Dynamic Sets

- 다음과 같은 연산을 효율적으로 처리할 수 있도록 지원하는 자료구조
  - Dynamic set S 의 각 원소는 key 이외에 여러 개의 보조 데이터를 가지고 있다.
  - Dynamic set은 다음과 같은 쿼리(query)를 지원한다.
    - Search (S, k)
    - Mininum (S)
    - Maximum (S)
    - Successor (S, x)
    - Predecessor (S, x)
  - 또한, 다음과 같은 수정 연산자를 지원한다
    - Insert (S, x)
    - Delete (S, x)
  - 위의 모든 연산은 O(log n) 시간에 처리 가능해야 함

Dynamic Sets

- Dynamic Set 자료구조
  - Binary Search Tree - AVL Tree
  - 2-3 Tree - Red-Black Tree
  - Splay Tree

- 추가적인 연산이 가능한 자료구조
  - Interval Tree - Range Tree
  - Segment Tree - Fenwick Tree(BIT)

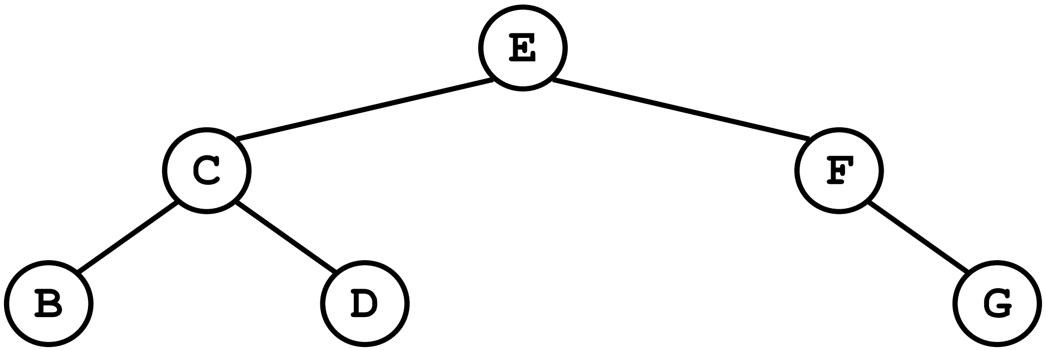
Binary Search Tree (BST)

- Dynamic Set을 처리하는 가장 기본적인 자료구조
- Tree구조를 지원하기 위하여 다음과 같은 필드 데이터를 가진다.
  - key : 원소의 순서를 정하기 위한 자료
  - left : pointer to left child (혹은 NULL)
  - right : pointer to right child (혹은 NULL)
  - p : pointer to a parent node (root는 NULL 을 가짐)

- BST는 다음과 같은 성질을 만족한다
  - Key[leftSubtree(x)] ≤ key[x] ≤ key[rightSubtree(x)]

Binary Search Tree (BST)

- BST의 예



- BST에서 각 원소를 정렬된 순서로 출력하는 함수 InorderTreeWalk()

```
InorderTreeWalk(x)
    inorderTreeWalk(x->left);
    print(x);
    inorderTreeWalk (x->right);
```

- 출력결과: B C D E F G

Binary Search Tree (BST)

- search(x, k): 노드 x를 루트로 하는 BST에서 주어진 key 값 k와 같은 key를 가지는 노드의 포인터를 리턴하는 함수

```
TreeSearch(x, k)    // recursive
        if (x == NULL  or  k == x->key)
            return x;
        if (k < x->key)
            return TreeSearch(x->left, k);
        else
            return TreeSearch(x->right, k);
```

```
TreeSearch(x, k)   // non-recursive
        while (x != NULL  and k != x->key)
            if (k < x->key)
                x = x->left;
            else
                x = x->right;
        return x;
```

Binary Search Tree (BST)

- TreeMinimum(x): 노드 x를 루트로 하는 BST에서 가장 작은 key를 찾는 함수
- TreeMaximum(x): 노드 x를 루트로 하는 BST에서 가장 큰 key를 찾는 함수

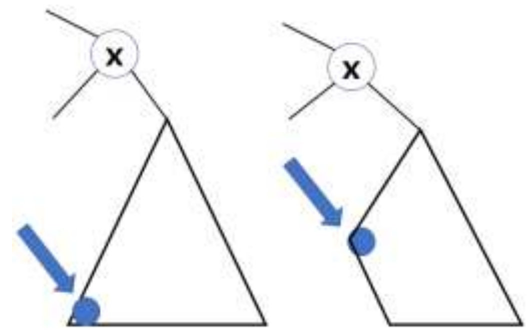
```
TreeMinimum(x)
        while (x->left != NULL)
            x = x->left;
        return x;
```

```
TreeMaximum(x)
        while (x->right != NULL)
            x = x->right;
        return x;
```

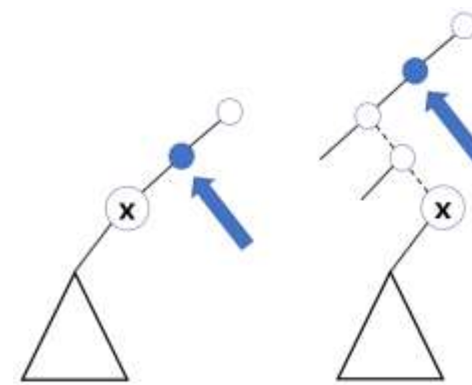
Binary Search Tree (BST)

- Successor(x): 주어진 노드 x의 key 보다 큰 key 중에서 가장 작은 key를 가지는 노드를 찾는 함수
- 다음과 같은 두 가지 경우

  (Case 1) x의 right subtree가 있는 경우
    - x의 right subtree에서 가장 작은 노드



  (Case 2) x의 right subtree가 없는 경우
    - x의 가장 가까운 조상노드(ancessor) 중에서 그 조상 노드의 left child 가 또한 x의 조상 노드인 경우이다.



Binary Search Tree (BST)

- Successor(x) 함수: 주어진 노드 x의 key 보다 큰 key 중에서 가장 작은 key를 가지는 노드를 찾는 함수

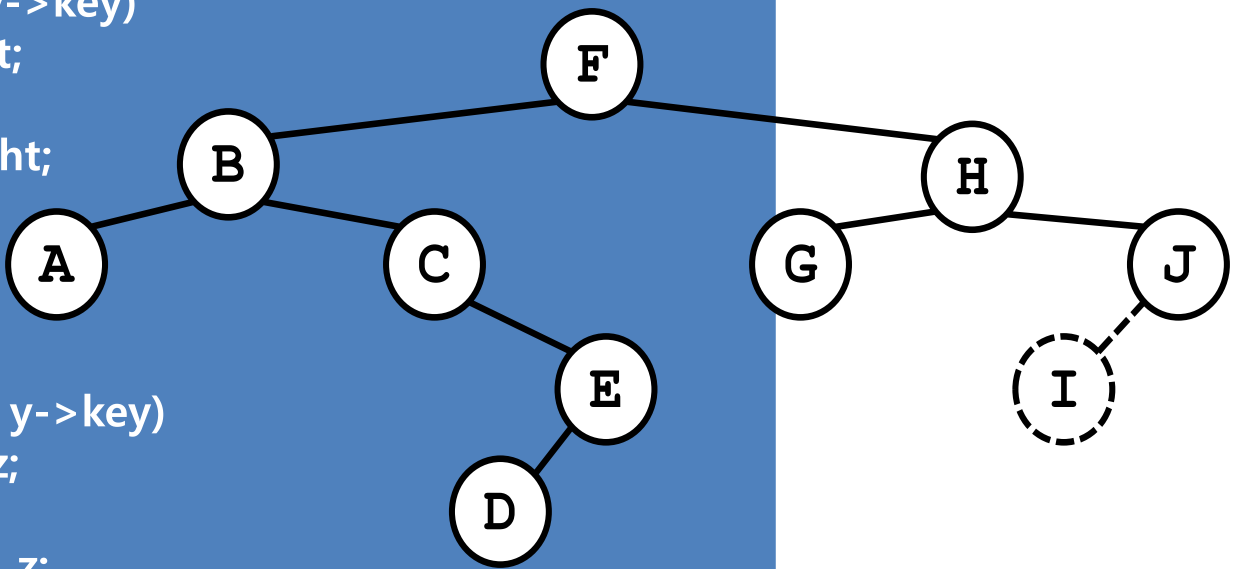
```
TreeSuccessor(x)
        if (x->right != NULL)
            return TreeMinimum(x->right);
        y = x->p;
        while (y != NULL and x == y->right){
            x = y;
            y = y->p;
        }
        return y;
```

- Predecessor(x): 주어진 노드 x의 key 보다 작은 key 중에서 가장 큰 key를 가지는 노드를 찾는 함수 → Successor()와 유사하게 만들 수 있음

Binary Search Tree (BST)

- TreeInsert(T,z) 함수: BST T에 임의의 노드 z를 입력하는 함수

```
TreeInsert(T, z)
        y = NULL;
        x = T.root;
        while (x != NULL){
             y = x;
             if (z->key < y->key)
                  x = x->left;
             else
                  x = x->right;
        }
        z->p = y;
        if (y == NULL)
             T.root = z;
        else if (z->key < y->key)
                  y->left = z;
        else
                  y->right = z;
```



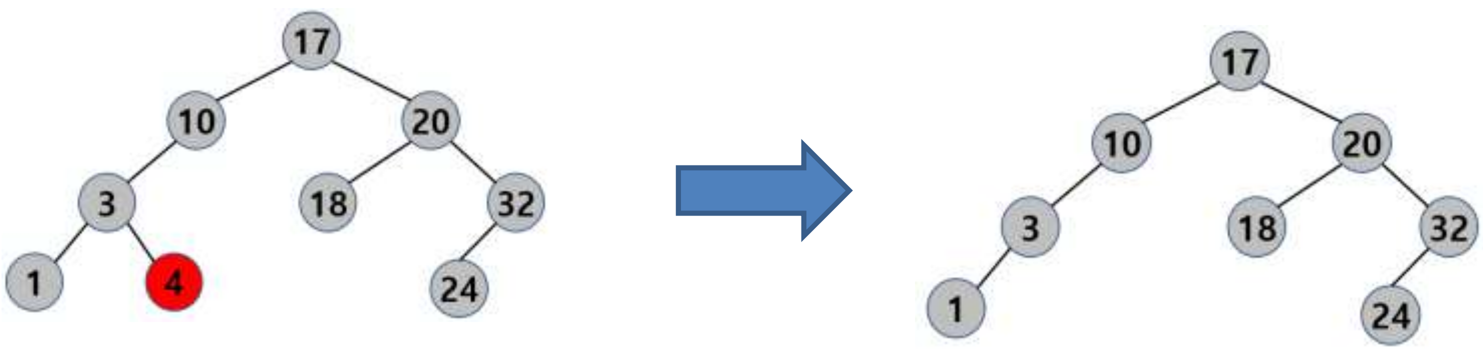
Binary Search Tree (BST)

- TreeDelete(T,x) 함수: BST T에 임의의 노드 x를 제거하는 함수
- 다음과 같은 세 가지 경우

  (Case 1) x가 child가 없는 경우

  (다음 그림에서 1, 4, 18, 24 처럼 단말 노드)

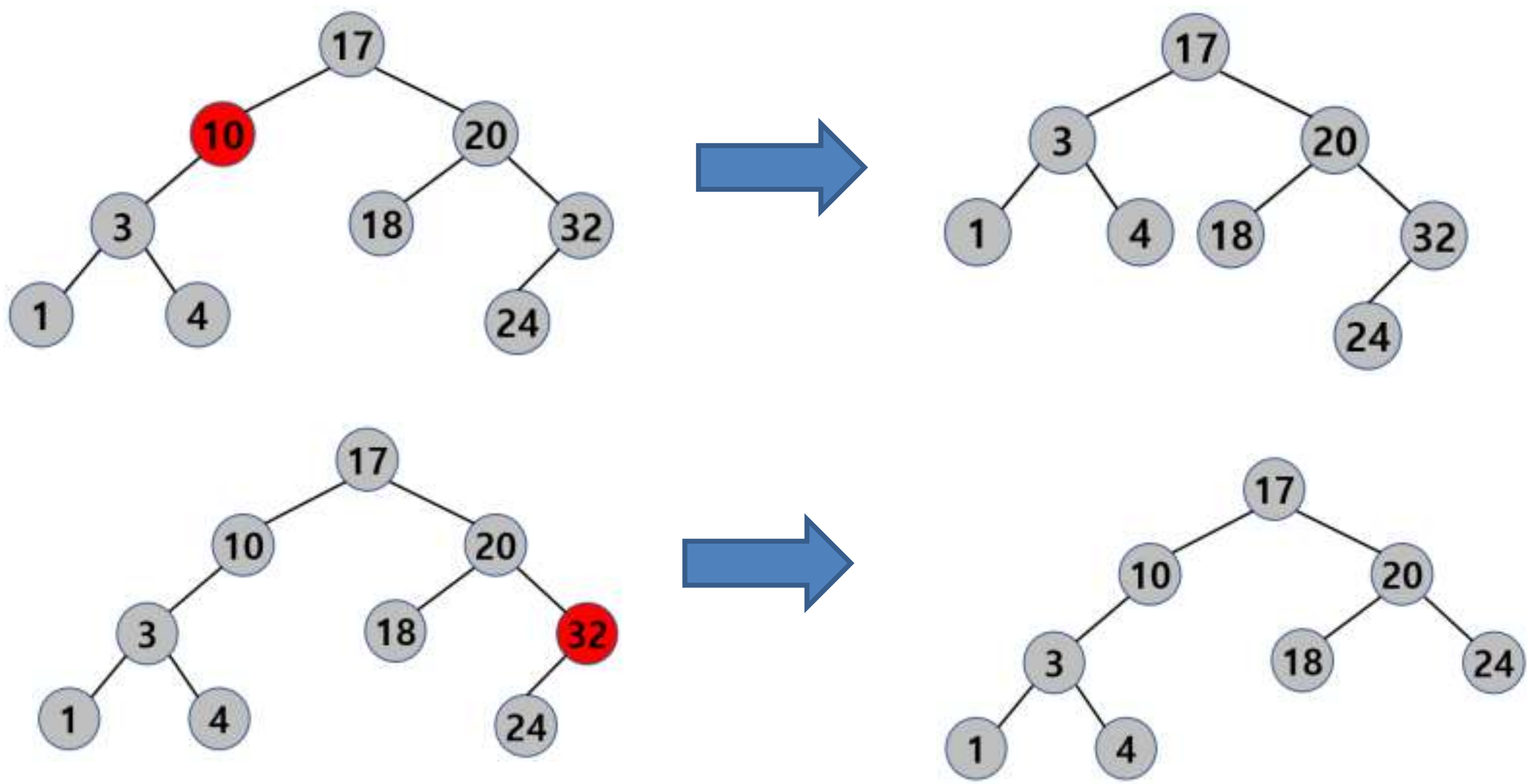
  - 단순히 x를 제거



Binary Search Tree (BST)

(Case 2) x가 1개의 child를 가지는 경우
(다음 그림에서 10, 32)

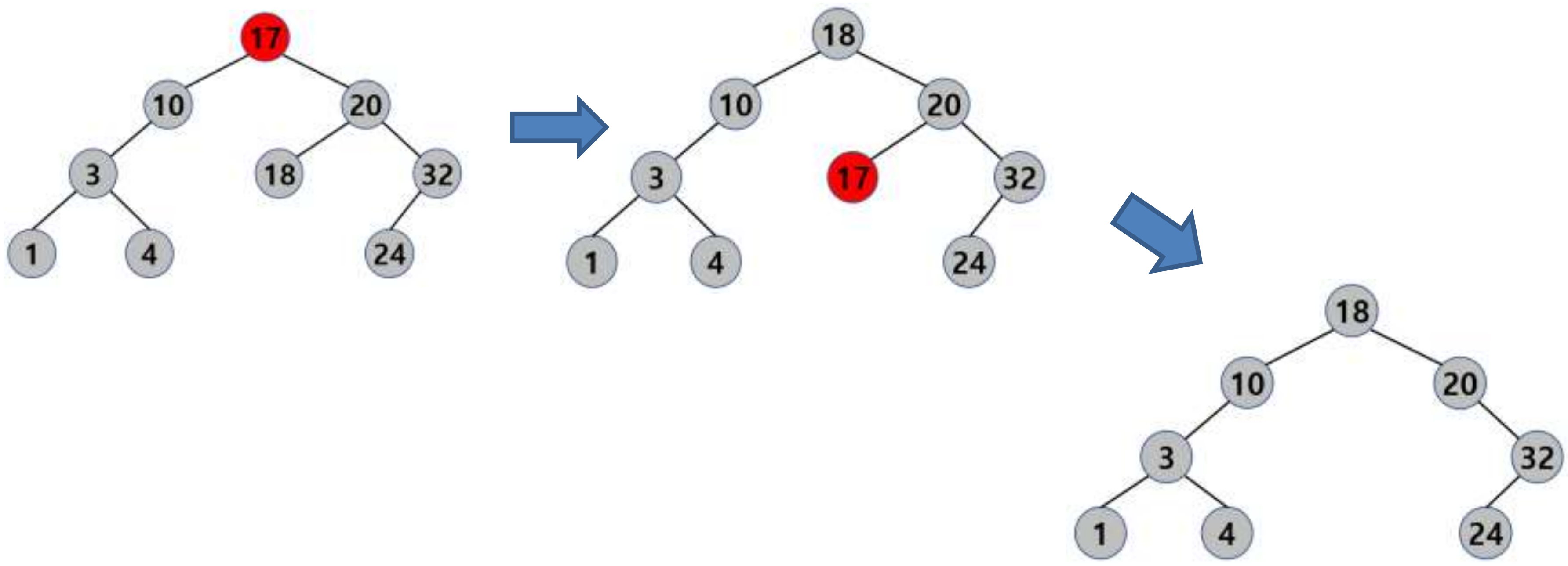
- x의 자식 노드를 x의 부모노드의 자식노드로 만든 후 x를 제거



Binary Search Tree (BST)

(Case 3) x가 2개의 child를 가지는 경우
(다음 그림에서 3, 17, 20)

- x를 x의 successor와 그 위치를 맞바꾸고, x에 대하여 위 (case 1), (case 2) 경우를 적용하여 x를 제거한다.



