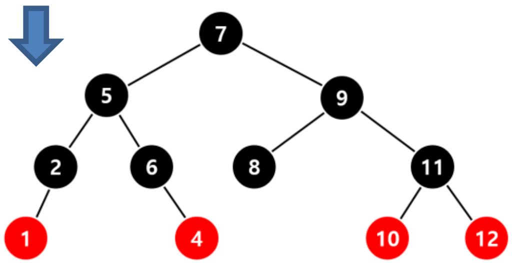
Key가 4인 노드 삭제

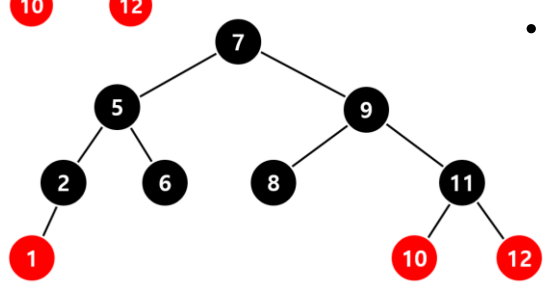


4의 successor인 5 와 위치 교환 후, 4를 삭제

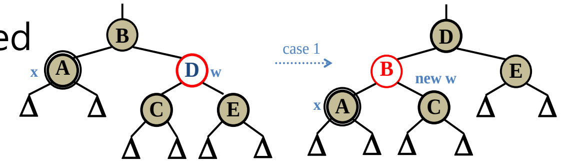


4의 successor인 6 과 위치 교환 후, 4를 삭제

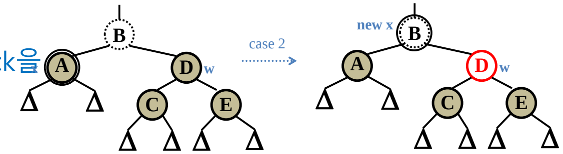




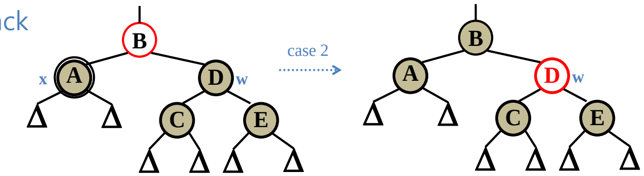
- Case 1: w == Red
  - w ← Black
  - P(x) ← Red
  - LeftRotate(p(x))



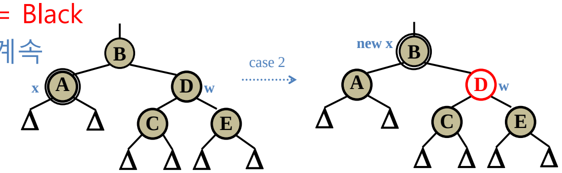
- Case 2: w의 두 자식이 모두 black
  - w ← Red
  - x의 여분의 Black을 부모에게 옮김



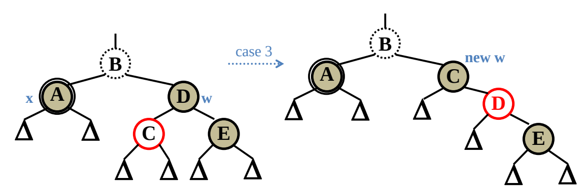
  - Case 2-1: P(x) == Red
  - P(x) ← Black
  - Done



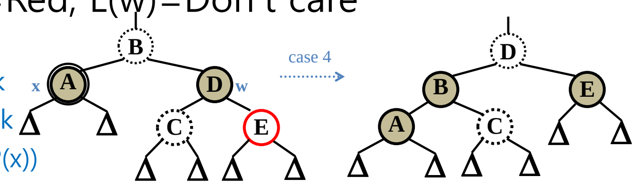
  - Case 2-2: P(x) == Black
  - P(x)를 x로 보고 계속
  - 이때 P(x)는 이중 Black을 가짐



- Case 3: R(w) == Black, L(w) == Red
  - w ← Red
  - L(w) ← Black
  - RightRotate(w)

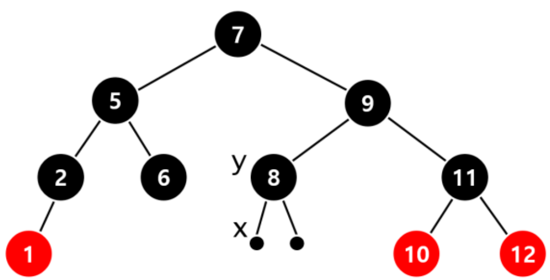


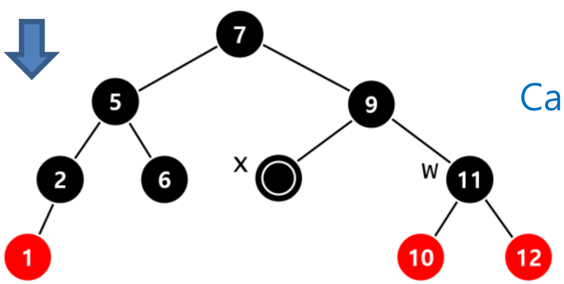
- Case 4: R(w)=Red, L(w)=Don't care
  - w ← P(x)색
  - P(x) ← Black
  - R(w) ← Black
  - LeftRotate(P(x))
  - Done



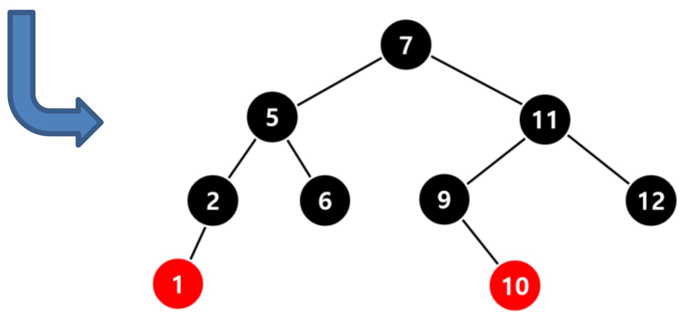
Deletion Examples

Key가 8인 노드 삭제

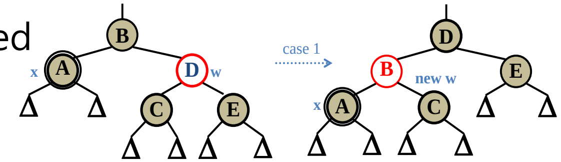




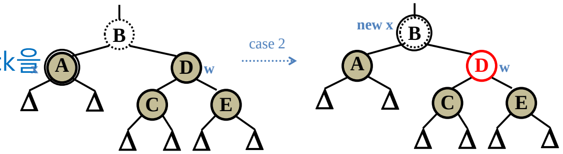
Case 4



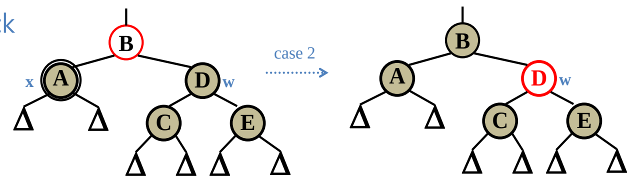
- Case 1: w == Red
  - w ← Black
  - P(x) ← Red
  - LeftRotate(p(x))



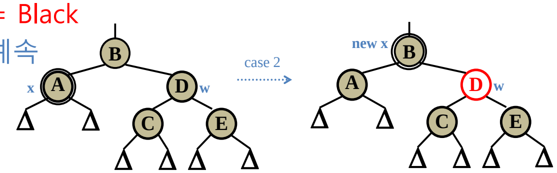
- Case 2: w의 두 자식이 모두 black
  - w ← Red
  - x의 여분의 Black을 부모에게 옮김



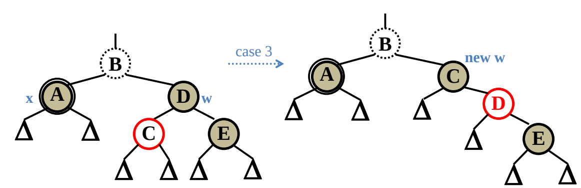
  - Case 2-1: P(x) == Red
  - P(x) ← Black
  - Done



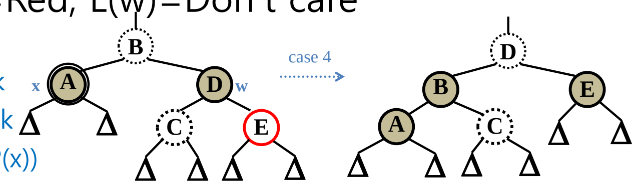
  - Case 2-2: P(x) == Black
  - P(x)를 x로 보고 계속
  - 이때 P(x)는 이중 Black을 가짐



