Chapter 3: Searching

- Reading Assignment: Chapter 3.2

- Dictionary ADT – should support three operations, INSERT, DELETE and FIND

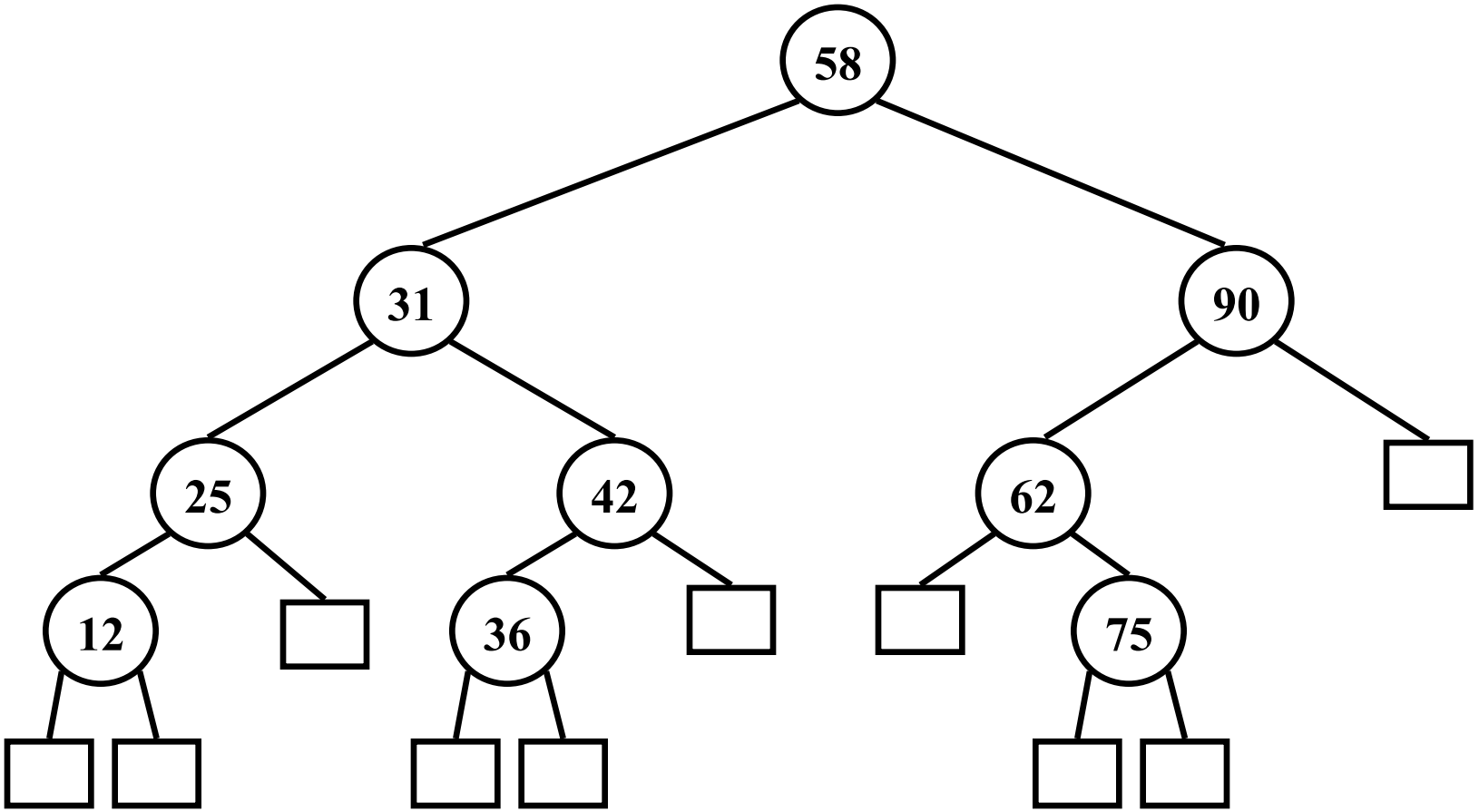
Implementation Strategies

	insert()	**delete()**	**member()**
Linear list (linked list, array, vector)	O(1)	O(n)	O(n)
Balanced binary search tree	O(log n)	O(log n)	O(log n)
Hash table	O(1)	O(1)	O(1)

Definition of Binary Search Trees

- A *binary search tree* is a binary tree in which each internal node *v* stores an element *e* such that the elements stored in the left subtree of *v* are less than or equal to *e*, and the elements stored in the right subtree of *v* are greater than or equal to *e*.

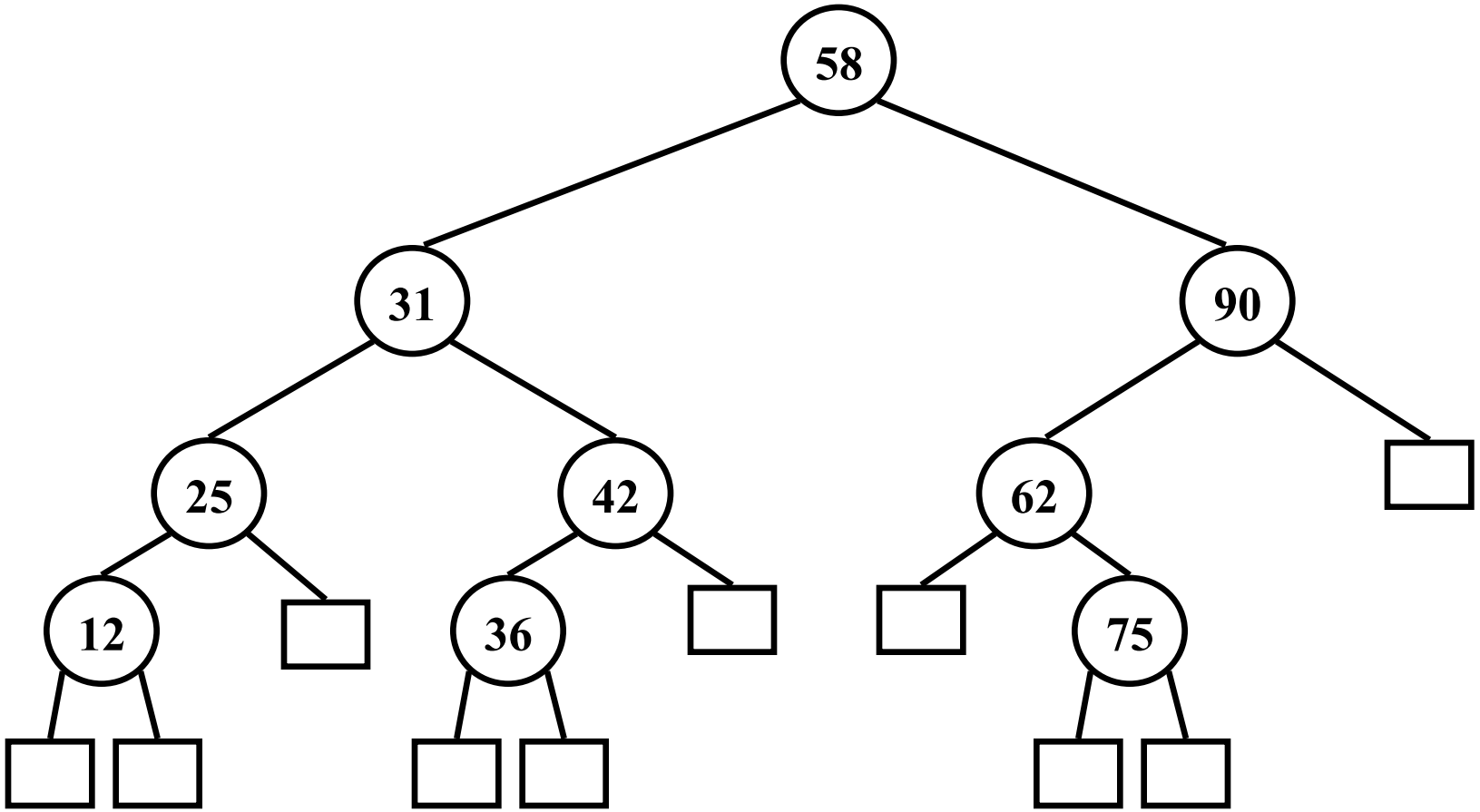
Example of a Binary Search Tree



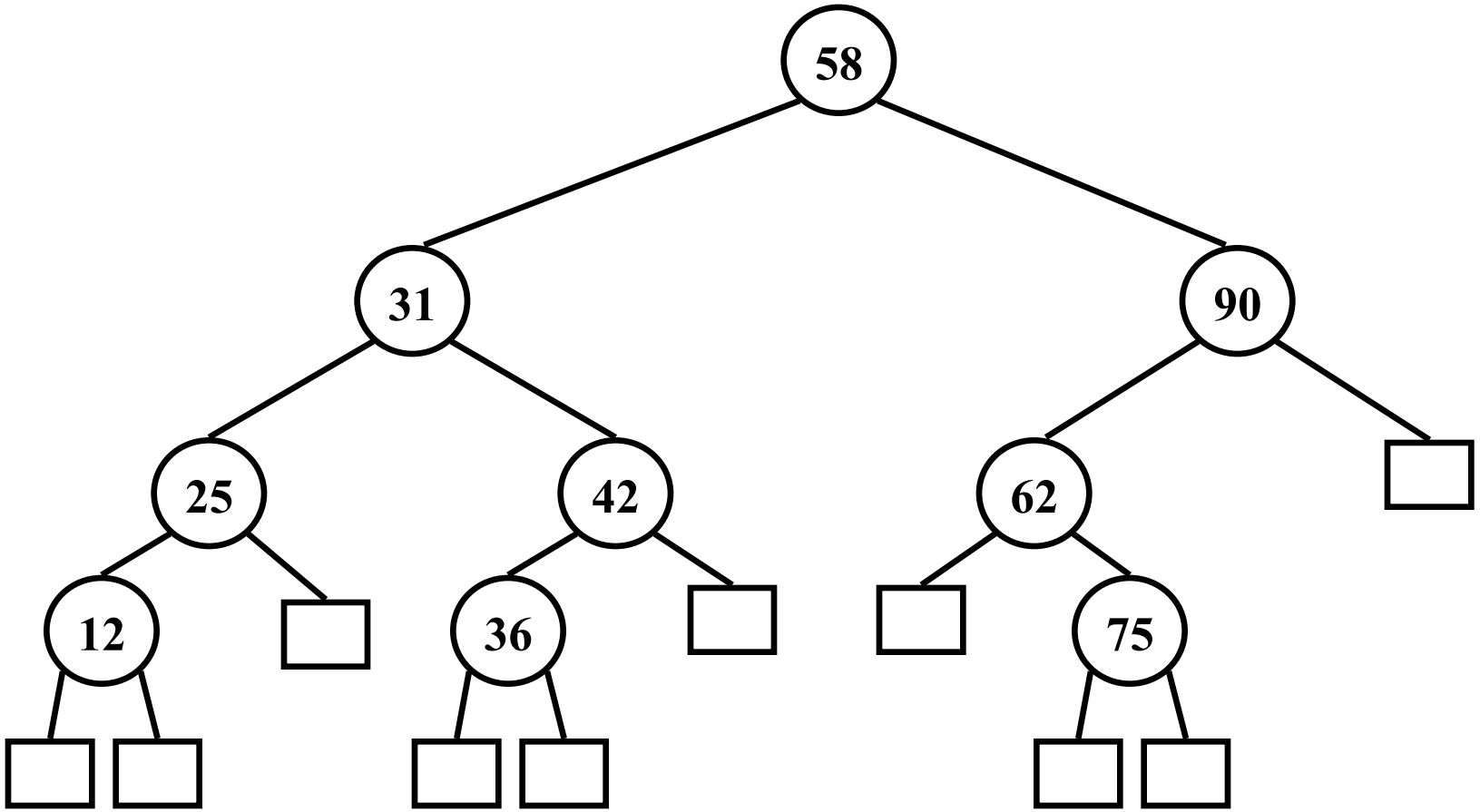
Definition of Binary Search Trees

- **More formally**: a *binary search tree* is a binary tree in which each internal node v of T stores an item (k,e) of a dictionary D, and keys stored at nodes in the left subtree of v are less than or equal to k, while keys stored at nodes in the right subtree of v are greater than or equal to k.

Example of a Binary Search Tree

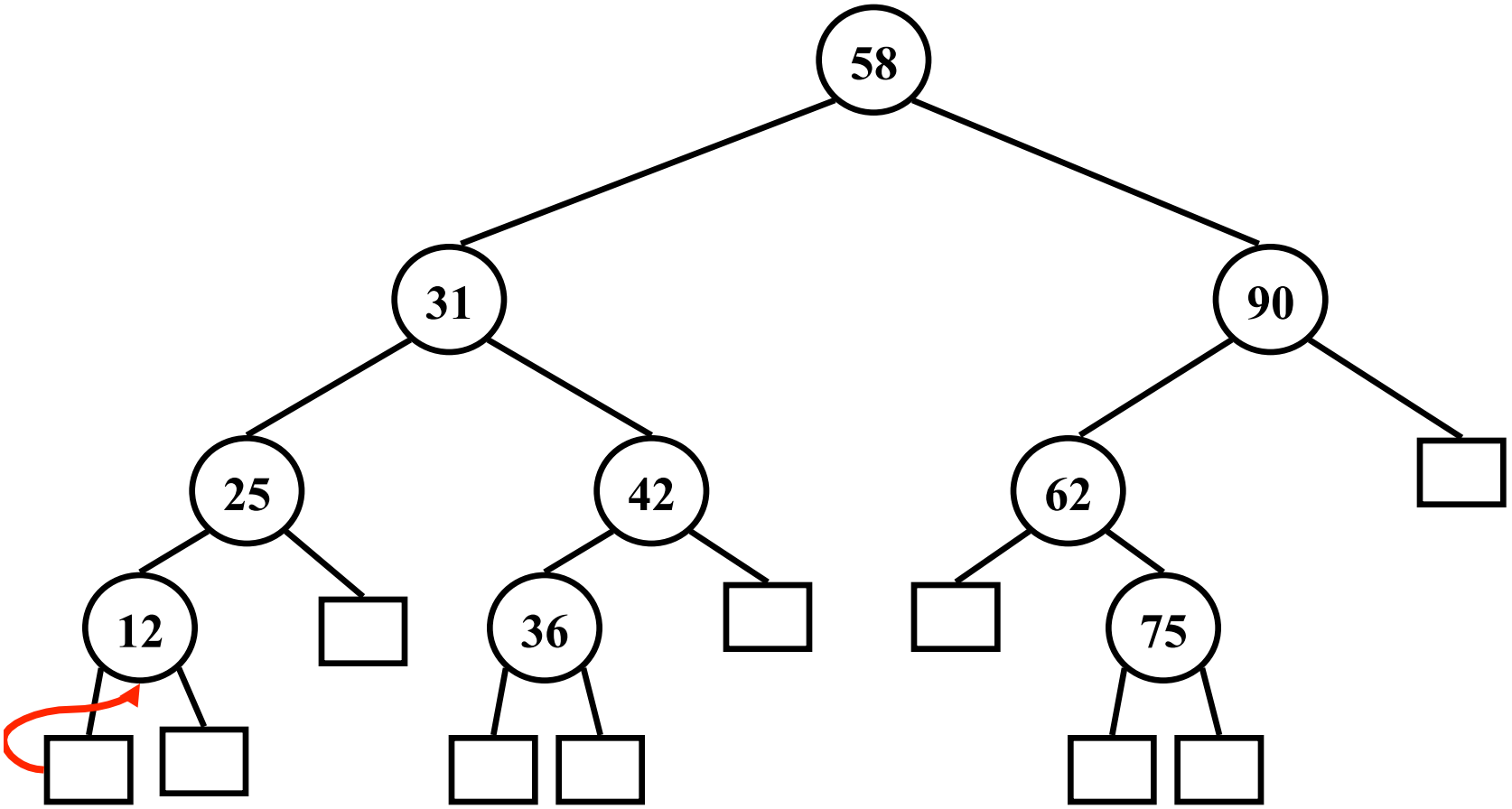


Inorder Traversal on a Binary Search Tree



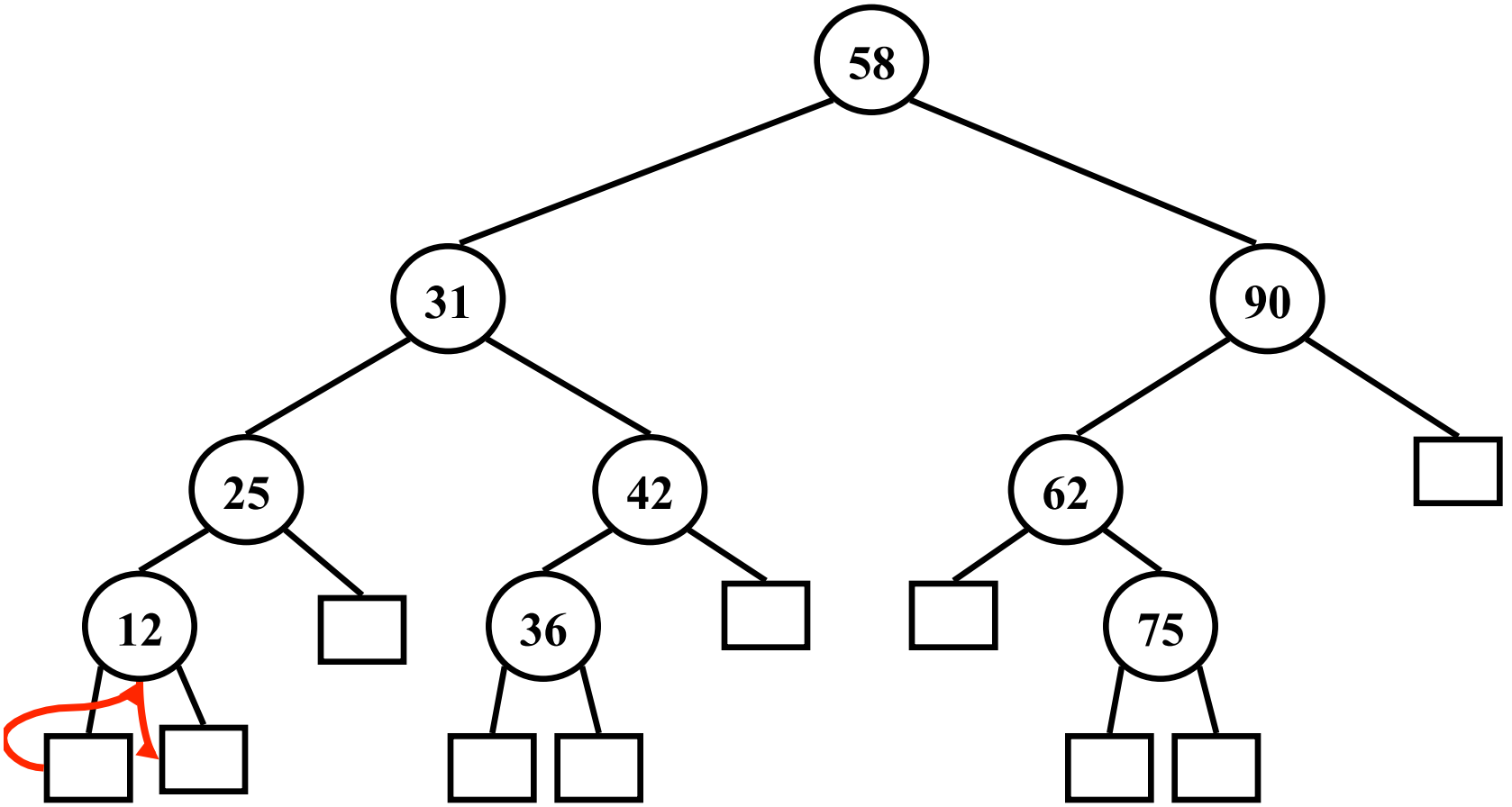
Inorder Traversal on a Binary Search Tree

12



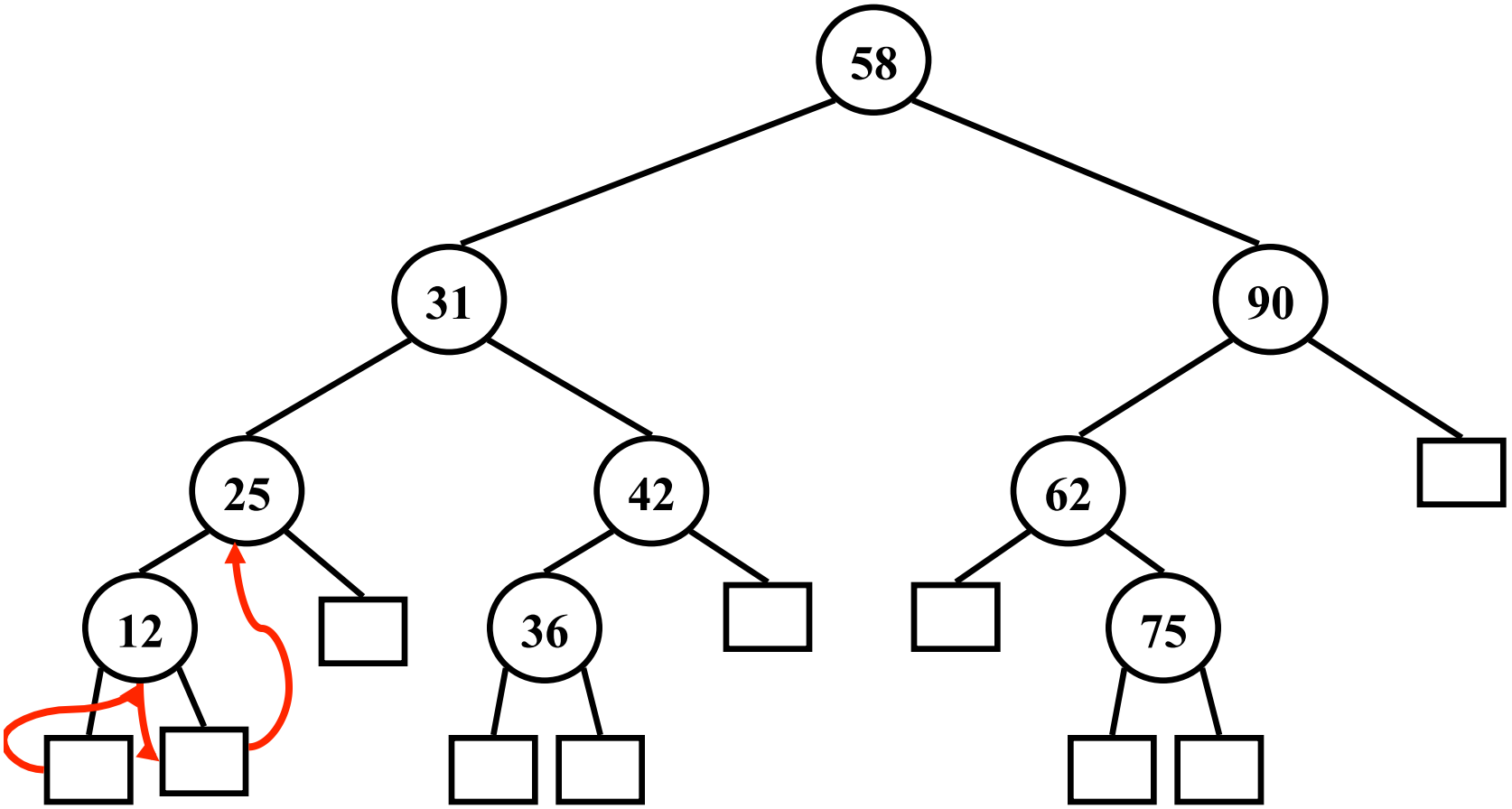
Inorder Traversal on a Binary Search Tree