Binary Search Tree (BST)

```
TreeDelete(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)
        S.root = x;
    else if (y == y->p->left)
        y->p->left = x;
    else
        y->p->right = x;

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

    return y;
```

Binary Search Tree (BST)

- Tree 높이가 h인 BST에서 Search(), Minimum(), Maximum(), Predecessor(), Successor(), Insert(), Delete() 는 모두 O(h) 시간 복잡도를 가진다.
- 원소의 개수가 n인 BST의 높이는 최악의 경우 (n-1)이다.
- 따라서, 위 연산 함수의 시간복잡도는 O(n)이다.

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.

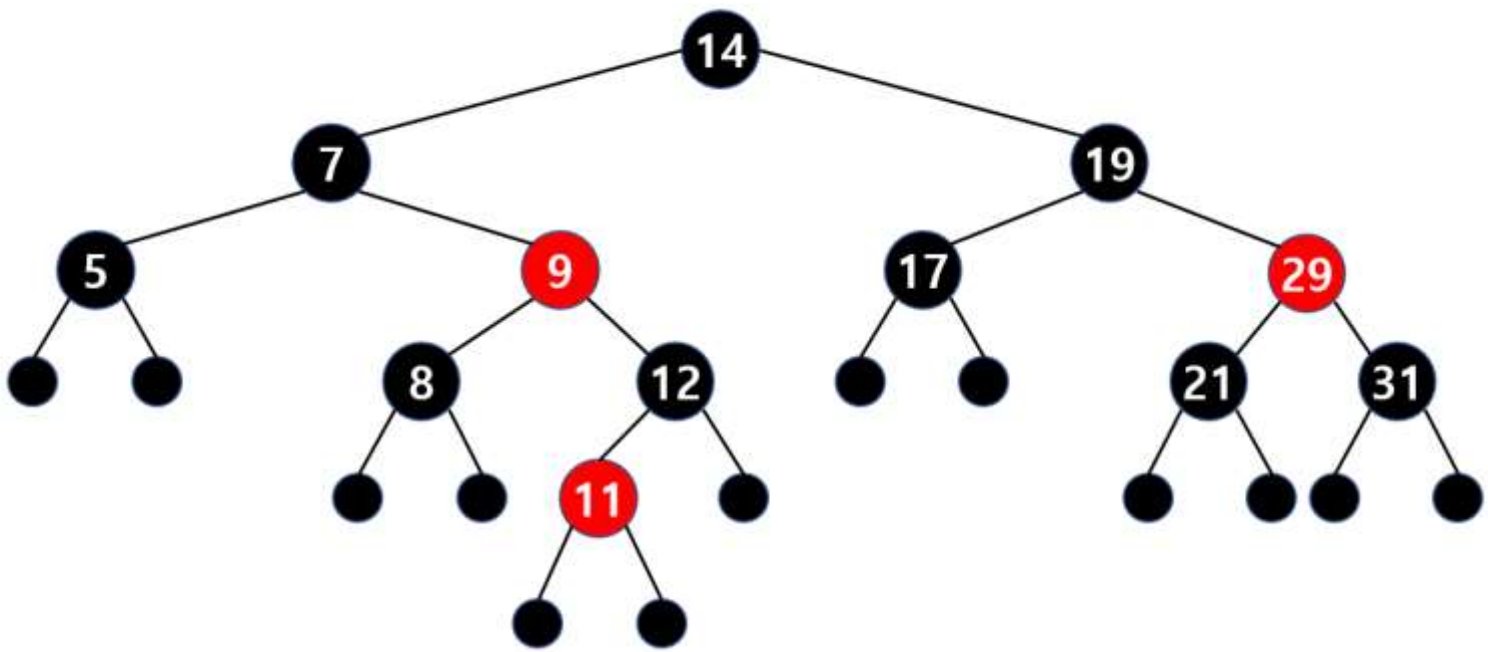
(성질 4) 어떤 노드로부터 이 노드의 모든 단말노드 까지의 경로는 모두 같은 개수의 Black 노드를 가진다.

(성질 5) 루트노드는 Black이다.

root 에서 leaf 까지의 Black노드 수가 같다.

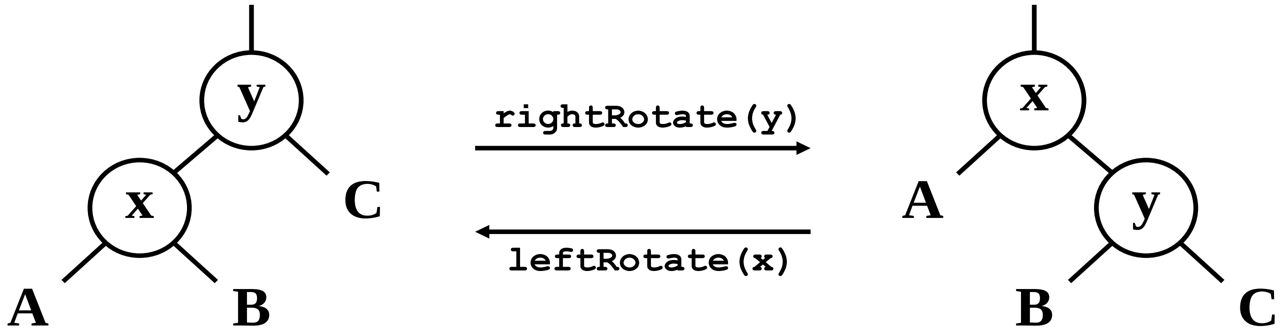
Red-Black Tree (RBT)