- Case 3: R(w) == Black, L(w) == Red
  - w ← Red
  - L(w) ← Black
  - RightRotate(w)

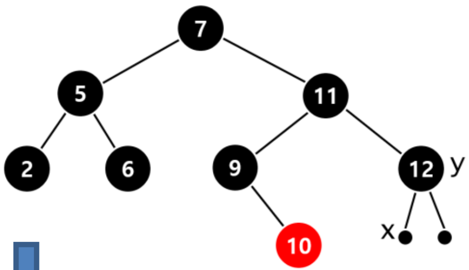


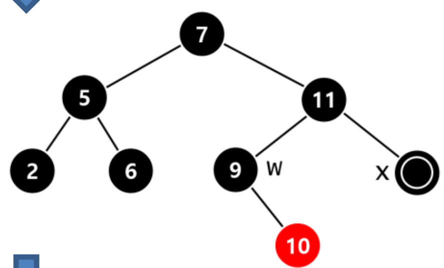
- Case 4: R(w)=Red, L(w)=Don't care
  - w ← P(x)색
  - P(x) ← Black
  - R(w) ← Black
  - LeftRotate(P(x))
  - Done



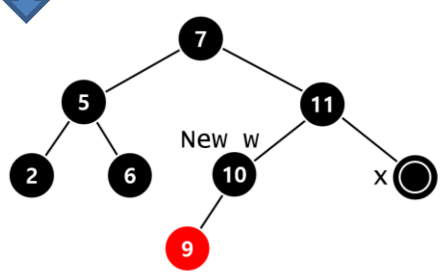
Deletion Examples

Key가 12인 노드 삭제

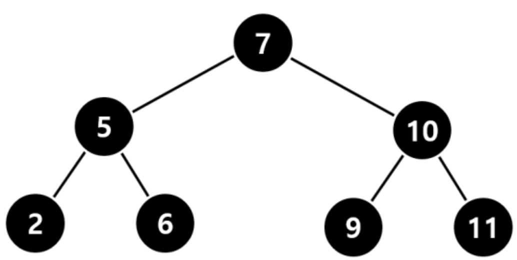




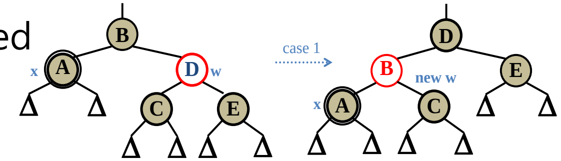
Case 3 (단, 모든 조건이 반대. 즉,
x = R(P(x)),
L(w) == Black
R(w) == Red



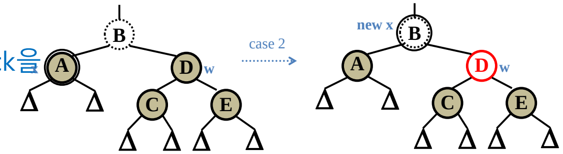
Case 4 (단, 모든 조건이 반대. 즉,
x = R(P(x)),
L(w) == Red



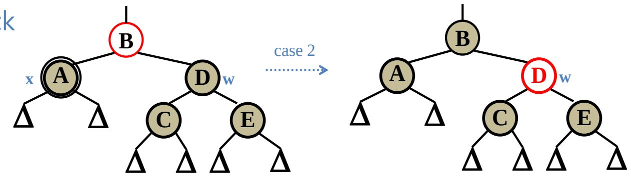
- Case 1: w == Red
  - w ← Black
  - P(x) ← Red
  - LeftRotate(p(x))



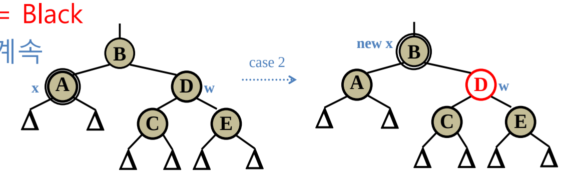
- Case 2: w의 두 자식이 모두 black
  - w ← Red
  - x의 여분의 Black을 부모에게 옮김



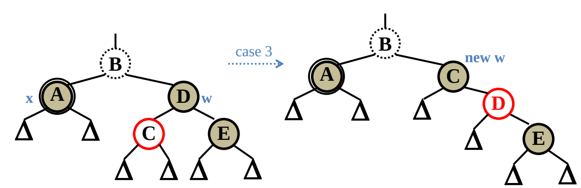
  - Case 2-1: P(x) == Red
  - P(x) ← Black
  - Done



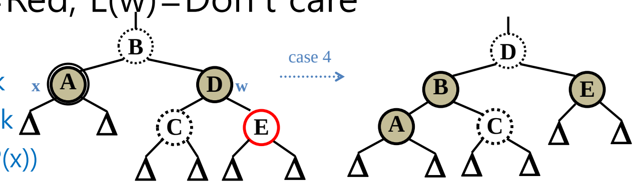
  - Case 2-2: P(x) == Black
  - P(x)를 x로 보고 계속
  - 이때 P(x)는 이중 Black을 가짐



- Case 3: R(w) == Black, L(w) == Red
  - w ← Red
  - L(w) ← Black
  - RightRotate(w)

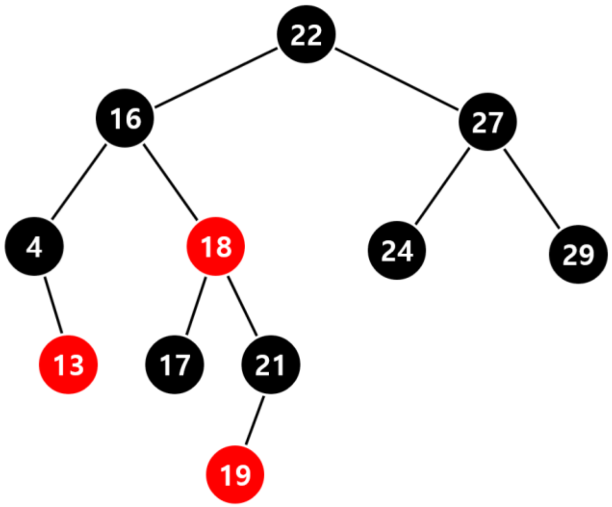


- Case 4: R(w)=Red, L(w)=Don't care
  - w ← P(x)색
  - P(x) ← Black
  - R(w) ← Black
  - LeftRotate(P(x))
  - Done

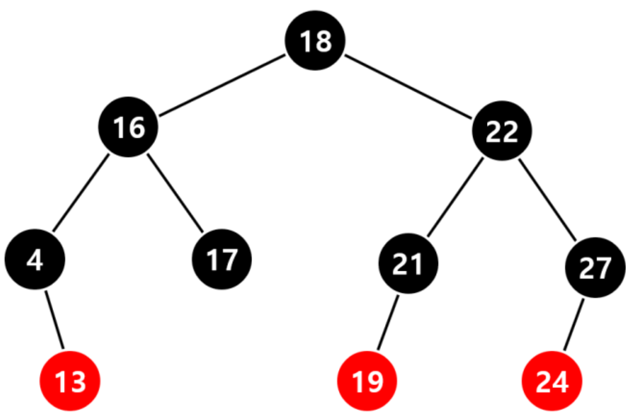


Deletion Exercise

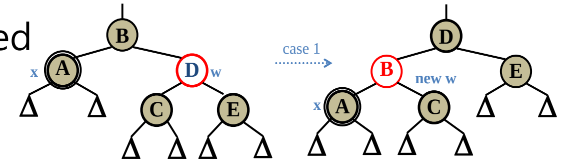
Key가 29인 노드 삭제



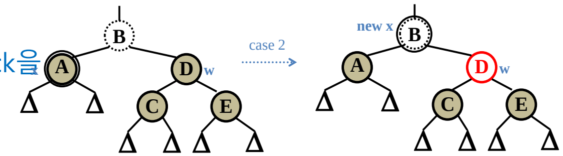
삭제 후 RBT



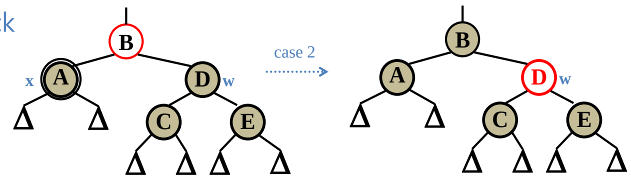
- Case 1: w == Red
  - w ← Black
  - P(x) ← Red
  - LeftRotate(p(x))