12



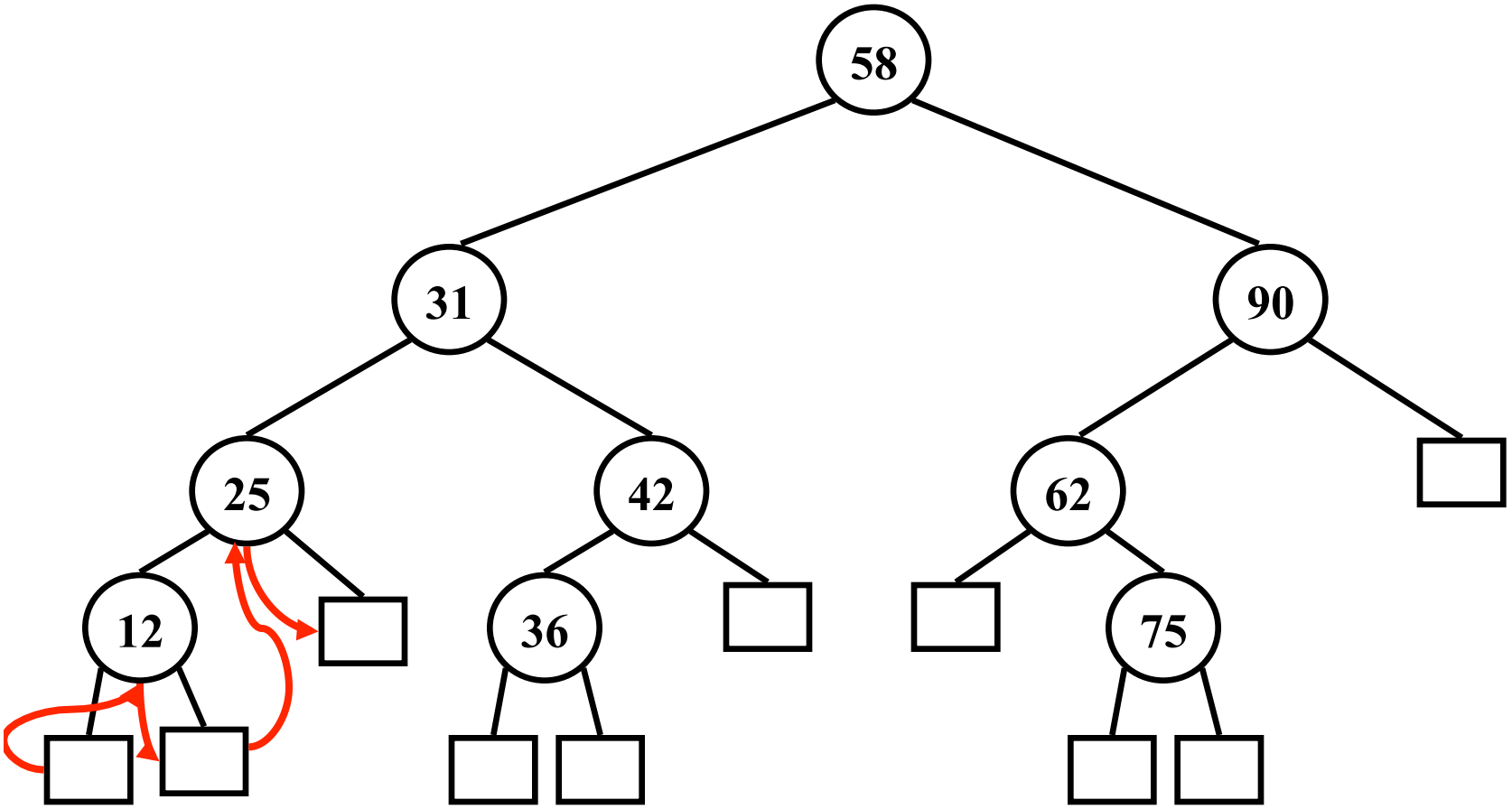
Inorder Traversal on a Binary Search Tree

12 **25**



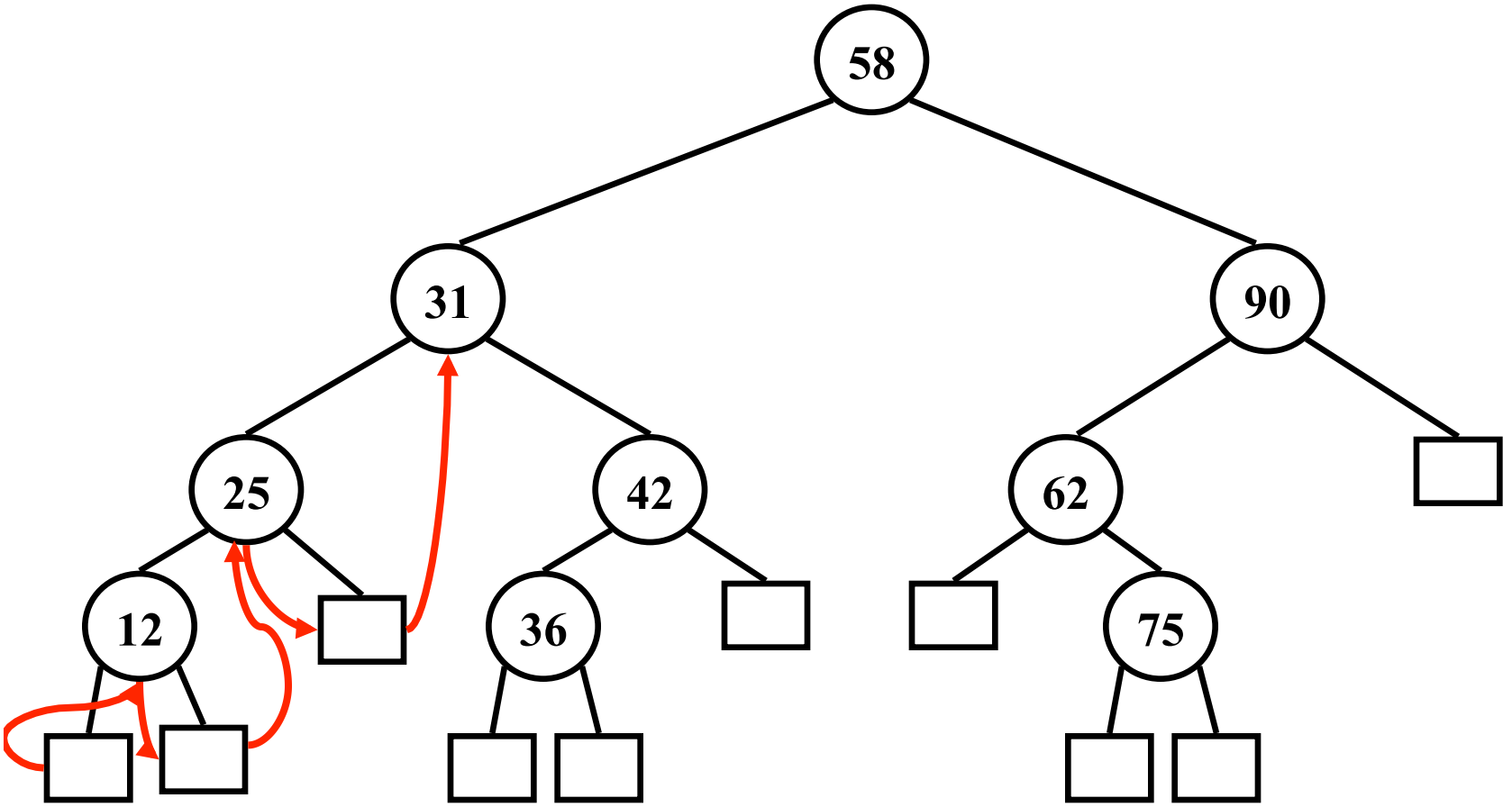
Inorder Traversal on a Binary Search Tree

12 **25**



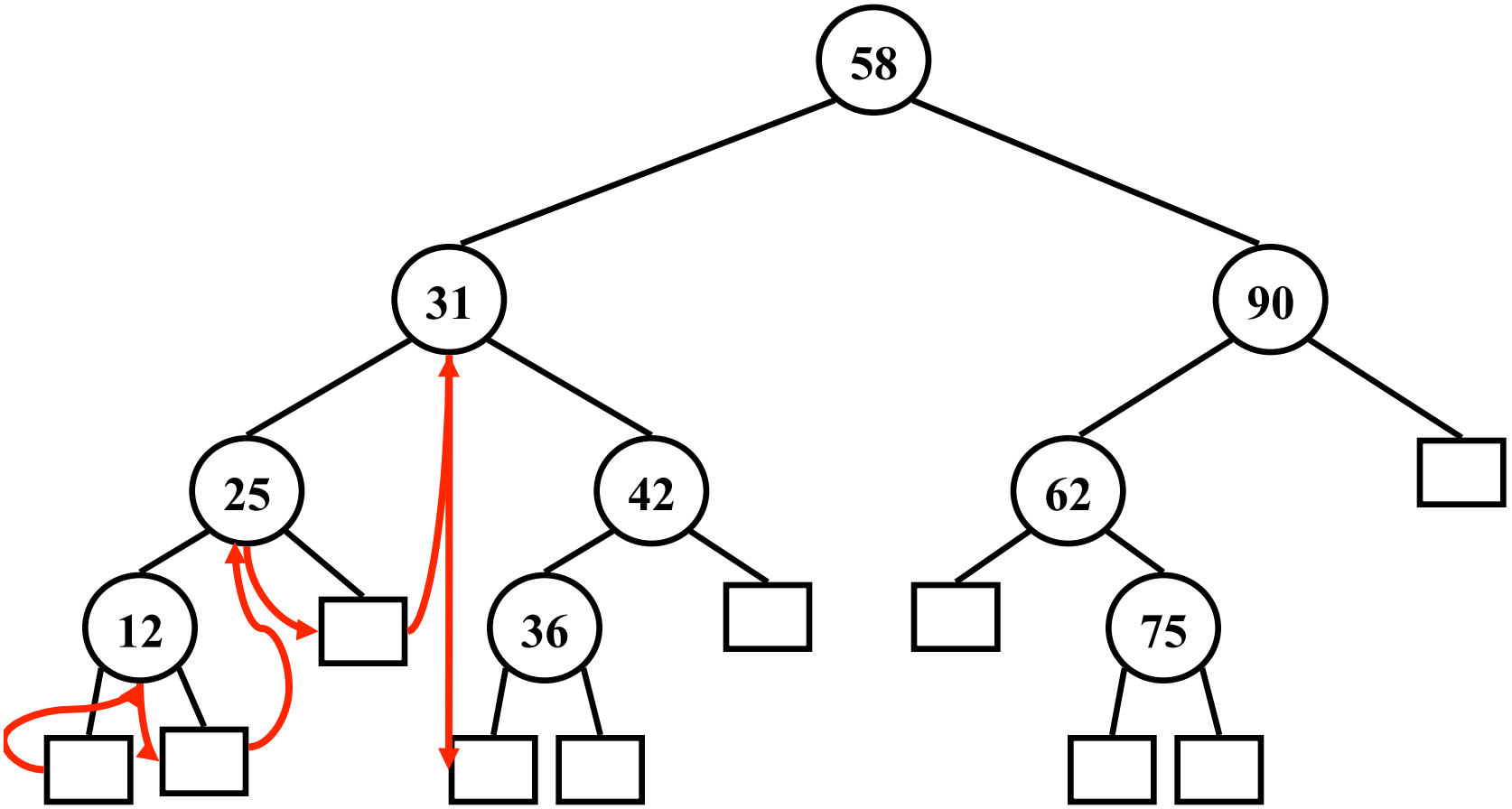
Inorder Traversal on a Binary Search Tree

12 25 31



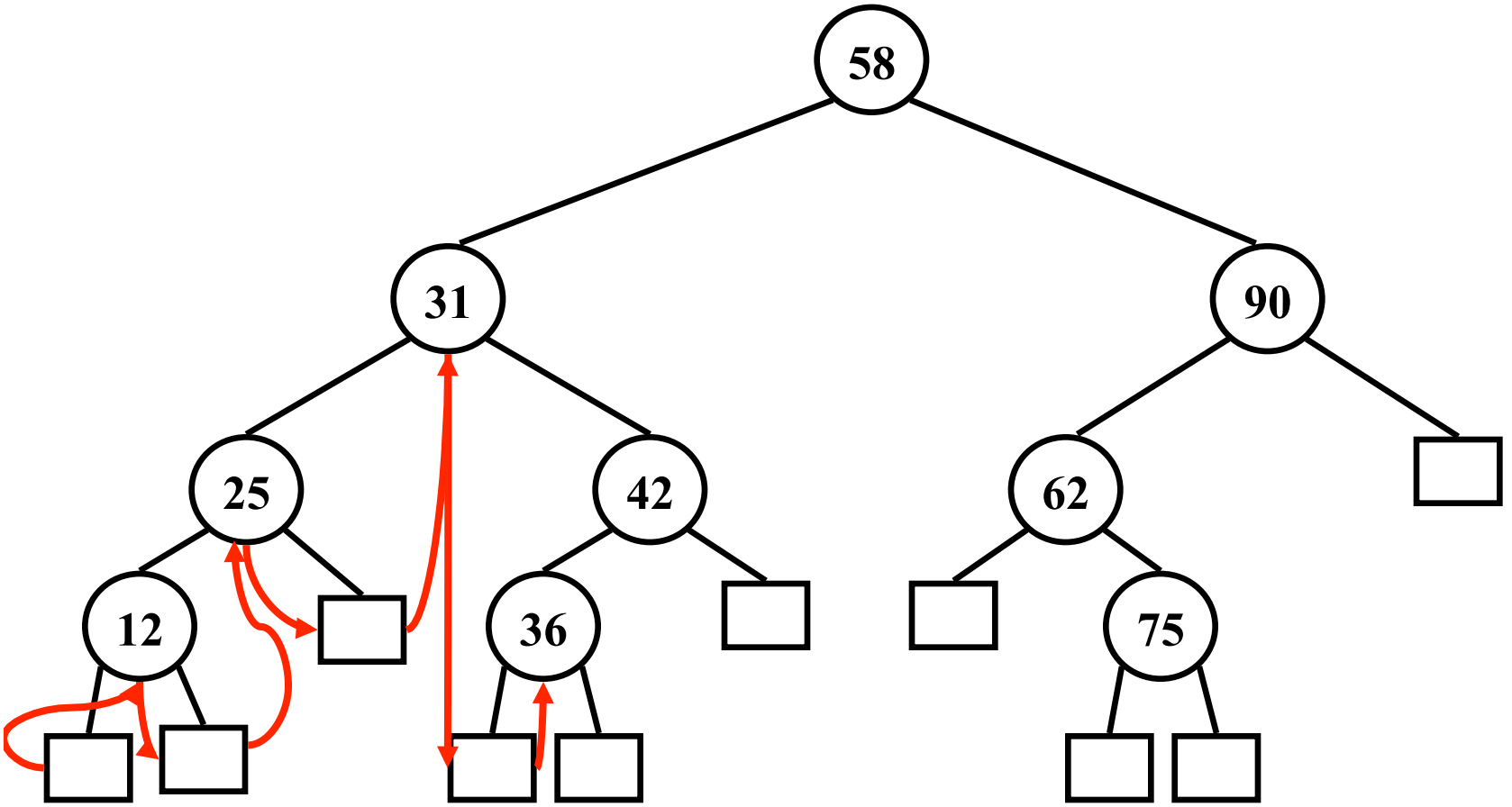
Inorder Traversal on a Binary Search Tree

12 25 31



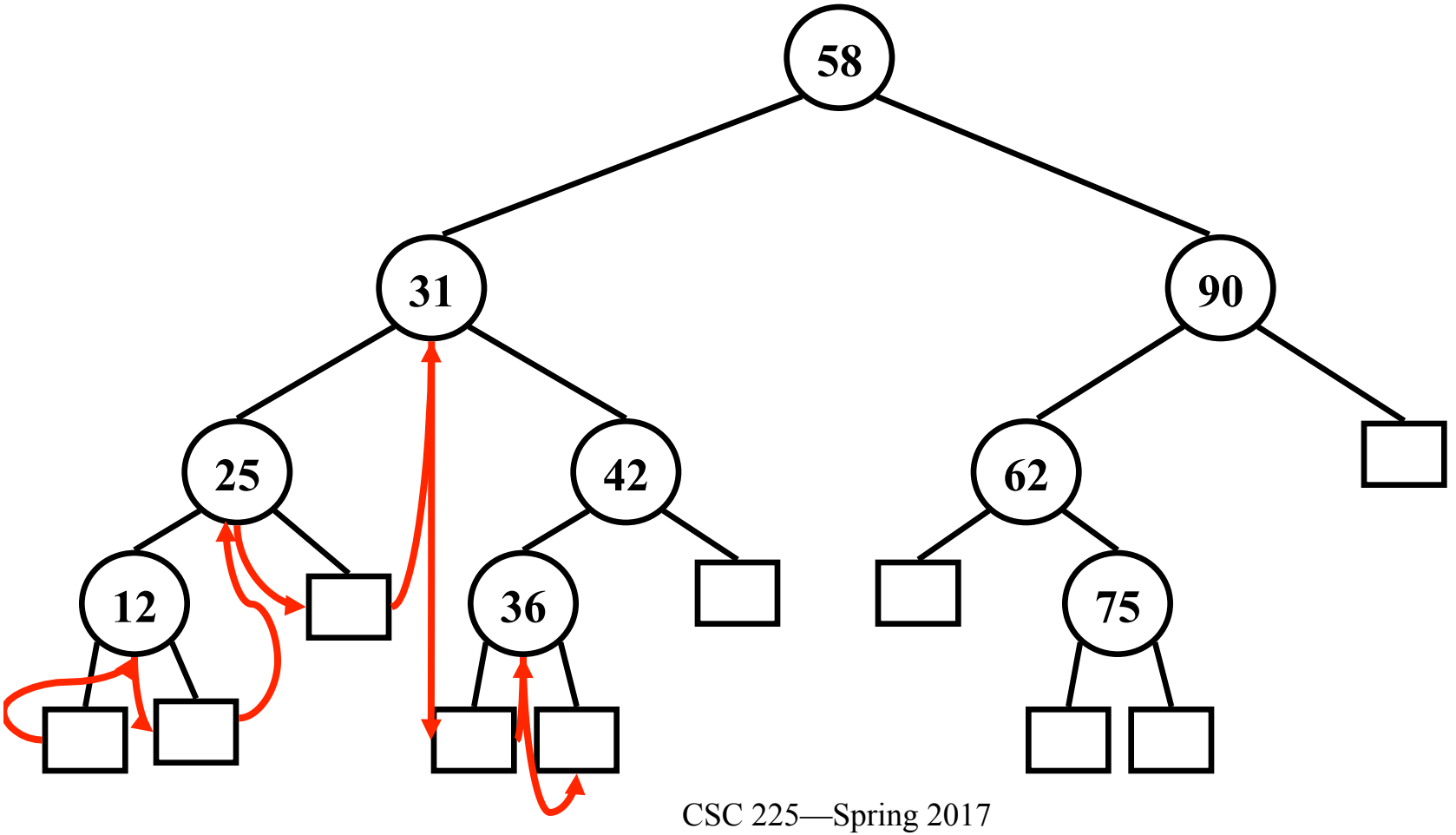
Inorder Traversal on a Binary Search Tree

12 25 31 36



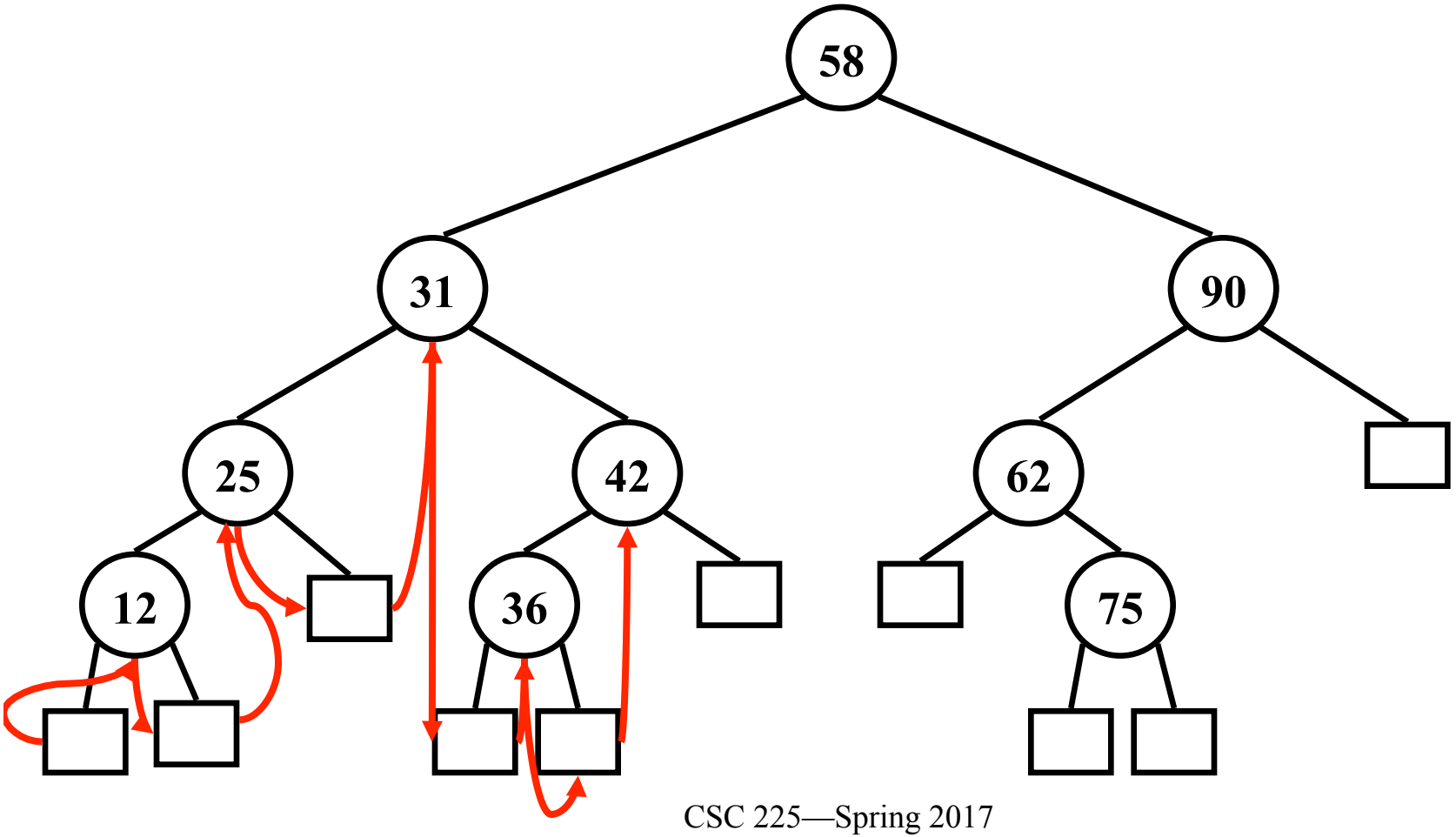
Inorder Traversal on a Binary Search Tree

12 25 31 36



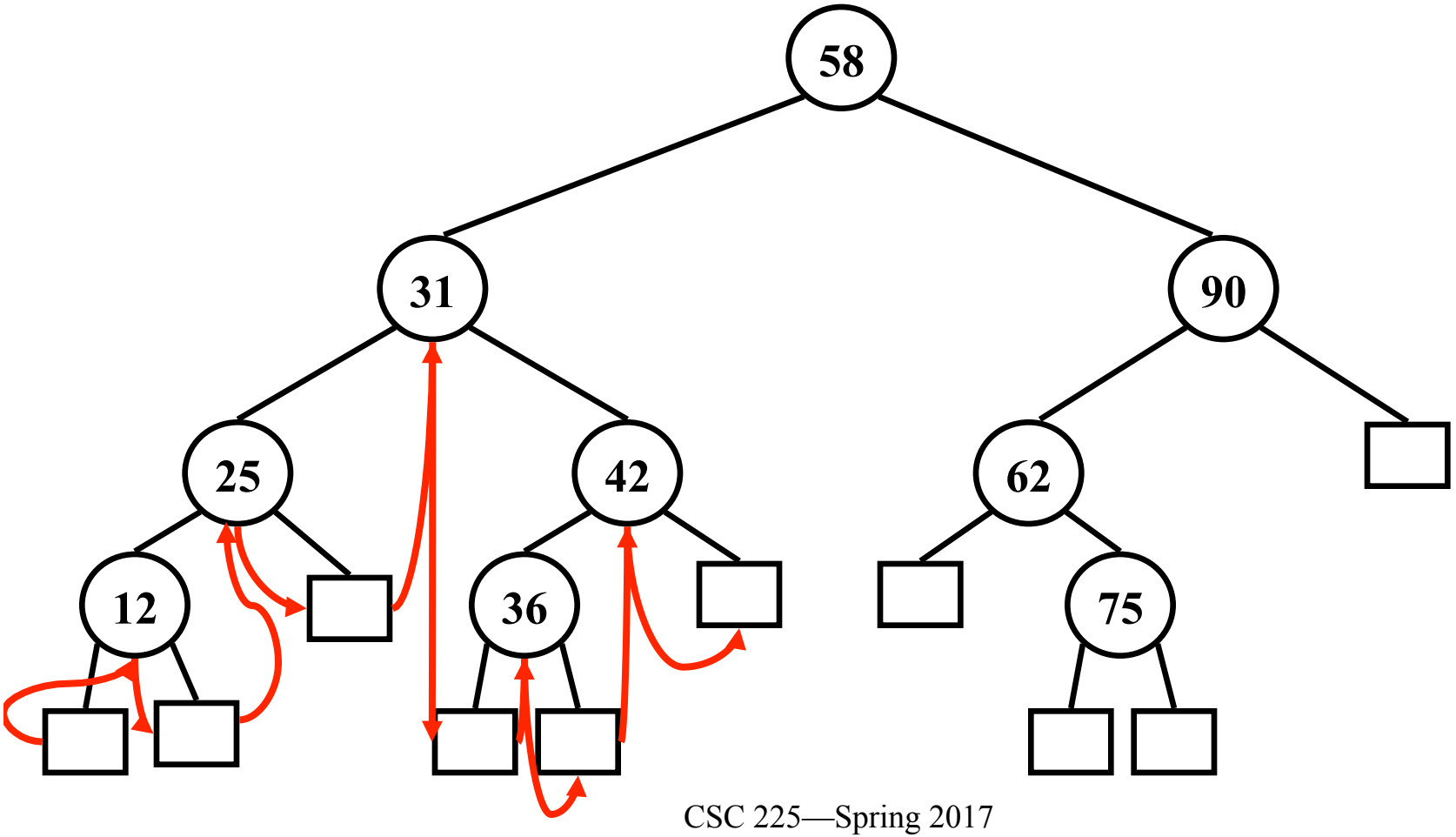
Inorder Traversal on a Binary Search Tree