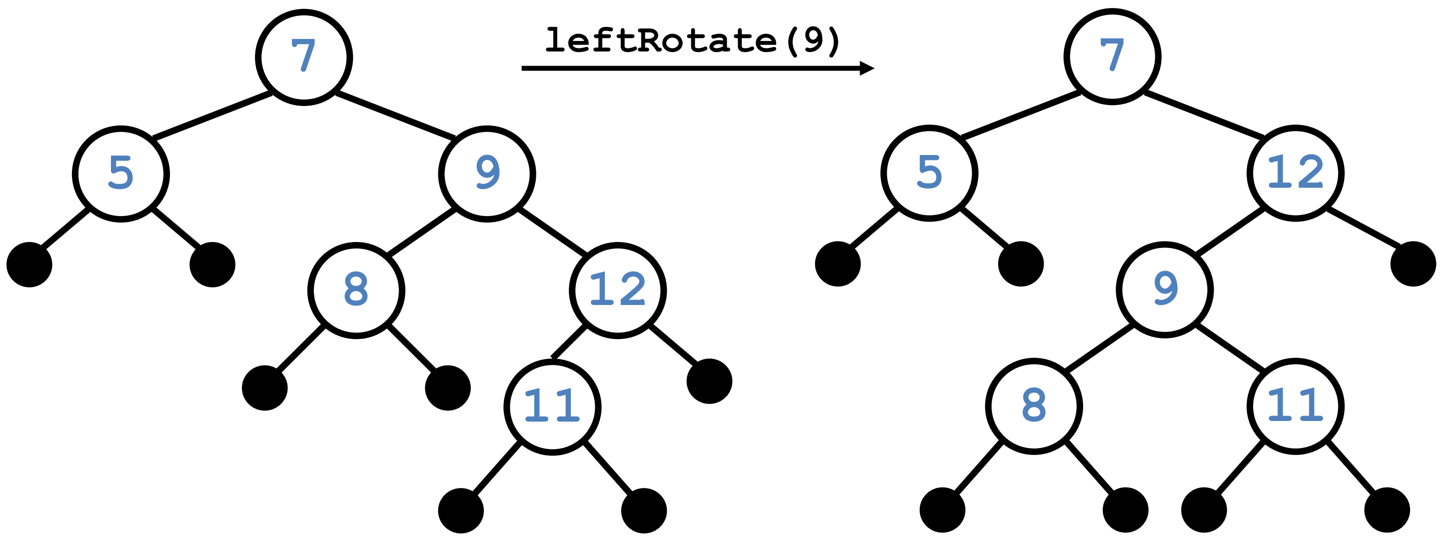
- RBT의 한 예



- 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

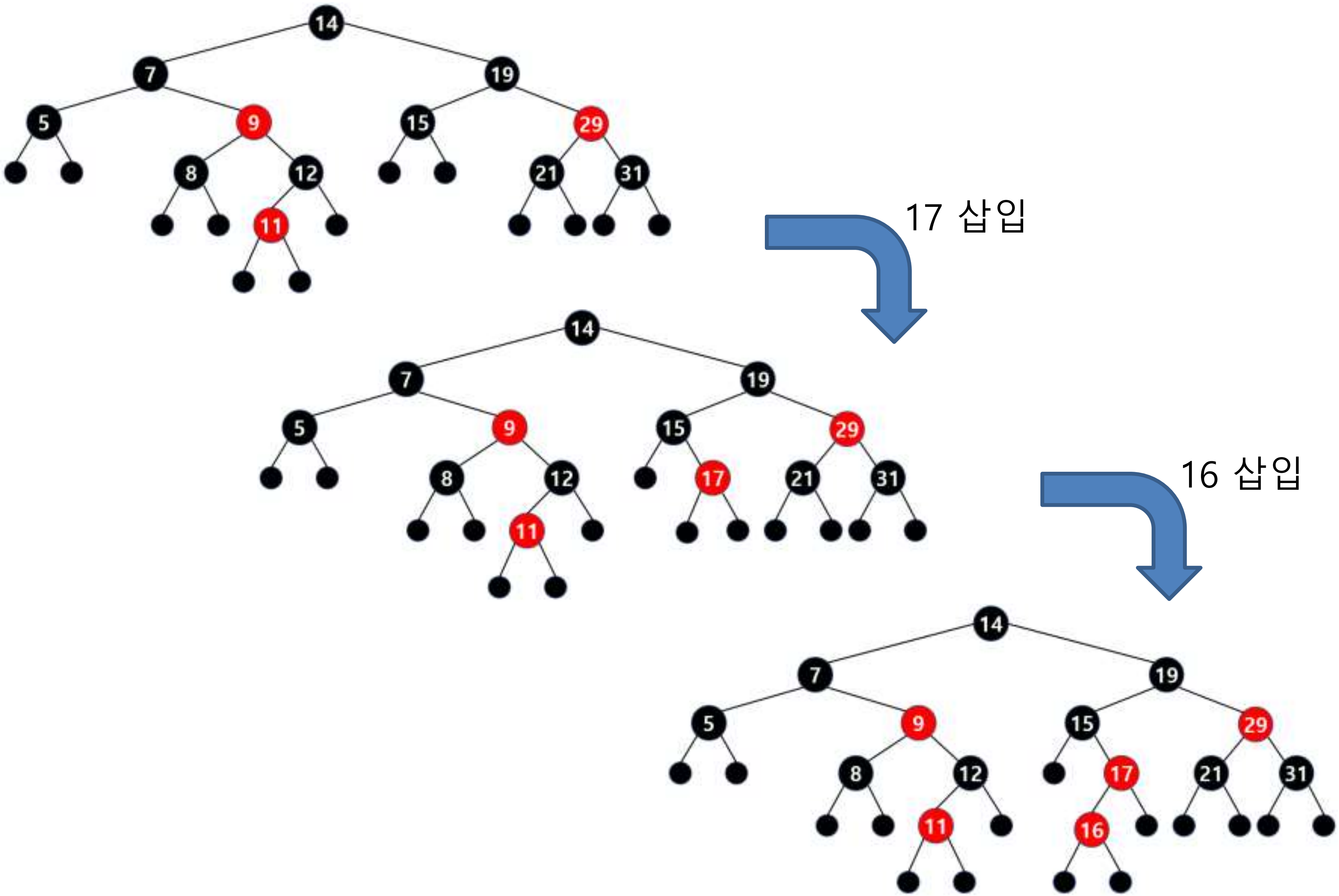
  (Step 1) x를 RBT에 삽입하고, x의 색을 red로 둠

  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



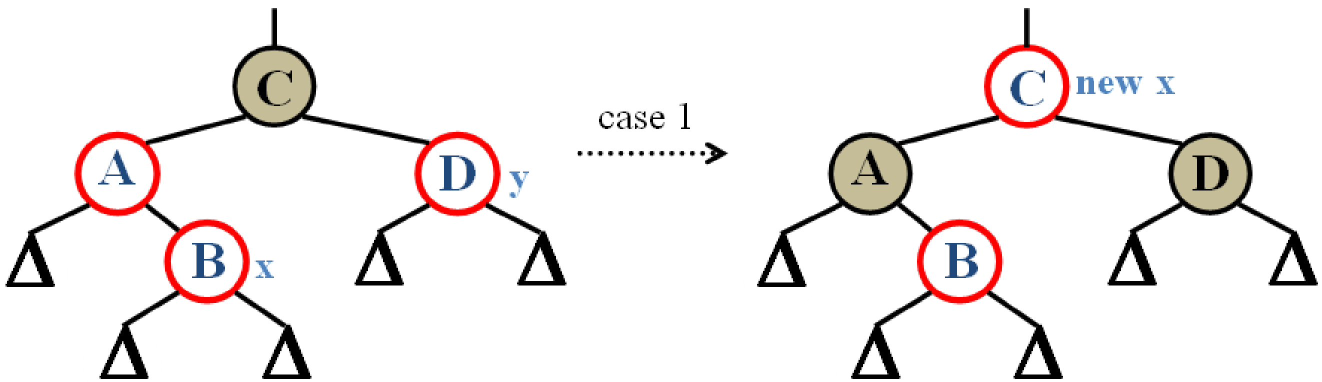
RBT - Insertion

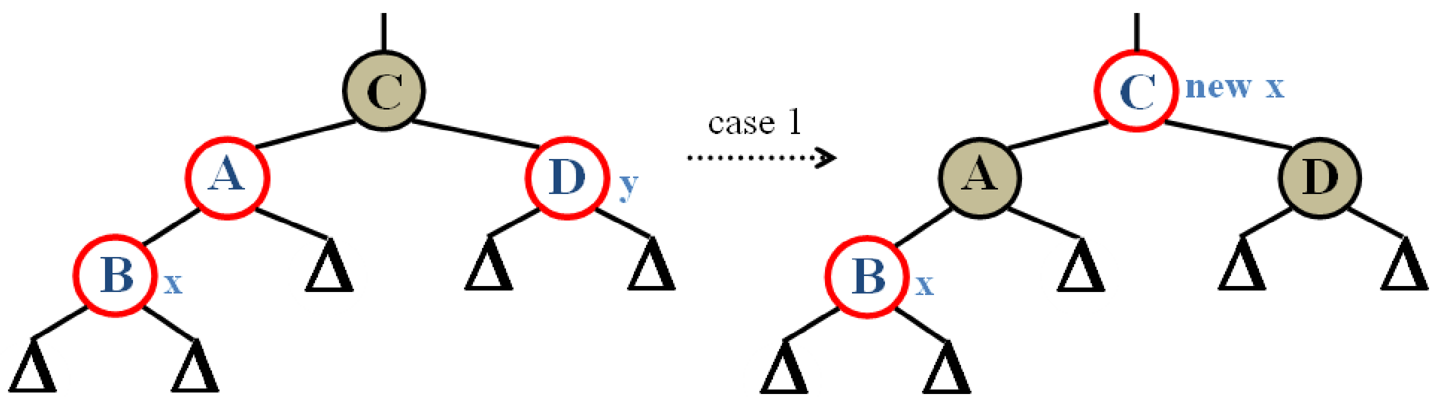
```
rbInsert(T, x)
  TreeInsert(T, x);
  x->color = RED;

  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

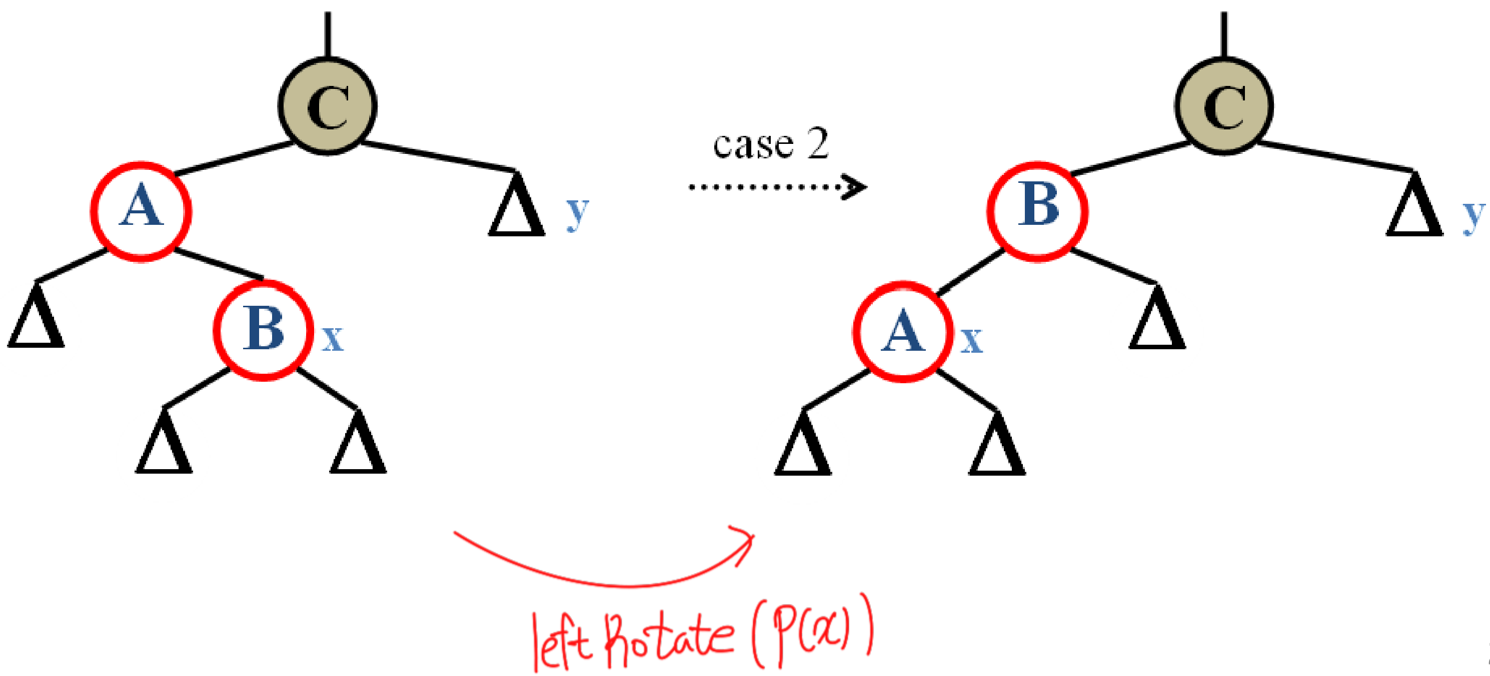




RBT - Insertion

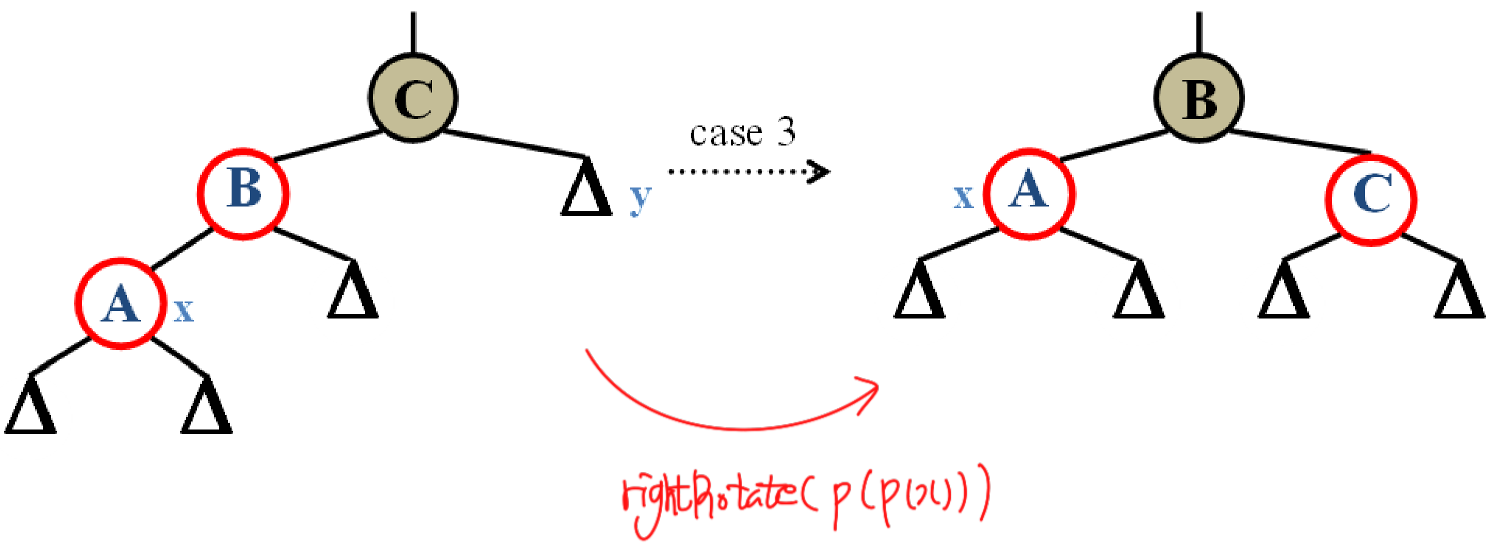
- (Case 2) x의 uncle (parent의 형제노드)의 색이 black이면서, x가 parent의 오른쪽 child인 경우

  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 이후에도 BST의 모든 성질을 만족한다.



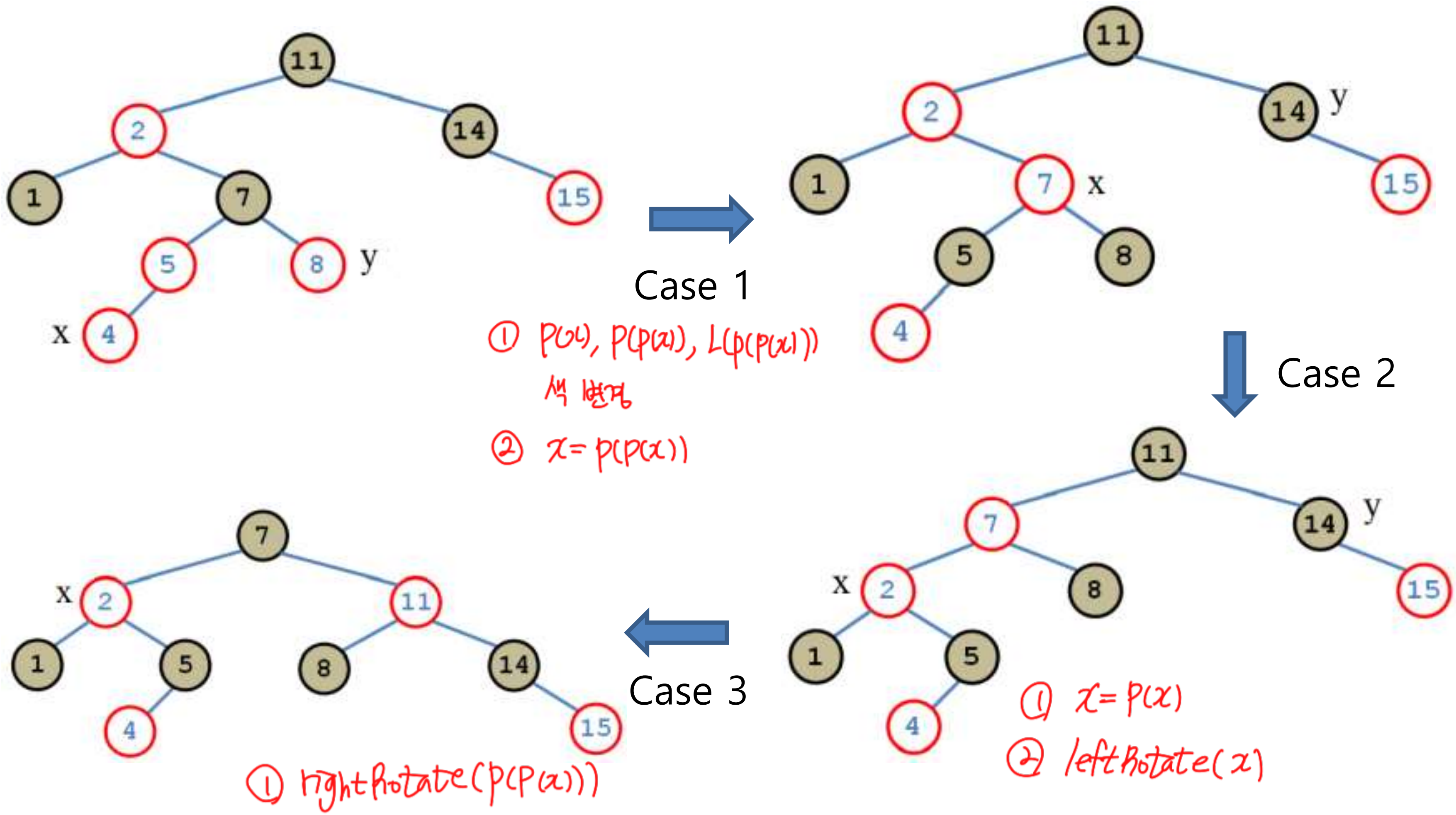
RBT - Insertion

- 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))
}
```

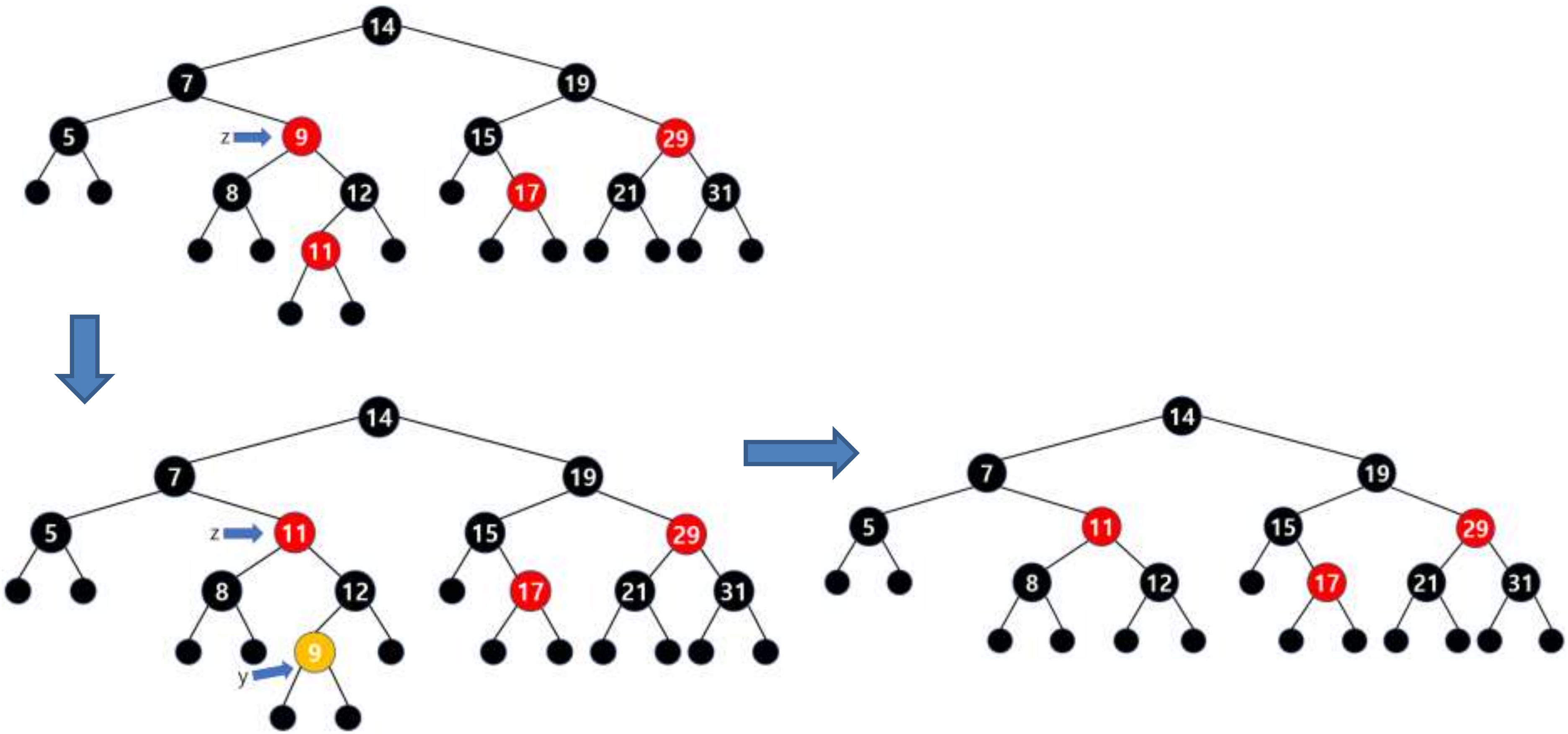
RBT - Insertion

- Key가 4인 노드를 BST에 삽입하는 예



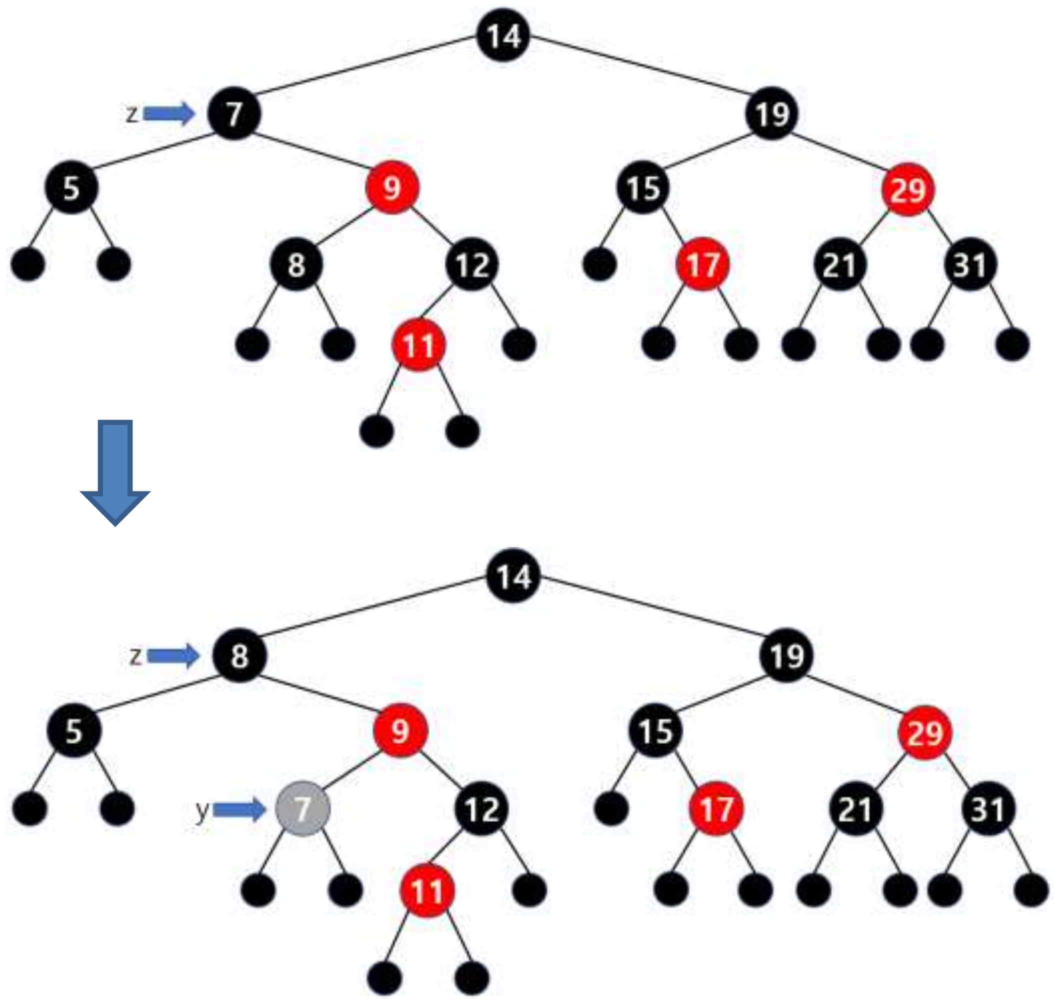
RBT - Deletion

- 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;
```

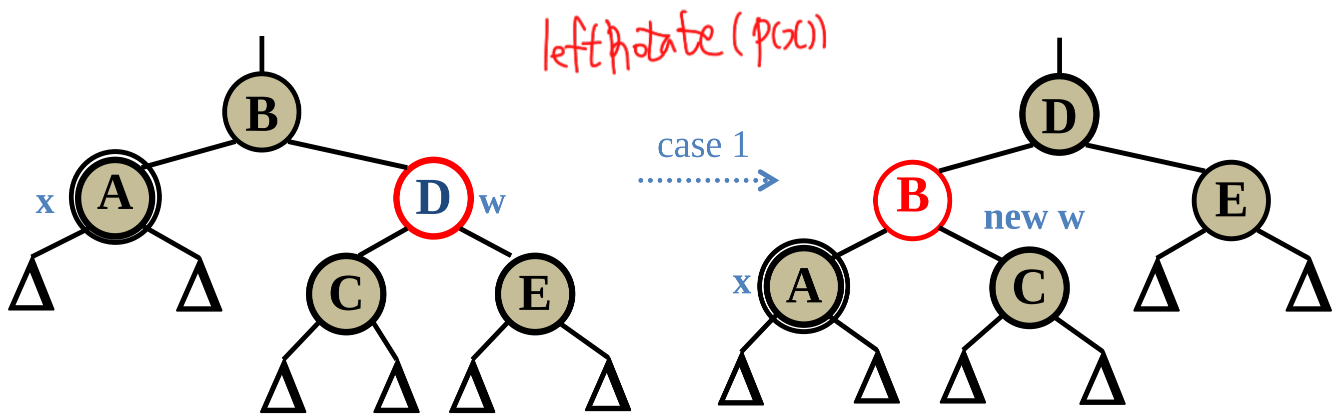
```
rbDeleteFixup(T, x)
  while (x != T.root && x->color == BLACK)
      if (x == x->p->left)
          w = x->p->right
          if (w->color == RED)
                w->color = BLACK;          // case 1
              x->p->color = RED;           // case 1
              leftRotate(T,x->p);          // 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

- 함수에서 x는 실제로 삭제된 노드y의 유일한 child.
- 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

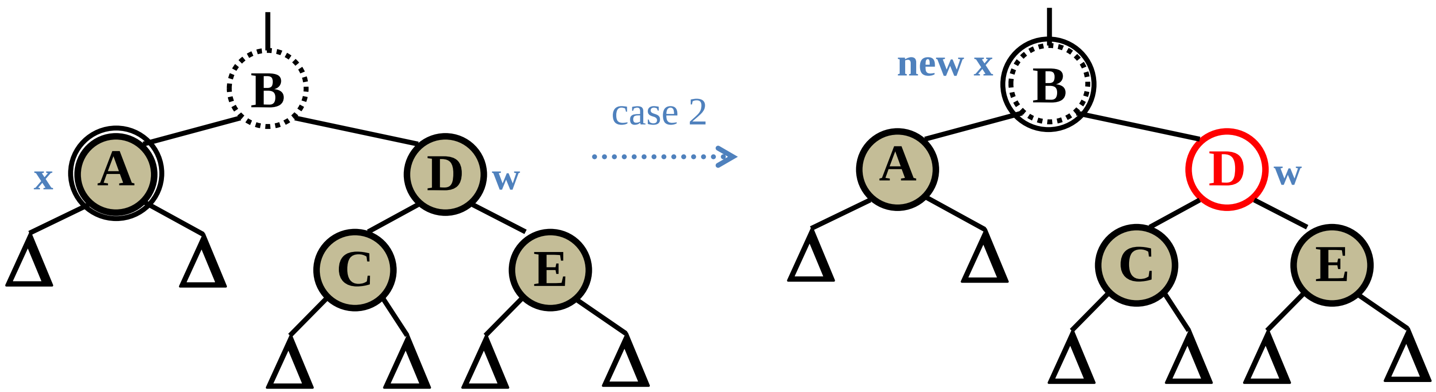
- (Case 1) x 의 형제노드 w의 색이 red인 경우 (x의 부모는 반드시 black 이다)
  - w를 black으로, x의 부모를 red로 바꾼 다음, leftRotate() 수행한 후 w를 다시 x의 형제 노드가 되도록 한다.
  - 이렇게 RBT의 구조를 바꾼 다음에, Case 1을 Case 2, 3, 4에 적용시킨다. 아래 그림에서 노드 x를 이중 원으로 표시한 것은 x가 여분의 black 을 가지고 있음을 나타낸다.