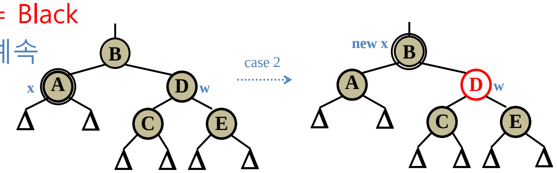
- Case 2: w의 두 자식이 모두 black
  - w ← Red
  - x의 여분의 Black을 부모에게 옮김



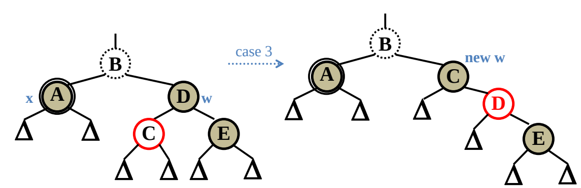
  - Case 2-1: P(x) == Red
  - P(x) ← Black
  - Done



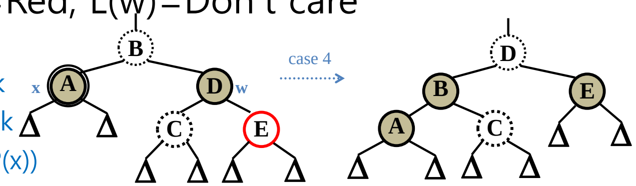
  - Case 2-2: P(x) == Black
  - P(x)를 x로 보고 계속
  - 이때 P(x)는 이중 Black을 가짐



- Case 3: R(w) == Black, L(w) == Red
  - w ← Red
  - L(w) ← Black
  - RightRotate(w)



- Case 4: R(w)=Red, L(w)=Don't care
  - w ← P(x)색
  - P(x) ← Black
  - R(w) ← Black
  - LeftRotate(P(x))
  - Done



RBT - Deletion

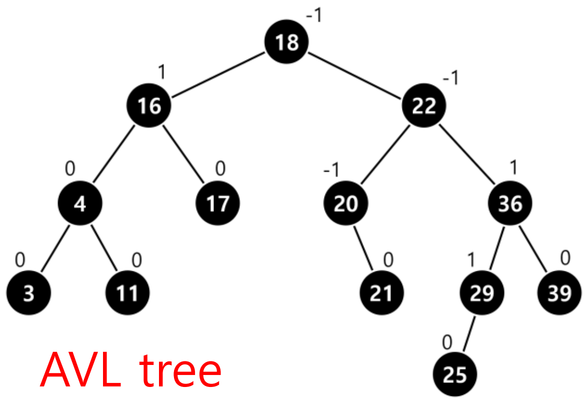
- delete 연산의 case 1, 3, 4에서 각 노드의 색 변화는 상수 번 변하며, 노드의 회전연산은 최대 3번 일어나게 된다.
- 그러나, case 2에서는 while 루프를 통하여 이중의 black 색을 가지는 노드를 계속 루트노드까지 올리게 된다.
- 따라서, delete 연산의 수행시간은 O(lg n)이다.

AVL Tree

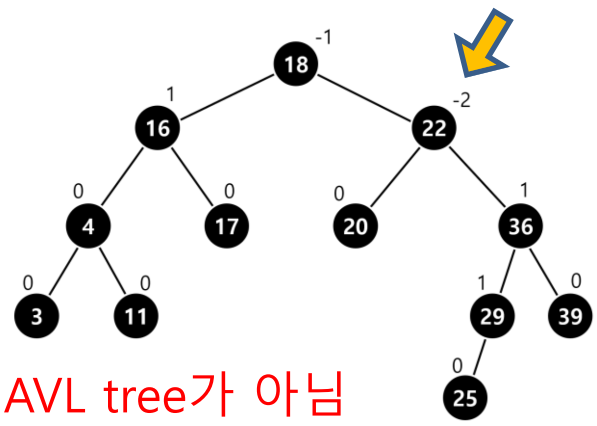
- AVL(**A**delson-**V**elsky and **L**andis) Tree: 본질적으로 BST (Binary Search Tree)이면서, RBT 처럼 균형트리(Balanced Tree) → Search, Insert, Delete 연산을 O(log n) 시간에 처리
- 트리의 높이를 이용하여 균형을 조정
- RBT 보다 상대적으로 균형이 더 잘 잡힌 트리

AVL Tree

- Balance Factor(BF) of node x:
  - BF(x) = H(Left subtree(x)) – H(Right subtree(x))
- H(x): x를 루트로 하는 서브트리의 높이
  - 높이: 노드 x에서 가장 먼 리프 노드까지의 경로에 존재하는 에지의 개수
- AVL tree
  - BF(x) ∈ {-1,0,1} for every node x in the tree