12 25 31 36 42



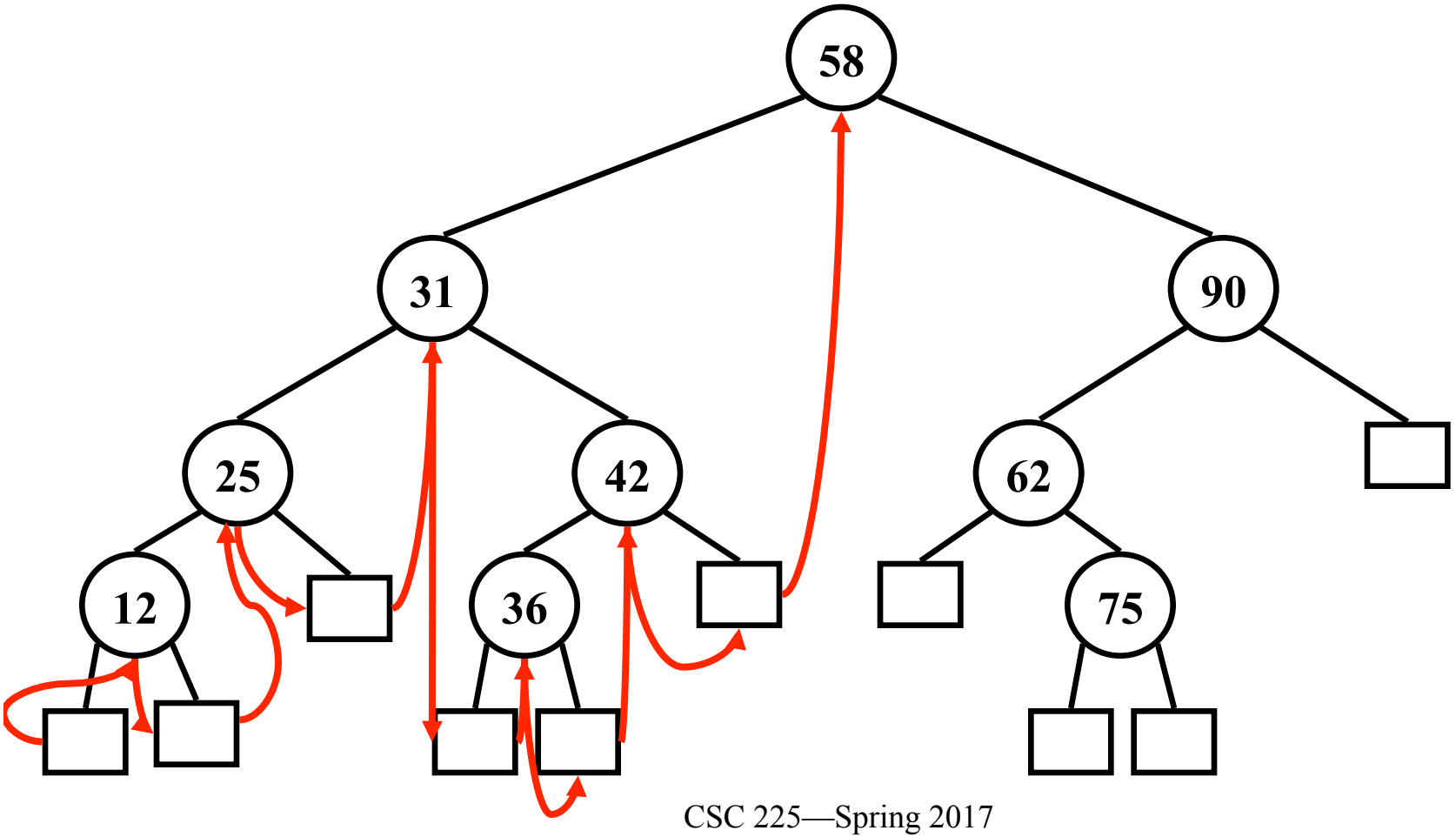
Inorder Traversal on a Binary Search Tree

12 25 31 36 42



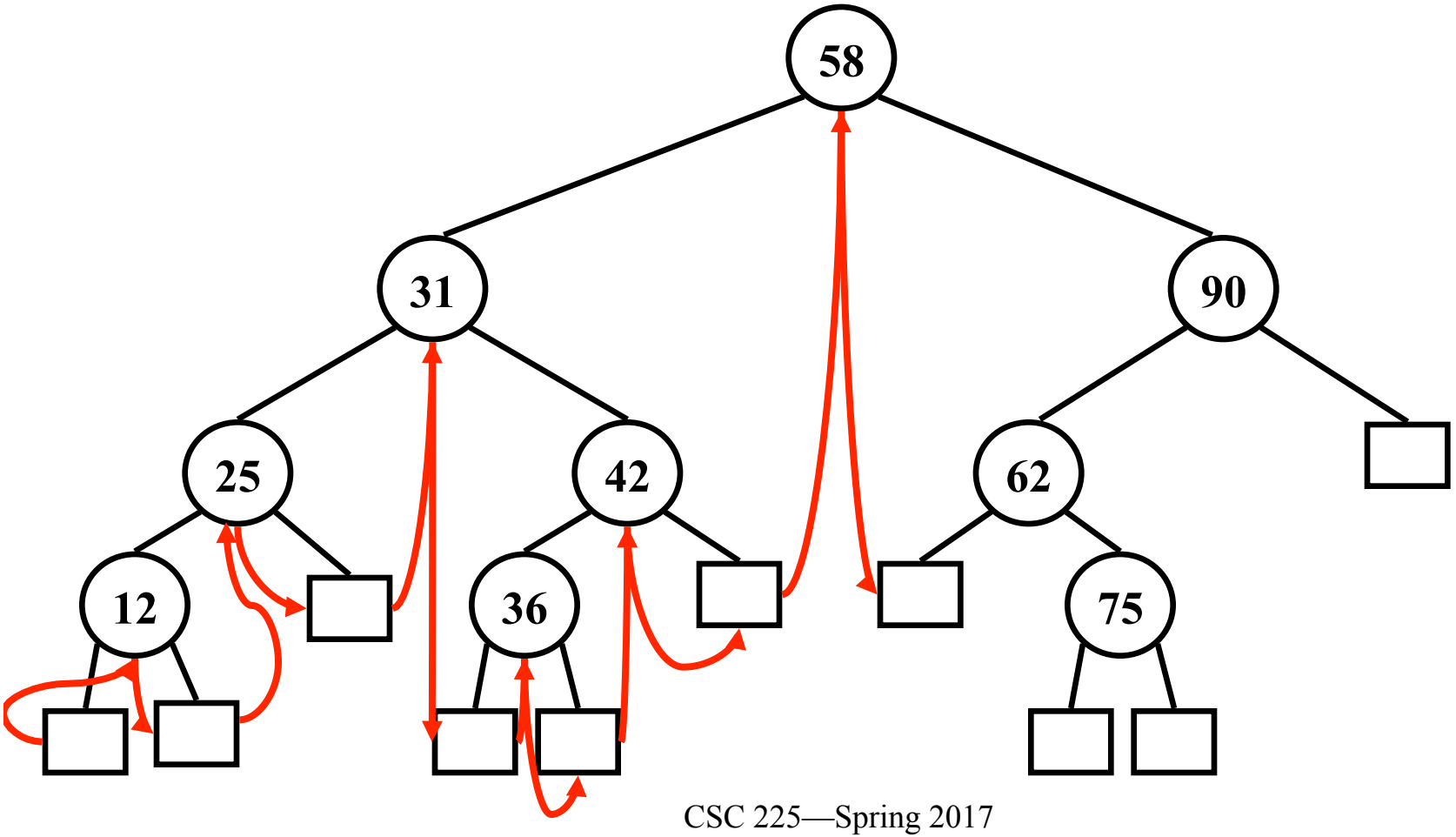
Inorder Traversal on a Binary Search Tree

12 25 31 36 42 58



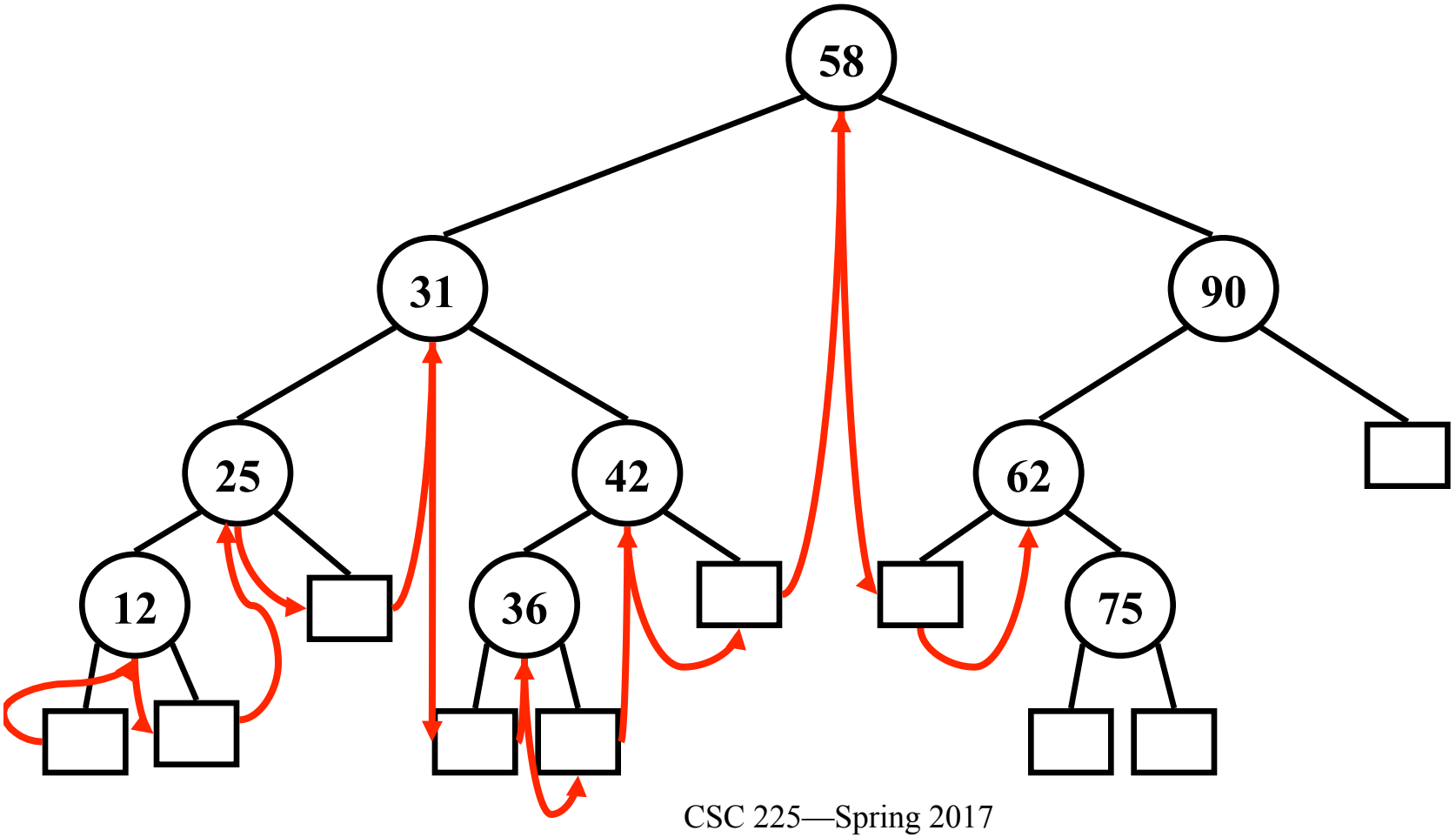
Inorder Traversal on a Binary Search Tree

12 25 31 36 42 58



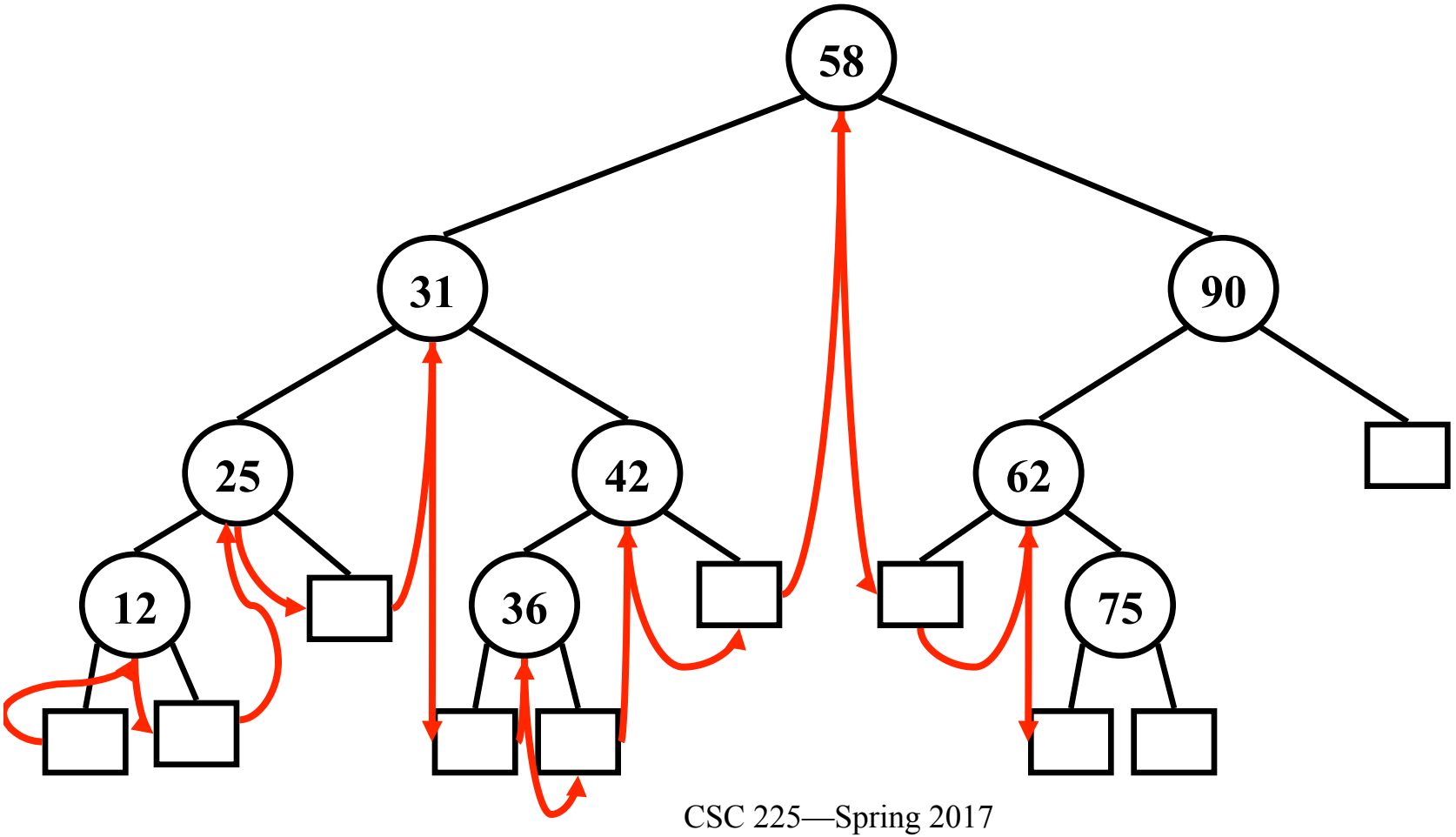
Inorder Traversal on a Binary Search Tree