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

RBT - Deletion

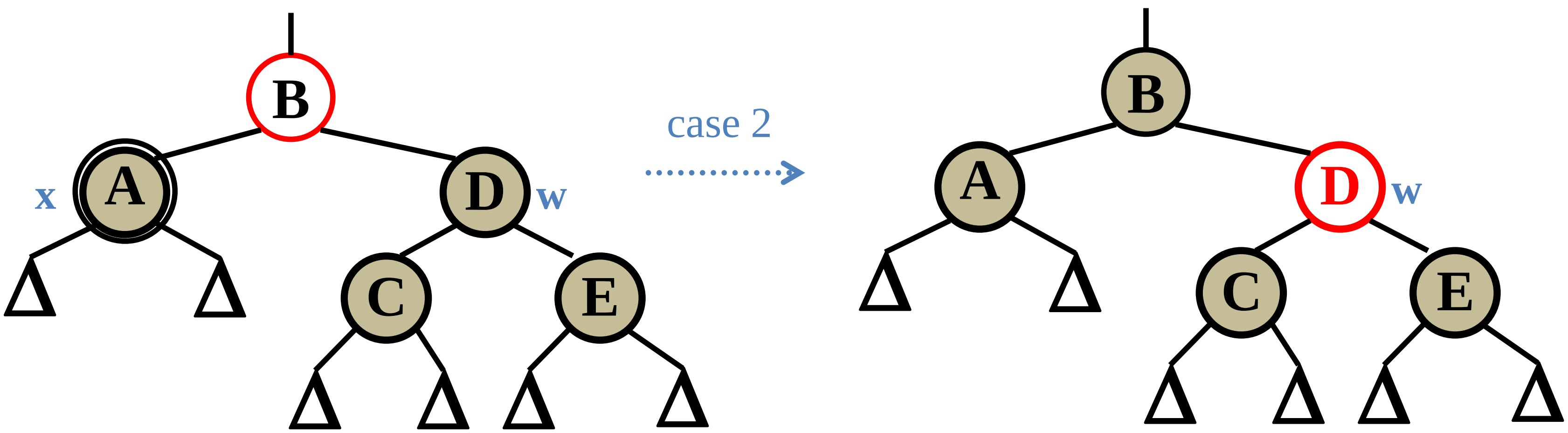
- (Case 2) w의 left, right child 모두 black인 경우
  - w의 색을 red로 바꾸고, x가 가지고 있던 여분의 black 을 x의 부모노드로 옮긴다.



  - 이 경우에는 x의 부모노드의 색에 따라 두 가지 경우가 발생한다

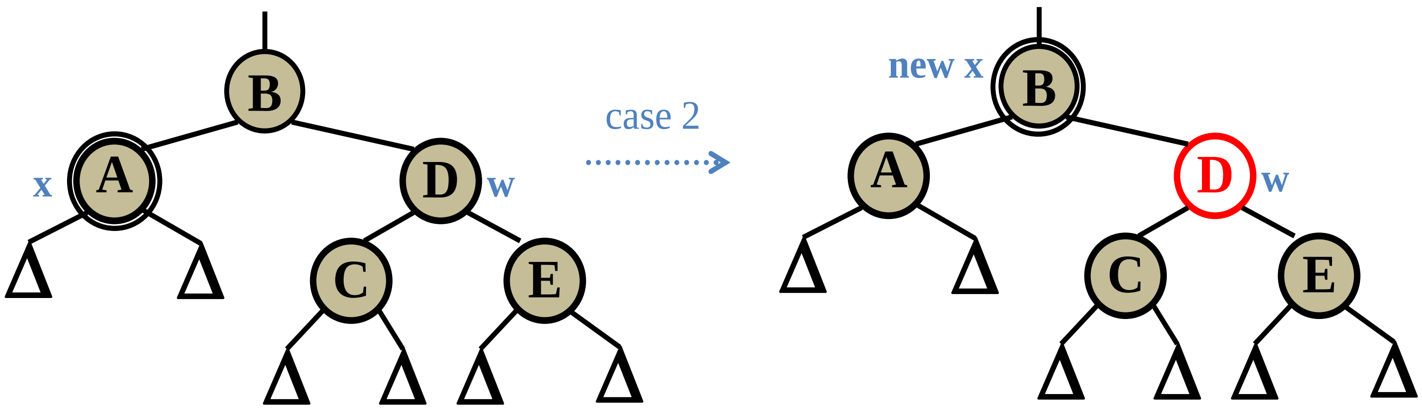
RBT - Deletion

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



RBT - Deletion

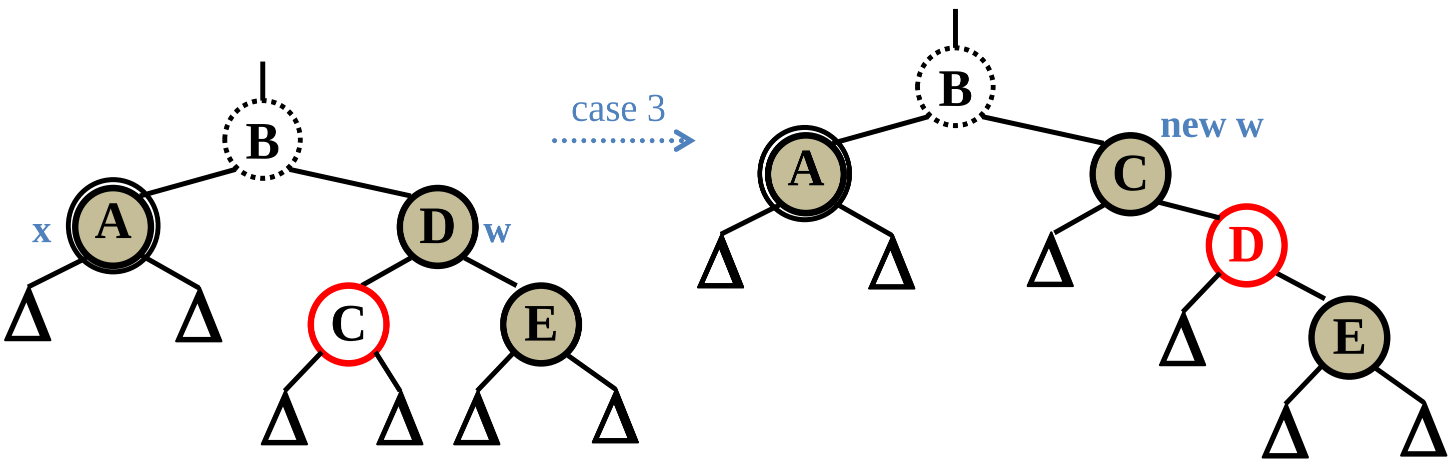
- (Case 2-2) x의 부모노드의 색이 black 인 경우
  - x의 부모노드가 x로부터 black으로 전달받아 여분의 black을 가지게 된다.
  - 이제 이 부모노드를 x로 정하고, 계속 loop를 수행.



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

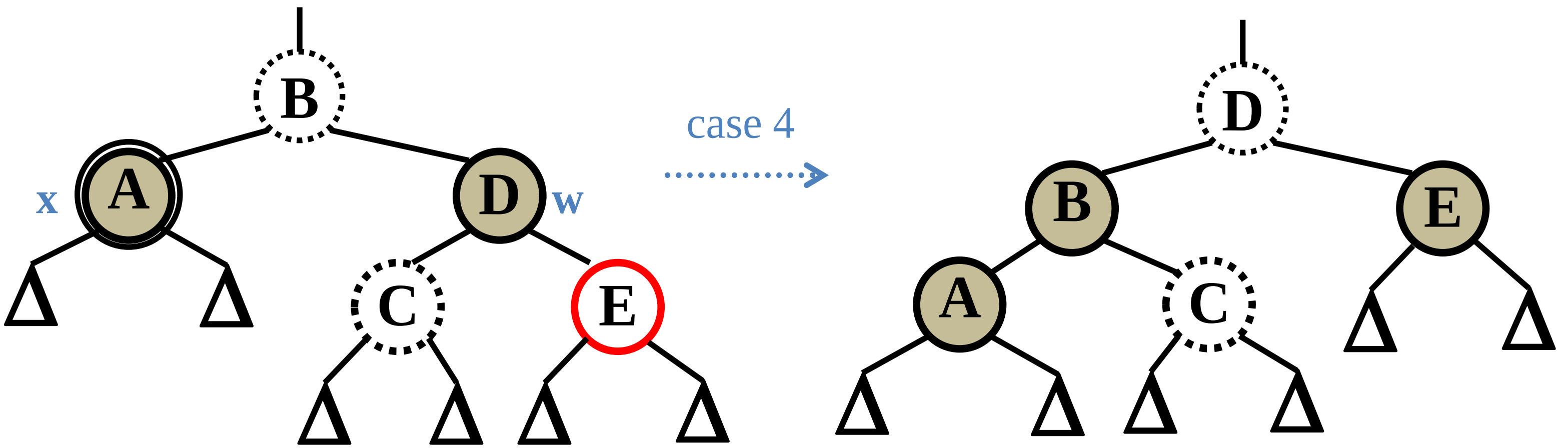
RBT - Deletion

- (Case 3) w의 오른쪽 child가 black인 경우. (따라서, 자동적으로 왼쪽 child는 red이다)
  - w의 right child가 red가 되게 하여 Case 4에서 처리하도록 한다.



RBT - Deletion

- (Case 4) w의 오른쪽 child가 red인 경우. (따라서, 왼쪽 child 는 red 또는 black 일 수 있다.)
  - x의 부모노드를 중심으로 왼쪽회전한 후, x가 가지고 있던 여분의 black을 위로 전달한다.



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

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.

Splay Tree

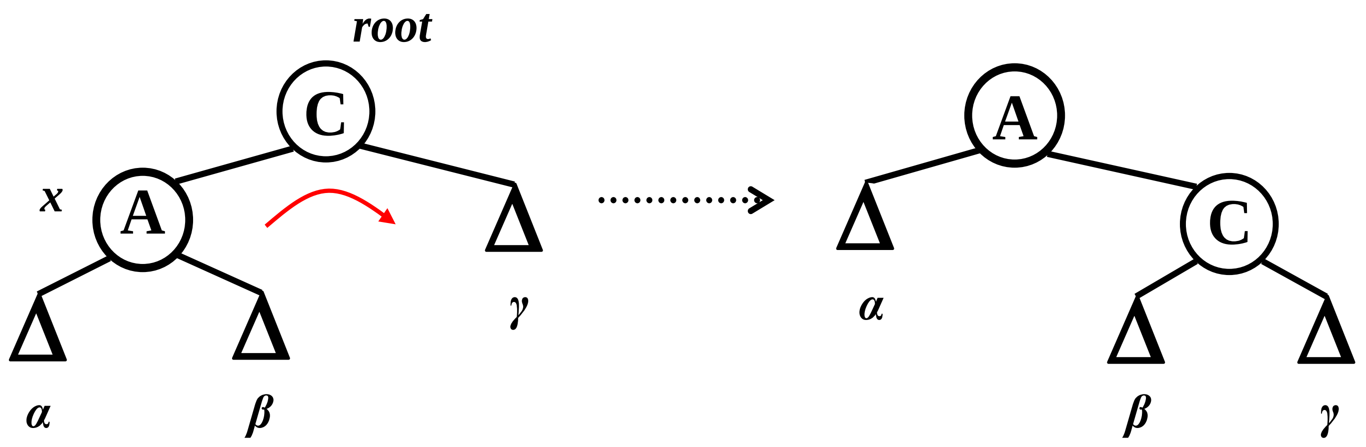
- 기본적으로 BST (Binary Search Tree)이면서, AVL tree, Red-Black Tree와 같은 균형트리(Balanced Tree)처럼 Search, Insert, Delete 연산을 O(log n) 시간에 처리
- Splay Tree가 기존의 균형트리와 다른 점은, 먼저 트리의 균형을 이루기 위한 추가적인 정보 (Red-Black Tree에서는 노드의 색)를 필요로 하지 않는다는 점
- 이러한 추가적인 정보 없이 Search, Insert, Delete 연산을 효율적으로 수행하기 위해 매 연산마다 트리의 구조를 재구성
- 이런 특징 때문에 Splay Tree를 Self-Reconstructing, Self-Adjusting, 혹은 Self-Organizing (자체 조정, 자체 재구성) Tree라 부른다.

Splay Tree

- 트리구조를 재구성하는 원칙:
  - 검색 혹은 삽입되는 노드가 루트가 되도록 재구성
  - 삭제되는 노드에 대해서는 이 노드에 인접한 노드(실제로 삭제되는 노드의 부모노드)를 루트가 되도록 재구성
- 이런 재구성 작업을 통하여 전체 트리가 균형 있게 만들어져 감
- 이와 같이 특정 노드를 회전시켜 루트까지 올리는 작업을 “splaying”이라고 함

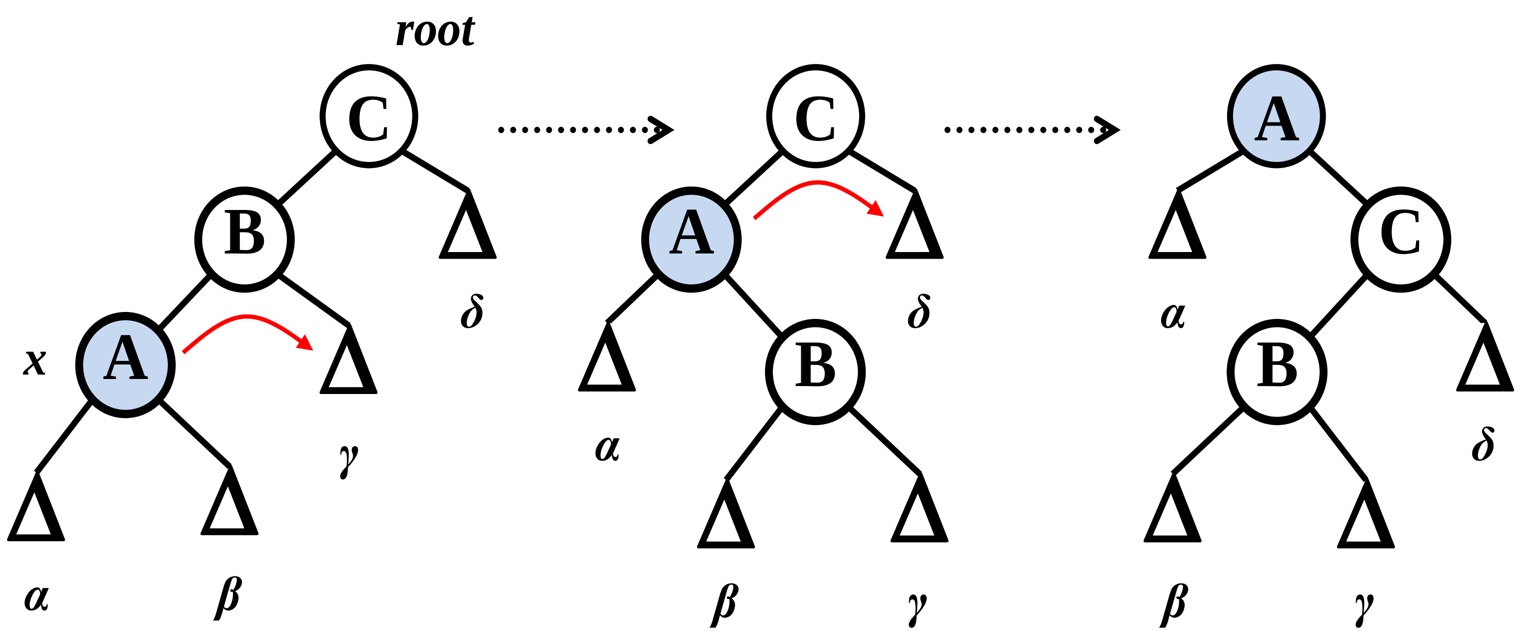
Splay Tree – 자체 조정방법

- 검색하는 노드를 트리의 루트노드로 올리는 방법은 연속적인 회전 연산을 적용시켜, 그 노드가 루트가 될 때까지 수행하면 된다.
- 예를 들어, 노드 x의 부모노드가 루트인 경우에는 아래와 같이 한 번의 회전을 통하여 x를 루트 노드로 만들 수 있다.