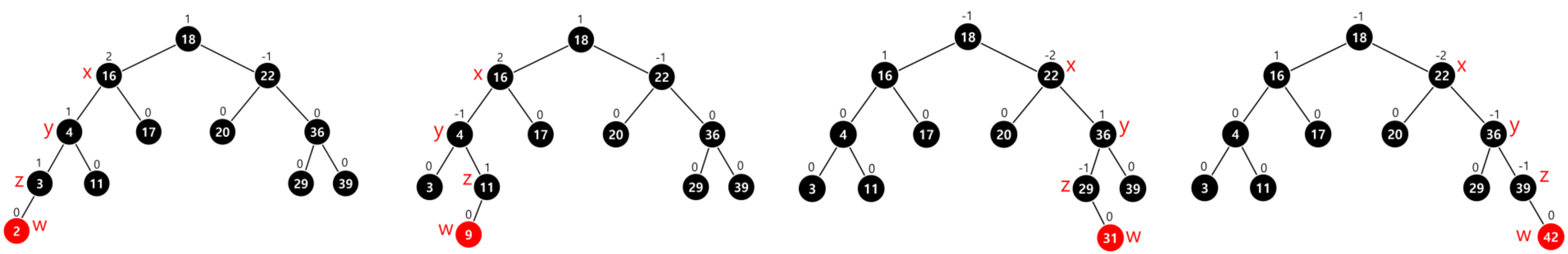
AVL tree



AVL tree가 아님

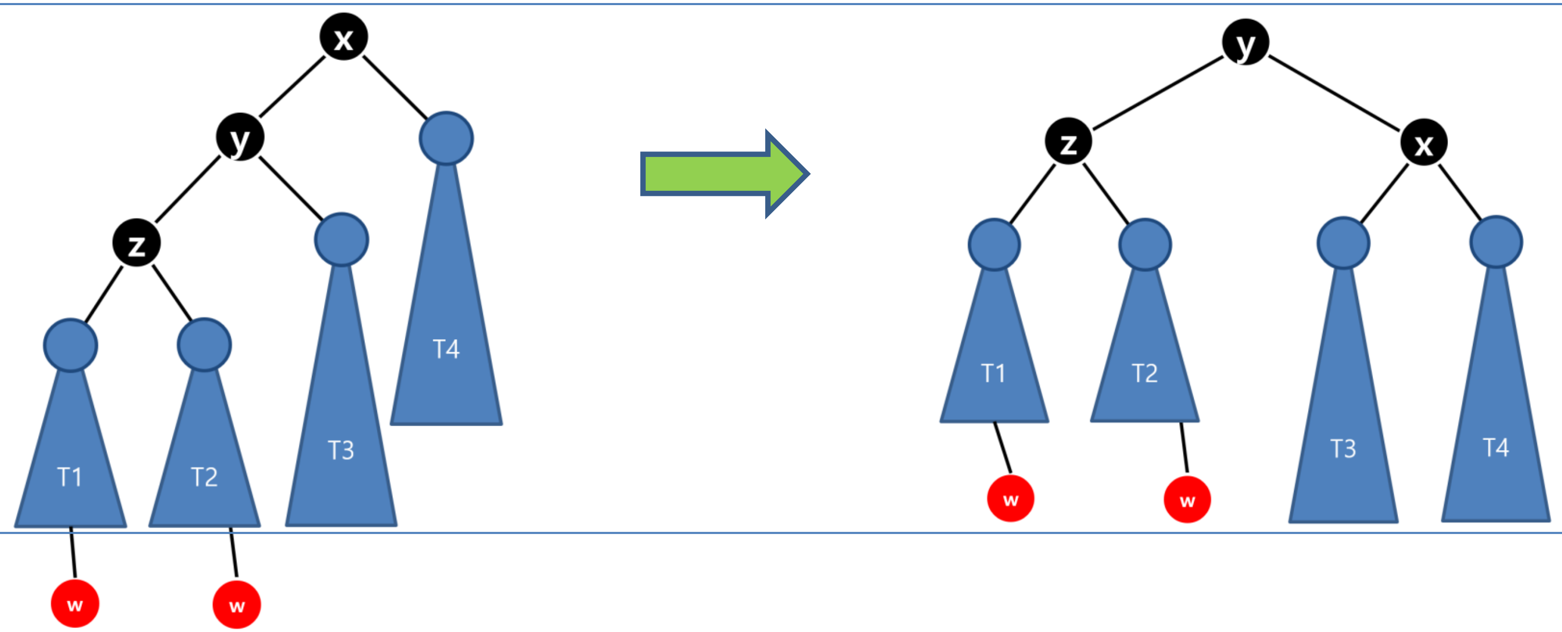
AVL Tree – Insertion

- BST에 입력하는 절차를 따라 입력
- 방금 입력된 노드를 w라 둠
- w로부터 루트로 검색해 가면서 BF 절대값이 2 이상이 되는 첫 노드를 x로, x의 자식을 y, y의 자식을 z로 둠 (BF 절대값이 2 이상인 노드가 없으면 그 자체로 이미 AVL tree 가 됨)
- 예: 4가지 케이스



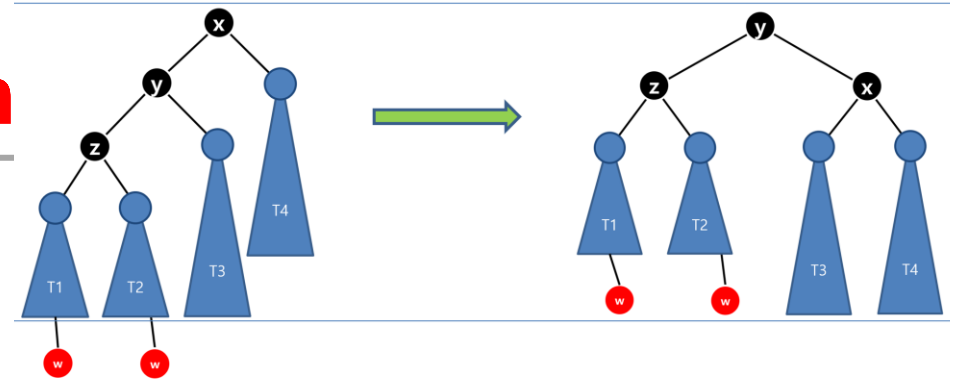
AVL Tree – Insertion

- Left Left Case
  - Right rotate (x)