12 25 31 36 42 58 62

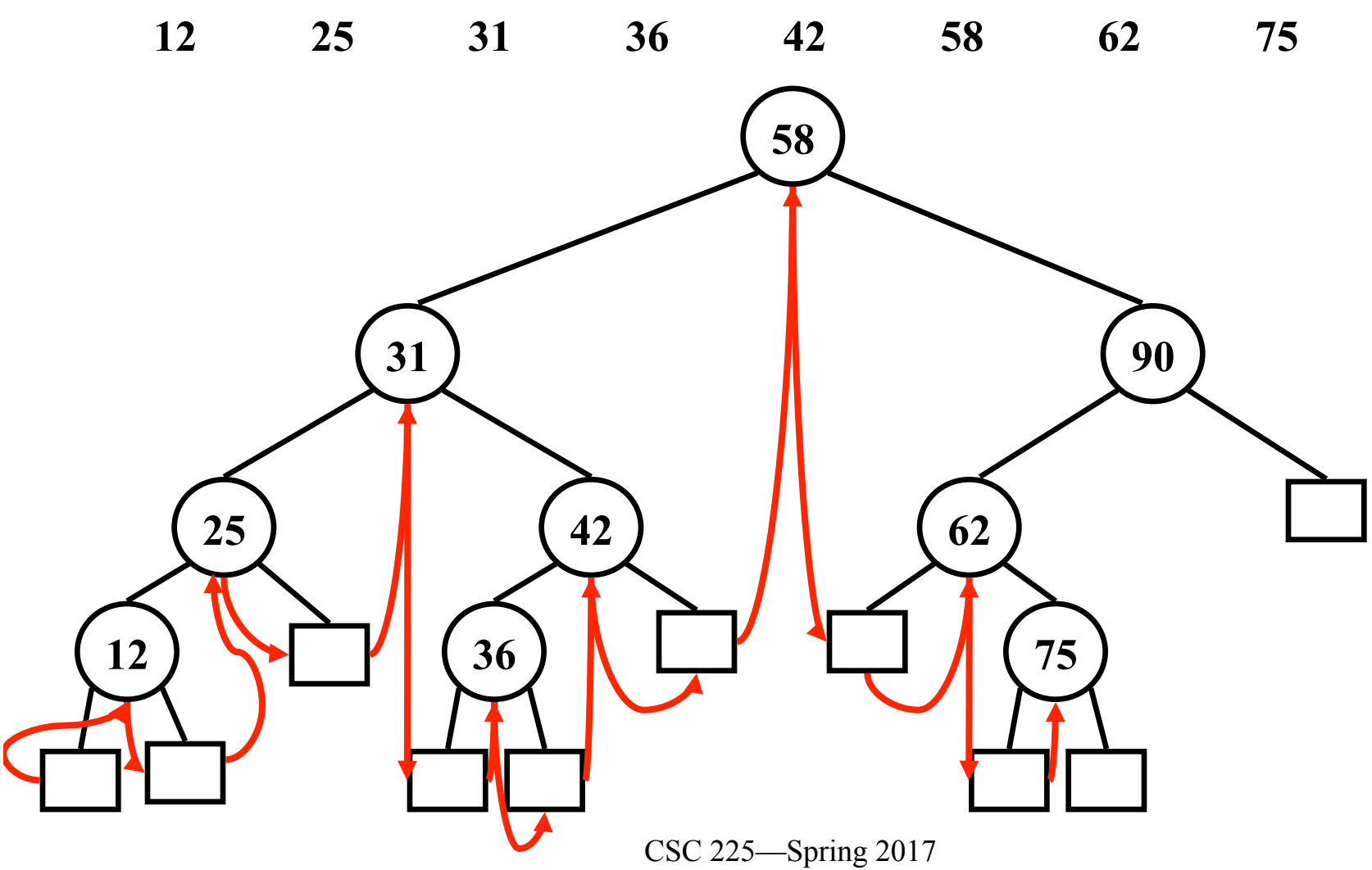


Inorder Traversal on a Binary Search Tree

12 25 31 36 42 58 62

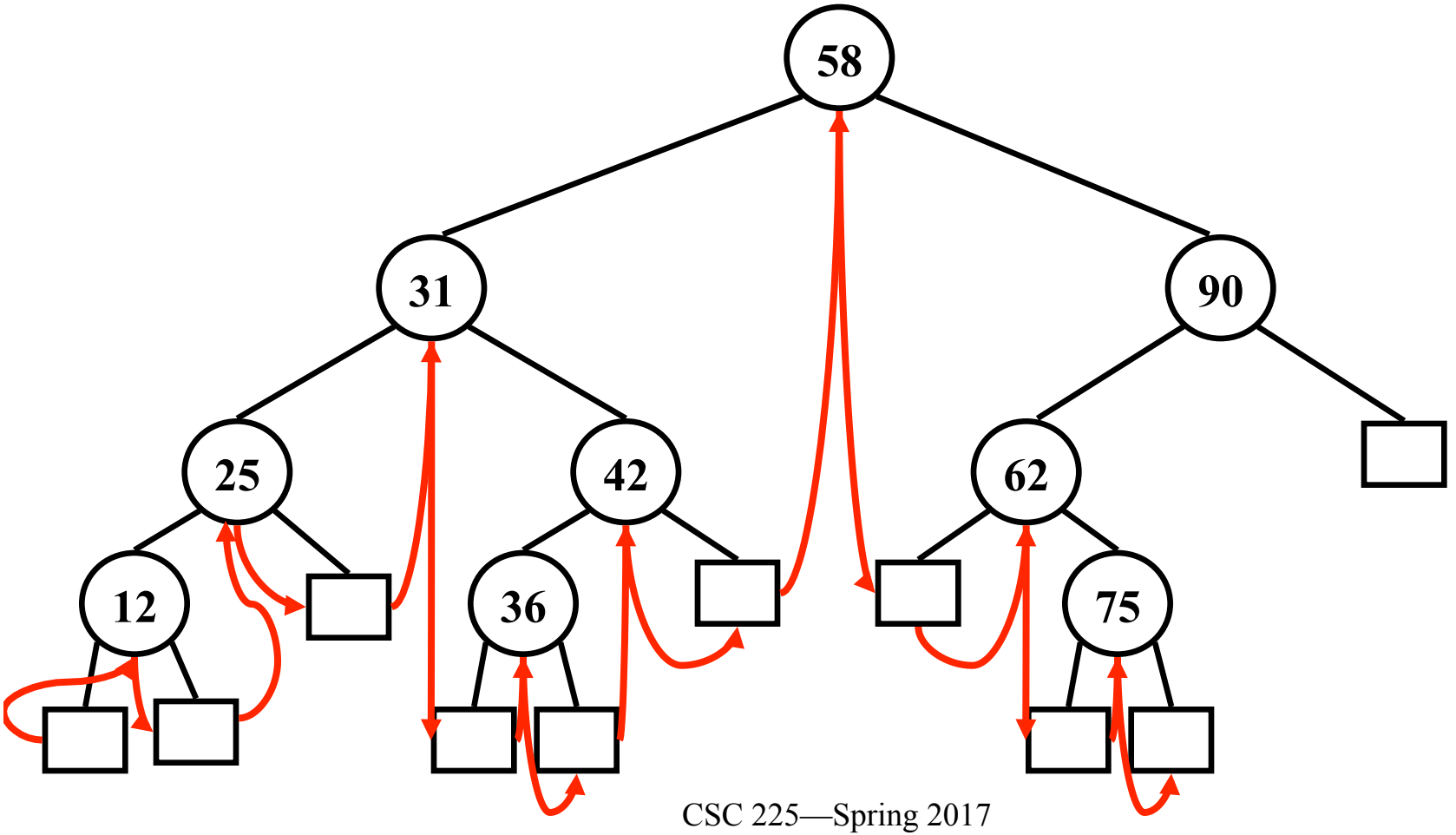


Inorder Traversal on a Binary Search Tree



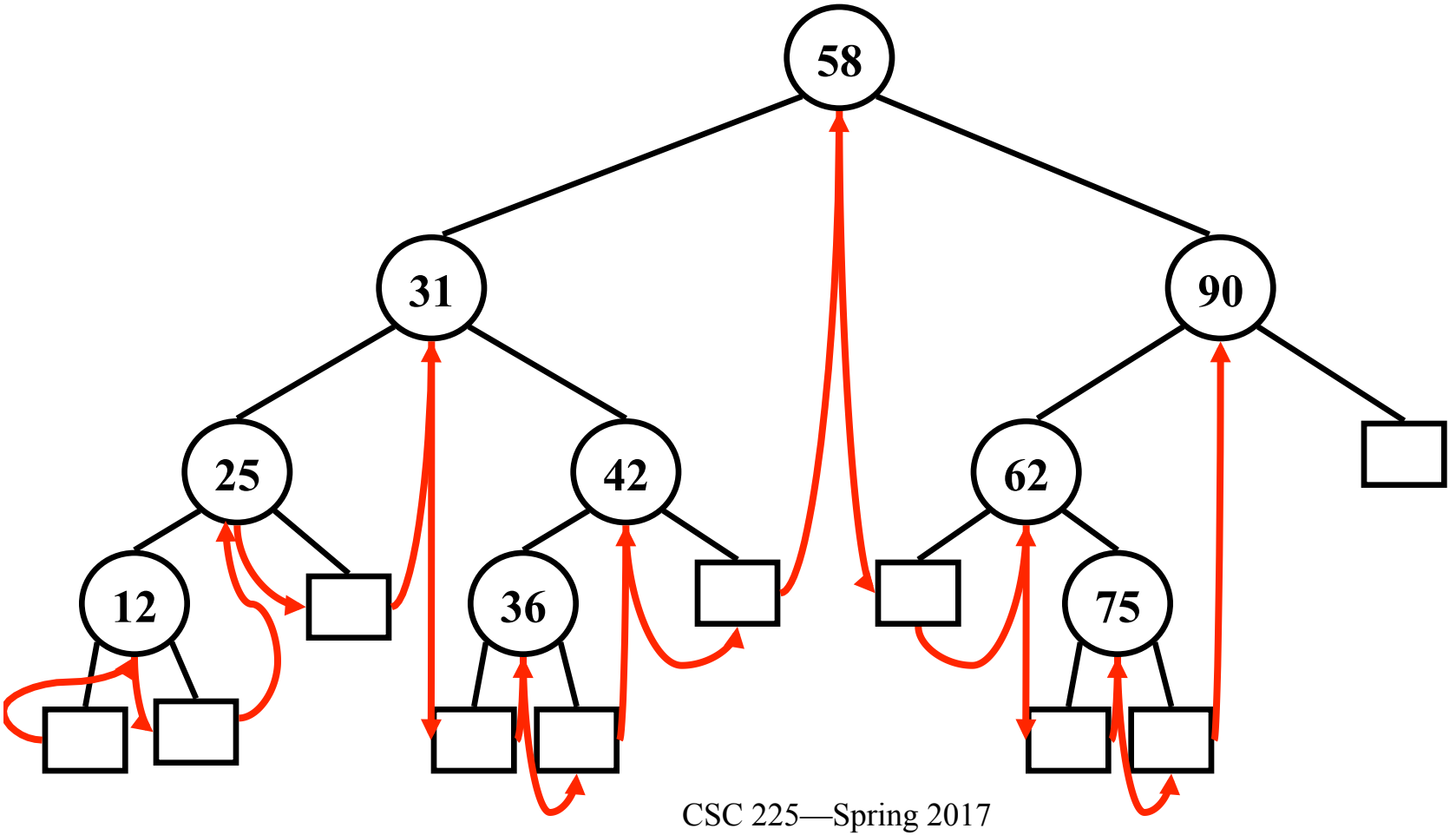
Inorder Traversal on a Binary Search Tree

12 25 31 36 42 58 62 75



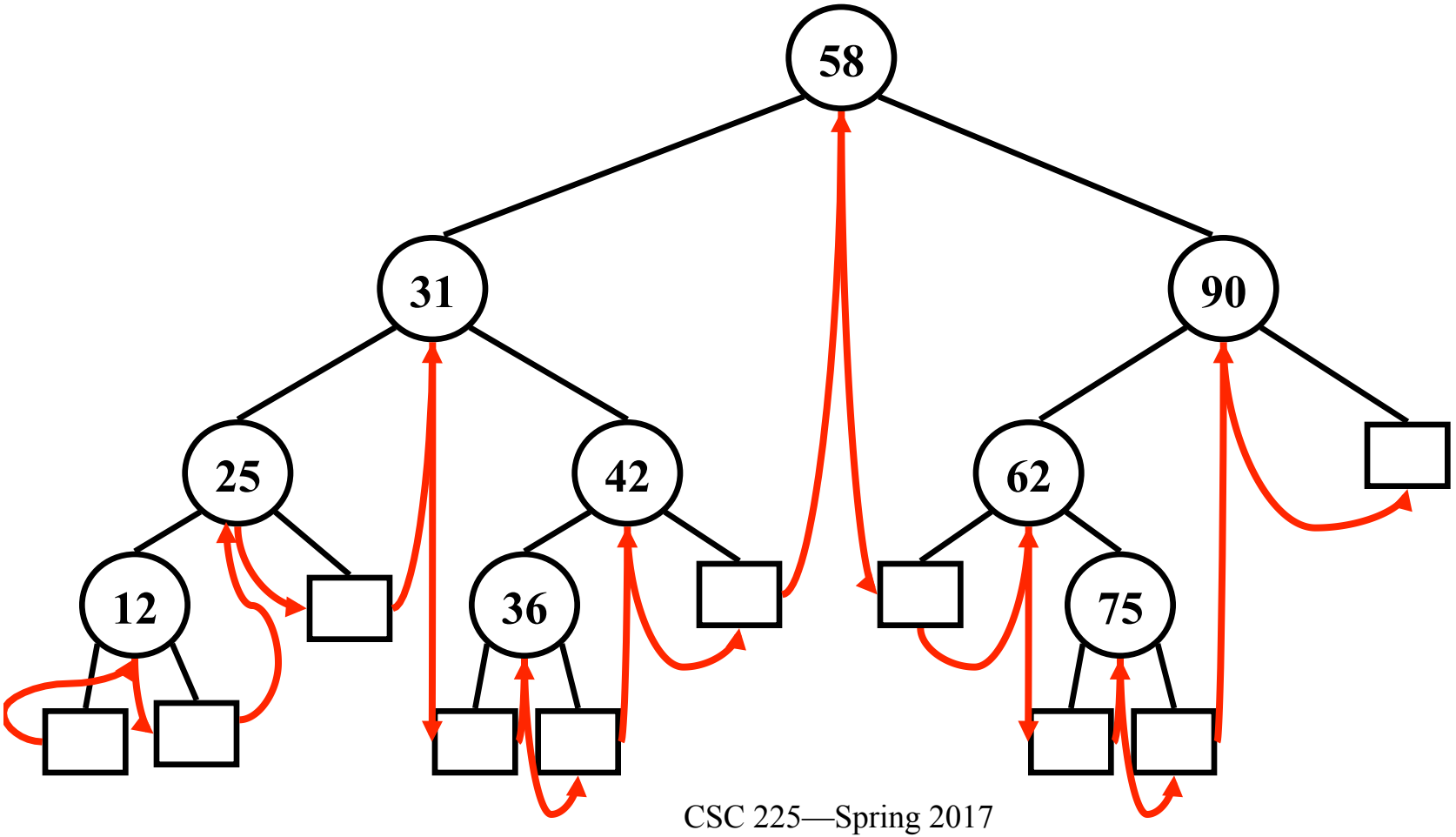
Inorder Traversal on a Binary Search Tree

12 25 31 36 42 58 62 75 90



Inorder Traversal on a Binary Search Tree

12 25 31 36 42 58 62 75 90

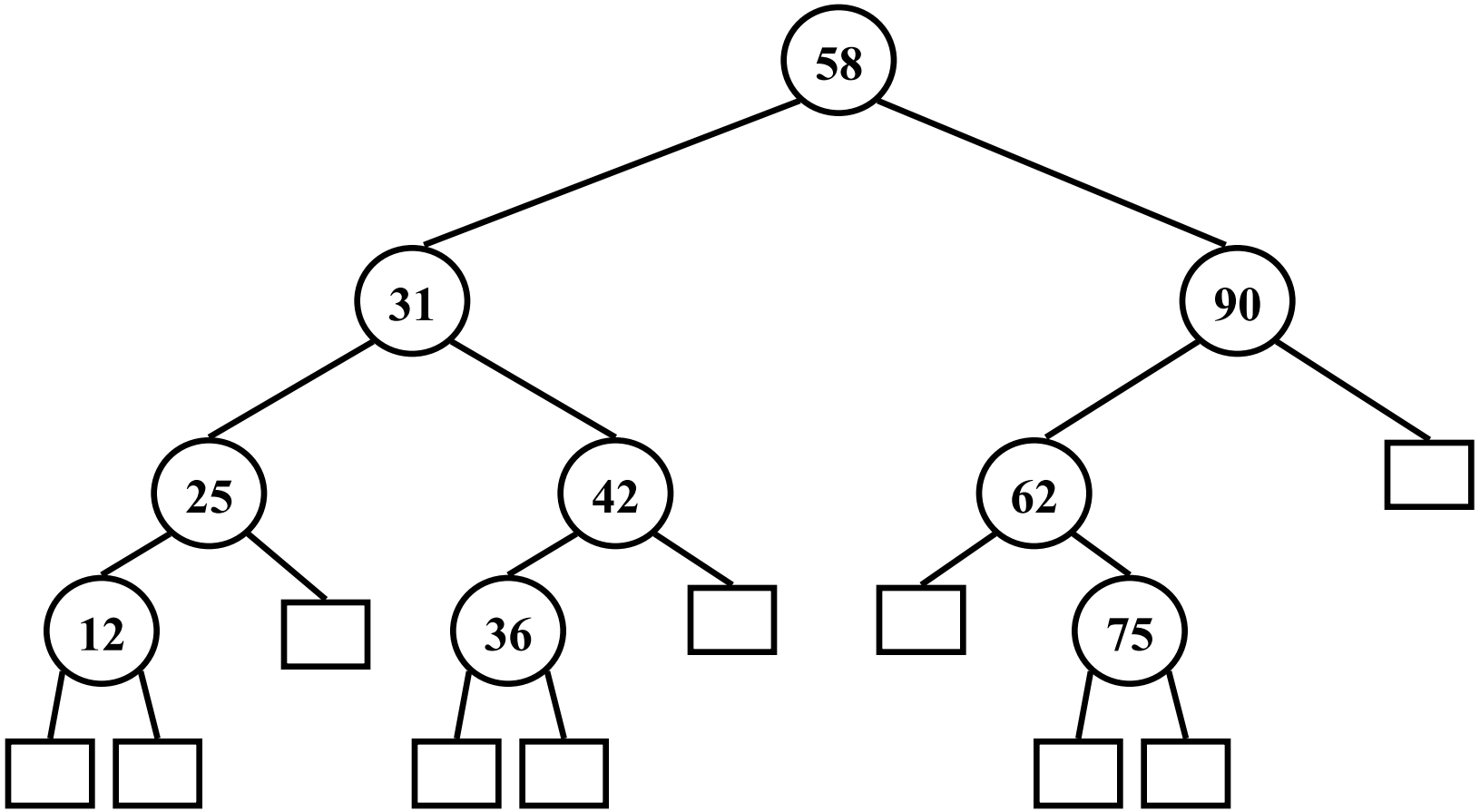


Inorder Traversal on a Binary Search Tree

- visits the elements stored in *nondecreasing* order

Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



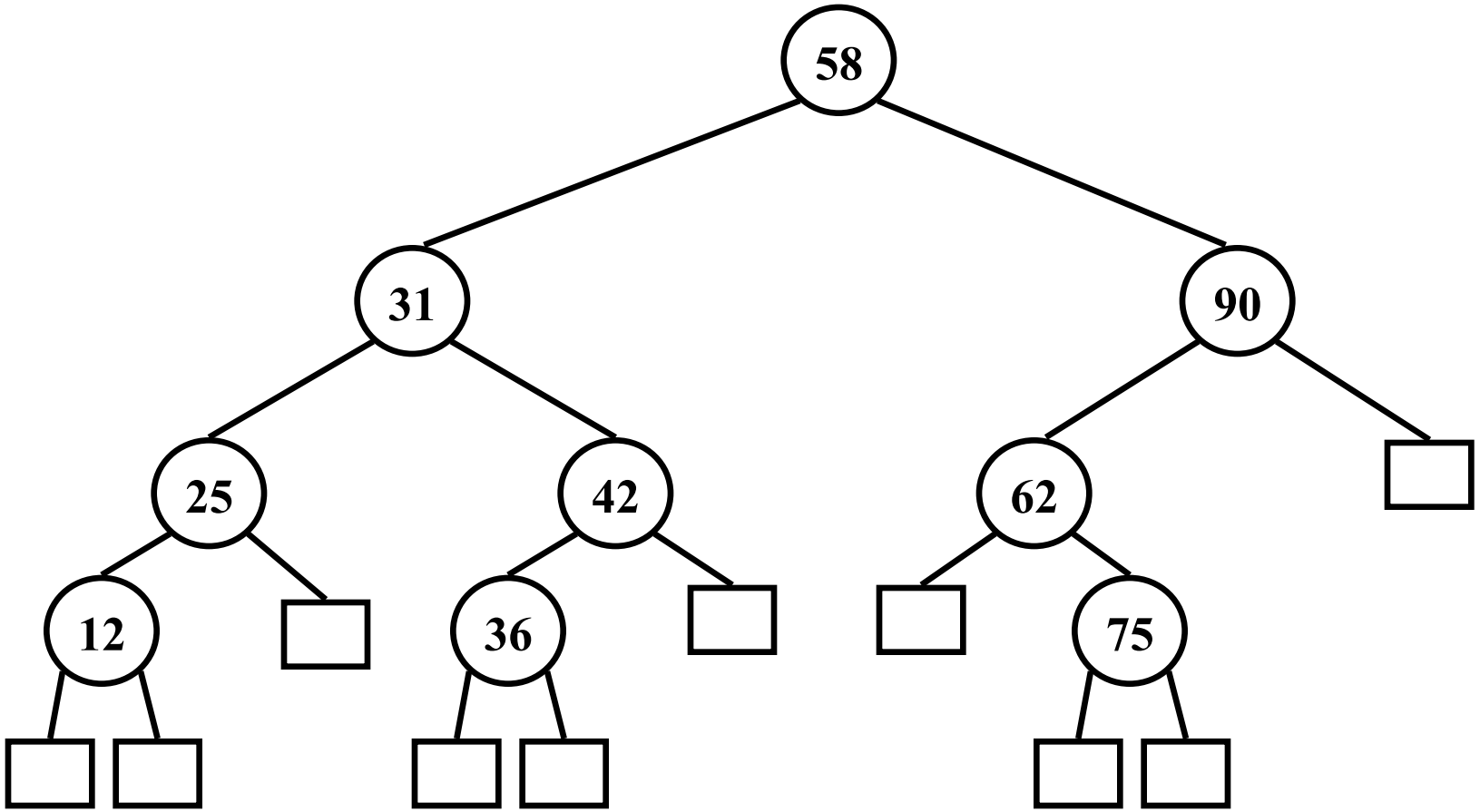
Algorithm TreeSearch(k, v)

Input: A search tree k, and a node v of a binary search tree T.

Output: A node w of the subtree $T(v)$ of T rooted at v, such that either w is an internal node storing key or w is the external node where an item with key k would belong if it existed.

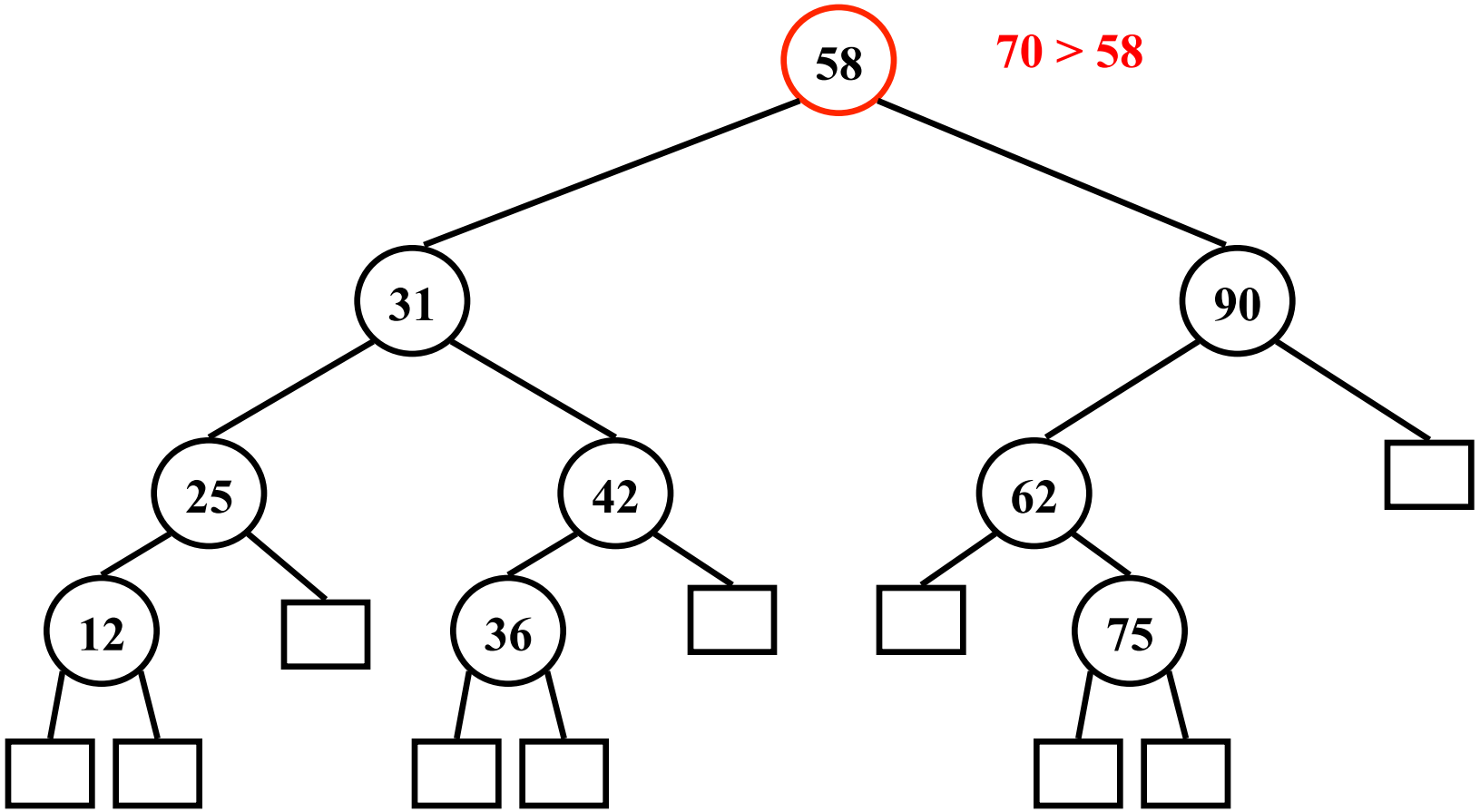
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



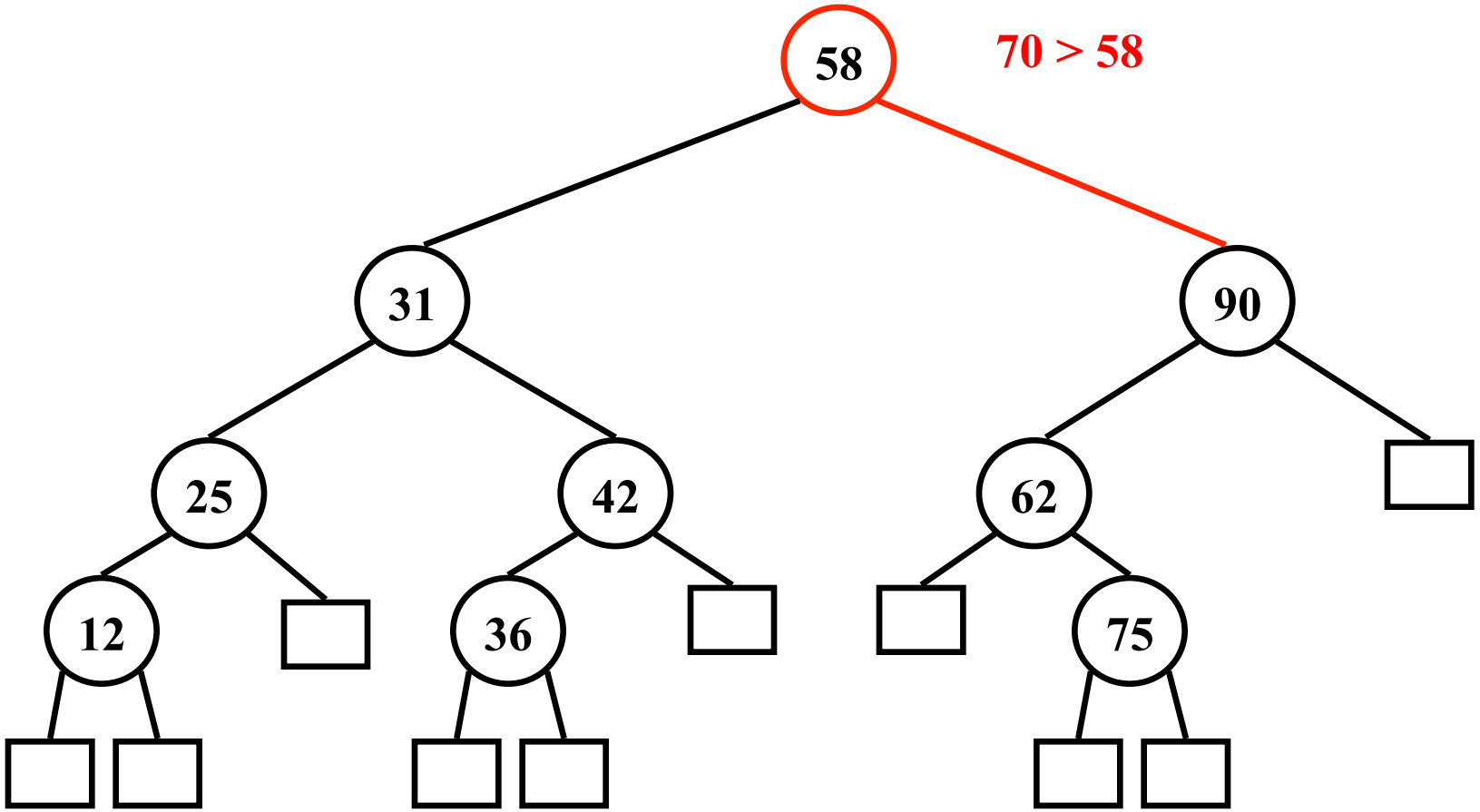
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



