```
Alg isExternal(w)
        input node w
        output boolean

1. if (w.left = ∅ and w.right = ∅)
        return True
   else {w.left ≠ ∅ or w.right ≠ ∅}
        return False
```

```
Alg isInternal(w)
        input node w
        output boolean

1. if (w.left ≠ ∅ or w.right ≠ ∅)
        return True
   else {w.left = ∅ and w.right = ∅}
        return False
```

```
Alg sibling(w)
        input node w
        output sibling of w

1. if (isRoot(w))
        invalidNodeException()              {root has no sibling}
2. if (leftChild(parent(w)) = w)
        return rightChild(parent(w))
   else
        return leftChild(parent(w))
```

```
Alg inOrderSucc(w)
        input internal node w
        output inorder successor of w

1. w ← rightChild(w)
2. if (isExternal(w))
        invalidNodeException()              {No inorder successor}
3. while (isInternal(leftChild(w)))
        w ← leftChild(w)
4. return w
```

```
Alg reduceExternal(z)
        input external node z
        output the node replacing the parent node of the removed node z

1. w ← z.parent
2. zs ← sibling(z)
3. if (isRoot(w))
   < 1 >  root ← zs                                  {renew root}
        zs.parent ← ∅
   else
        g ← w.parent
        zs.parent ← g
        if (w = g.left)
                g.left ← zs
        else {w = g.right}
                g.right ← zs

4. putnode(z)                                        {deallocate node z}
5. putnode(w)                                        {deallocate node w}
6. return zs
```

```
Alg insertItem(k)
        input AVL tree T, key k
        output none

1. w ← treeSearch(root(), k)                    {삽입 키를 저장한 노드 찾기}

2. if (isInternal(w))                           {이미 존재하면 반환}
        return
   else
        Set node w to k
        expandExternal(w)
        searchAndFixAfterInsertion(w)
        return
```

```
Alg searchAndFixAfterInsertion(w)
    input internal node w
    output none

{Update heights and search for imbalance}
1. w.left.height, w.right.height, w.height
    ← 0, 0, 1
2. if (isRoot(w))
        return
3. z ← w.parent
4. while (updateHeight(z) & isBalanced(z))
        if (isRoot(z))
            return
        z ← z.parent
5. if (isBalanced(z))
        return
{Fix imbalance}
6. if (z.left.height > z.right.height)
        y ← z.left
    else {z.left.height < z.right.height}
        y ← z.right
7. if (y.left.height > y.right.height)
        x ← y.left
    else {y.left.height < y.right.height}
        x ← y.right
8. restructure(x, y, z)
9. return
                                    {Total O(log n)}
```

Alg *restructure*(x, y, z)
input internal node x, y, z, s.t. y is the parent of x and z is the parent of y
output internal node

{Let (a, b, c) be an inorder listing of the nodes x, y, and z and let (T_0, T_1, T_2, T_3) be an inorder listing of the four subtrees of x, y, and z not rooted at x, y, or z}
1. **if** ($key(z) < key(y) < key(x)$)
    $a, b, c \leftarrow z, y, x$
    $T_0, T_1, T_2, T_3 \leftarrow a$.**left**, b.**left**, c.**left**, c.**right**
  elseif ($key(x) < key(y) < key(z)$)
    $a, b, c \leftarrow x, y, z$
    $T_0, T_1, T_2, T_3 \leftarrow a$.**left**, a.**right**, b.**right**, c.**right**
  elseif ($key(z) < key(x) < key(y)$)
    $a, b, c \leftarrow z, x, y$
    $T_0, T_1, T_2, T_3 \leftarrow a$.**left**, b.**left**, b.**right**, c.**right**
  else {$key(y) < key(x) < key(z)$}
    $a, b, c \leftarrow y, x, z$
    $T_0, T_1, T_2, T_3 \leftarrow a$.**left**, b.**left**, b.**right**, c.**right**

{Replace the subtree rooted at z with a new subtree rooted at b}
2. **if** (*isRoot*(z))
    root $\leftarrow b$
    b.**parent** $\leftarrow$ *Null*
  elseif (z.**parent**.**left** = z)
    z.**parent**.**left** $\leftarrow b$
    b.**parent** $\leftarrow z$.**parent**
  else {z.**parent**.**right** = z}
    z.**parent**.**right** $\leftarrow b$
    b.**parent** $\leftarrow z$.**parent**

{Let T_0 and T_1 be the left and the right subtree of a, resp.}
3. a.**left**, a.**right** $\leftarrow T_0, T_1$
4. T_0.**parent**, T_1.**parent** $\leftarrow a$
5. *updateHeight*(a)

{Let T_2 and T_3 be the left and the right subtree of c, resp.}
6. c.**left**, c.**right** $\leftarrow T_2, T_3$
7. T_2.**parent**, T_3.**parent** $\leftarrow c$
8. *updateHeight*(c)

{Let a and c be the left and the right child of b, resp.}
9. b.**left**, b.**right** $\leftarrow a, c$
10. a.**parent**, c.**parent** $\leftarrow b$
11. *updateHeight*(b)