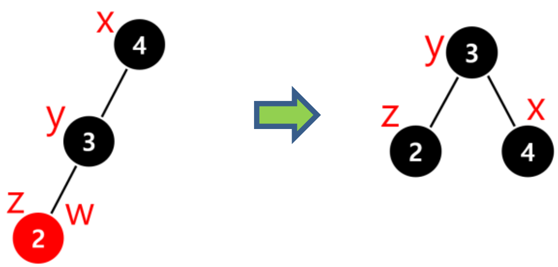


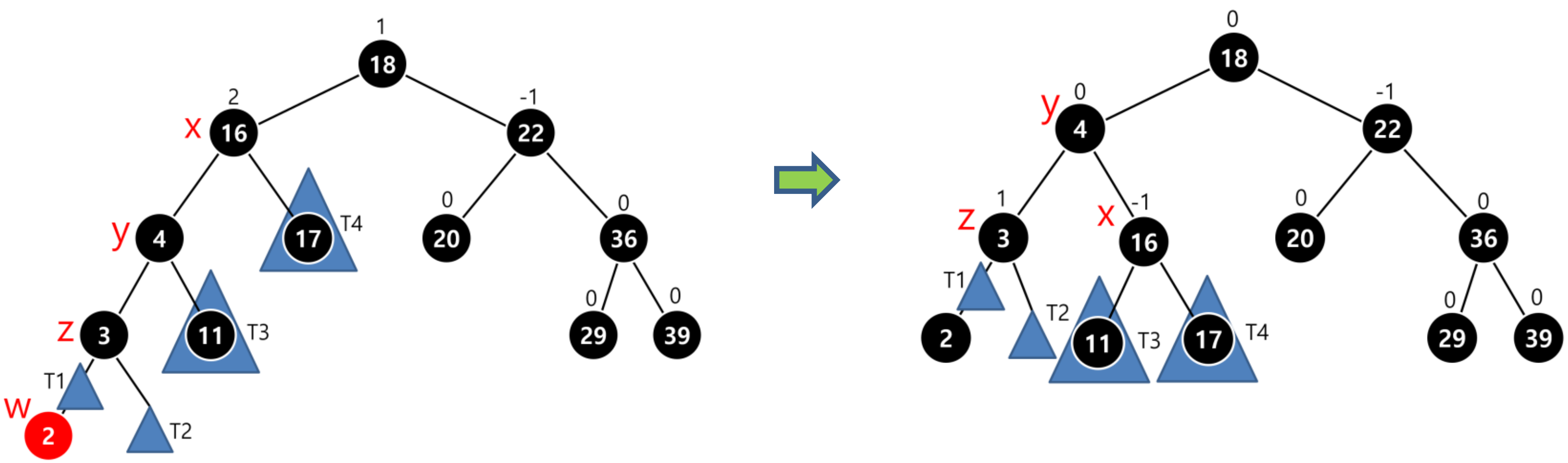
- Note: 트리의 높이가 유지됨

AVL Tree – Insertion



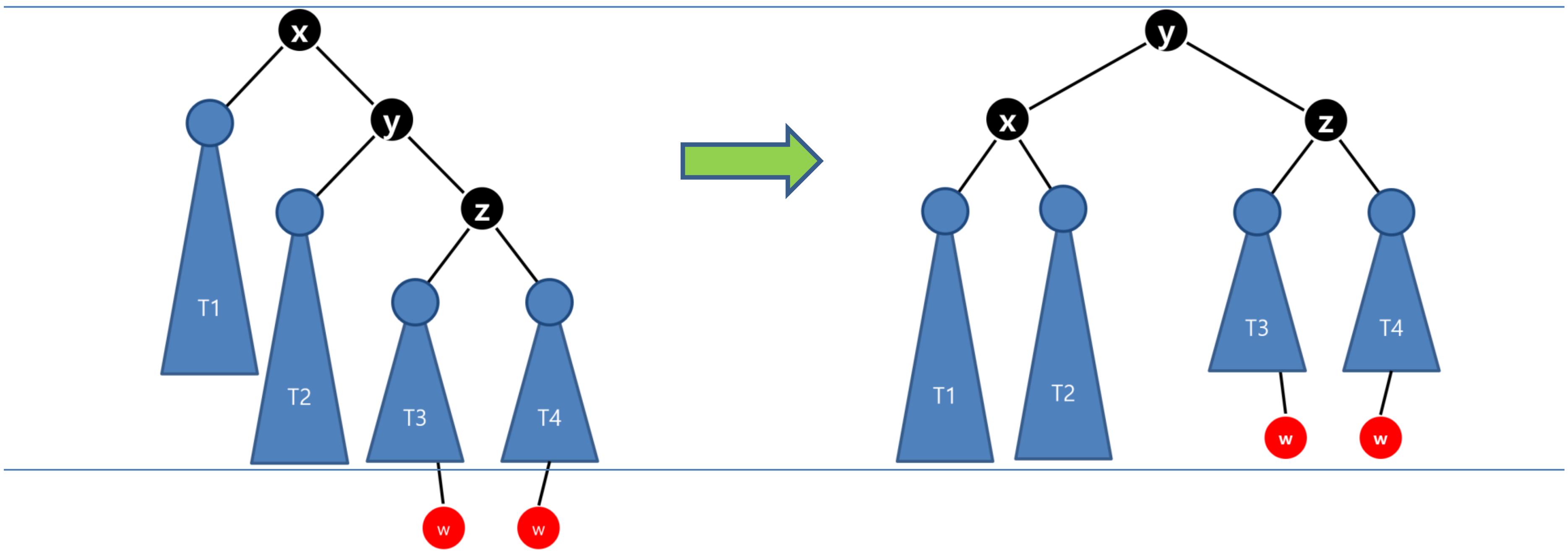
- Left Left Case 예





AVL Tree – Insertion

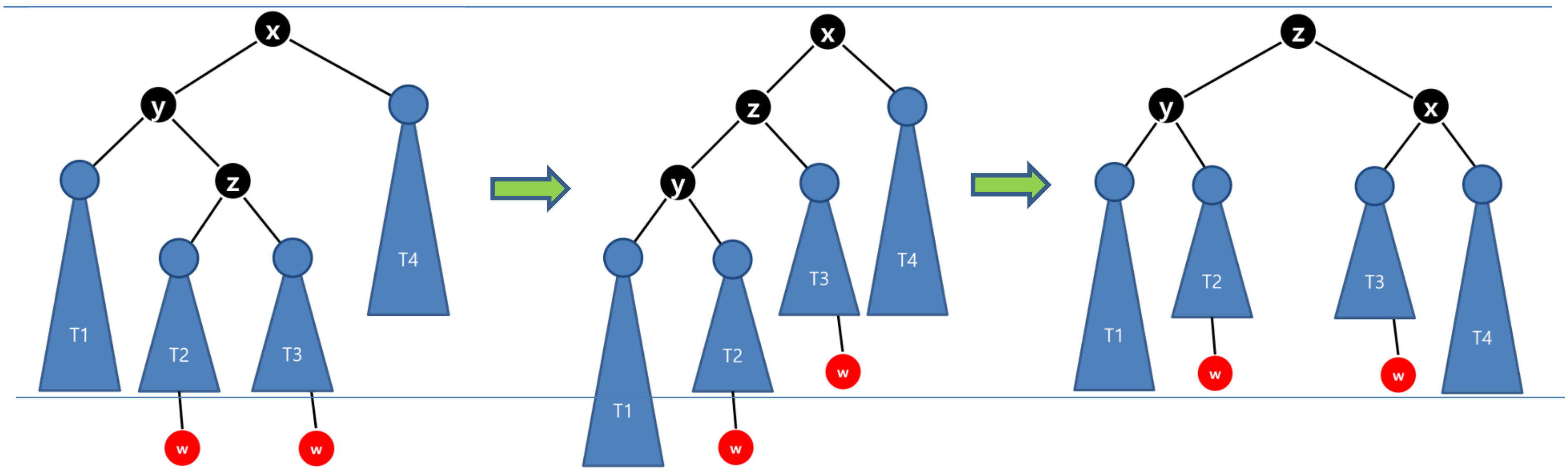
- Right Right Case
  - Left rotate (x)



- Note: 트리의 높이가 유지됨

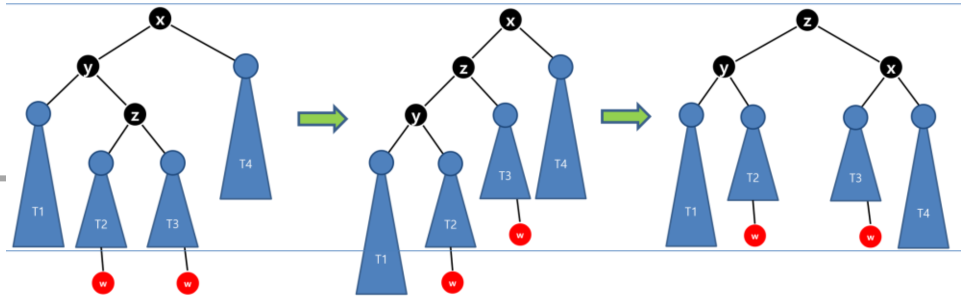
AVL Tree – Insertion

- Left Right Case
  - Left rotate (y) → Right rotate (x)

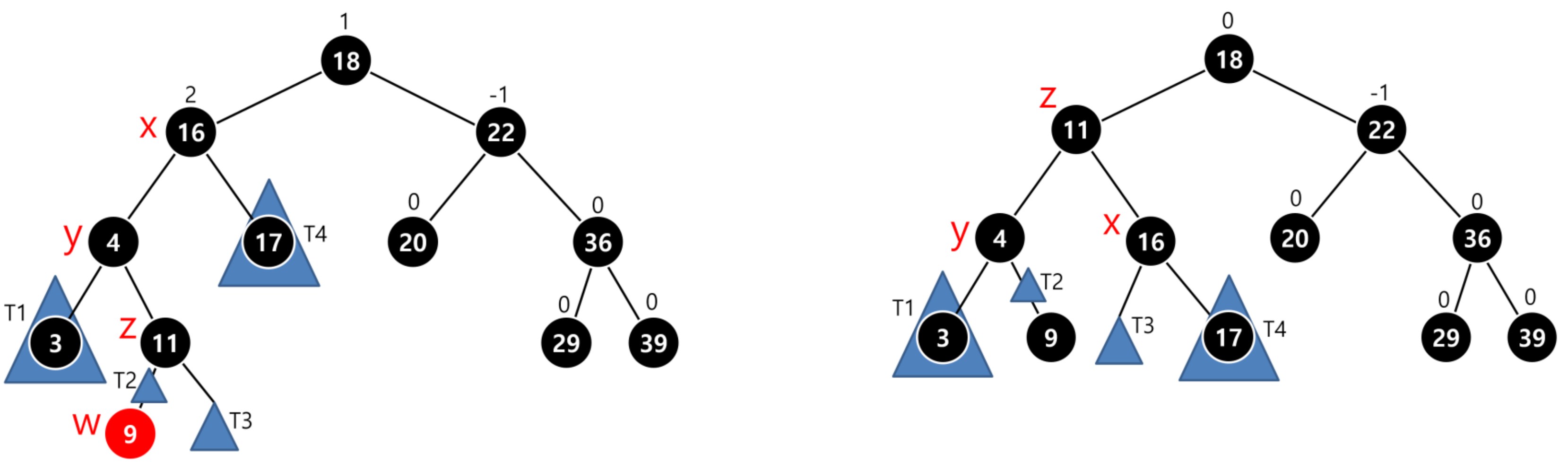


- Note: 트리의 높이가 유지됨

AVL Tree – Insertion

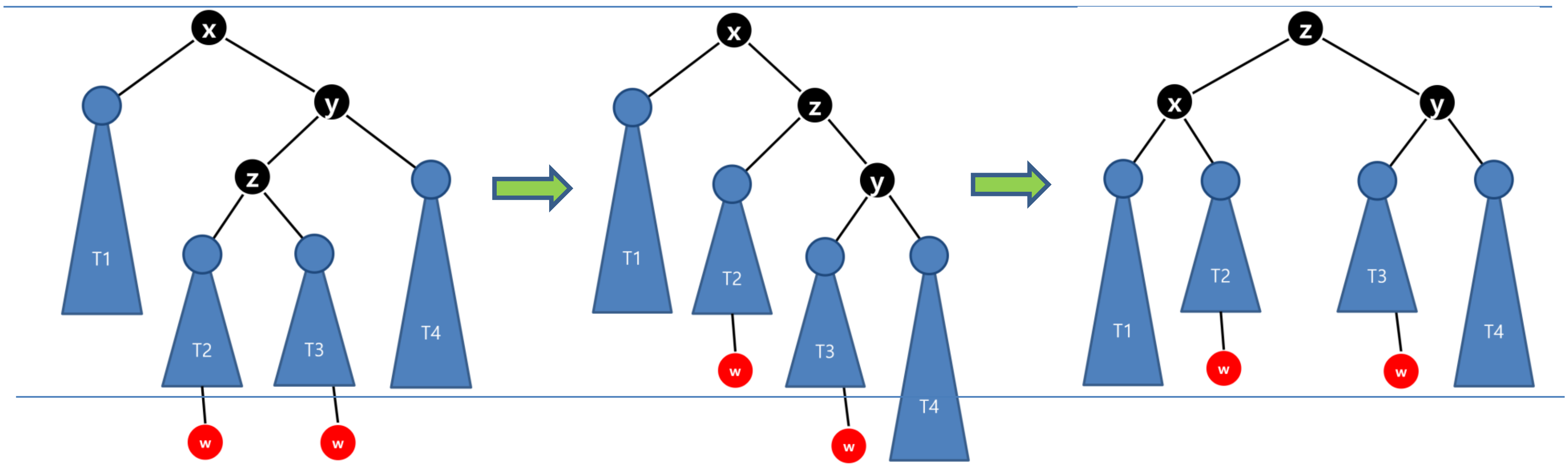


- Left Right Case 예



AVL Tree – Insertion

- Right Left Case
  - Right rotate (y) → Left rotate (x)