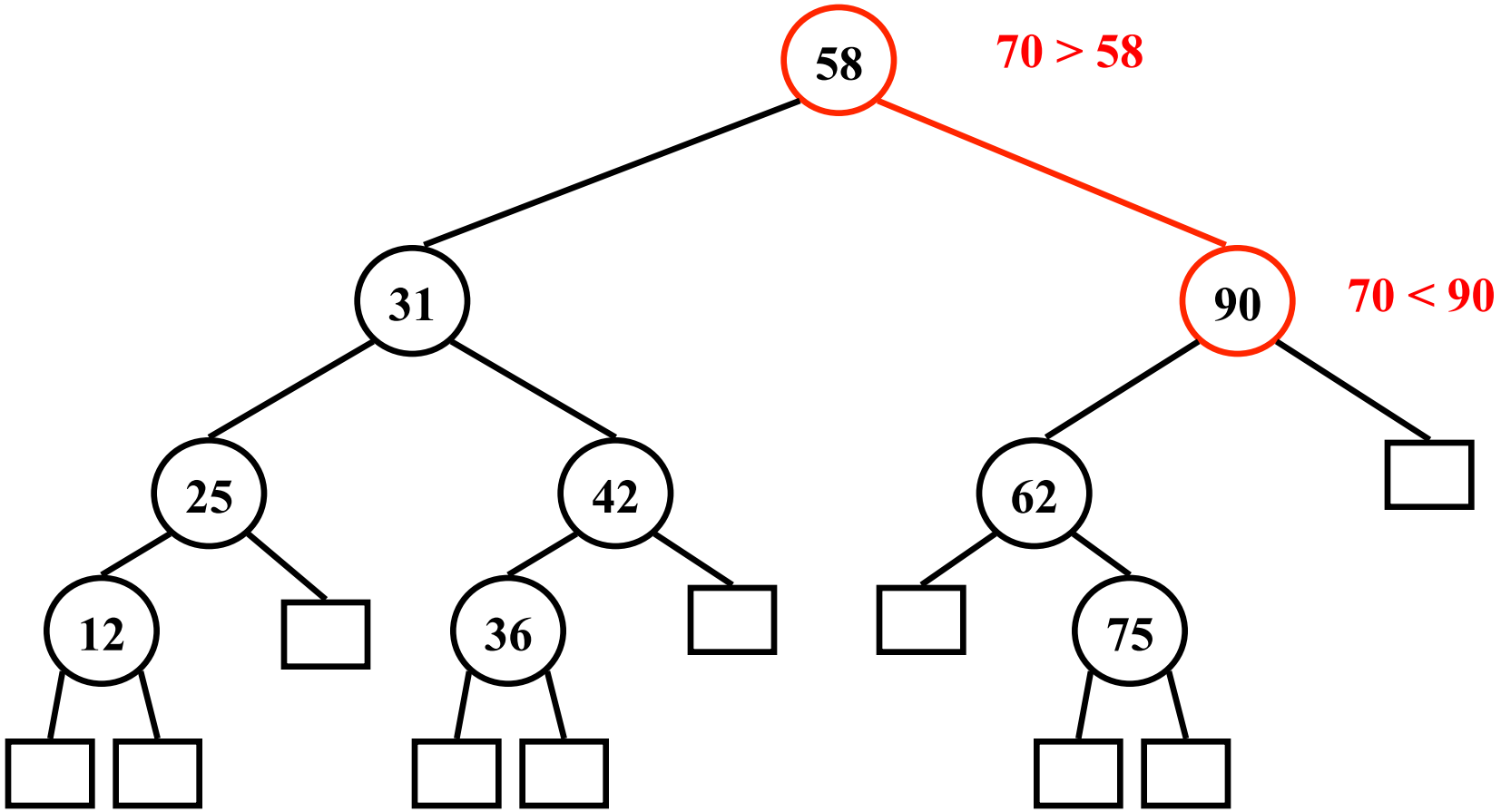
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



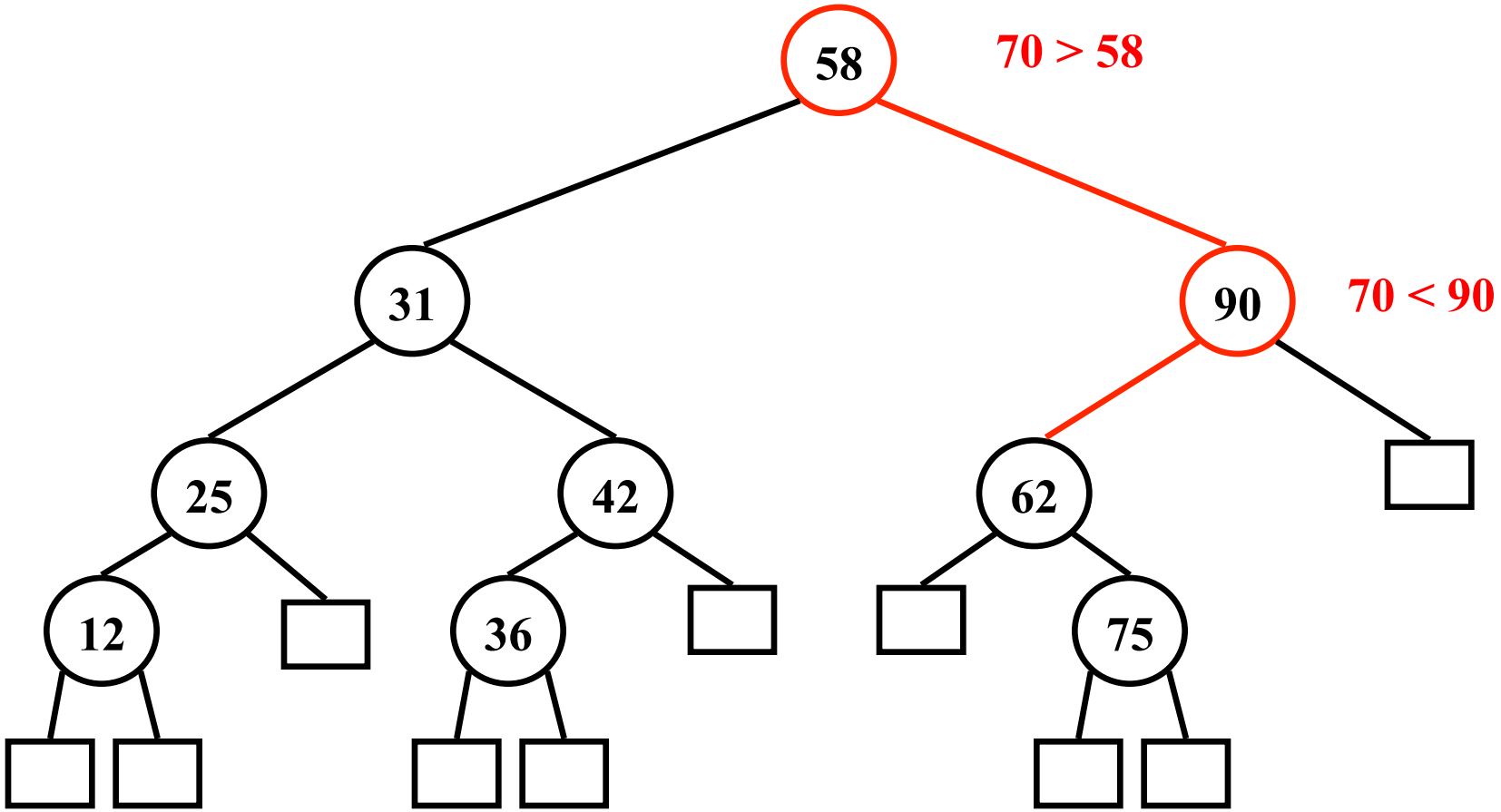
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



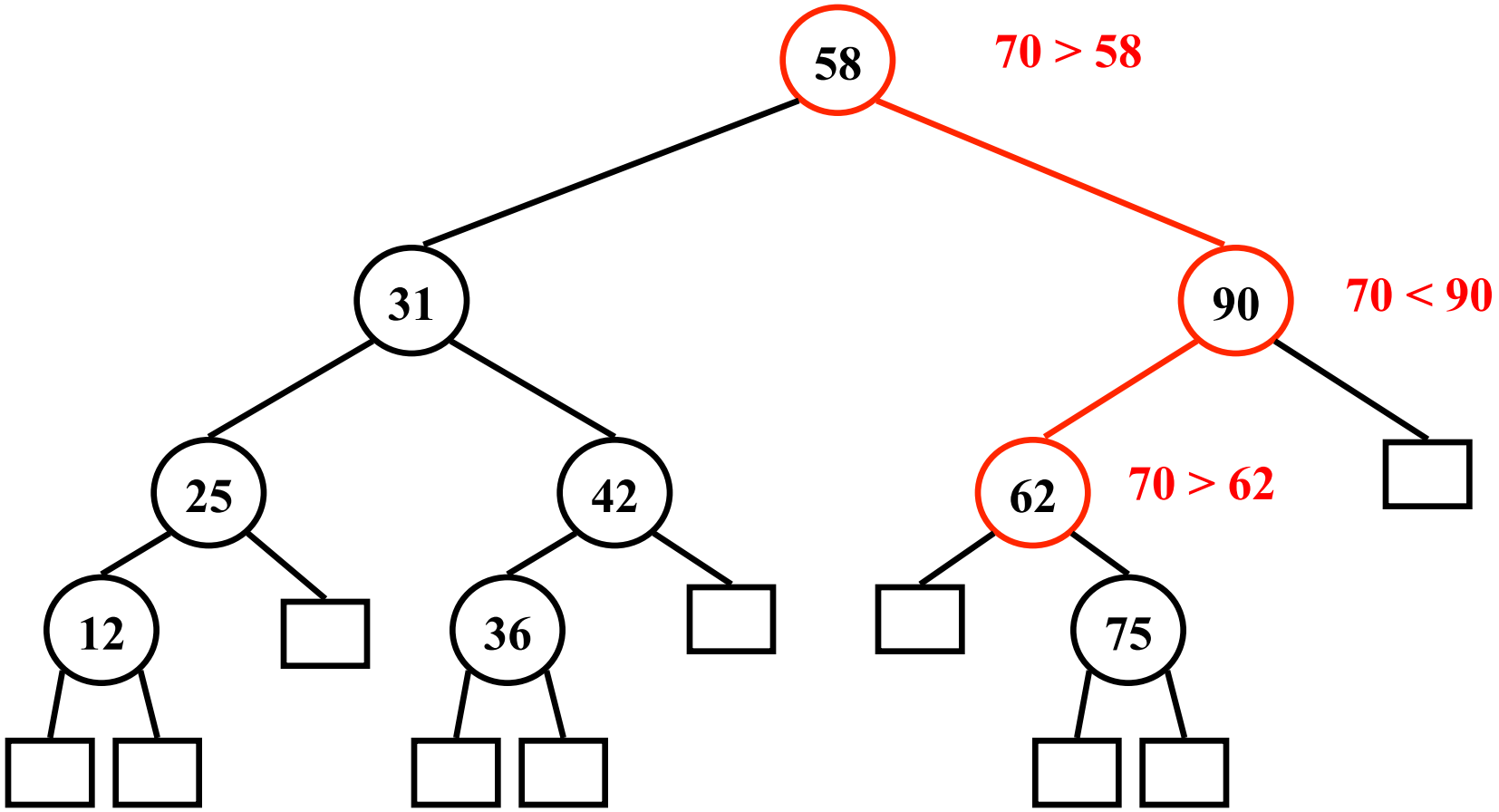
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



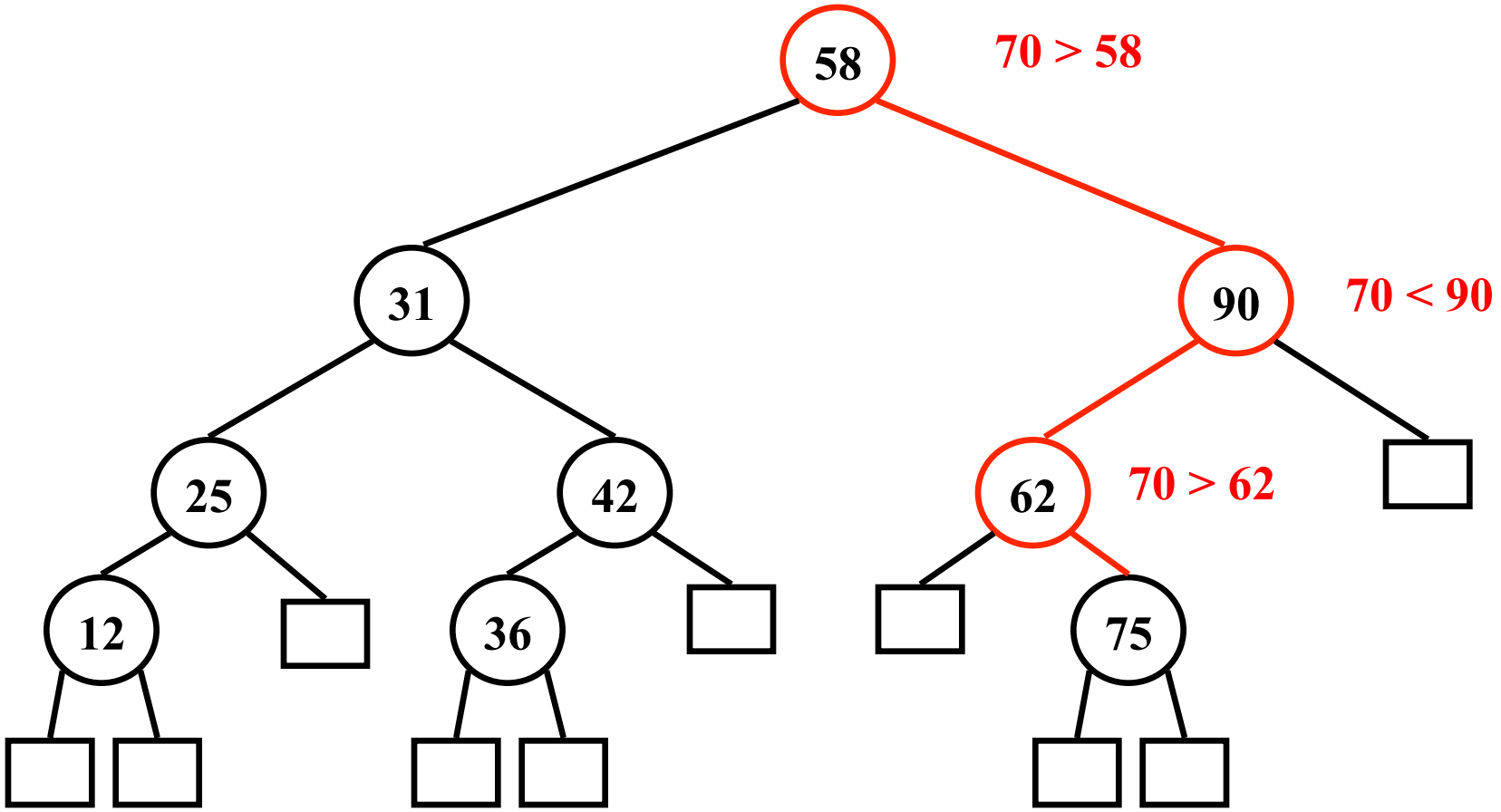
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



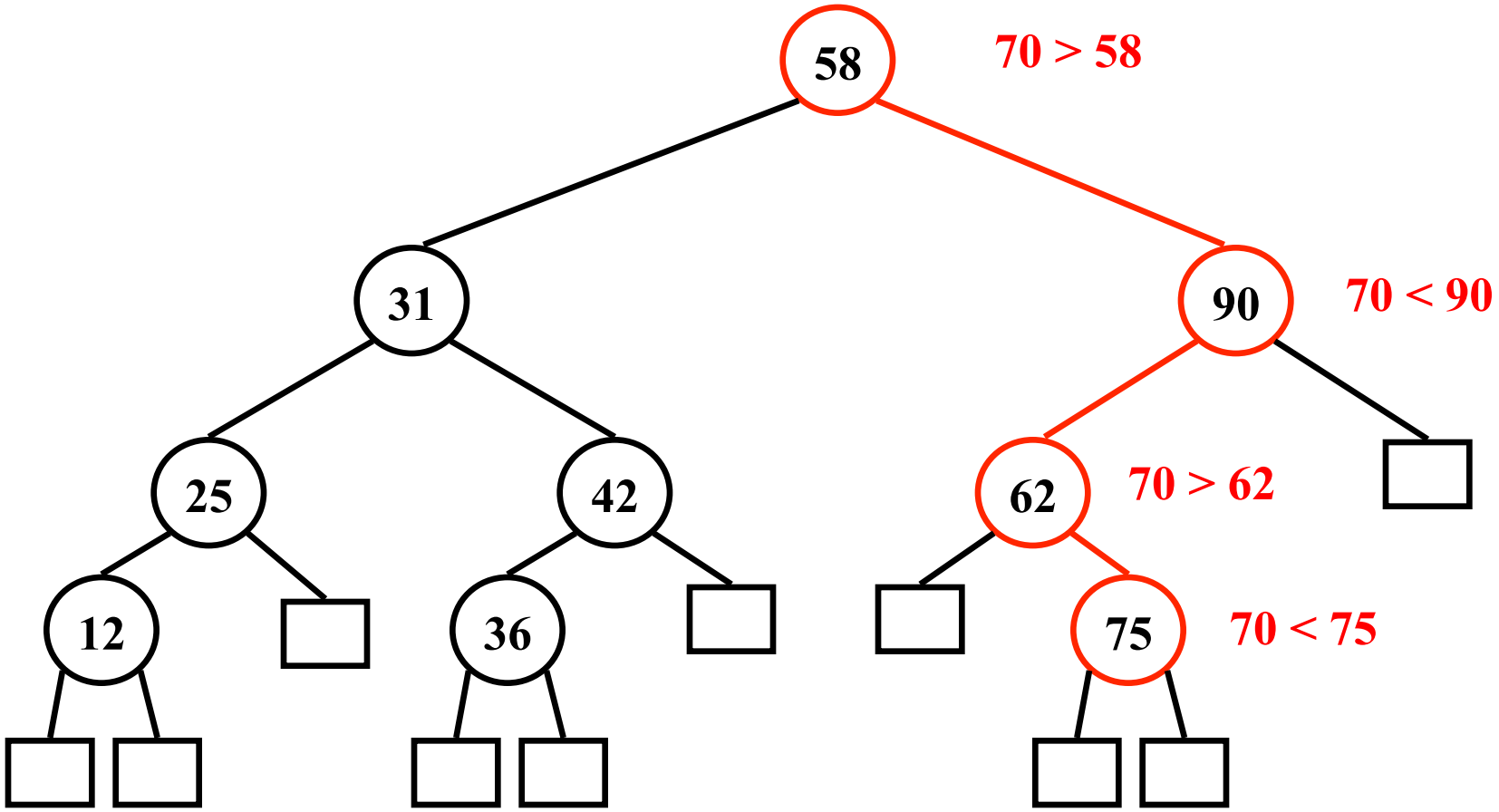
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



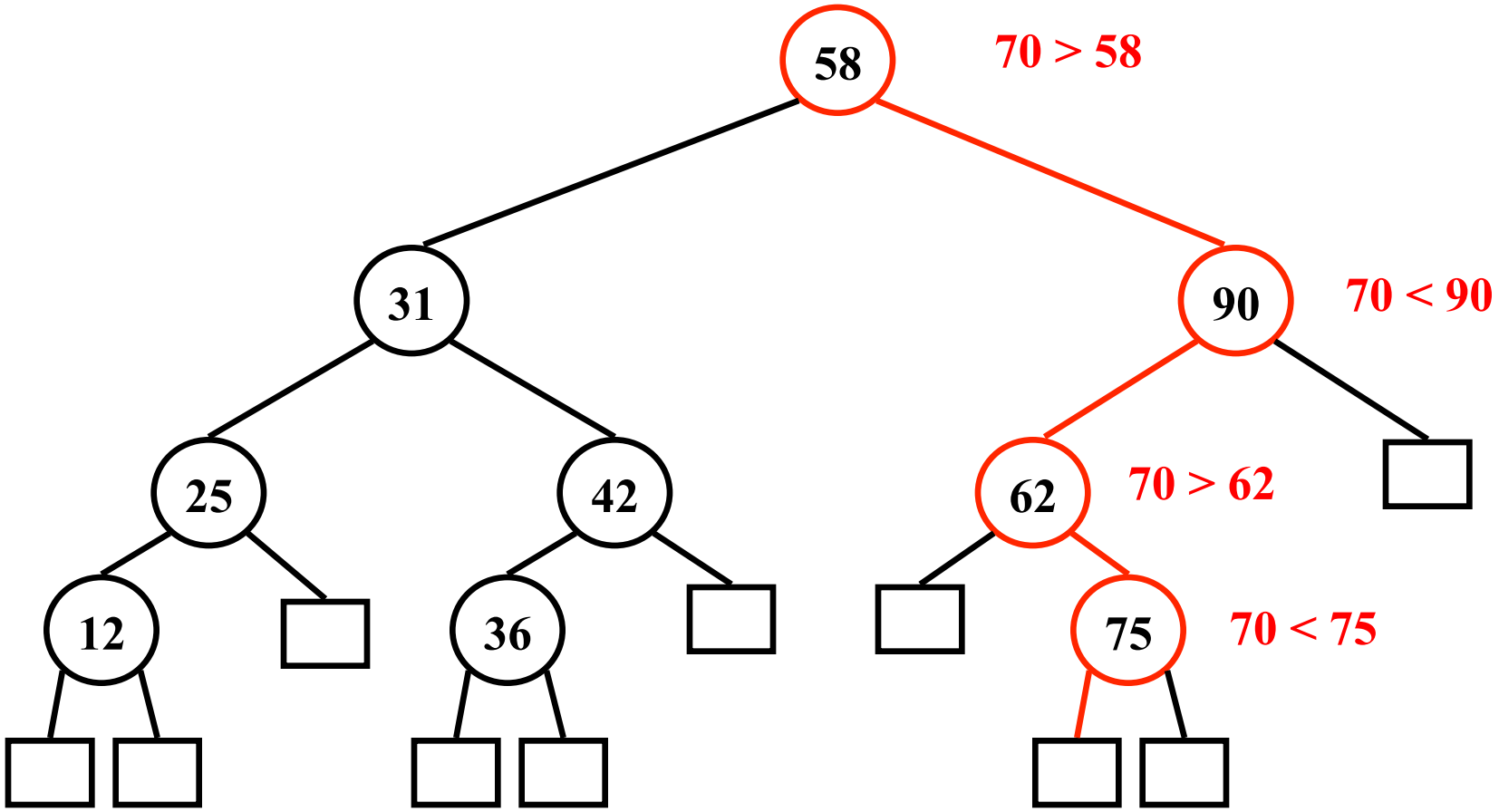
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



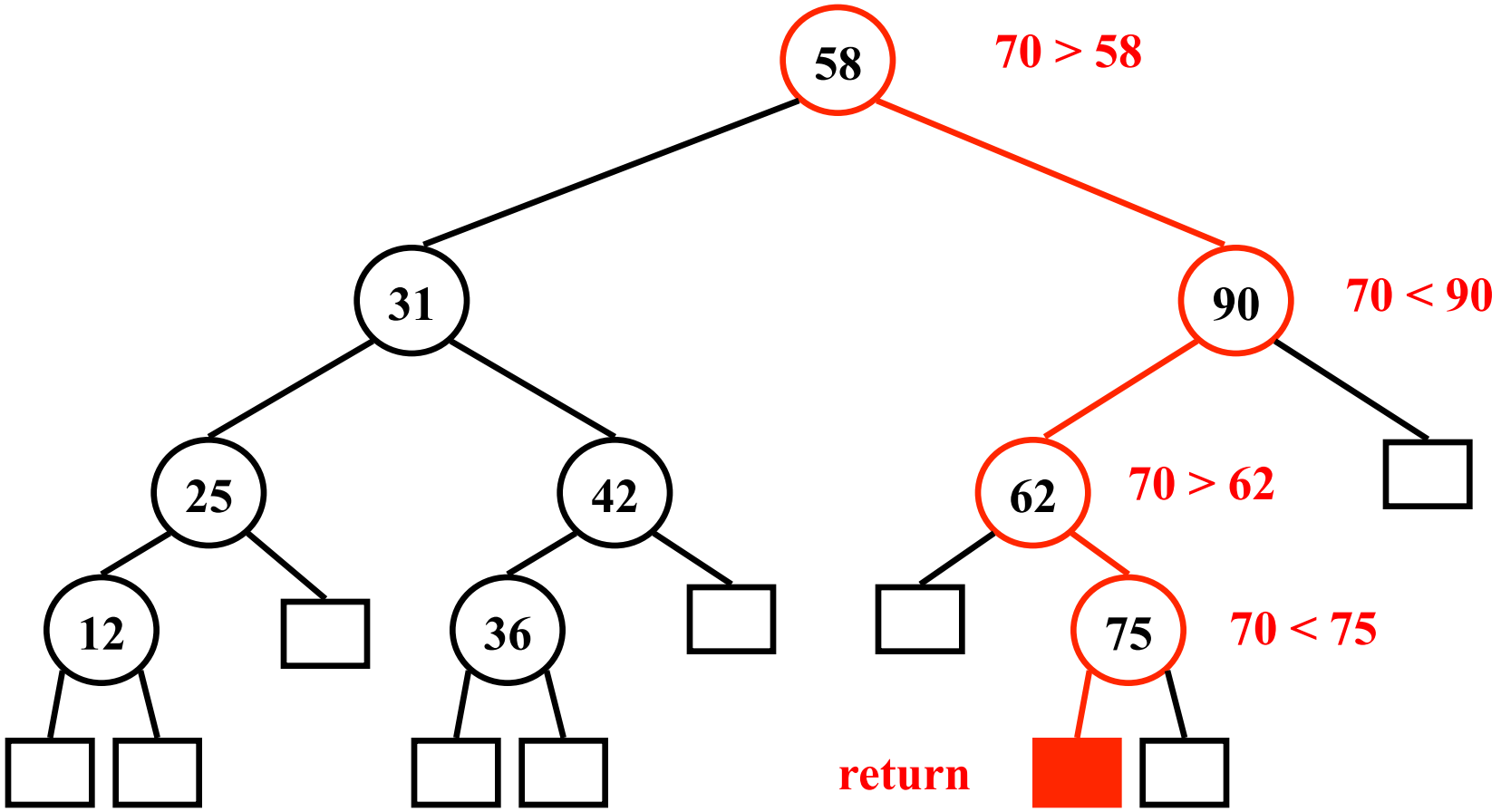
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