12. **return** b

{Total $\mathbf{O}(1)$}

```
Alg updateHeight(w)
    input internal node w
    output boolean

1. h ← max(w.left.height, w.right.height) + 1
2. if (h ≠ w.height)
       w.height ← h
       return True
   else
       return False

Alg isBalanced(w)
    input internal node w
    output boolean

1. return |w.left.height – w.right.height| < 2
```

```
Alg restructure(x)
        input a node x of a binary search tree T that has both a parent y
        and a grandparent z
        output tree T after restructuring involving nodes x, y and z
 1. y ← x.parent
 2. z ← y.parent

{x,y,z의 4가지 경우에 따라 a, b, c 와 T0, T1, T2, T3를 결정}
 3. if (z.key < y.key < x.key)
        a, b, c ← z, y, x
        T0, T1, T2, T3 ← a.left, b.left, c.left, c.right
 4. elseif (x.key < y.key < z.key)
        a, b, c ← x, y, z
        T0, T1, T2, T3 ← a.left, a.right, b.right, c.right
 5. elseif (z.key < x.key < y.key)
        a, b, c ← z, x, y
        T0, T1, T2, T3 ← a.left, b.left, b.right, c.right
 6. else                                                  {y.key < x.key < z.key}
        a, b, c ← y, x, z
        T0, T1, T2, T3 ← a.left, b.left, b.right, c.right

 7. if (isRoot(z))                  {a, b, c 와 T0, T1, T2, T3를 균형 있게 개조}
        root ← b
        b.parent ← ∅
 8. elseif (z.parent.left = z)
        z.parent.left ← b
        b.parent ← z.parent
 9. else                                                  {z.parent.right = z}
        z.parent.right ← b
        b.parent ← z.parent

10. a.left ← T0                                           {노드 a의 자식 조정}
11. T0.parent ← a
12. a.right ← T1
13. T1.parent ← a
14. a->height = max(T0->height, T1->height) + 1;     {노드 a의 높이 업데이트}

15. c.left ← T2                                           {노드 c의 자식 조정}
16. T2.parent ← c
17. c.right ← T3
18. T3.parent ← c
19. c->height = max(T2->height, T3->height) + 1;     {노드 c의 높이 업데이트}

20. b.left ← a                                            {노드 b의 자식 조정}
21. a.parent ← b
22. b.right ← c
23. c.parent ← b
24. b->height = max(a->height, c->height) + 1;       {노드 b의 높이 업데이트}

25. return b
```

```
Alg searchAndRepairAfterInsertion(w)
        input internal node w
        output none

 1. z ← w
 2. while (heightUpdateAndBalanceCheck(z))            {부모노드로 올라가면서
        if (isRoot(z))                                 높이 업데이트 및 균형 검사}
                return;
        z ← z.parent

 3. if (z.left.height > z.right.height)                      {y 자식노드 선택}
        y ← z.left
 4. else {z.left.height < z.right.height}
        y ← z.right

 5. if (y.left.height > y.right.height)                      {x 자식노드 선택}
        x ← y.left
 6. elseif (y.left.height < y.right.height)
        x ← y.right
 7. else {y.left.height = y.right.height)
        x ← y.left or y.right                        {x를 w의 조상 노드로 선택}

 8. restructure(x)                                                  {개조}

 9. return
```

```
Alg heightUpdateAndBalanceCheck(w)              {다른방식으로 구현하여도 됨}
        input internal node w
        output boolean

 1. if (w = ∅)
        return true

 2. l ← w.left                                {부모노드로 올라가면서 균형 검사}
 3. r ← w.right
 4. w.height ← max(r.height, l.height) + 1          {노드 w의 높이 업데이트}

 5. balance ← |r.height - l.height|                          {균형 정도 계산}
 6. return balance < 2                                       {균형 판단}
```

```
Alg removeElement(k)
        input AVL tree T, key k
        output key

1. w ← treeSearch(root(), k)                    {삭제 키를 저장한 노드 찾기}
2. if (isExternal(w))                           {그런 노드가 없으면 반환}
        return NoSuchKey
3. z ← leftChild(w)
4. if (!isExternal(z))
        z ← rightChild(w)
5. if (isExternal(z))                           {case 1}
        zs ← reduceExternal(z)
   else                                         {case 2}
        y ← inOrderSucc(w)
        z ← leftChild(y)
        Set node w to key(y)
        zs ← reduceExternal(z)
6. searchAndFixAfterRemoval(parent(zs))
7. return k
```

```
Alg searchAndFixAfterRemoval(z)
    input internal node z
    output none

{Update heights and search for imbalance}
1. while (updateHeight(z) & isBalanced(z))
        if (isRoot(z))
            return
        z ← z.parent
2. if (isBalanced(z))
        return
{Fix imbalance}
3. if (z.left.height > z.right.height)
        y ← z.left
    else {z.left.height < z.right.height}
        y ← z.right
4. if (y.left.height > y.right.height)
        x ← y.left
    elseif (y.left.height < y.right.height)
        x ← y.right
    else {y.left.height = y.right.height}
        if (z.left = y)
            x ← y.left
        else {z.right = y}
            x ← y.right
5. b ← restructure(x, y, z)
6. if (isRoot(b))
        return
7. searchAndFixAfterRemoval(b.parent)
                                {Total O(log n)}
```

```
Alg searchAndRepairAfterRemoval(w)
        input internal node w
        output none

 1. z ← w                                          {부모노드로 올라가면서 균형 검사}
 2. while (heightUpdateAndBalanceCheck(z))
        if (isRoot(z))
                return;
        z ← z.parent

 3. if (z.left.height > z.right.height)                        {y 자식노드 선택}
        y ← z.left
 4. else {z.left.height < z.right.height}
        y ← z.right

 5. if (y.left.height > y.right.height)                        {x 자식노드 선택}
        x ← y.left
 6. elseif (y.left.height < y.right.height)
        x ← y.right
 7. < 2 > elseif (z.left = y)
        x ← y.left
 8. else {z.right = y}
        x ← y.right

 9. b ← restructure(x)                                                  {개조}

10. searchAndRepairAfterRemoval(b)              {전역적인 균형을 맞추기 위해 재귀}

11. return
```