- Note: 트리의 높이가 유지됨

AVL Tree – Insertion

- https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

```cpp
// C++ program to insert a node in AVL tree
#include<bits/stdc++.h>
using namespace std;
// An AVL tree node
class Node
{
    public:
    int key;
    Node *left;
    Node *right;
    int height;
};

// A utility function to get the height of the tree
int height(Node *N)
{
    if (N == NULL) return 0;
    return N->height;
}
// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}
```

```cpp
/* Helper function that allocates a
   new node with the given key and
   NULL left and right pointers. */
Node* newNode(int key)
{
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially
                      // added at leaf
    return(node);
}
```

AVL Tree – Insertion

```
// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
Node *rightRotate(Node *y)
{
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
                    height(y->right)) + 1;
    x->height = max(height(x->left),
                    height(x->right)) + 1;

    // Return new root
    return x;
}
```

```
// A utility function to left
// rotate subtree rooted with x
// See the diagram given above.
Node *leftRotate(Node *x)
{
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left),
                    height(x->right)) + 1;
    y->height = max(height(y->left),
                    height(y->right)) + 1;

    // Return new root
    return y;
}
```

```
// Get Balance factor of node N
int getBalance(Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
```

```
// A utility function to print preorder
// traversal of the tree.
// The function also prints height
// of every node
void preOrder(Node *root)
{
    if(root != NULL)
    {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

AVL Tree – Insertion

```
// Recursive function to insert a key
// in the subtree rooted with node and
// returns the new root of the subtree.
Node* insert(Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

   /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                           height(node->right));

   /* 3. Get the balance factor of this ancestor
        node to check whether this node became
        unbalanced */
    int balance = getBalance(node);

    // 우측에서 코드가 계속됨
```

```
    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}
```

AVL Tree – Insertion

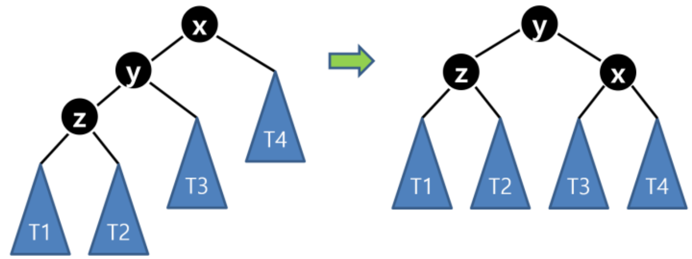
```
// Driver Code
int main()
{
    Node *root = NULL;

    /* Constructing tree given in
    the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    cout << "Preorder traversal of the "
            "constructed AVL tree is \n";
    preOrder(root);