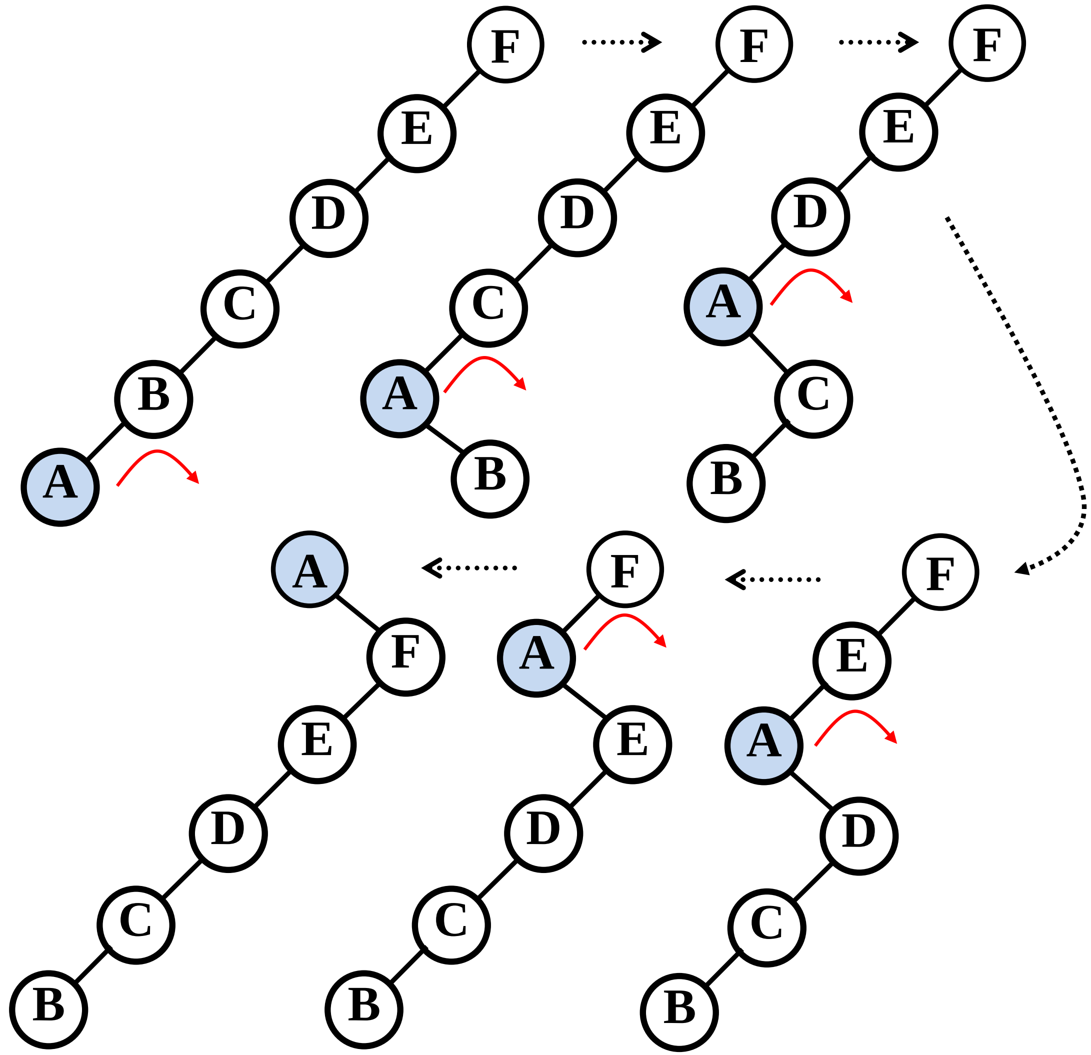
Splay Tree – 자체 조정방법

- 노드 x의 부모노드가 루트가 아닌 경우 한 번 이상의 연속적인 회전이 필요



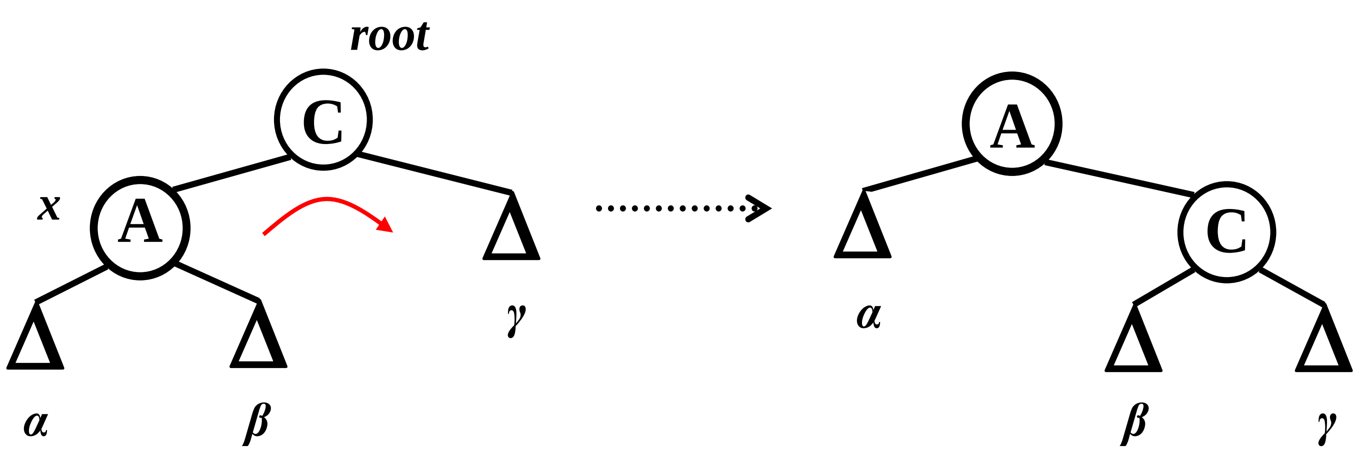
Splay Tree – 자체 조정방법

- 특정 노드 x를 연속적으로 회전시키면서 루트로 올리는 작업의 단점
  - 자체조정된 트리가 균형잡힌 모양이 되지 않을 수 있다



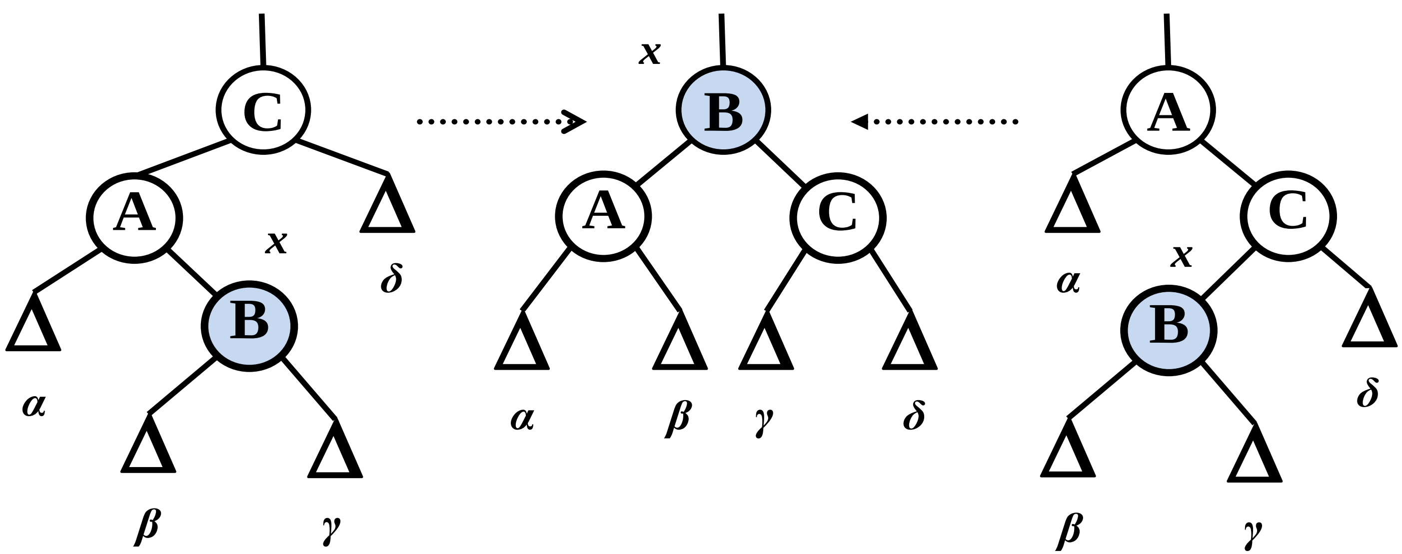
Splay Tree – 자체 조정방법

- (Case 1) Zig step
  - 노드 x의 부모노드가 루트인 경우에는 한 번의 회전을 통하여 x를 루트노드로 만들 수 있다.
  - 이 경우를 Zig step 이라 한다.
  - 이 경우에는 왼쪽 또는 오른쪽 회전을 하게 된다..



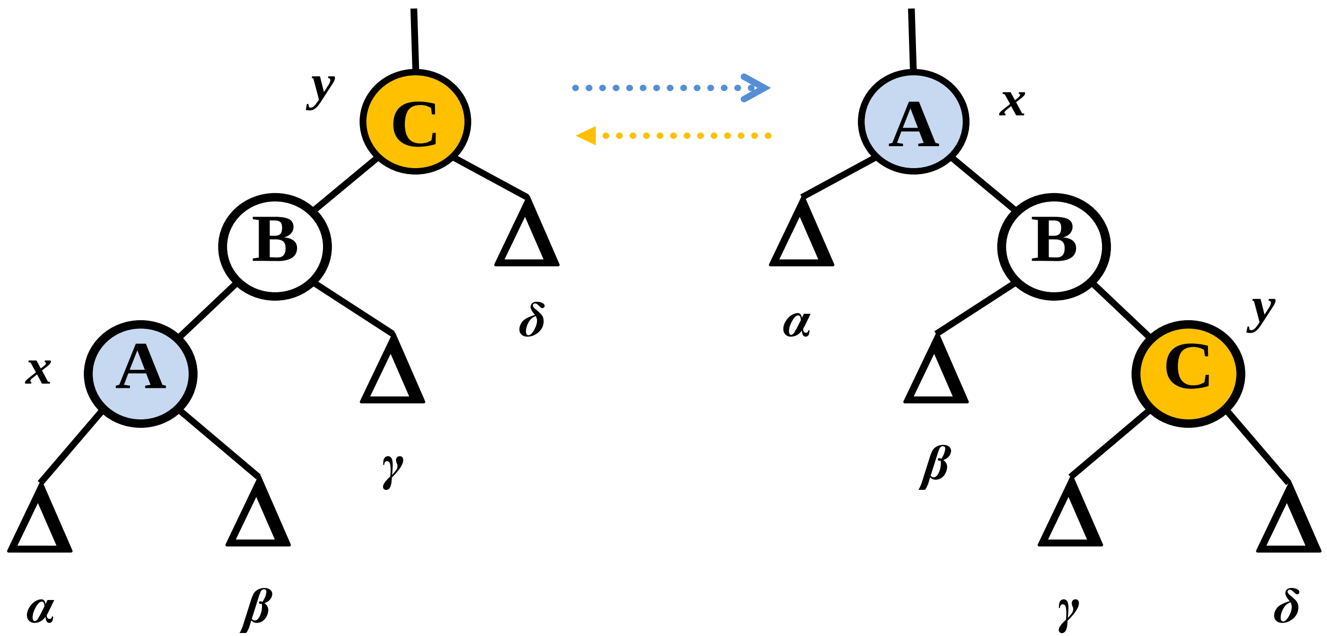
Splay Tree – 자체 조정방법

- (Case 2) Zig-Zag step
  - x의 부모노드가 루트가 아니고 x와 x의 부모노드가 서로 다른 방향의 자식인 경우
  - 두 번의 서로 다른 방향의 회전을 통하여 x 를 트리의 위쪽으로 옮긴다.



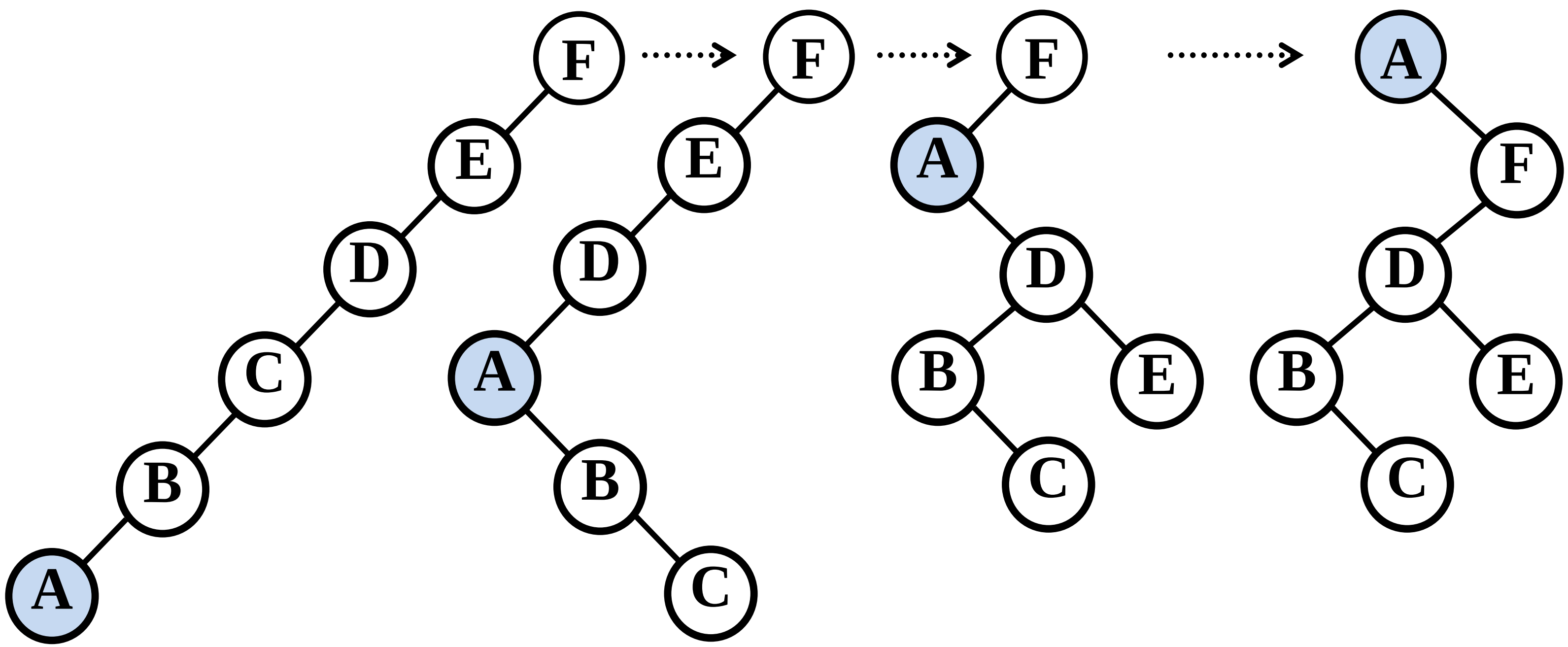
Splay Tree – 자체 조정방법

- (Case 3) Zig-Zig step
  - x와 x의 부모노드 모두가 같은 방향의 자식인 경우
  - 세 노드를 같은 방향으로 동시에 회전



Splay Tree – 자체 조정방법

- 위의 세 단계를 한쪽으로 치우친 tree에 적용하면 다음과 같은 작업이 이루어짐
- 트리가 원래의 트리보다 조금 균형잡힌 모양으로 변함을 알 수 있다.



Splay Tree – Search, Insert, Delete

- Splay 작업은 BST에 적용되는 검색, 삽입, 삭제 연산을 수행한 이후에 매번 노드 적용
  - Search(T, x) : 트리 T에 x가 존재하는 경우에는 key x를 가지는 노드에 대하여 splay 작업이 이루어지며, x가 트리에 존재하지 않는 경우에는 x를 찾기 위하여 가장 마지막에 접근한 노드에 대하여 splay 작업을 적용
  - Insert(T, x) : 트리 T에 key x를 insert 한 이후에, insert된 노드에 splay 작업을 적용
  - Delete(T, x): 트리 T에서 key x를 가진 노드를 delete 한 후, 실제로 delete 된 노드의 부모 노드에 splay 작업을 적용
- worst time complexity: O(n)
- 전체 연산을 수행하는 시간을 고려하면 (amortized time complexity): O(m log n)

Dynamic Order Statistic (동적 순서 통계데이터)

- 통계학에서 k-th order statistic (k-번째 순서 통계데이터)란 k-번째로 작은 데이터를 말함 → O(n) 시간에 가능
- RBT와 같은 균형트리에 저장하게 되면, k-번째로 작은 데이터는 O(lg n) 시간에 가능
- 또한, RBT를 이용하면, 데이터의 집합에 데이터를 추가/제거하는 동적인 작업을 통해서도, k-번째 작은 데이터를 O(lg n) 시간에 가능
- 또한, 주어진 데이터가 전체 데이터에서 크기가 몇 번째, 즉, 순위(rank)를 O(lg n) 시간에 찾을 수 있음

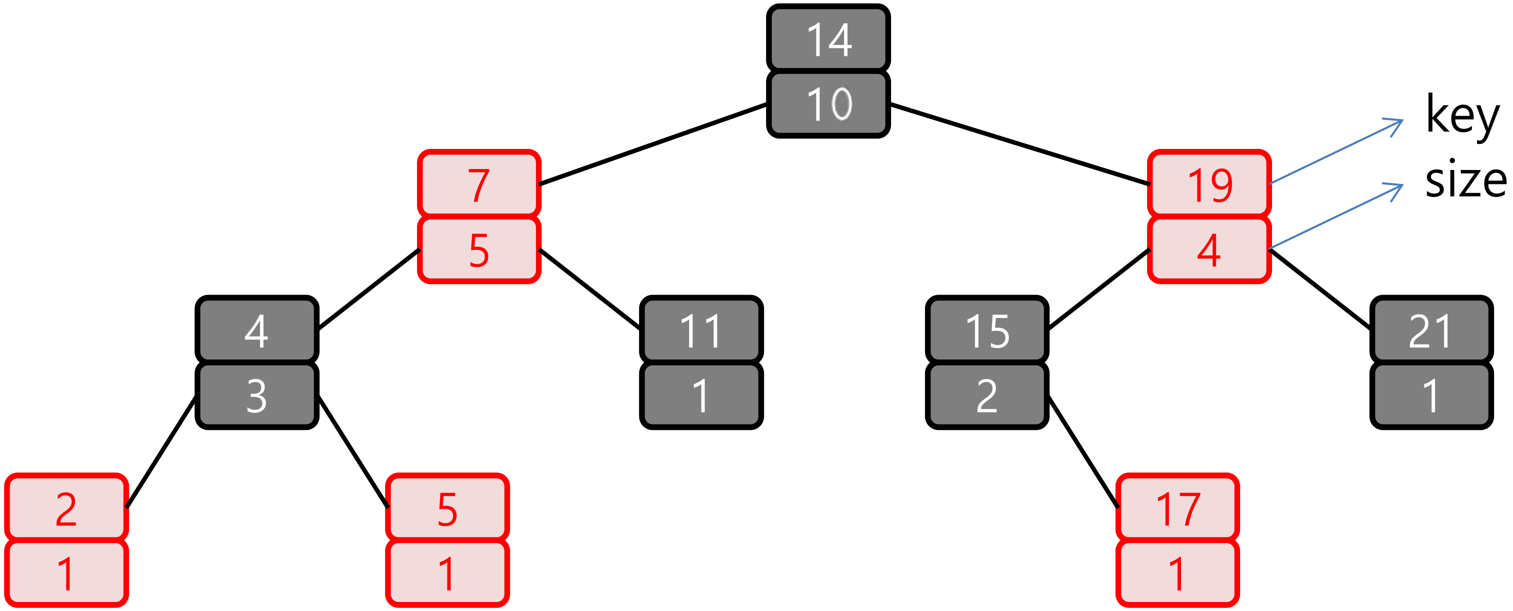
Order-Statistic Tree(OST)