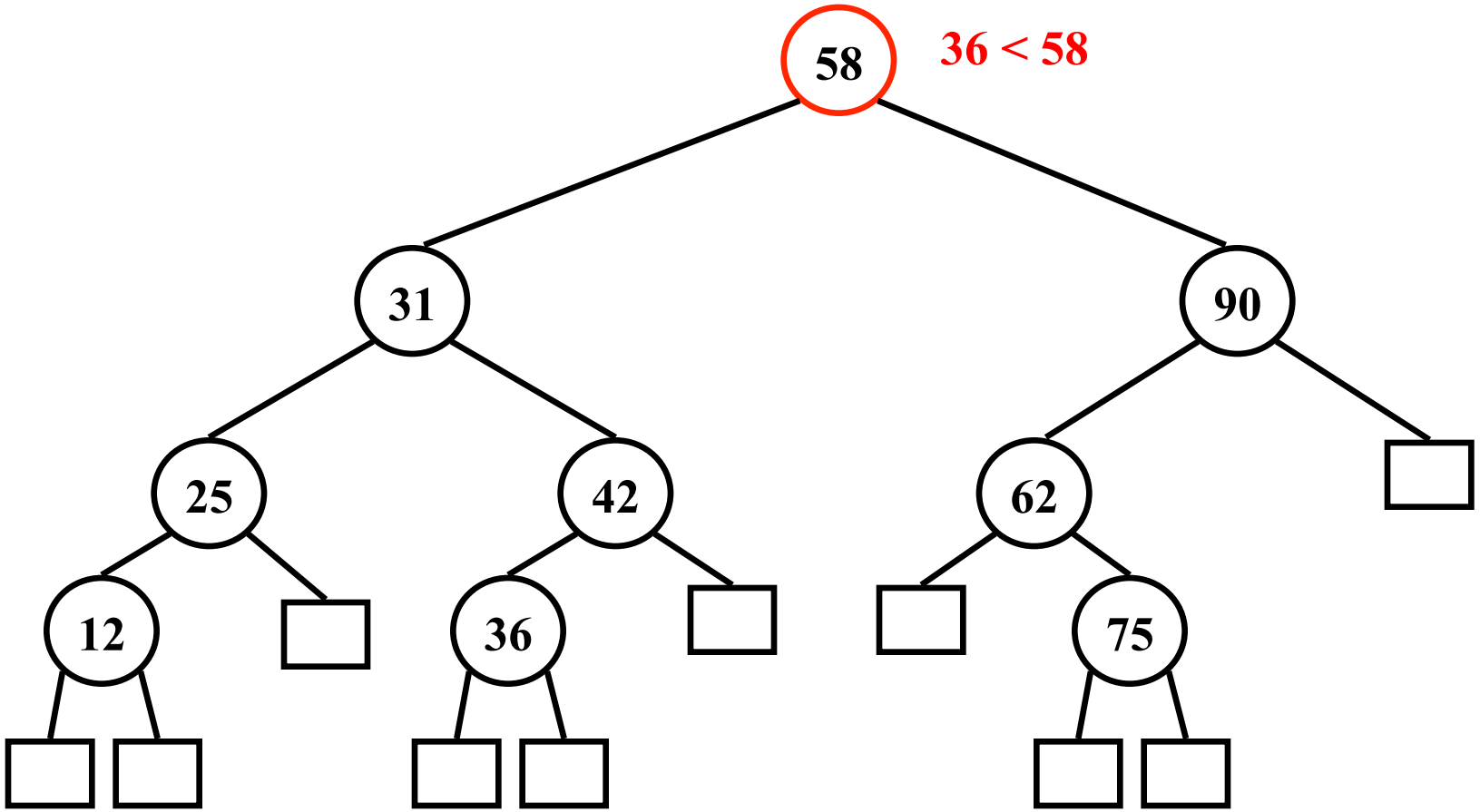
Search an element in a Binary Search Tree

findElement(70): TreeSearch(70,*T*.root())



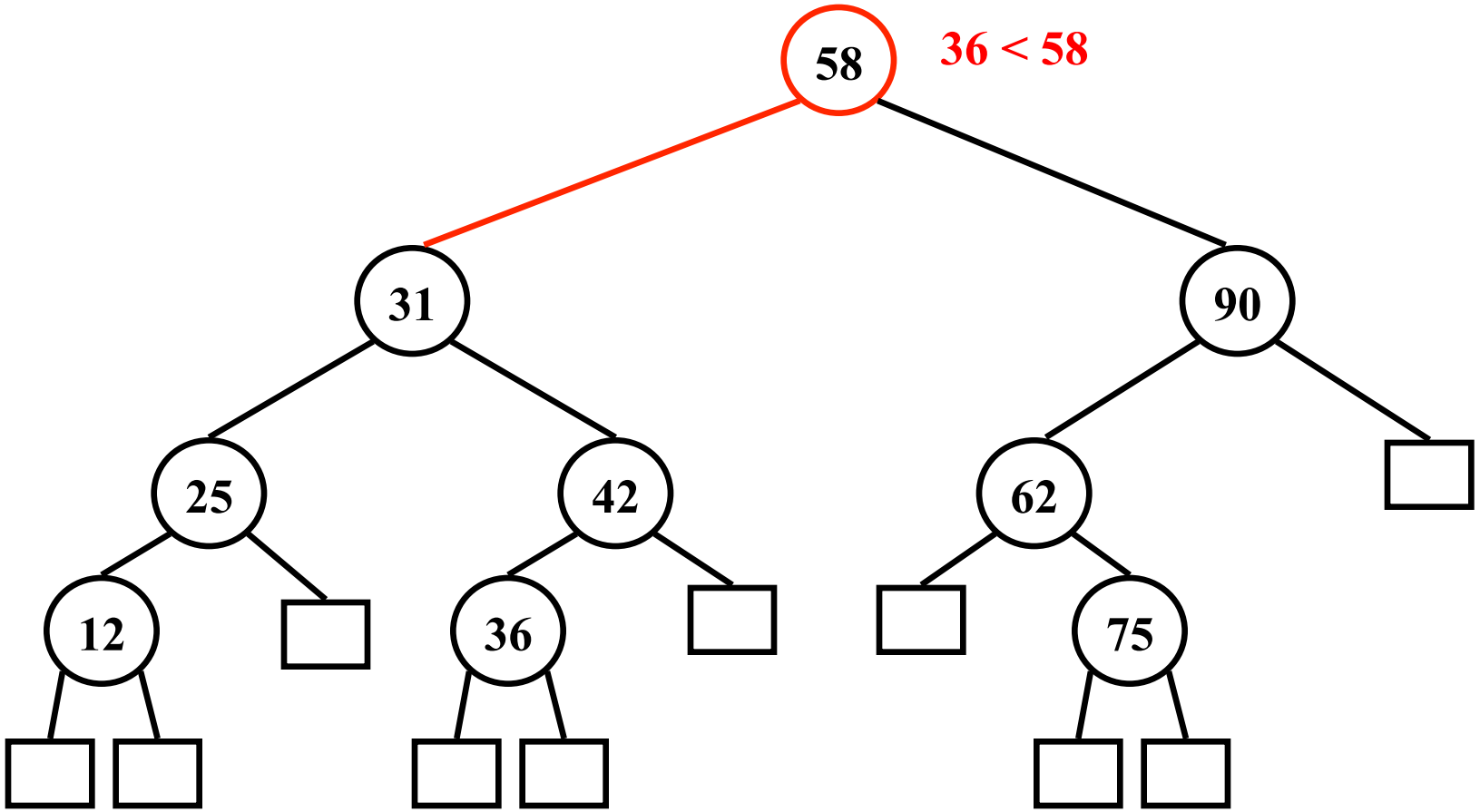
Search an element in a Binary Search Tree

findElement(36)



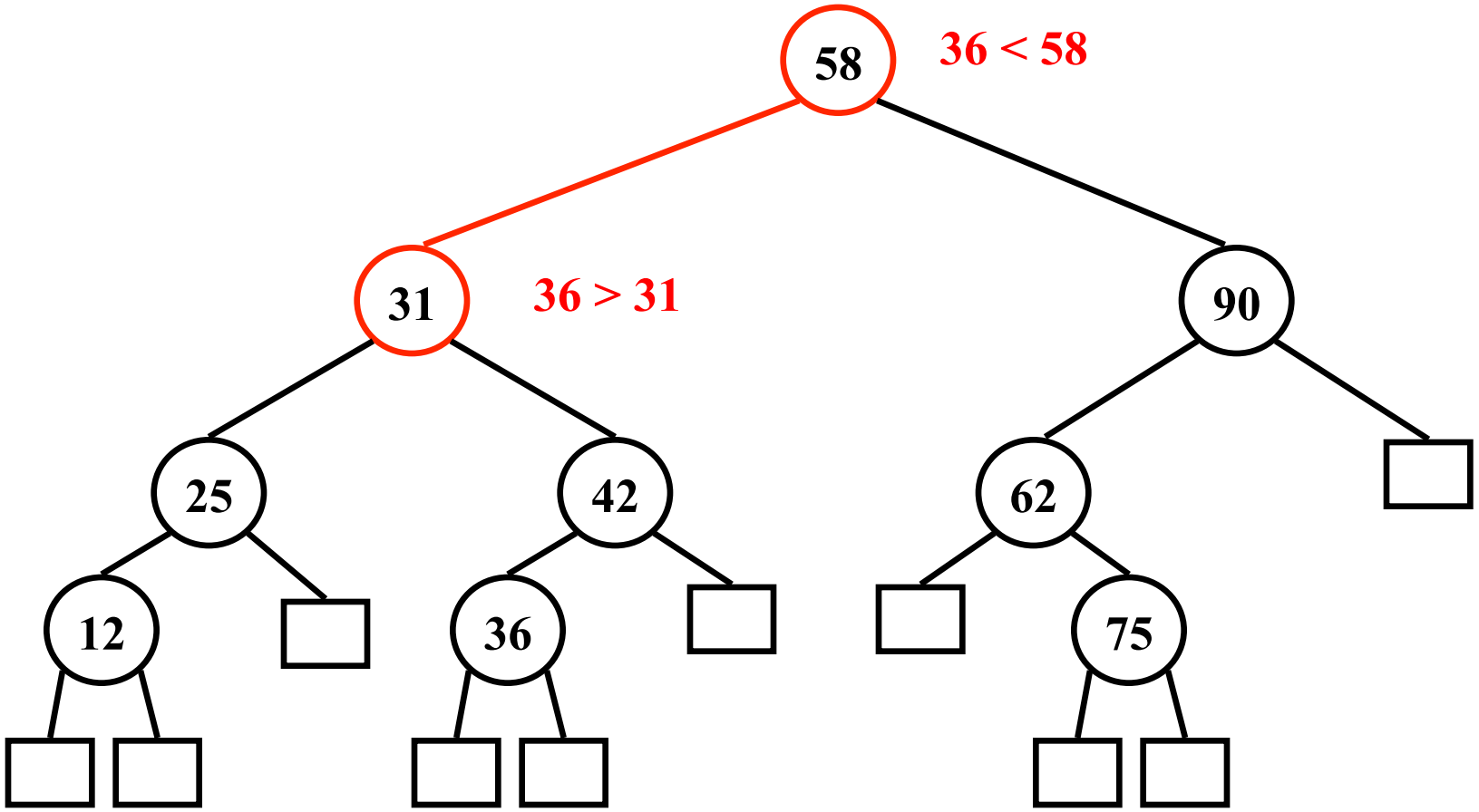
Search an element in a Binary Search Tree

findElement(36)



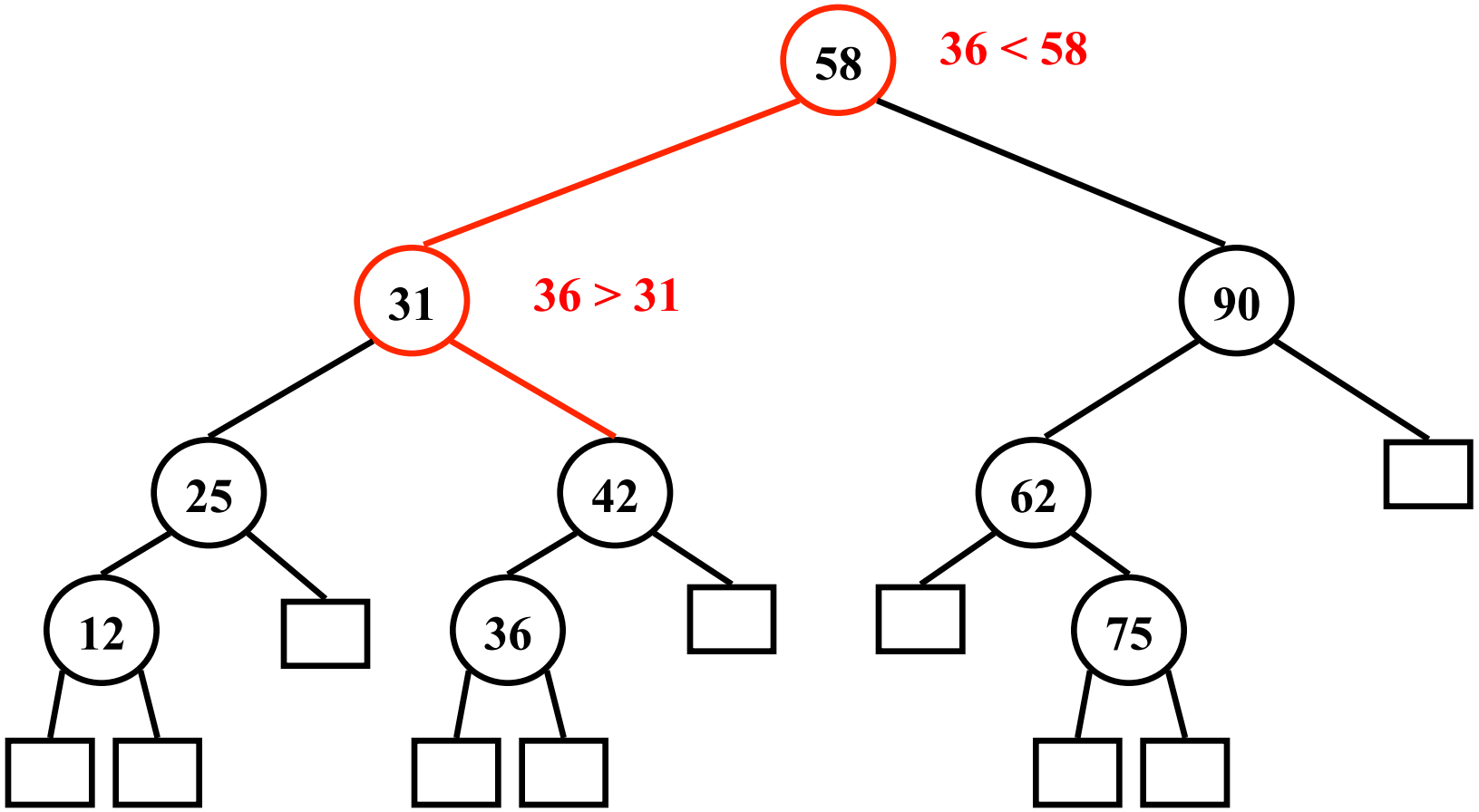
Search an element in a Binary Search Tree

findElement(36)



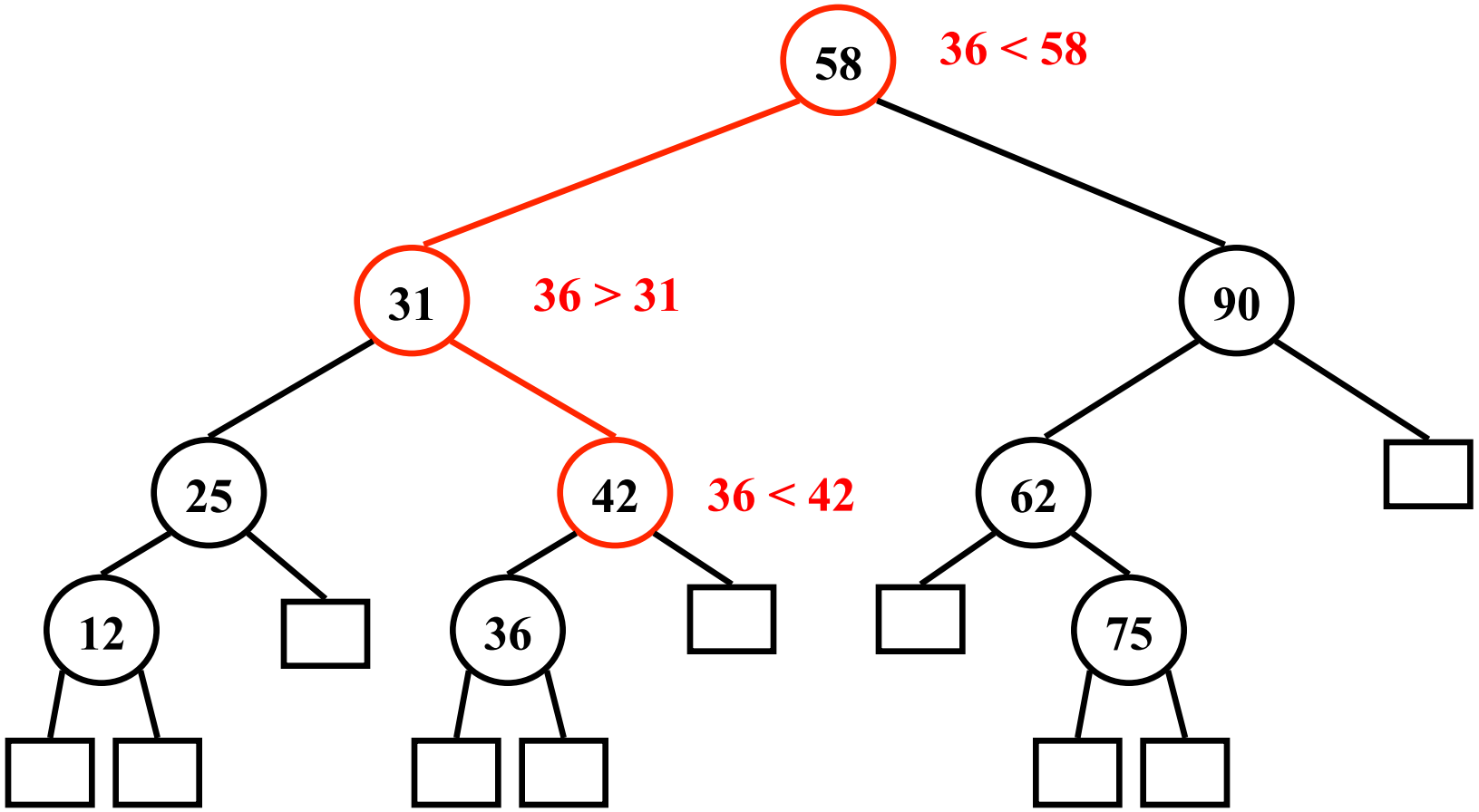
Search an element in a Binary Search Tree

findElement(36)



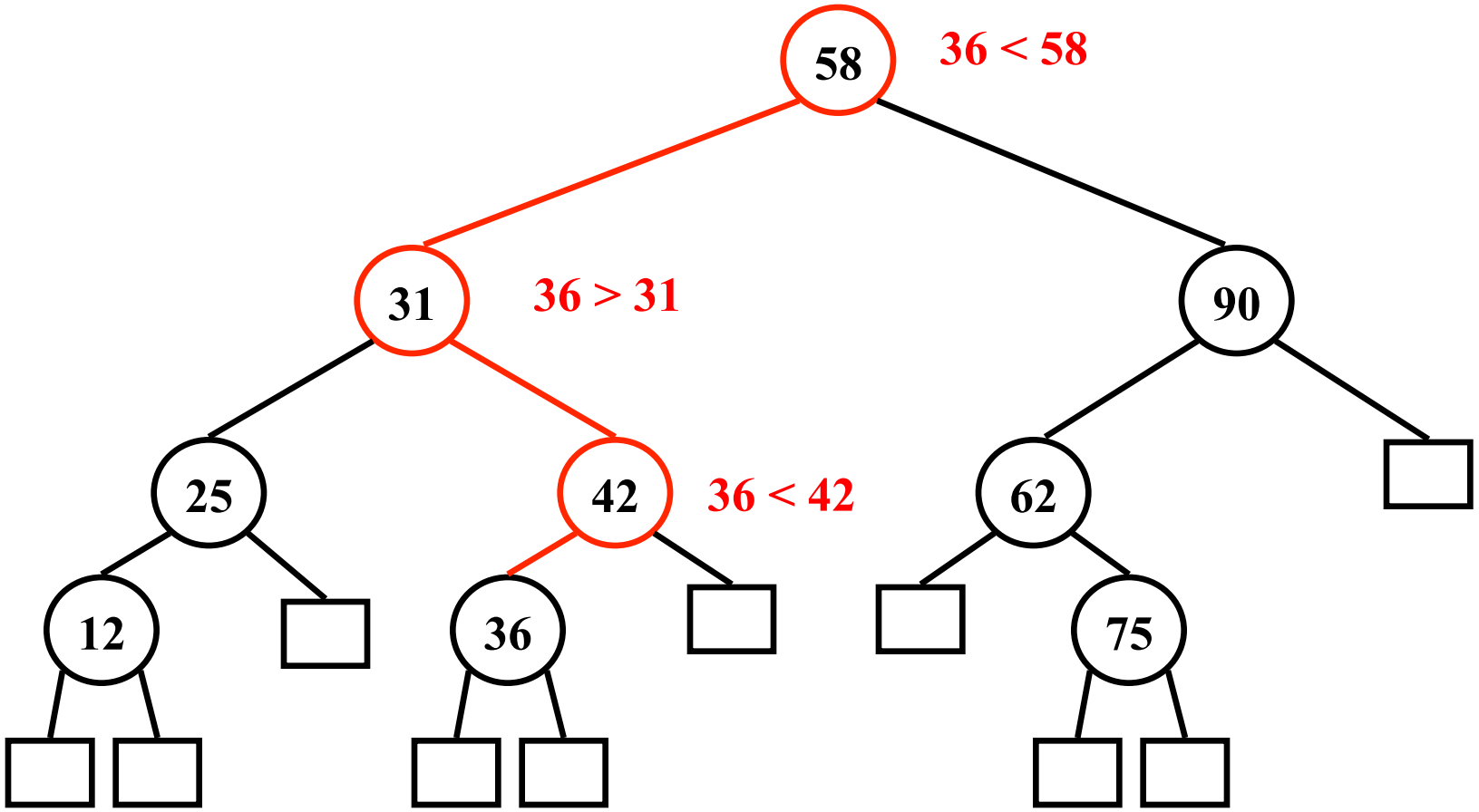
Search an element in a Binary Search Tree

findElement(36)