    return 0;
}


// This code is contributed by rathbhupendra
```

AVL Tree – Deletion

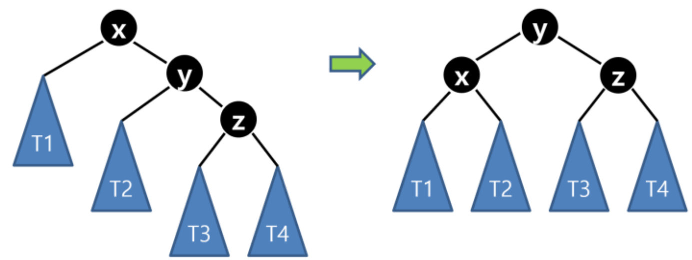
- BST에서 삭제하는 절차를 따라 삭제
- 실제로 삭제된 노드의 부모를 w라 둠
- w로부터 루트로 검색해 가면서 BF 절대값이 2 이상이 되는 첫 노드를 x로 둠
- x의 두 자식 중 높이가 높은 것을 y로 둠
- y의 두 자식 중 높이가 높은 것을 z로 둠
- Note: BF 절대값이 2 이상인 노드가 없으면 그 자체로 이미 AVL tree 가 됨

AVL Tree – Deletion

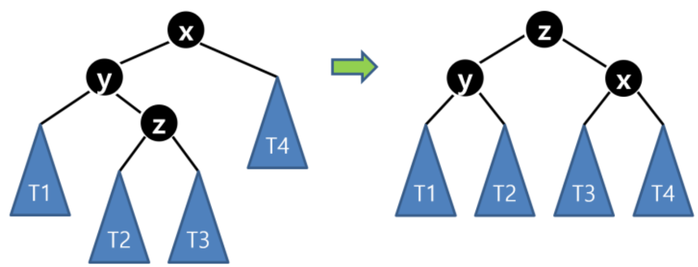
- Left Left Case



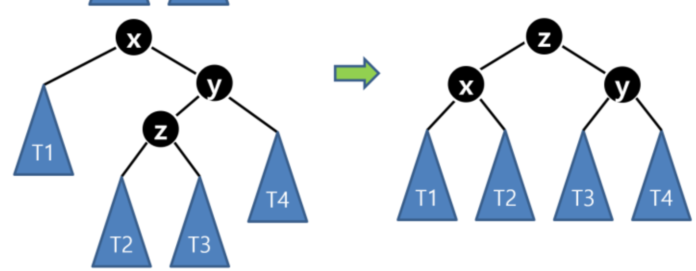
- Right Right Case



- Left Right Case



- Right Left Case



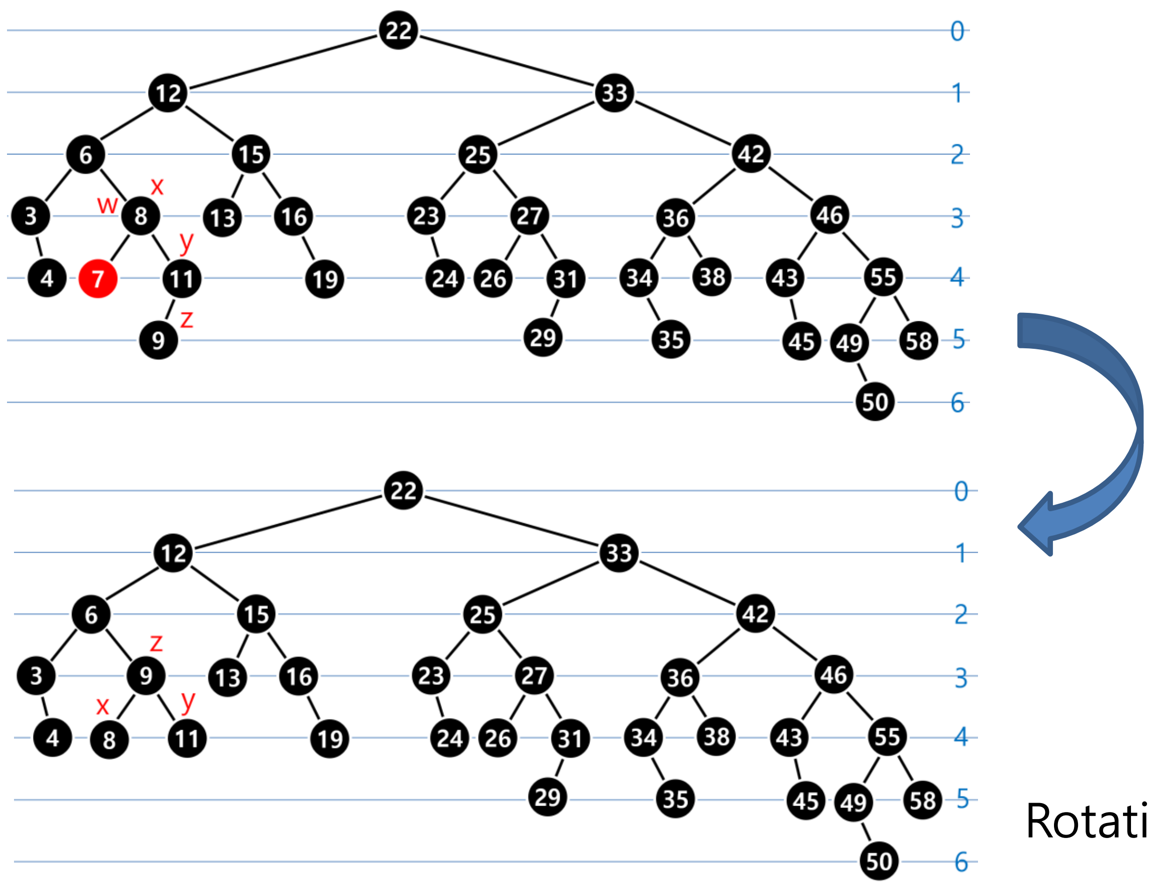
  - 모든 경우 높이가 줄어 듬

AVL Tree – Deletion

- Rotation 후 높이가 감소할 수도 있기 때문에, insertion 경우와는 다르게 처리
- 노드 x에서 rotation 후, 방금 회전한 서브트리의 루트를 w로 둠
- w가 루트가 아니면 아래 과정을 반복
  - w에서 루트까지 높이 검사를 하여 트리가 불균형인지 검사
  - 불균형이면 케이스에 따라 회전한 후, 방금 회전한 서브트리의 루트를 w로 둠

AVL Tree – Deletion

- 예: 아래 그림에서 노드 7 삭제

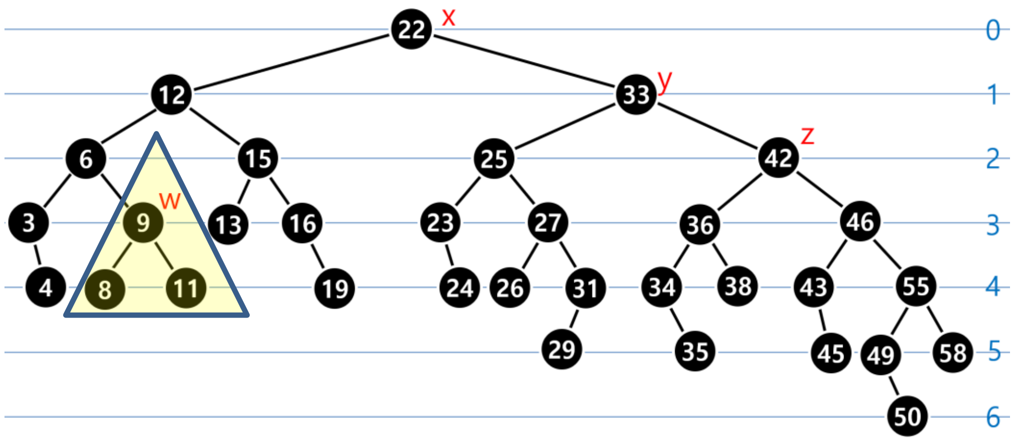


Right Left Case

Rotation 후 트리는 불균형

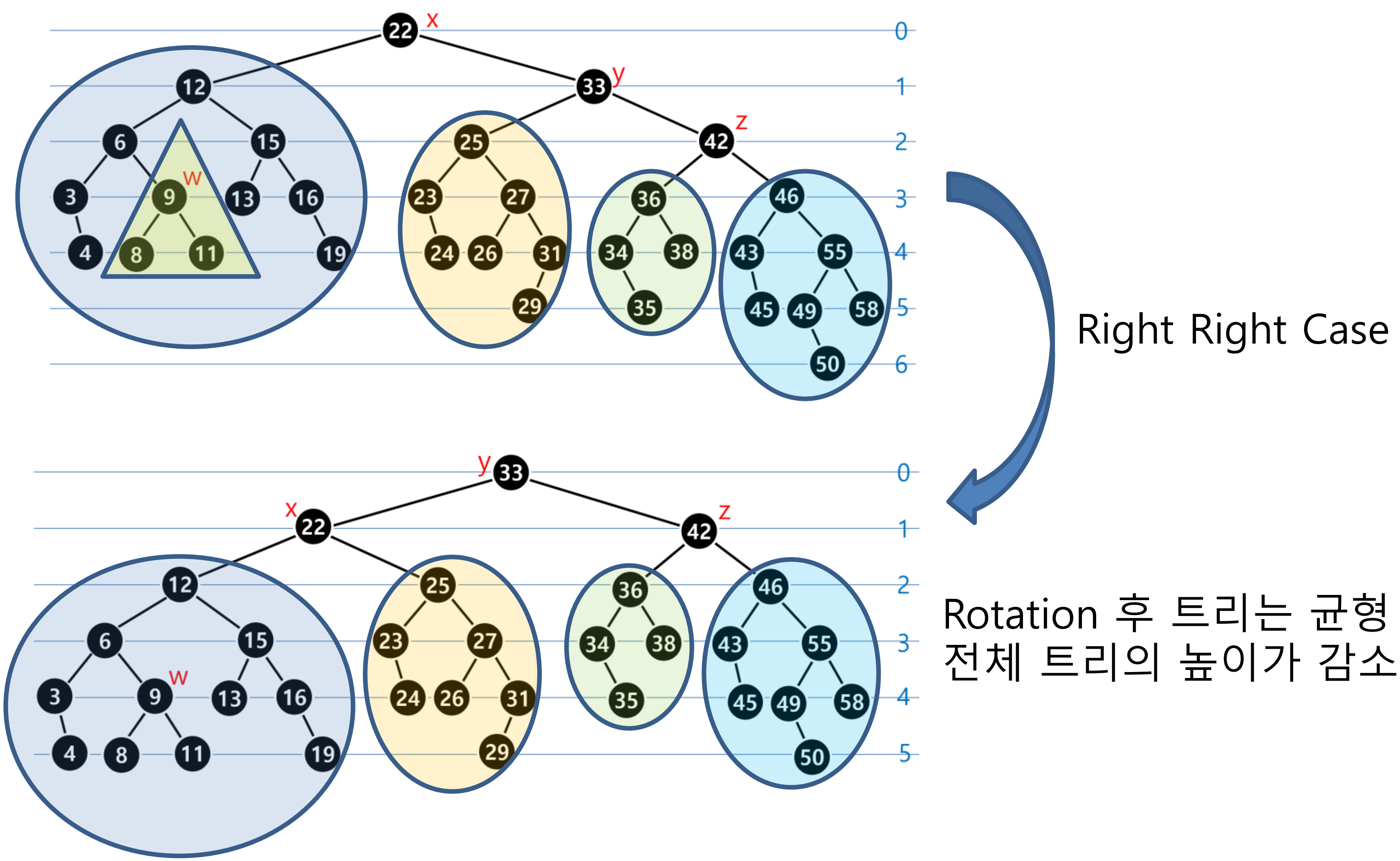
AVL Tree – Deletion

- 노드 7 삭제 → rotation : 여전히 불균형



AVL Tree – Deletion

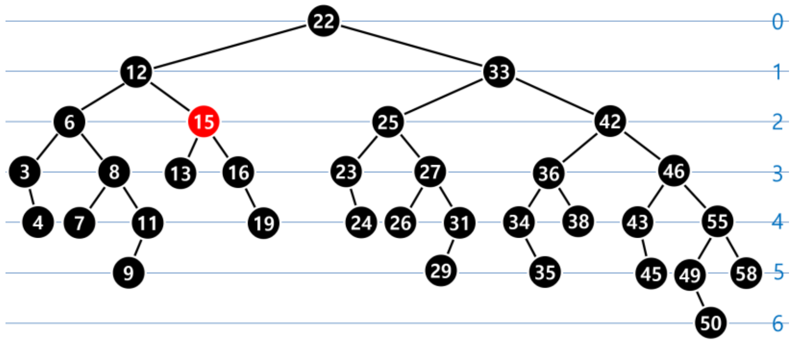
- 노드 7 삭제 → rotation : 여전히 불균형



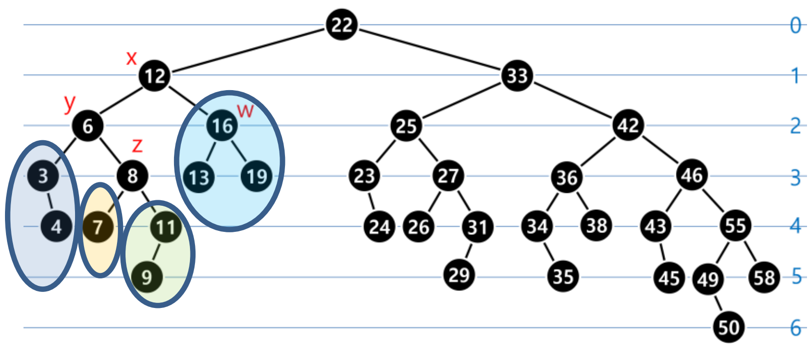
Rotation 후 트리는 균형
전체 트리의 높이가 감소

AVL Tree – Deletion

- 예: 아래 그림에서 노드 15 삭제

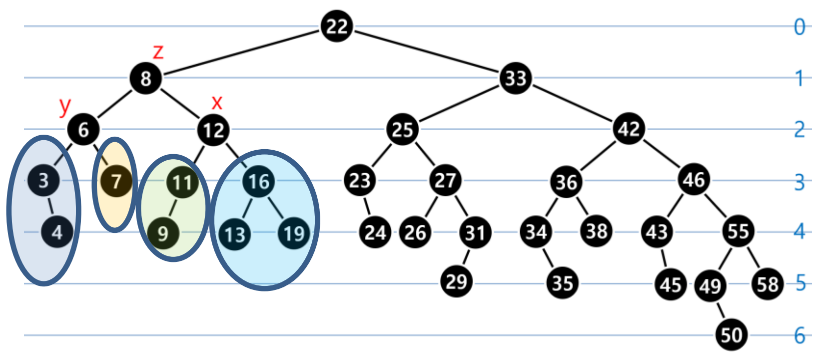


15의 successor 16과 위치 교환 후
15 삭제



삭제된 노드의 부모 노드 16을 w로 둠
전체 트리는 불균형

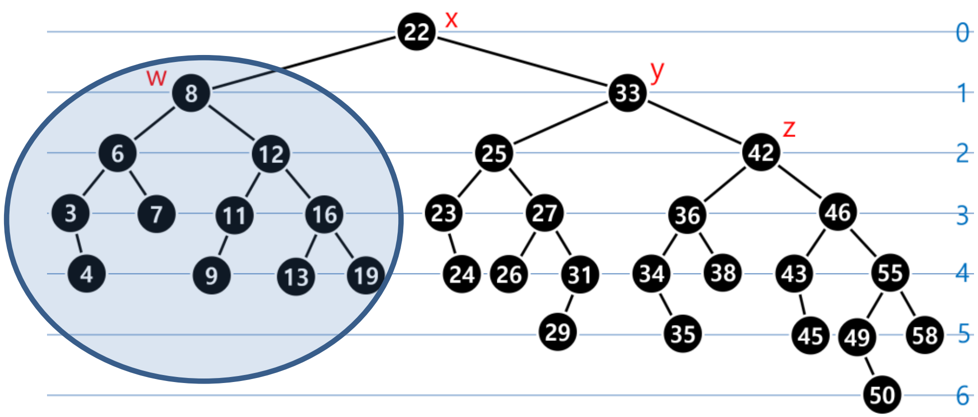
Left Right Case



Rotation 후 트리는 불균형

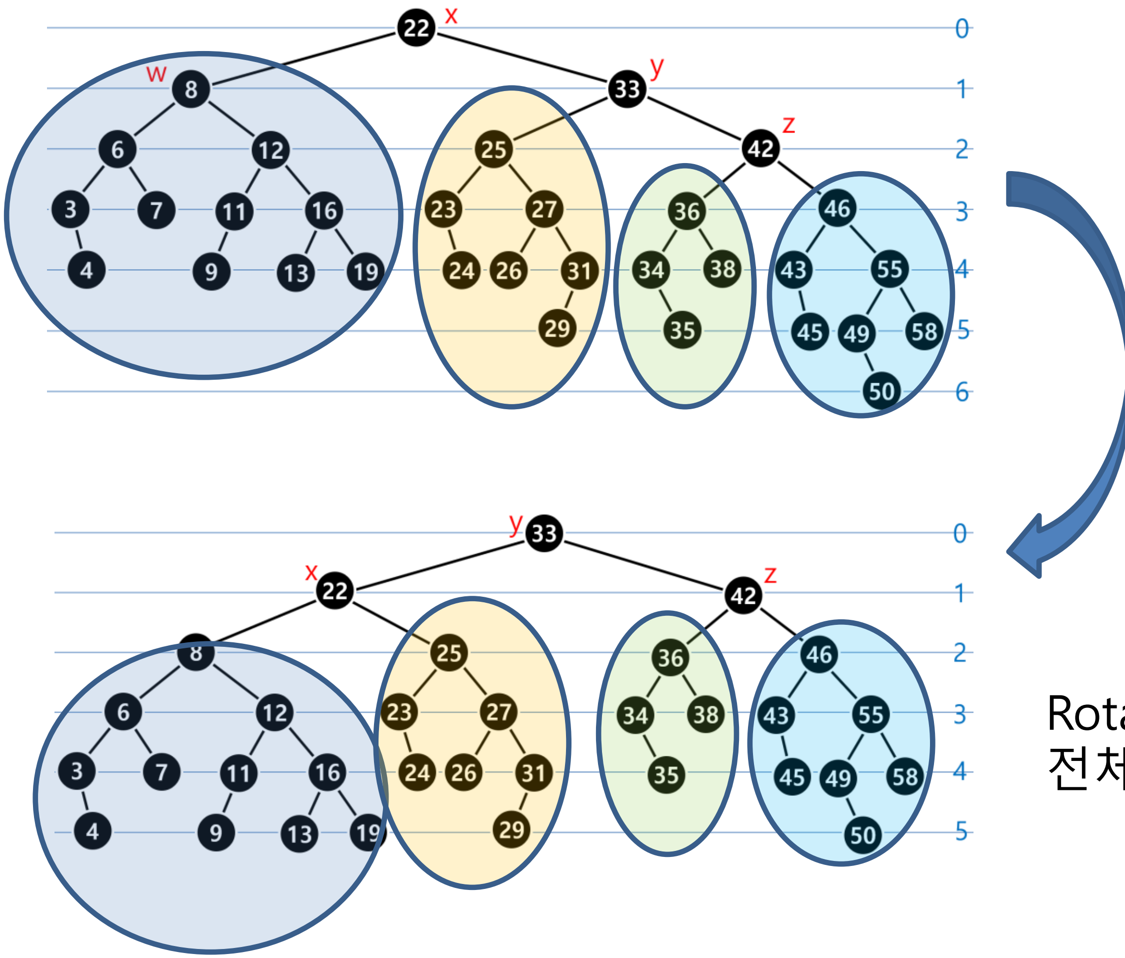
AVL Tree – Deletion

- 노드 15 삭제 → rotation : 여전히 불균형



AVL Tree – Deletion

- 노드 15 삭제 → rotation : 여전히 불균형



Right Right Case

Rotation 후 트리는 균형
전체 트리의 높이가 감소

AVL Tree – Deletion

- https://www.geeksforgeeks.org/avl-tree-set-2-deletion/