- order-statistic tree(OST) : 동적 순서 통계데이터(dynamic order statistic)를 처리하기 하기 위한 자료구조
- OST는 기본적으로 RBT로 만듬
- RBT의 각 노드가 가지는 필드에는 기본적으로 key, color, p(부모노드로의 포인터), left(left child로의 포인터), right 이외에도 size를 가진다.
- 필드 size는 이 노드를 루트노드로 하는 subtree의 모든 노드의 개수를 나타낸다.
- 노드 x의 size는 다음과 같이 정의된다

  x->size = x->left->size + x->right->size + 1

Order-Statistic Tree(OST)

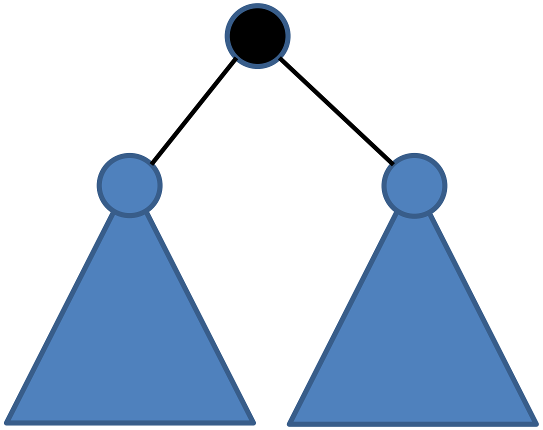
- OST의 예



Order-Statistic Tree(OST)

- OST T에서 순위 k의 데이터(즉 k-번째 작은 데이터)를 검색하는 OSselect() 함수
- 이 함수는 OSselect(T.root, k) 로 처음 호출됨

```
OSselect(x, k)
    r = x->left->size + 1;
    if (r == k)
        return x;
    else if (r > k)
        return OSselect(x->left, k);
    else
        return OSselect(x->right, k-r);
```



Order-Statistic Tree(OST)

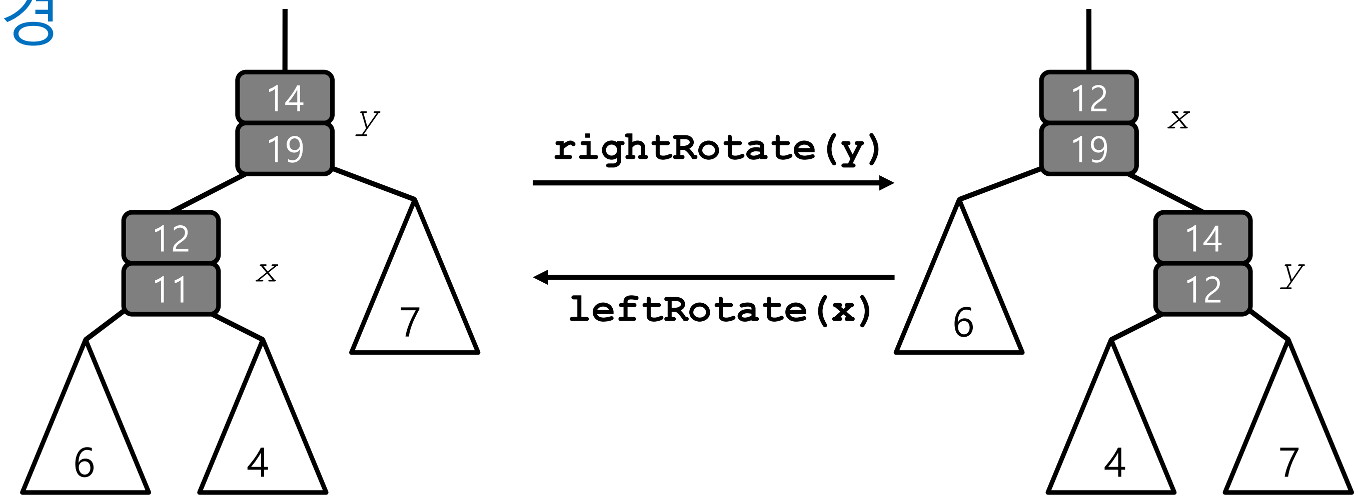
- OST T 에서 주어진 데이터의 순위를 계산하는 OSrank() 함수

```
OSrank(T, x)
    r = x->left->size + 1;
    y = x;
   while (y != T.root)
        if (y == y->p->right)
            r += y->p->left ->size + 1;
        y = y->p;
    return r;
```

- OSselect(), OSrank() : O(lg n) 시간

Order-Statistic Tree(OST)

- T에 insert하는 경우 size 계산
  - 입력된 데이터는 T의 루트 노드에서 시작하여 단말노드로 내려가게 되는데, 이 때 지나가는 모든 노드의 size 값을 1 증가시키고, 입력된 데이터를 저장하는 단말노드의 size 값을 1로 만든다.
  - 그 다음으로는 RBT를 회전을 통하여 트리의 구조를 바꾸게 되는데, 각 회전 연산에 따라 각 노드의 size값 변경



Order-Statistic Tree(OST)

- T에서 delete하는 경우 size 계산도 유사
  - RBT에서와 같이 노드를 삭제한다. 실제로 delete 되는 노드의 부모노드를 y라 하면, 노드 y부터 시작하여 T의 루트노트까지 올라가면서 모든 노드의 size 값을 1 감소시킨다.
  - 그 다음으로는 RBT를 회전을 통하여 트리의 구조를 바꾸게 되는데, 각 회전 연산에 따라 각 노드의 size값 변경

Range Tree

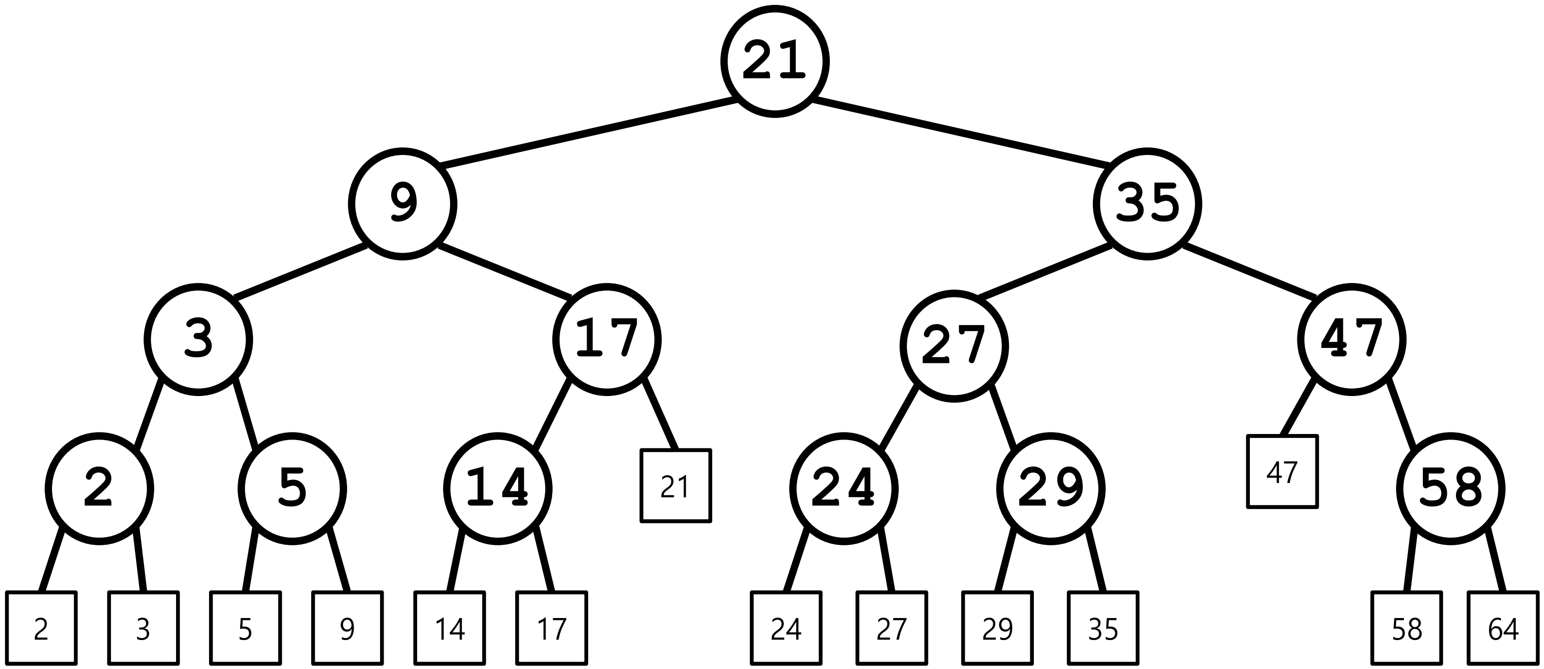
- 1차원 직선상에 n개의 실수 $S = \{a_1, a_2, \ldots, a_n\}$이 주어졌을 때, 주어진 폐구간(query interval) $I=[b_1, b_2]$에 포함되는 S의 모든 원소에 관해 계산하는 문제를 Range query라고 부름
- Range query에는 두 종류의 문제
  - 모든 점을 계산(또는 reporting)하는 range reporting 문제
  - 점의 개수 만을 계산하는 range counting 문제
- 이러한 range query 문제에서는 집합 S 의 모든 원소가 동적으로 insert, delete 될 수 있으며, 주어진 많은 수의 range query를 효율적으로 처리할 수 있는 자료구조가 필요
- 이러한 자료구조를 range tree라고 함

1차원 Range Tree

- 1차원 Range Tree는 RBT와 같은 균형트리를 근간으로 다음과 같이 만듬
  - Range Tree의 단말노드는 n개의 실수 $\{a_1, a_2, \ldots, a_n\}$를 원소로 한다.
  - Range Tree는 내부노드는 (n-1)개의 실수 $\{a_1, a_2, \ldots, a_{n-1}\}$ 를 원소로 한다.

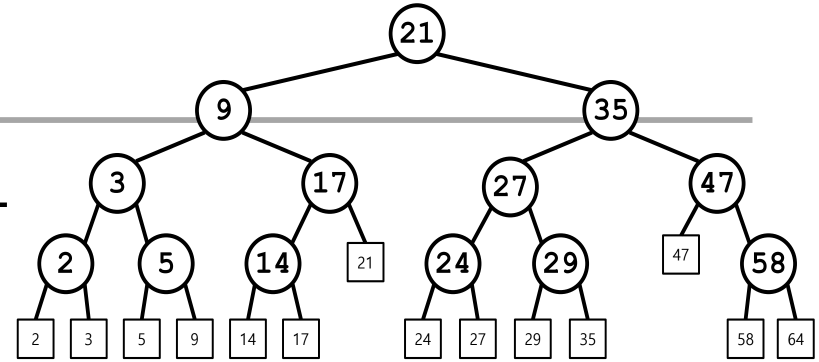
1차원 Range Tree

- 예: S = {2, 3, 5, 9, 14, 17, 21, 24, 27, 29, 35, 47, 58, 64} 를 원소로 하는 range tree



각 내부 노드는 왼쪽 서브트리에 있는 단말노드 중 가장 값이 큰 것을 저장함

1차원 Range Tree

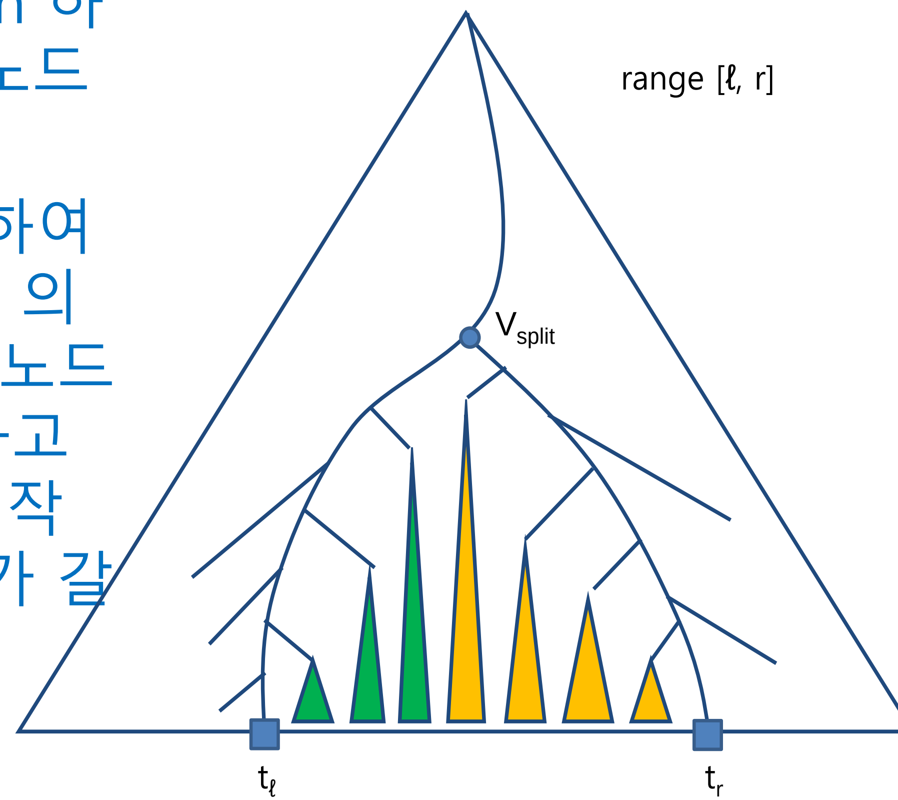


[28,60]

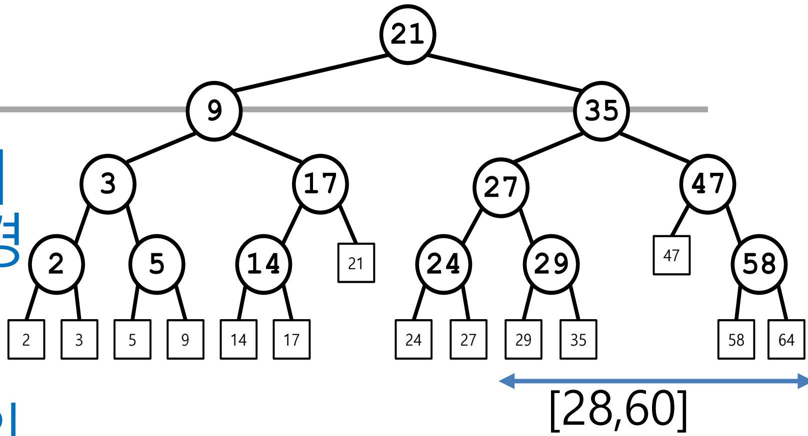
- range [ℓ, r] 에 포함되는 모든 점들을 구하기 위한 range reporting query의 실행

  (step 1) ℓ, r에 대해 각각 search 하면서 최종적으로 도달한 단말노드를 각각 t_ℓ, t_r 로 둠

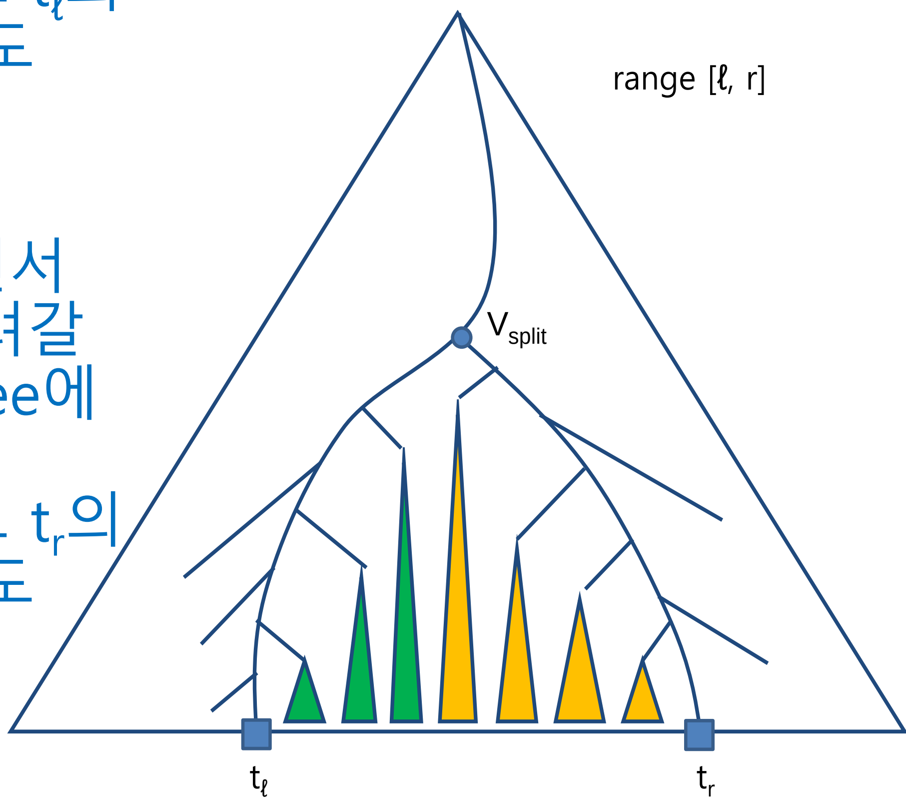
  (step 2) 루트노드로부터 시작하여 단말노드로 내려가면서 노드 x 의 key 값이 [ℓ, r] 에 포함되는 첫 노드를 구한다. 이 노드를 v_{split} 이라고 부르자. 이 노드는 루트에서 시작하여 단말노드 t_ℓ, t_r 로의 경로가 갈라지는 노드 임



1차원 Range Tree



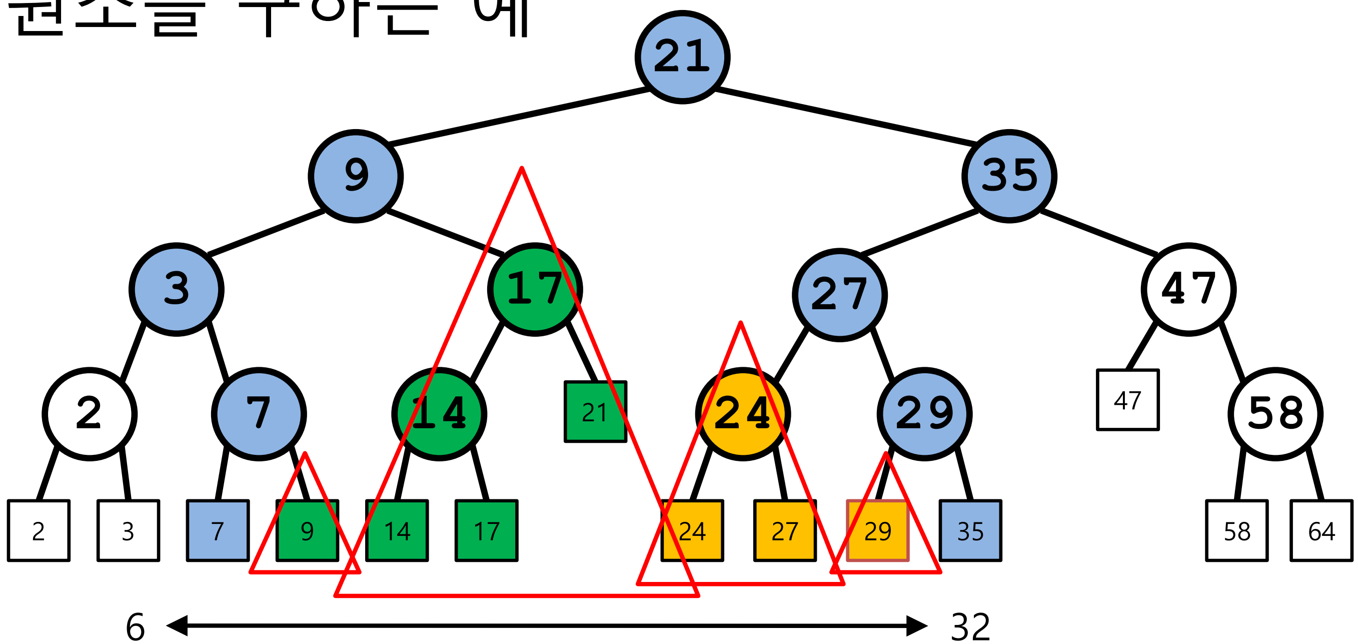
(step 3) v_{split}으로부터 t_ℓ로 내려가면서 어떤 노드 x의 왼쪽 child 로 내려갈 경우에는 이 노드 x의 오른쪽 subtree에 속하는 모든 단말노드의 점들을 report 한다. 최종적으로 단말노드 t_ℓ의 key가 range에 포함되면 이 노드도 report 한다.