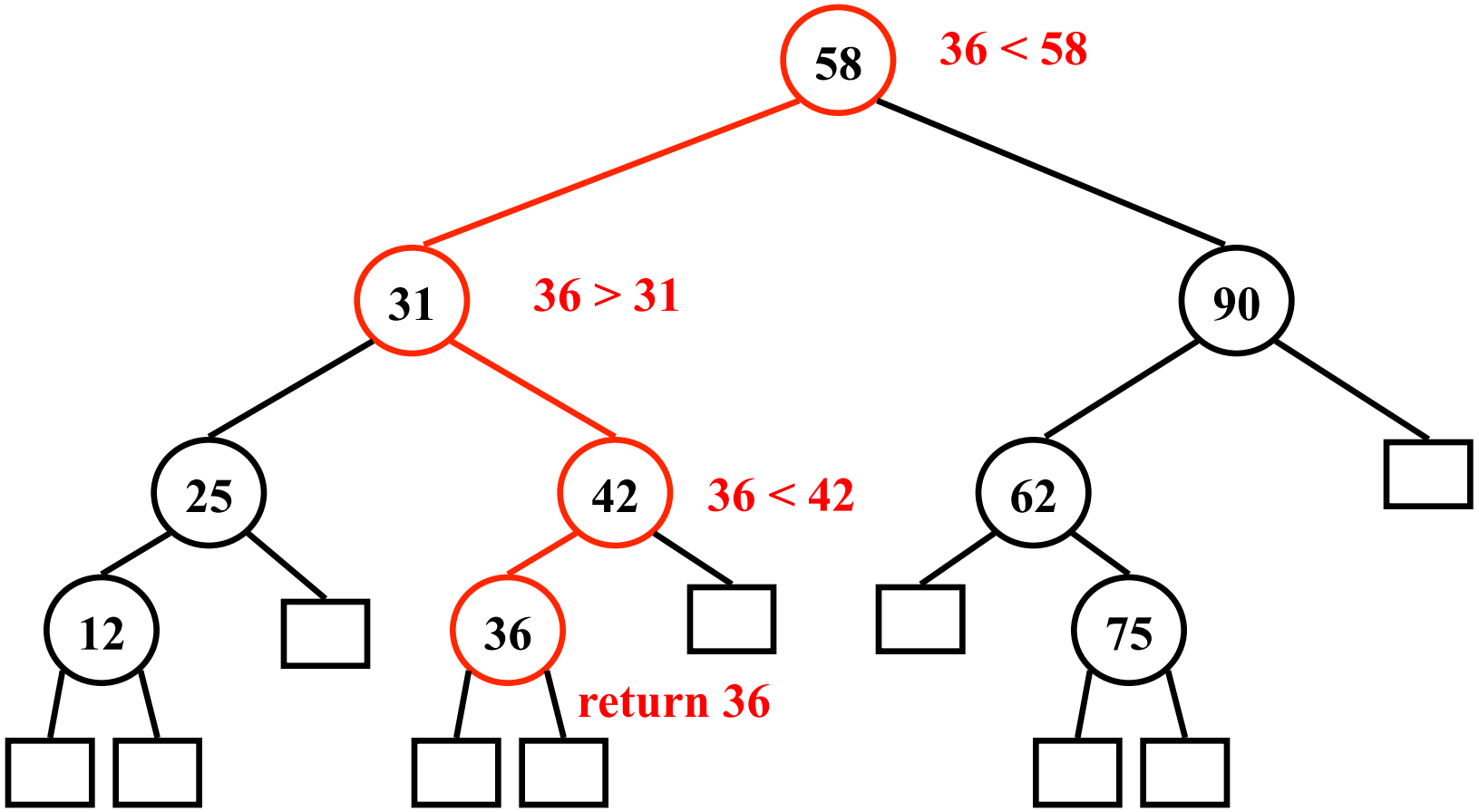
Search an element in a Binary Search Tree

findElement(36)



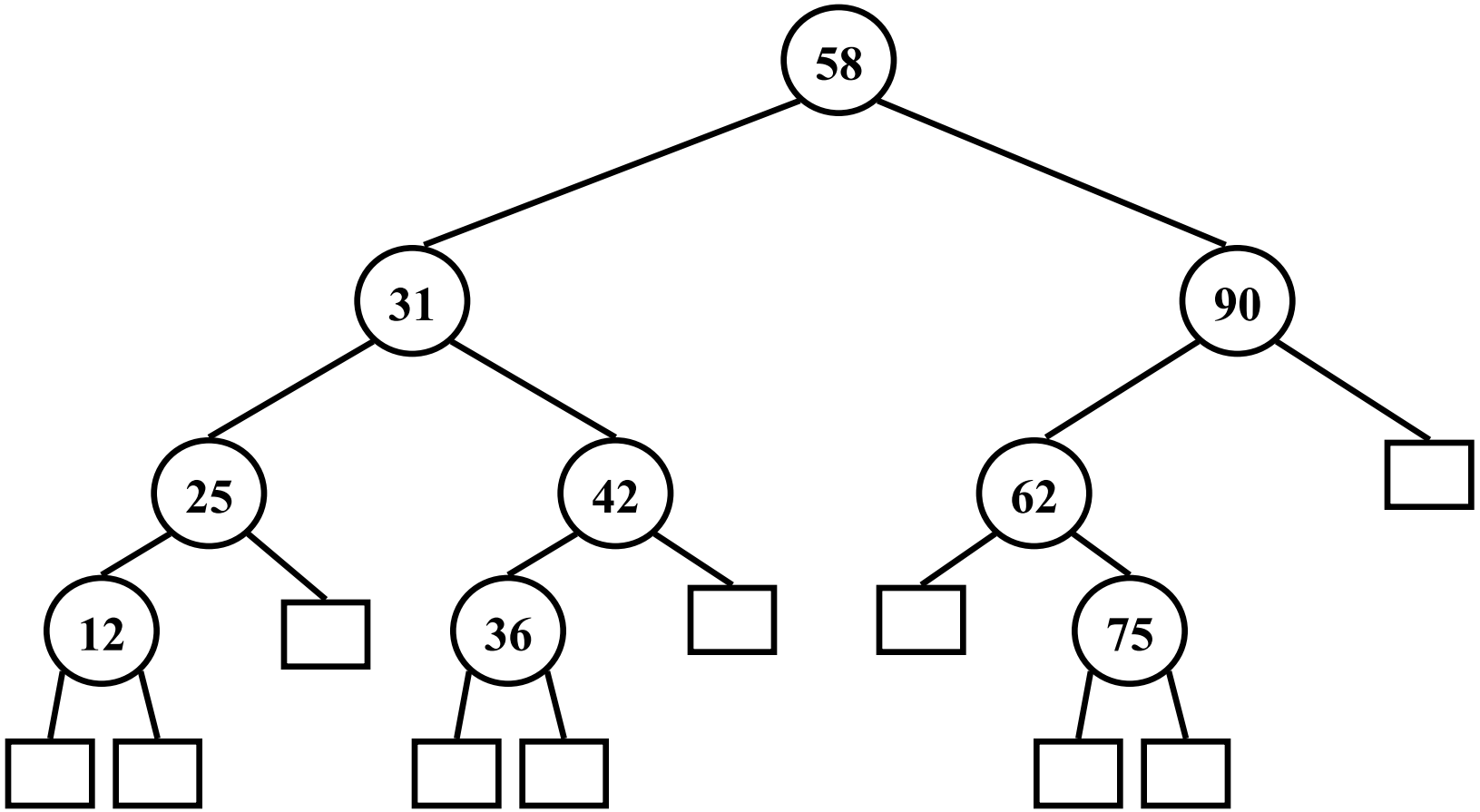
Search an element in a Binary Search Tree

findElement(36)



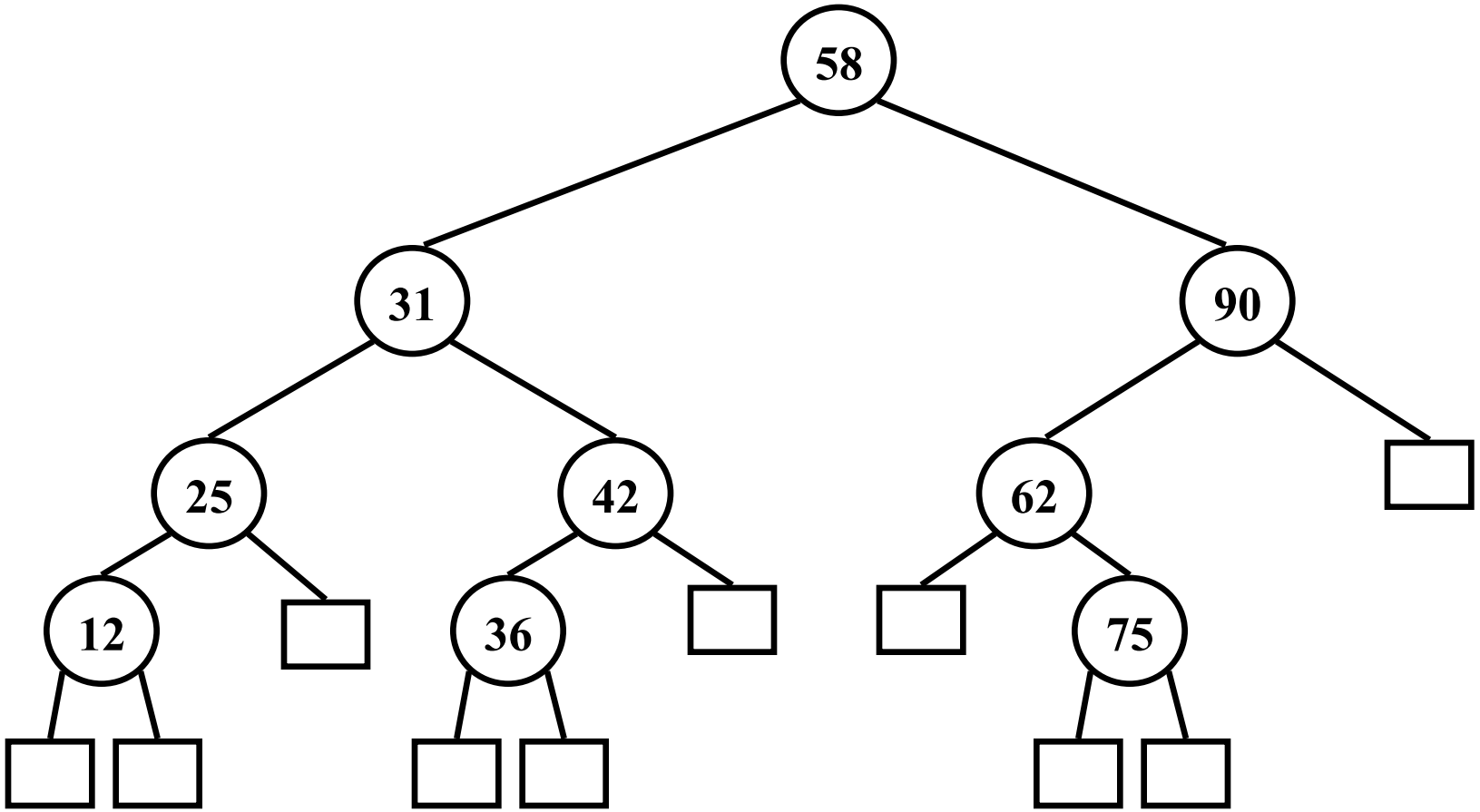
Insertion in a Binary Search Tree

InsertItem(78,*e*)



Insertion in a Binary Search Tree

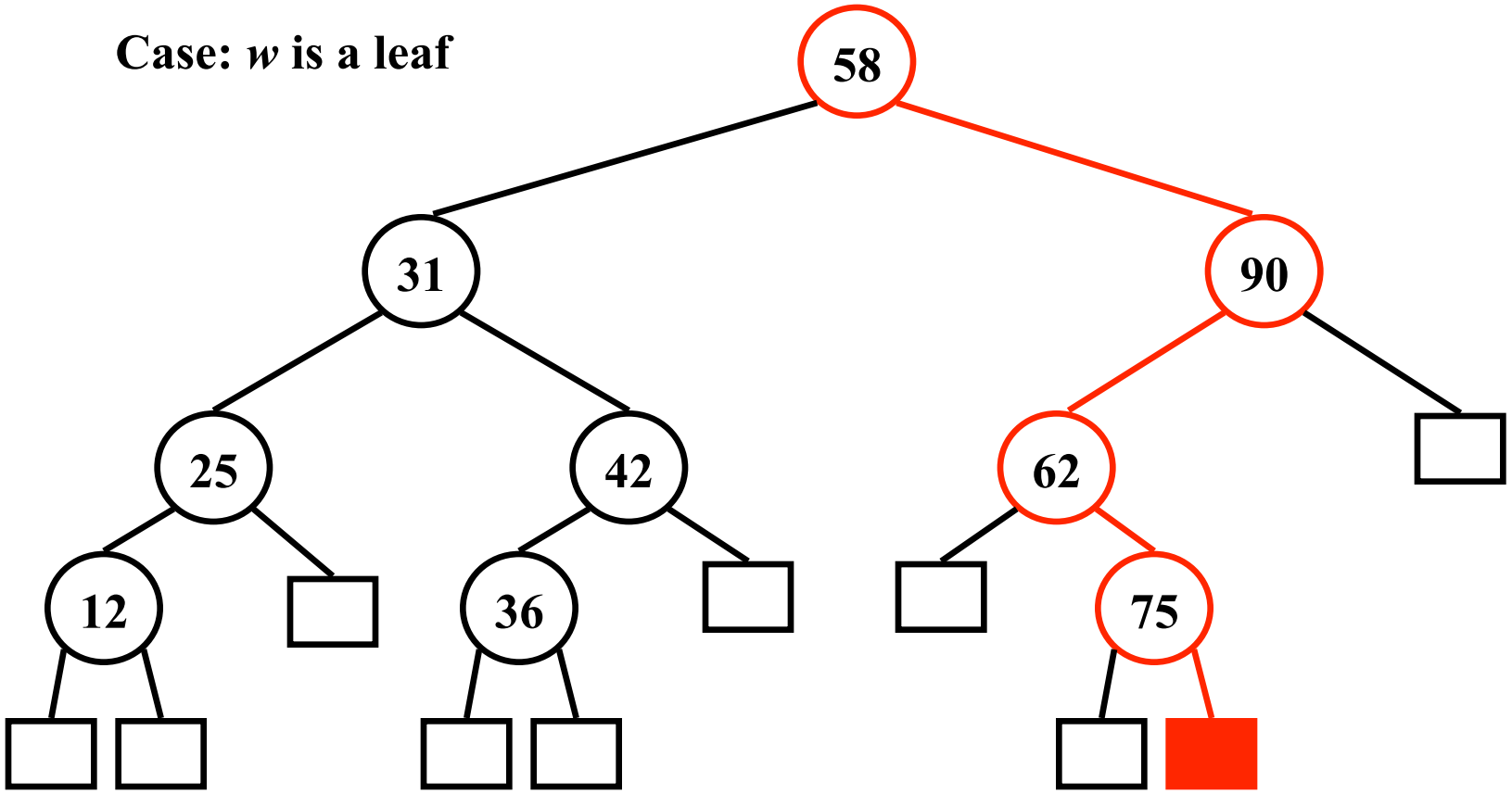
InsertItem(78,*e*): *w* ← TreeSearch(78,*T*.root())



Insertion in a Binary Search Tree

InsertItem(78,*e*): *w* ← TreeSearch(78,*T*.root())

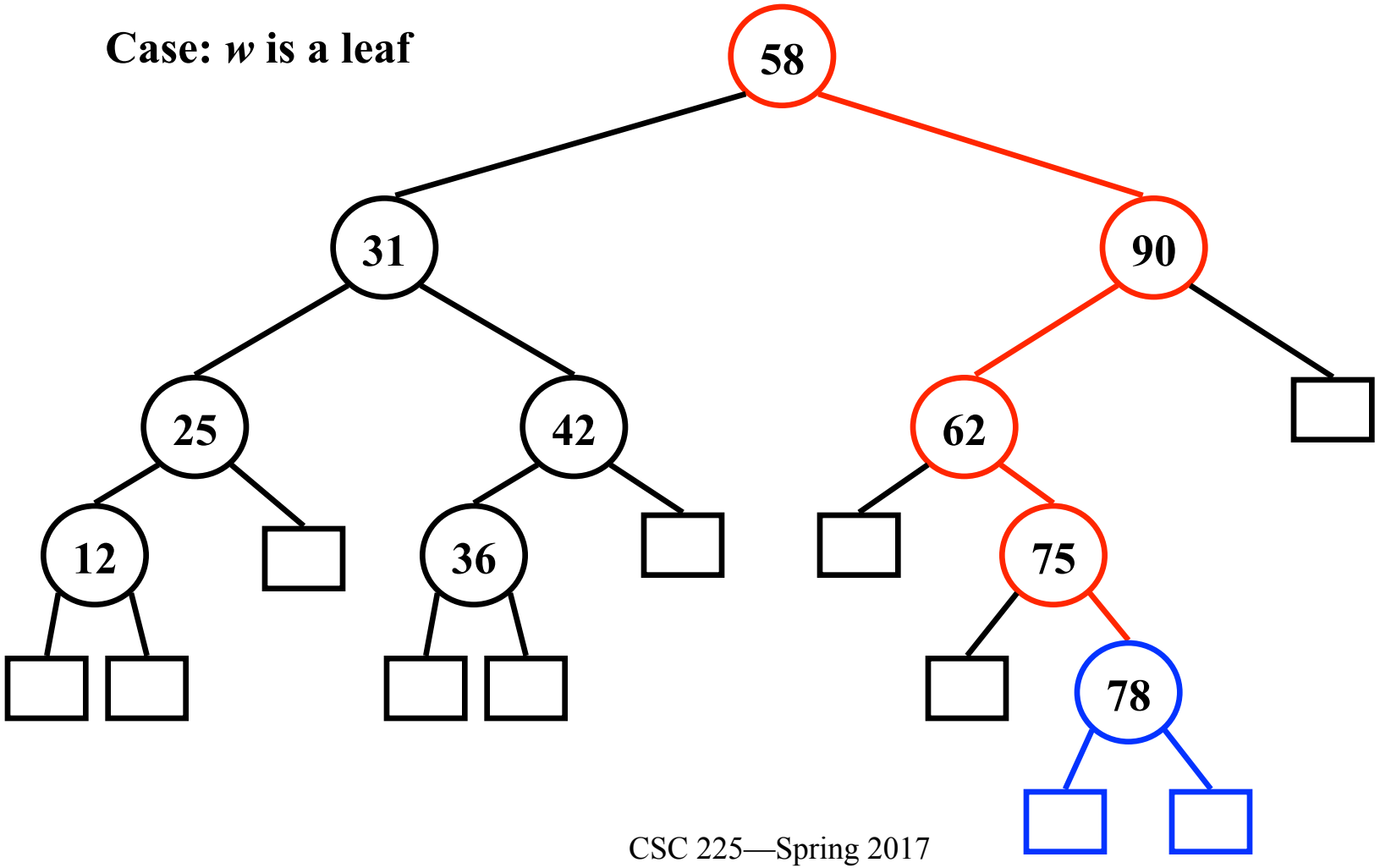
Case: *w* is a leaf



Insertion in a Binary Search Tree

InsertItem(78,*e*): *w* ← TreeSearch(78,*T*.root())

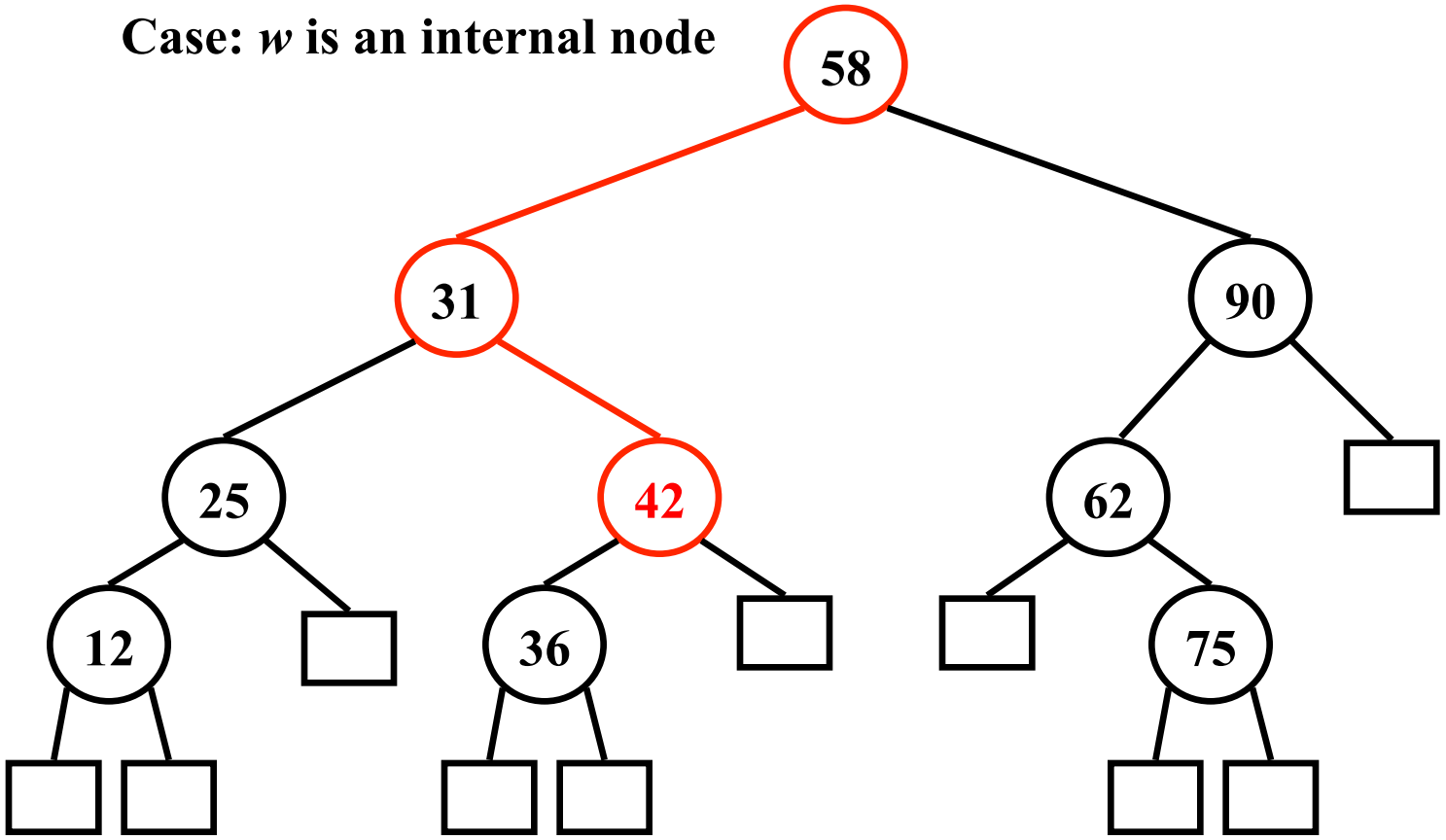
Case: *w* is a leaf



Insertion in a Binary Search Tree

InsertItem(42,*e*): *w* ← TreeSearch(42,*T*.root())

Case: *w* is an internal node

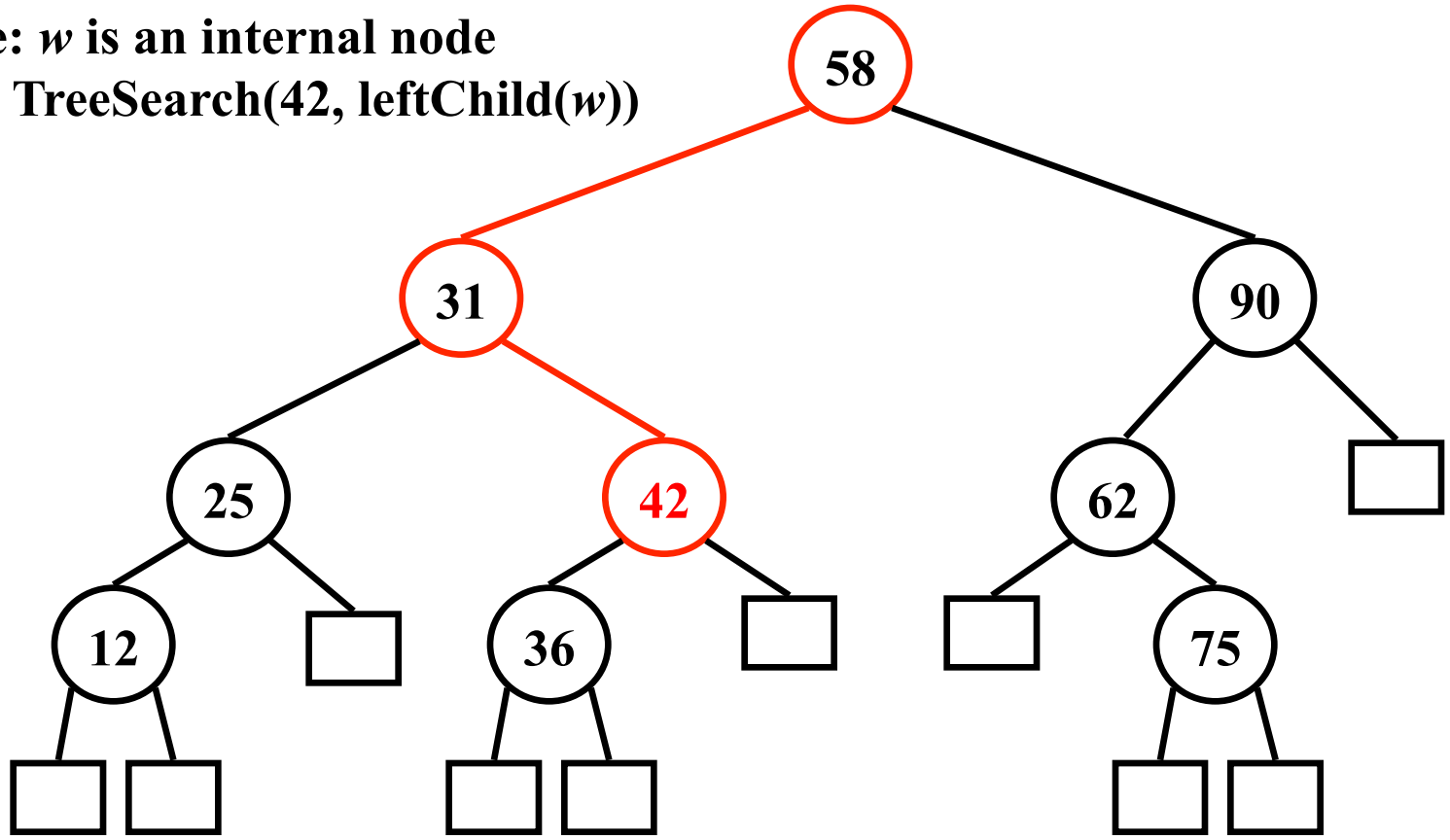


Insertion in a Binary Search Tree

InsertItem(42,*e*): $w \leftarrow$ TreeSearch(42,*T*.root())