```
// Driver Code
int main()
{

    Node *root = NULL;

    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);

    cout << "Preorder traversal of the "
            "constructed AVL tree is \n";
    preOrder(root);

    root = deleteNode(root, 10);

    cout << "\nPreorder traversal after"
         << " deletion of 10 \n";
    preOrder(root);

    return 0;
}

// This code is contributed by rathbhupendra
```

- **Note**: 여기서 제시된 코드에서는 삭제할 노드의 successor 대신 predecessor와 교체함 (함수 minVlaueNode() 를 이용)

```
/* Given a non-empty binary search tree, return the node with
minimum key value found in that tree.
Note that the entire tree does not need to be searched. */

Node * minValueNode(Node* node)
{
    Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}
```

AVL Tree – Deletion

```
// Recursive function to delete a node with given key
// from subtree with given root. It returns root of the
// modified subtree.
Node* deleteNode(Node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the
    // root's key, then it lies in left subtree
    if ( key < root->key )
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the
    // root's key, then it lies in right subtree
    else if( key > root->key )
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then
    // This is the node to be deleted
    else {
        // node with only one child or no child
        if( (root->left == NULL) || (root->right == NULL) ) {
            Node *temp = root->left ? root->left : root->right;

            // No child case
            if (temp == NULL) {
                temp = root;
                root = NULL;
            }
            else // One child case
            *root = *temp; // Copy the contents of
                           // the non-empty child
            free(temp);

            // 우측에서 계속
```

```
        } else {
            // node with two children: Get the inorder
            // successor (smallest in the right subtree)
            Node* temp = minValueNode(root->right);
            // Copy the inorder successor's data to this node
            root->key = temp->key;
            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }
    // If the tree had only one node then return
    if (root == NULL) return  root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = 1 + max(height(root->left),
                           height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF
    // THIS NODE (to check whether this node became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}
```

Comparison between RBT & AVL

- AVL trees provide **faster lookups** than Red Black Trees because they are more strictly balanced.
- Red Black Trees provide **faster insertion and removal** operations than AVL trees as fewer rotations are done due to relatively relaxed balancing.
- AVL trees store **balance factors or heights** with each node, thus requires storage for an integer per node whereas Red Black Tree requires only 1 bit of information per node.
- Red Black Trees are used in most of the language libraries like map, multimap, multiset in C++ whereas AVL trees are used in **databases** where faster retrievals are required.