(step 4) 마찬가지로 t_r로 내려가면서 어떤 노드 x의 오른쪽 child로 내려갈 경우에는 이 노드 x의 왼쪽 subtree에 속하는 모든 단말노드의 점들을 report 한다. 최종적으로 단말노드 t_r의 key가 range에 포함되면 이 노드도 report 한다.

1차원 Range Tree

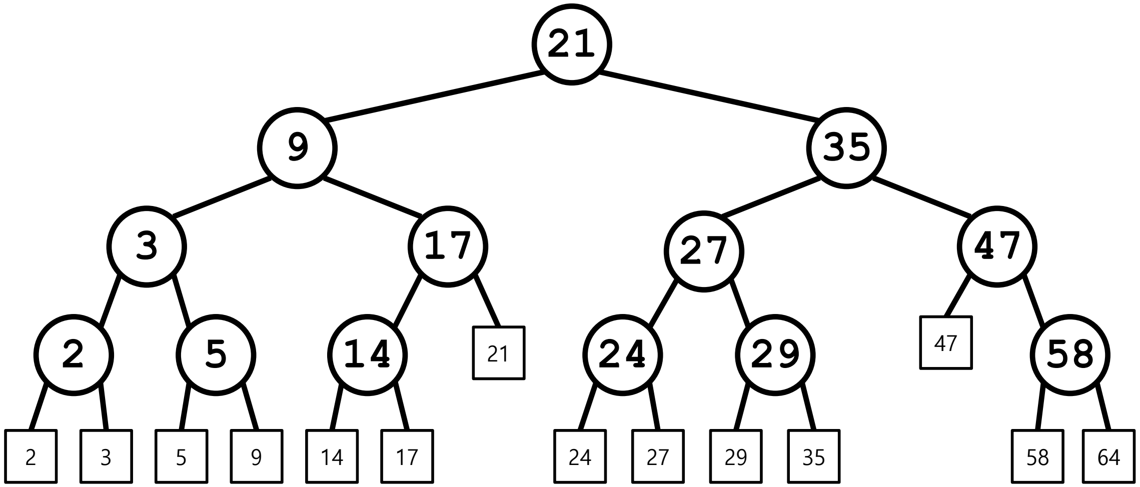
- S = {2, 3, 5, 9, 14, 17, 21, 24, 27, 29, 35, 47, 58, 64} 를 원소로 하는 range tree에서 구간 [6, 32] 에 속하는 원소를 구하는 예



- Range Query Report: O(log n + k) 여기서, k는 report 되는 원소의 개수

1차원 Range Tree

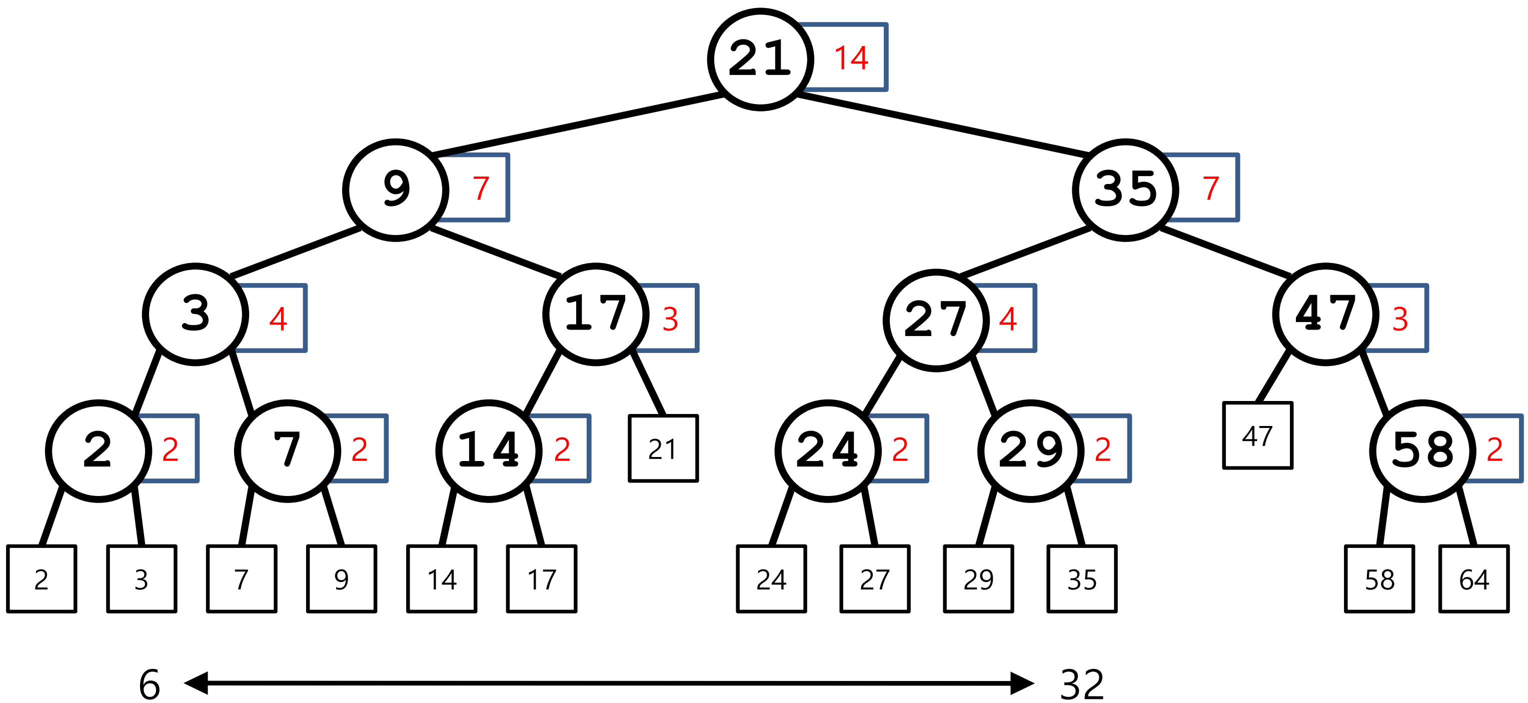
- 주어진 range tree



- 새로운 원소 23, 80, 19, 40이 차례로 추가되면?
- 기존의 원소 9, 27, … 삭제되면?

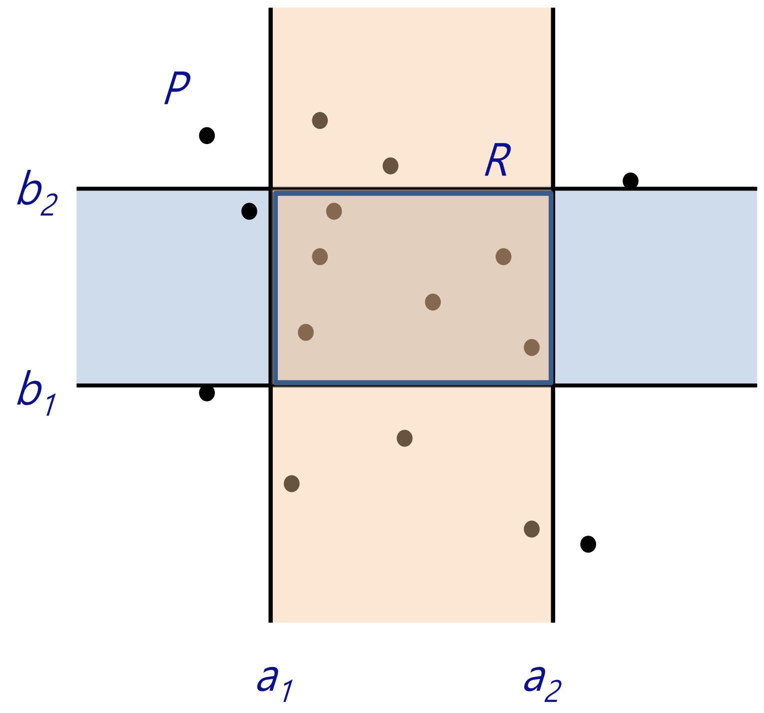
1차원 Range Tree

- Range Counting Query는 Range Tree의 각 내부 노드에 그 노드를 루트로 하는 subtree의 단말노드 개수를 저장하면 됨
- Range Counting Query : O(log n)



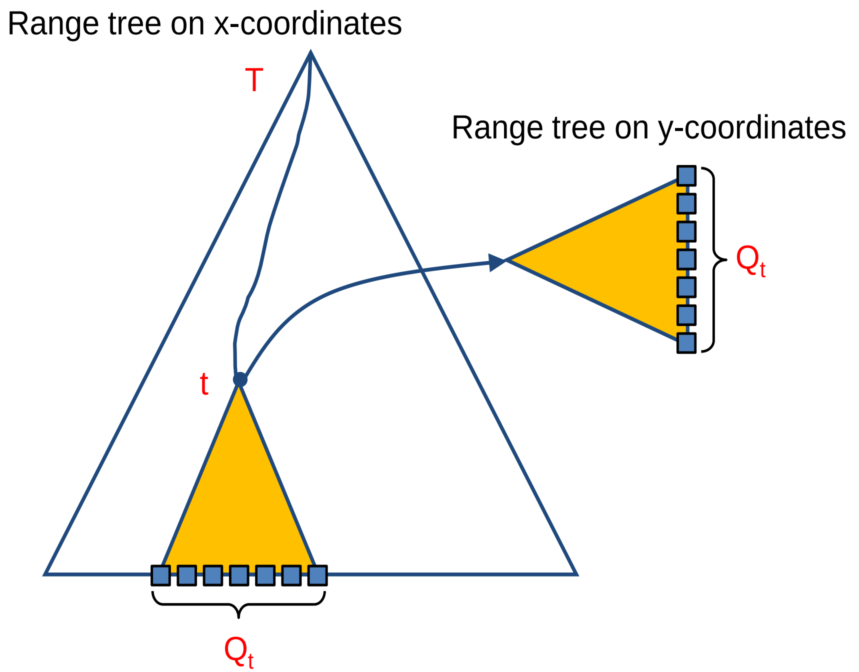
2차원 Range Tree

- 2차원 평면 상에 n 개의 점 $P = \{p_1, p_2, \ldots, p_n\}$ 이 주어졌을 때, 주어진 직사각형 $R=[a_1, a_2] \times [b_1, b_2]$ 에 포함되는 P 의 모든 점을 계산하는 문제
- 집합 P 의 모든 점이 동적으로 insert, delete 될 수 있음



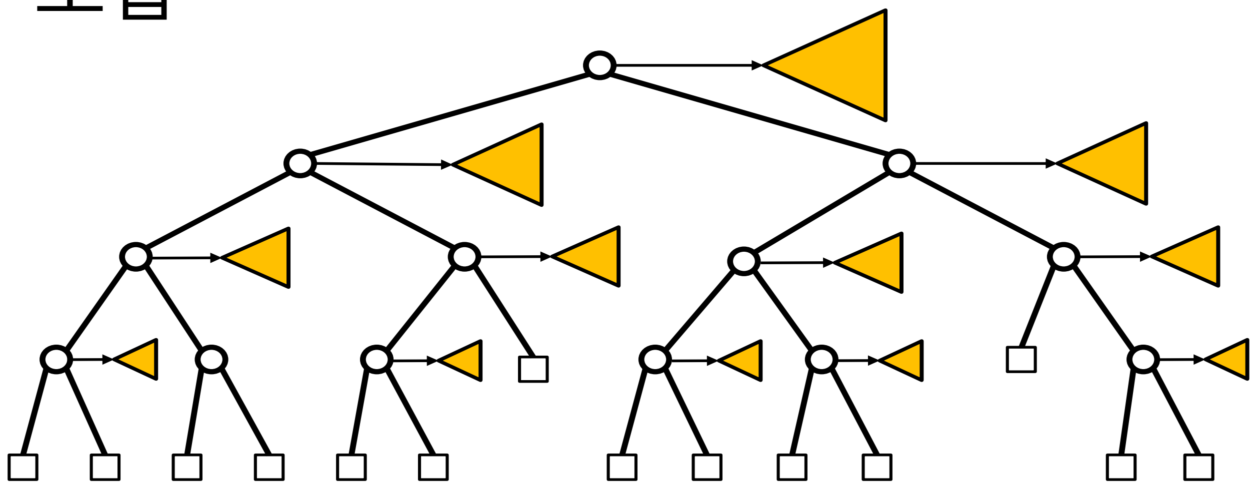
2차원 Range Tree

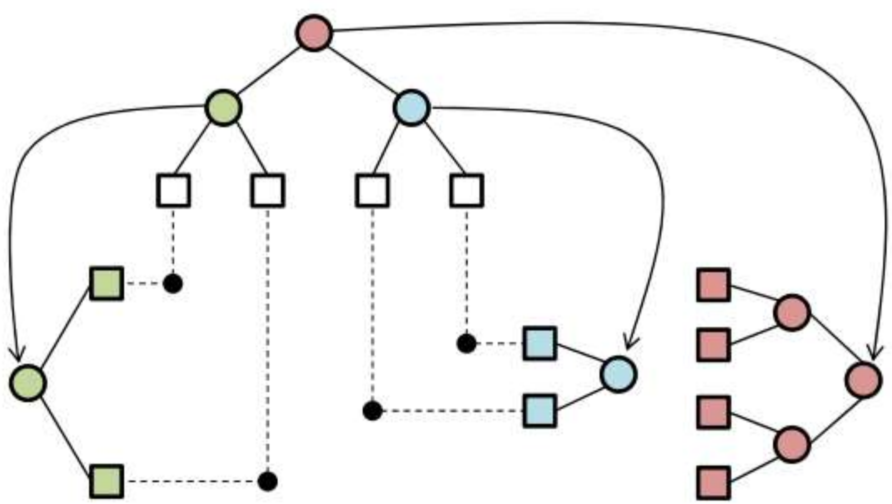
- 2차원 range tree 구성
  - 먼저 P 에 속하는 모든 점들의 x-좌표를 기준으로 1차원 range tree T를 만든다. 이 트리에서는 그림에서와 같이 x-좌표가 $[a_1, a_2]$ 인 수직구간에 포함되는 모든 점을 계산하는 query 를 처리할 수 있다.
  - 그 다음에 위에서 만든 트리 T 의 모든 내부 노드 t 마다, t 를 루트로 하는 T의 모든 단말노드에 속하는 점들의 집합 Q_t 를 정의할 수 있다. 이 때, Q_t 에 속하는 모든 점들에 대하여 y-좌표를 기준으로 1차원 range tree 를 만들어 이 트리를 노드 t에 연결한다.



2차원 Range Tree

- 위의 방법으로 구성된 2차원 range tree의 개략적인 모습





2차원 Range Tree

- 2차원 Range Tree 알고리즘

```
Build2dRangeTree(P)
    Ty = P에 속하는 점들의 y-좌표를 기준으로 만든 1차원 range tree
    if (P에 한 개의 점만 있는 경우)
        한 점으로 이루어진 leaf 노드를 만들고, Ty를 이 노드에 연결
    else
        P에 속하는 점들을 x좌표의 가장 중앙에 있는 점 pm을 기준으로
        그 왼쪽에 있는 점들의 집합 Pl 와 오른쪽에 있는 점들의 집합
        Pr로 나눈다.
        점 pm 을 원소로 하는 노드 v 를 만들고, Ty를 이 노드에 연결
        v->left = Build2dRangeTree(Pl);
        v->right = Build2dRangeTree(Pr);
    return v;
```

2차원 Range Tree

- 알고리즘 시간 복잡도 분석

  - If t is a node of x-tree:
    - t.val: cut value
    - t.left, t.right: child
    - t.ytree: y-tree
  - If t is a leaf of x-tree:
    - t.pt: point
    - t.ytree: a y-tree with a single point

BuildXTree (S) //S: point set

1. If |S|=1, return leaf t where
   1. t.pt and t.ytree are the point of S
2. x be median of X coordinates of all points in S
3. L (R) be subset of S whose X coordinates are no greater than (greater than) x
4. Return node t where
   1. t.val = x
   2. t.left = BuildXTree (L)
   3. t.right = BuildXTree (R)
   4. t.ytree = MergeYTree (t.left.ytree, t.right.ytree)

O(n)

2T(n/2)

O(n)

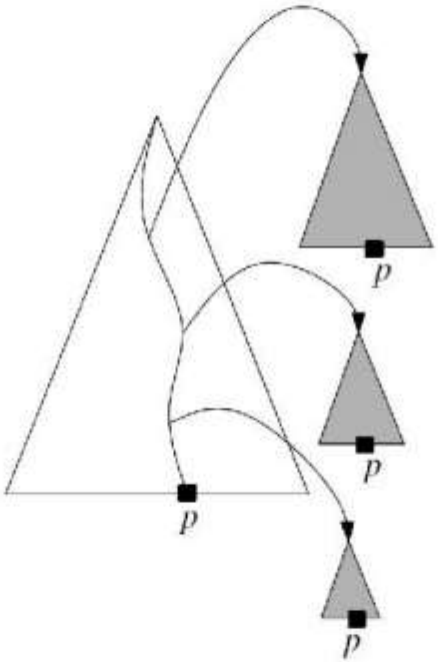
$T(n) = O(n) + 2T(n/2)$
$= O(n \log n)$

2차원 Range Tree

- Space complexity:
  - Size of each tree (x- or y-) is linear to # of leaves
  - Let T_i be # of trees of which p_i is a leaf, total space is

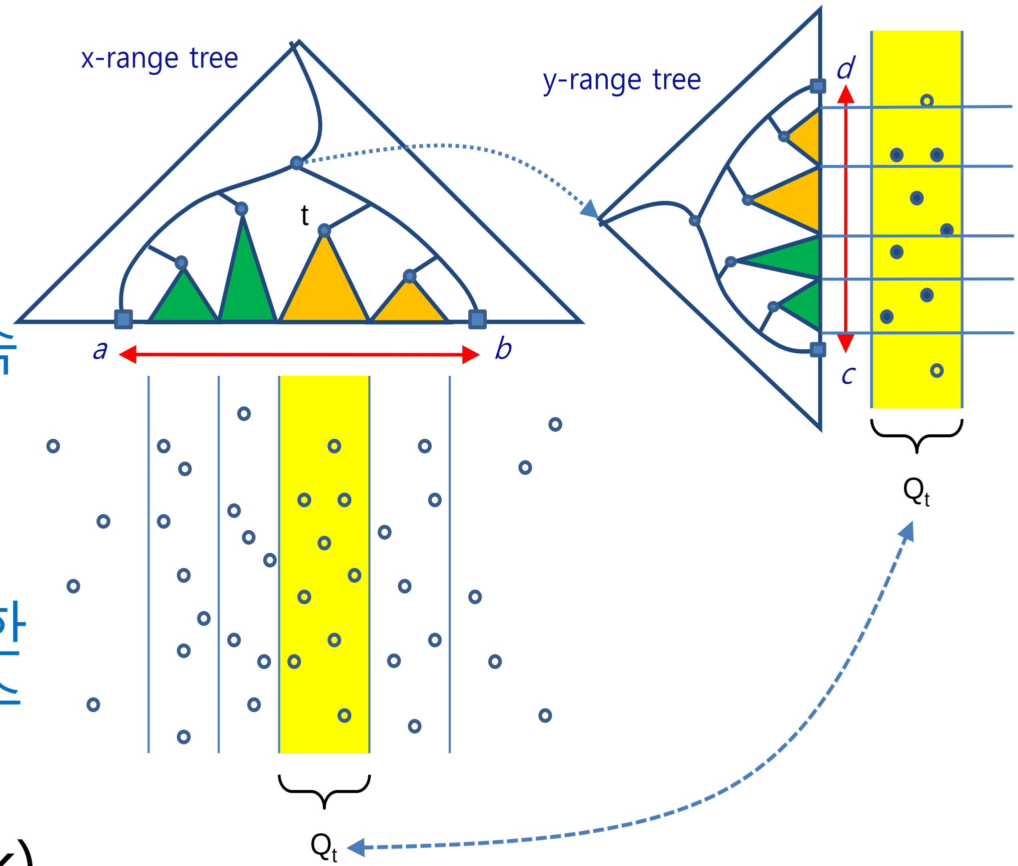
$$O(\sum_{i=1}^{n} T_i)$$

  - $T_i = O(\log n)$
  - Total space is $O(n \log n)$



2차원 Range Tree

- 2차원 range query가 1차원 range query와 다른 점:
  - 1차원 range query에서 어떤 노드 x의 각 subtree에 속하는 원소를 report 하는 부분 대신에, 노드 x에 연결된 y-좌표로 만들어진 1차원 range tree에 대하여 다시 한 번 더 1차원 range tree를 수행하는 것이다.
- 시간복잡도: $O(\log^2 n + k)$



Interval Tree

- 1차원 직선상에서 폐구간(closed interval) [a, b]
  두 실수 a, b(a≤b) 에 대하여 두 실수 사이(a, b를 포함하여)의 모든 실수 집합을 나타냄
- 두 구간 [a, b], [c, d]가 서로 겹치는 경우
  (a≤d) AND (c≤b) 인 경우

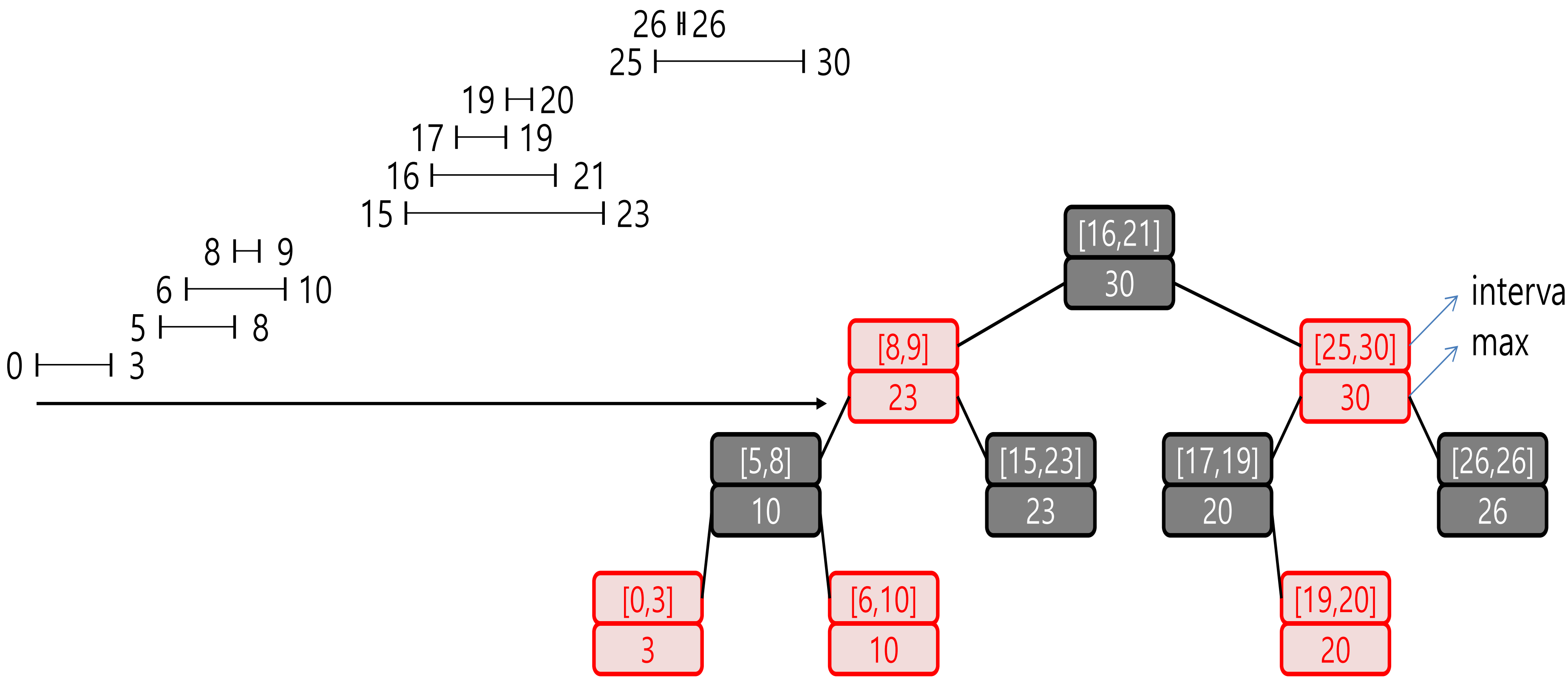
- Interval tree는 아래 연산을 지원하는 자료구조

  - IntervalInsert(T,x) : Interval 트리 T에 폐구간을 저장하는 노드 x를 추가한다.
  - IntervalDelete(T,x) : Interval 트리 T에서 노드 x를 제거한다.
  - IntervalSearch(T,i) : 폐구간 i 와 서로 겹치는 모든 구간을 계산한다.

- Interval Tree는 기본적으로 RBT와 같은 균형트리를 기반으로 만들어 짐

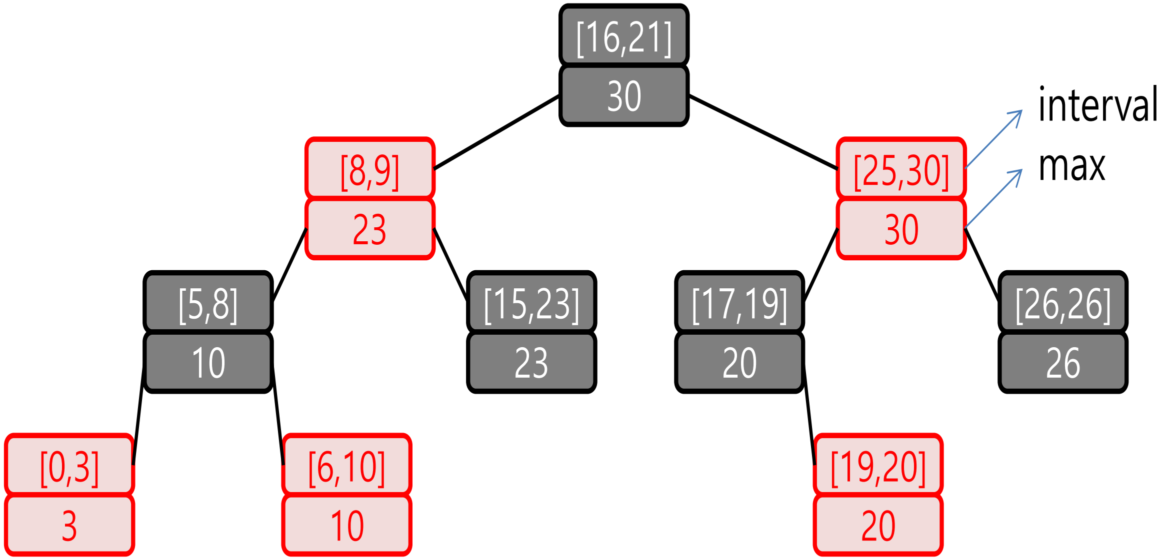
Interval Tree

- 아래와 같은 10개의 구간에 대한 Interval Tree
- 각 구간 [low, high]의 low 값을 키 값으로 사용



Interval Tree

- IntervalInsert(), IntervalDelete() 연산
  - O(logn) 시간에 가능
- IntervalSearch()
  - 구간 i 와 겹치는 어떤 구간을 찾는 연산
  - O(logn) 시간에 가능
  - Ex:
    - [11,13]
    - [20,24]
    - [31,32]
    - [11,15]
    - [24,24]



Interval Tree for Point Querying

- 여러 개의 폐구간이 이미 주어지고, 한 개의 query 점이 주어졌을 때, 이 점을 포함하는 모든 폐구간을 구하는 문제
  - 앞에서 제시한 interval tree와는 구조가 다른 종류의 interval tree를 만들어 해결한다.

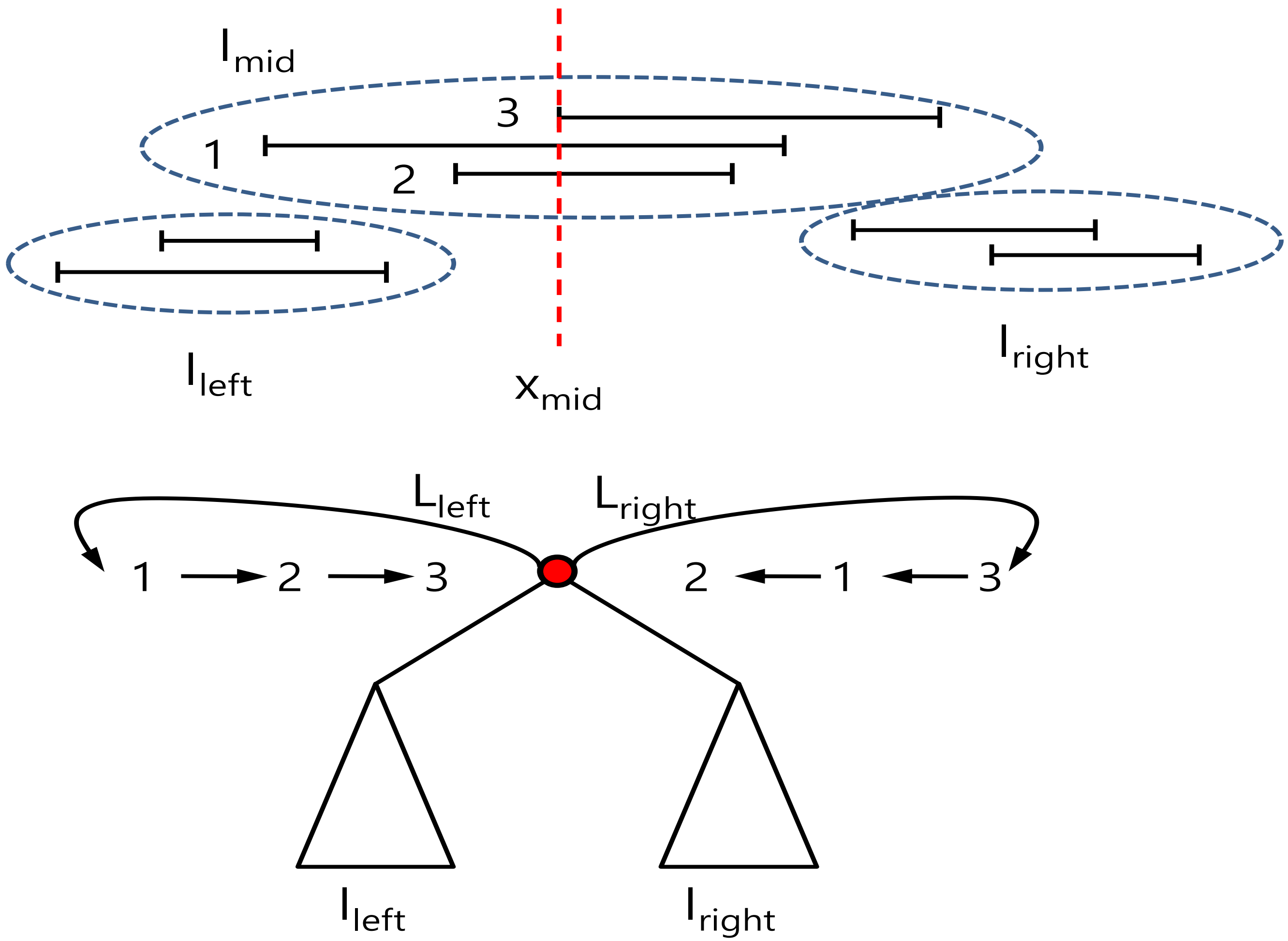
- n개의 폐구간 $I = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$ 주어졌을 때, x_{mid}를 2n개의 폐구간의 끝점의 가장 중앙인 점이라고 정의하자. 그러면, n개의 폐구간은 x_{mid}와의 관계에 의하여 다음과 같은 3개의 집합으로 나눌 수 있다.
  - I_{left} : x_{mid} 보다 완전히 왼쪽에 위치한 폐구간의 집합
  - I_{right} : x_{mid} 보다 완전히 오른쪽에 위치한 폐구간의 집합
  - I_{mid} : x_{mid} 를 포함하는 폐구간의 집합

Interval Tree for Point Querying

- 앞선 조건에 따라 Interval Tree를 다음과 같이 구성
  - (1) 만약 I=∅ 이면, Interval Tree는 단말노드만 가짐
  - (2) 그렇지 않으면, x_{mid} 값을 가지는 루트노드 v를 만들고, 아래와 같이 재귀적으로 루트노드의 subtree를 만듬
    - (2.1) I_{left} 를 이용하여 재귀적으로 Interval Tree를 만들고, 이 트리를 v의 left subtree로 한다.
    - (2.2) I_{right} 를 이용하여 재귀적으로 Interval Tree 를 만들고, 이 트리를 v의 right subtree로 한다.
    - (2.3) 루트노드 v 는 I_{mid} 에 속하는 모든 폐구간을 가지는 두 개의 리스트 $L_{left}(v)$, $L_{right}(v)$를 가진다. $L_{left}(v)$ 에는 I_{mid} 에 속하는 모든 폐구간이 왼쪽 끝점의 오름차순으로 정렬되어 있으며, $L_{right}(v)$ 에는 I_{mid} 에 속하는 모든 폐구간이 오른쪽 끝점의 내림차순으로 정렬되어 있다.

Interval Tree for Point Querying

- 7개의 폐구간으로 만들어진 Interval Tree의 예



Interval Tree for Point Querying

- Query point q_x를 포함하는 모든 폐구간을 찾는 연산

```
QueryIntervalTree(v, qx)
    if v is not a leaf
        if (qx < v->xmid)
          v->Lleft에 연결된 interval을 따라가면서 qx를 포함하는 모든
          interval을 출력하고, qx를 포함하지 않는 interval을 만나자마
          자 Lleft를 탐색하는 것을 종료한다.
          QueryIntervalTree(v->left, qx)
        else
          v->Lright에 연결된 interval을 따라가면서 qx를 포함하는 모
          든 interval을 출력하고, qx를 포함하지 않는 interval을 만나
          자마자 Lright를 탐색하는 것을 종료한다.
          QueryIntervalTree(v->right, qx)
```

- 수행시간 : O(log n + k)