Case: *w* is an internal node

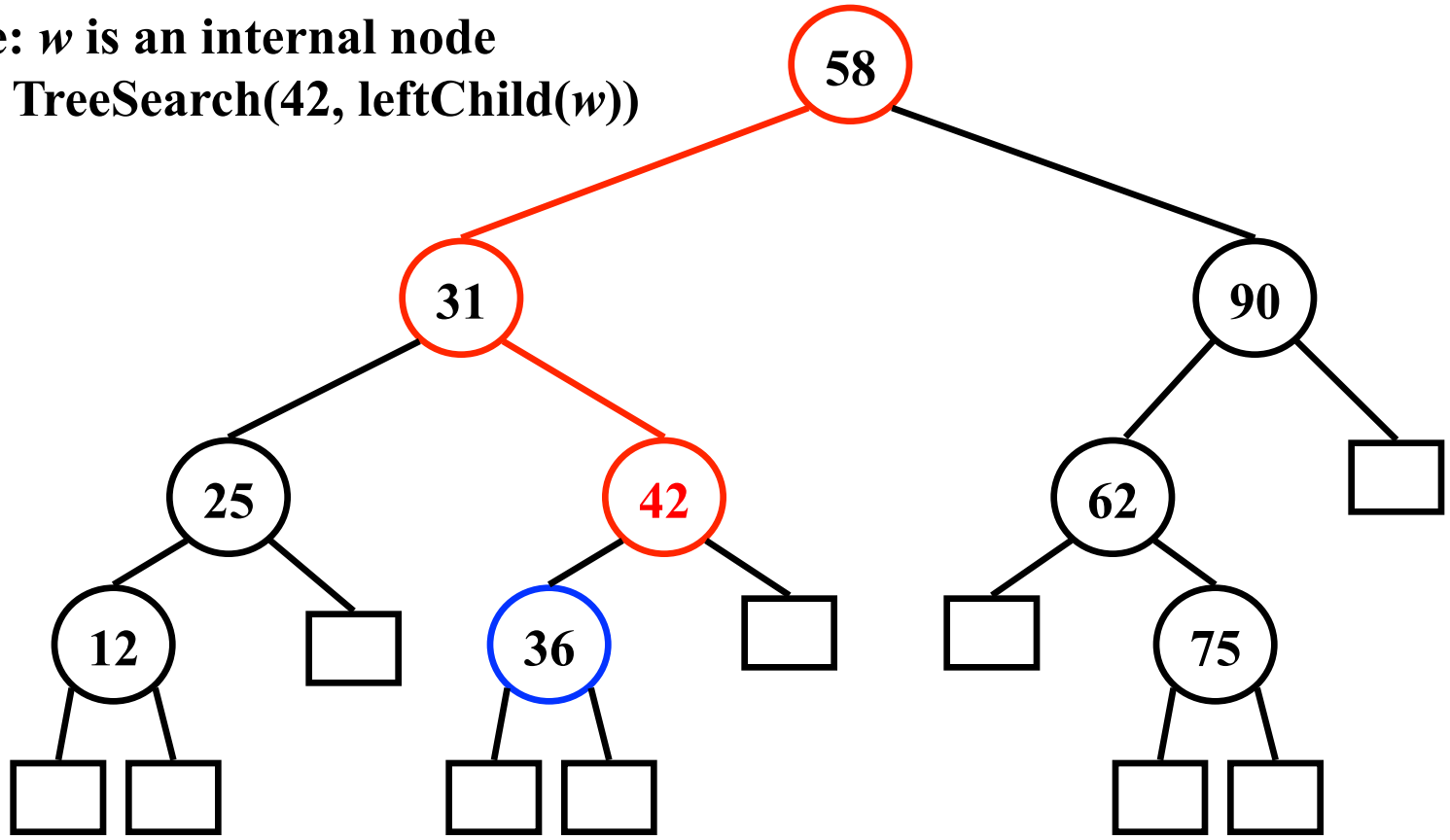
***w* ← TreeSearch(42, leftChild(*w*))**



Insertion in a Binary Search Tree

InsertItem(42,*e*): *w* ← TreeSearch(42,*T*.root())

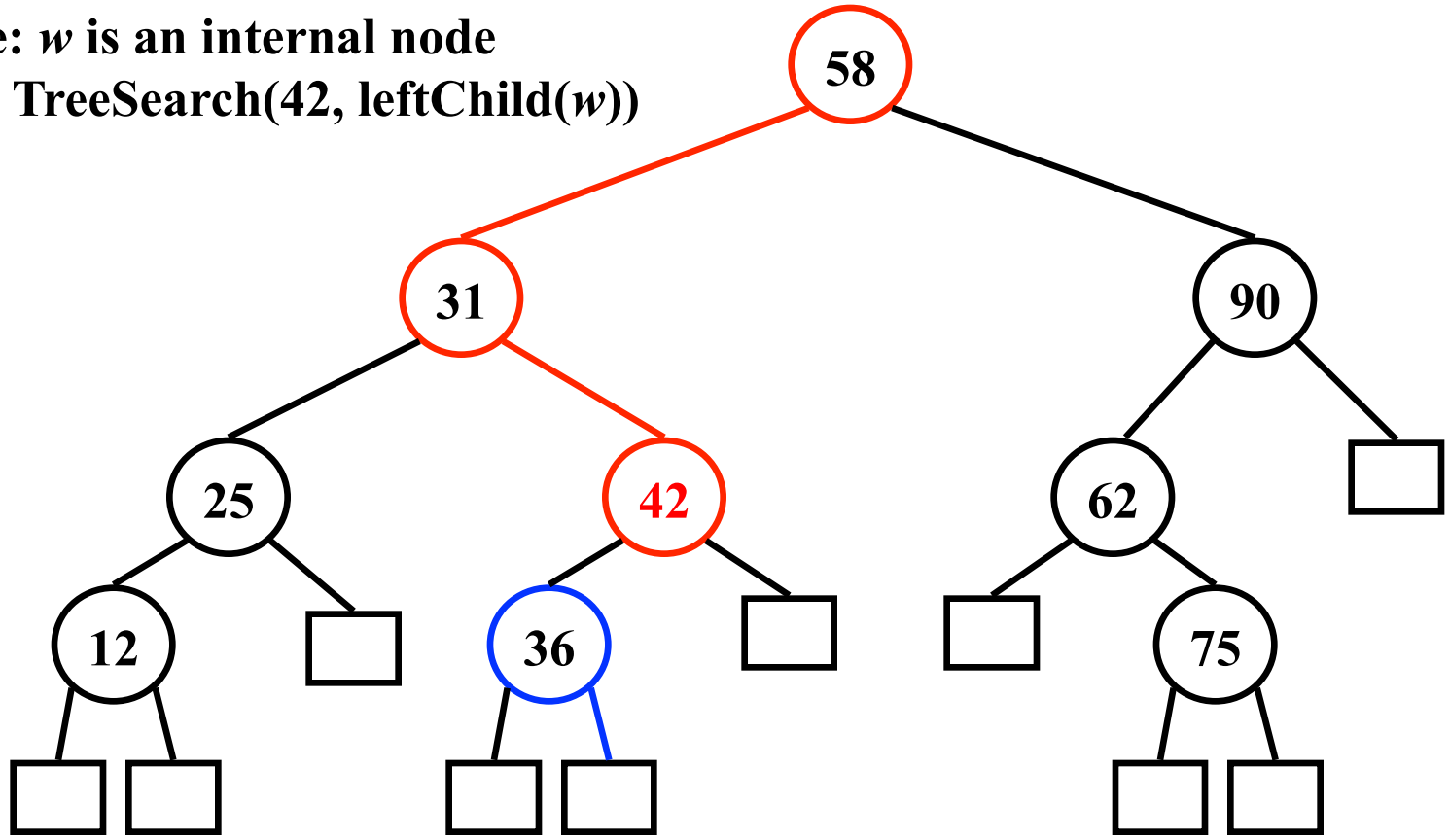
Case: *w* is an internal node
***w* ← TreeSearch(42, leftChild(*w*))**



Insertion in a Binary Search Tree

InsertItem(42,*e*): *w* ← TreeSearch(42,*T*.root())

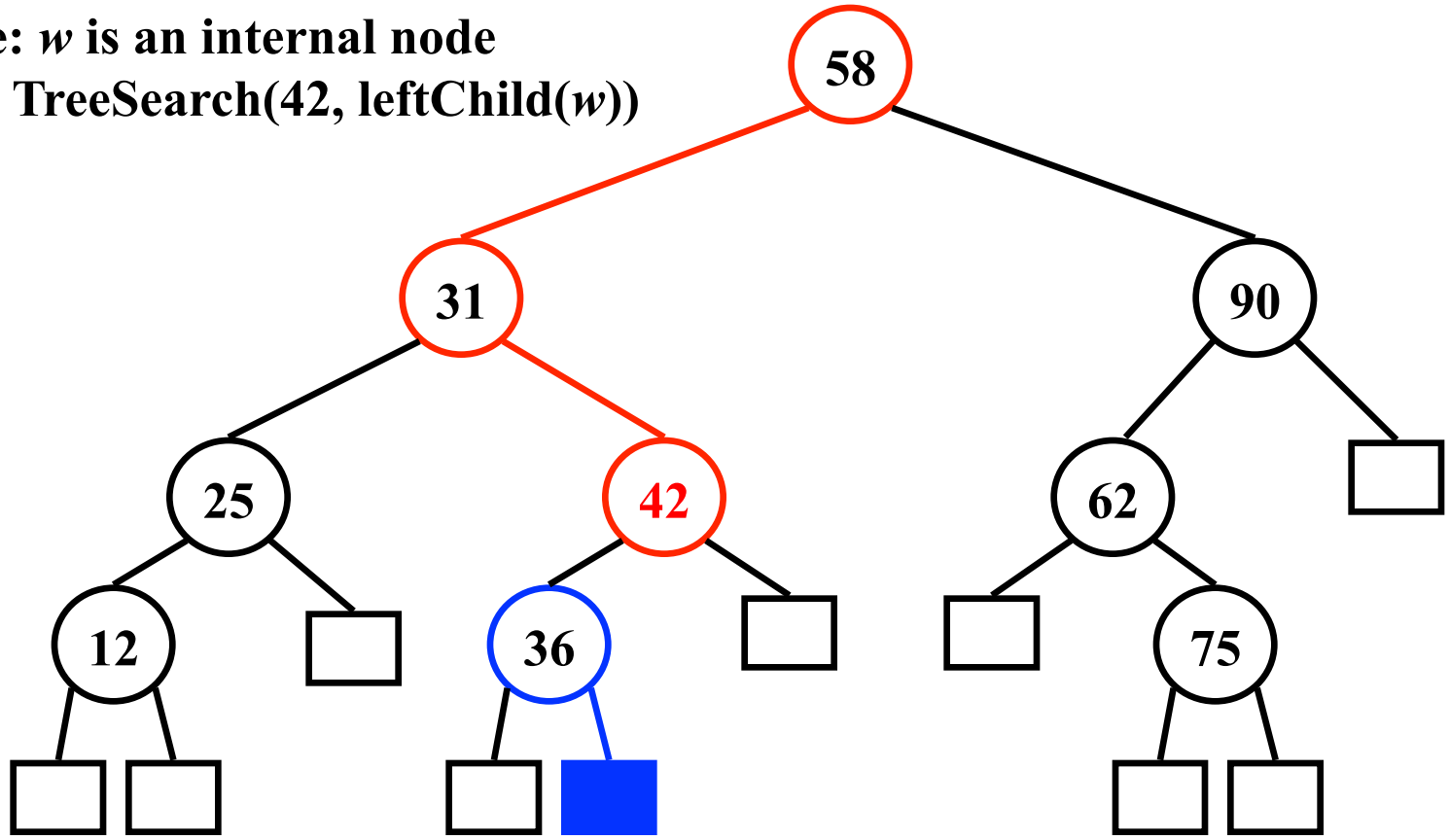
Case: *w* is an internal node
***w* ← TreeSearch(42, leftChild(*w*))**



Insertion in a Binary Search Tree

InsertItem(42,*e*): *w* ← TreeSearch(42,*T*.root())

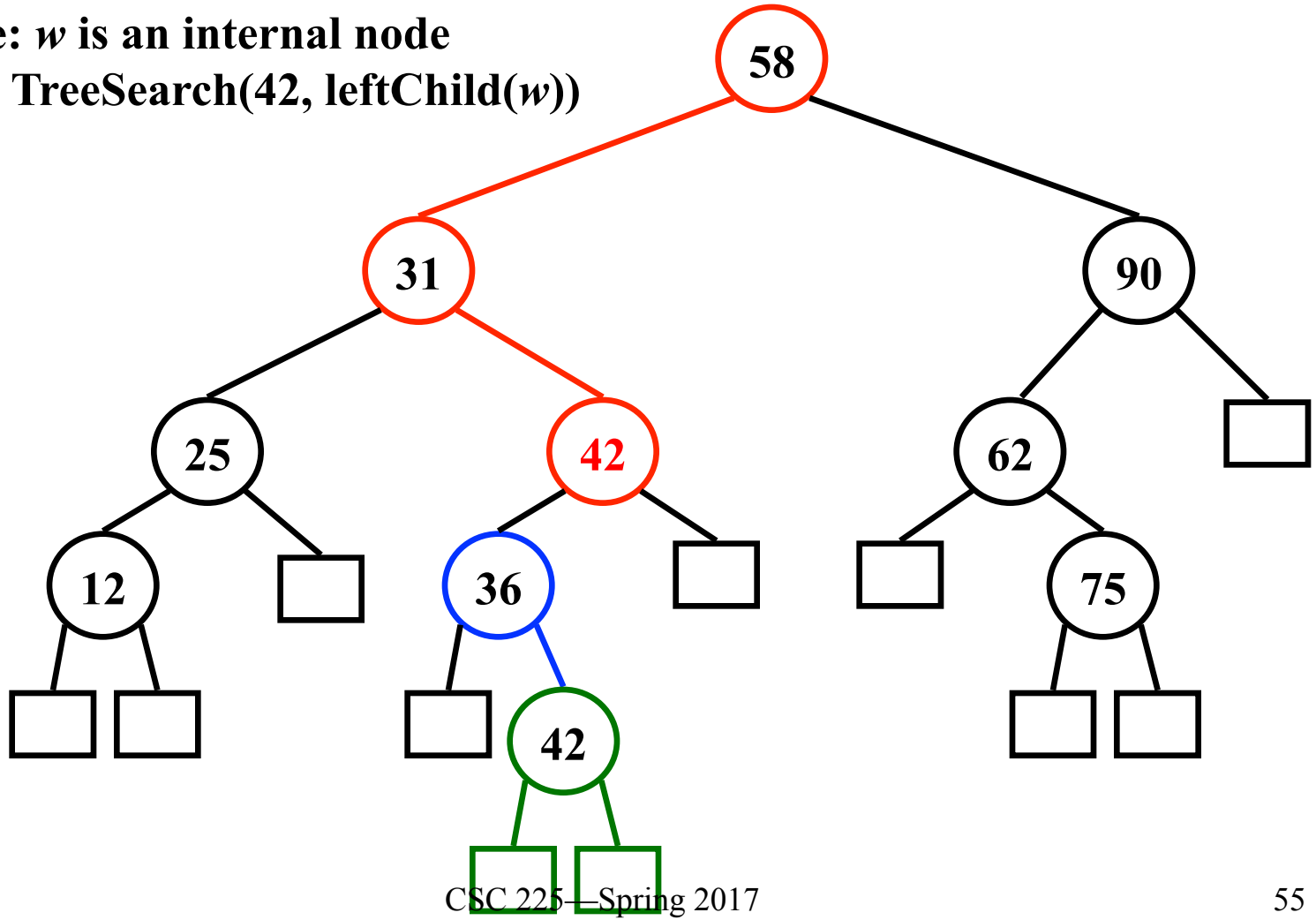
Case: *w* is an internal node
***w* ← TreeSearch(42, leftChild(*w*))**



Insertion in a Binary Search Tree

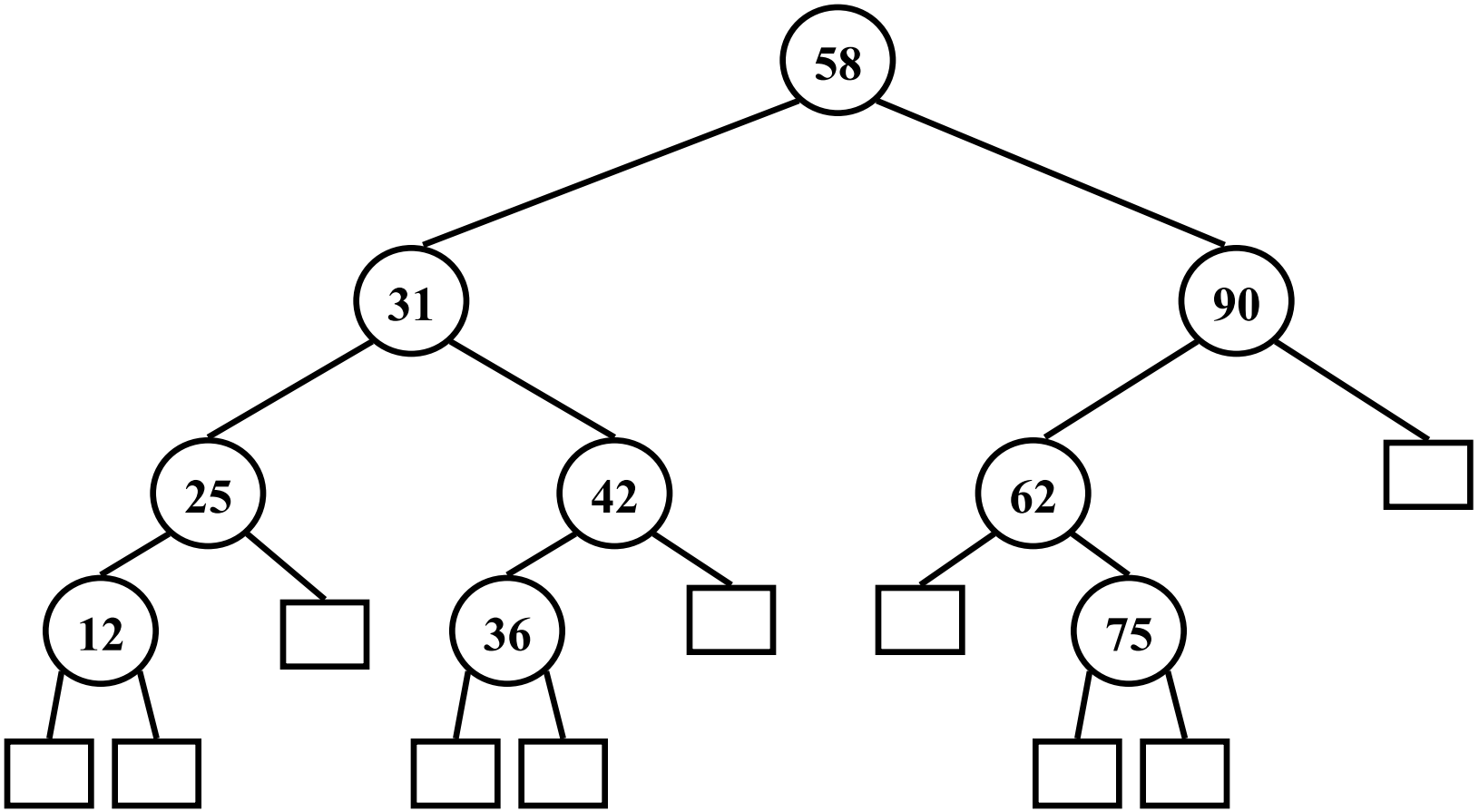
InsertItem(42,*e*): *w* ← TreeSearch(42,*T*.root())

Case: *w* is an internal node
***w* ← TreeSearch(42, leftChild(*w*))**



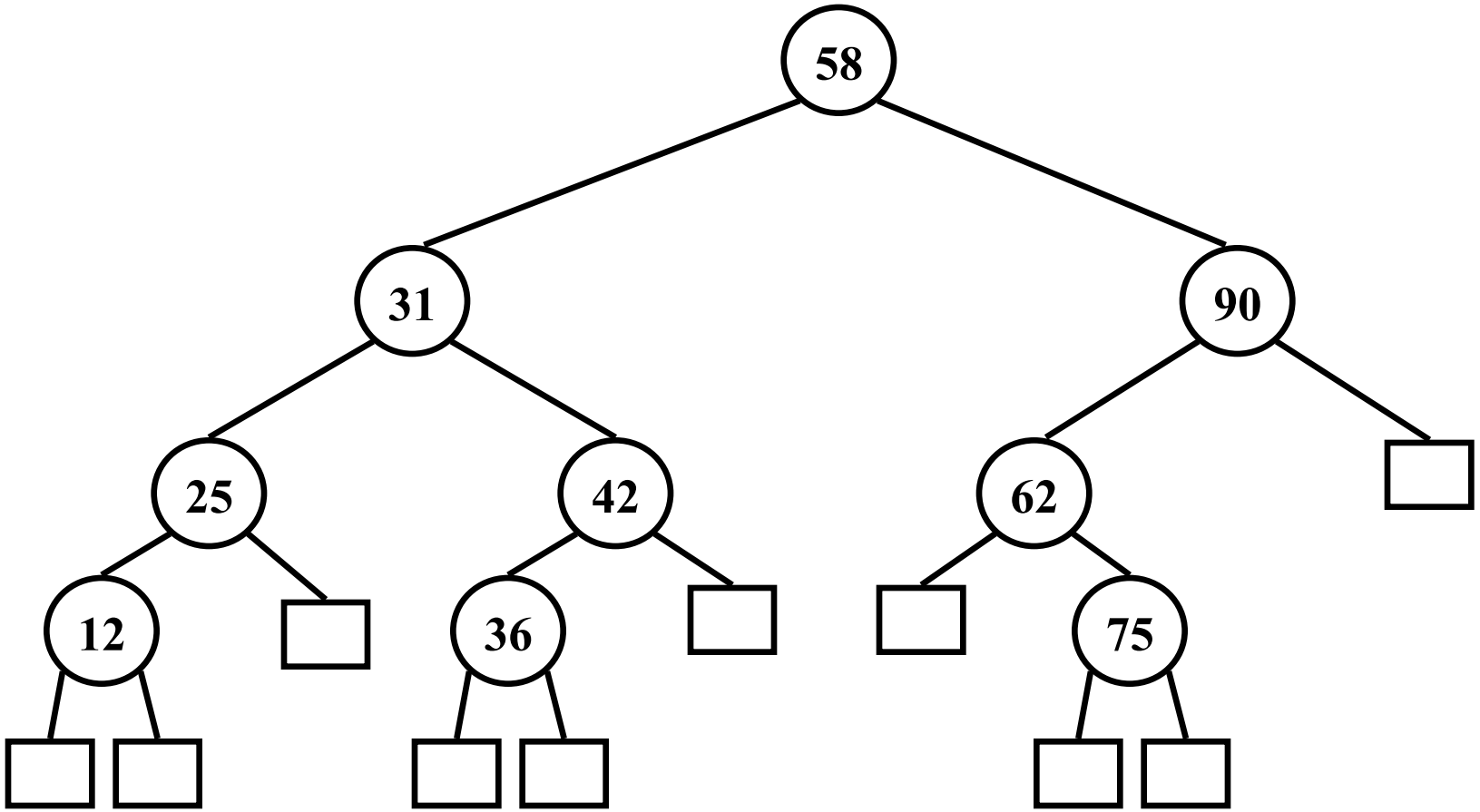
Removal in a Binary Search Tree

removeElement(37)



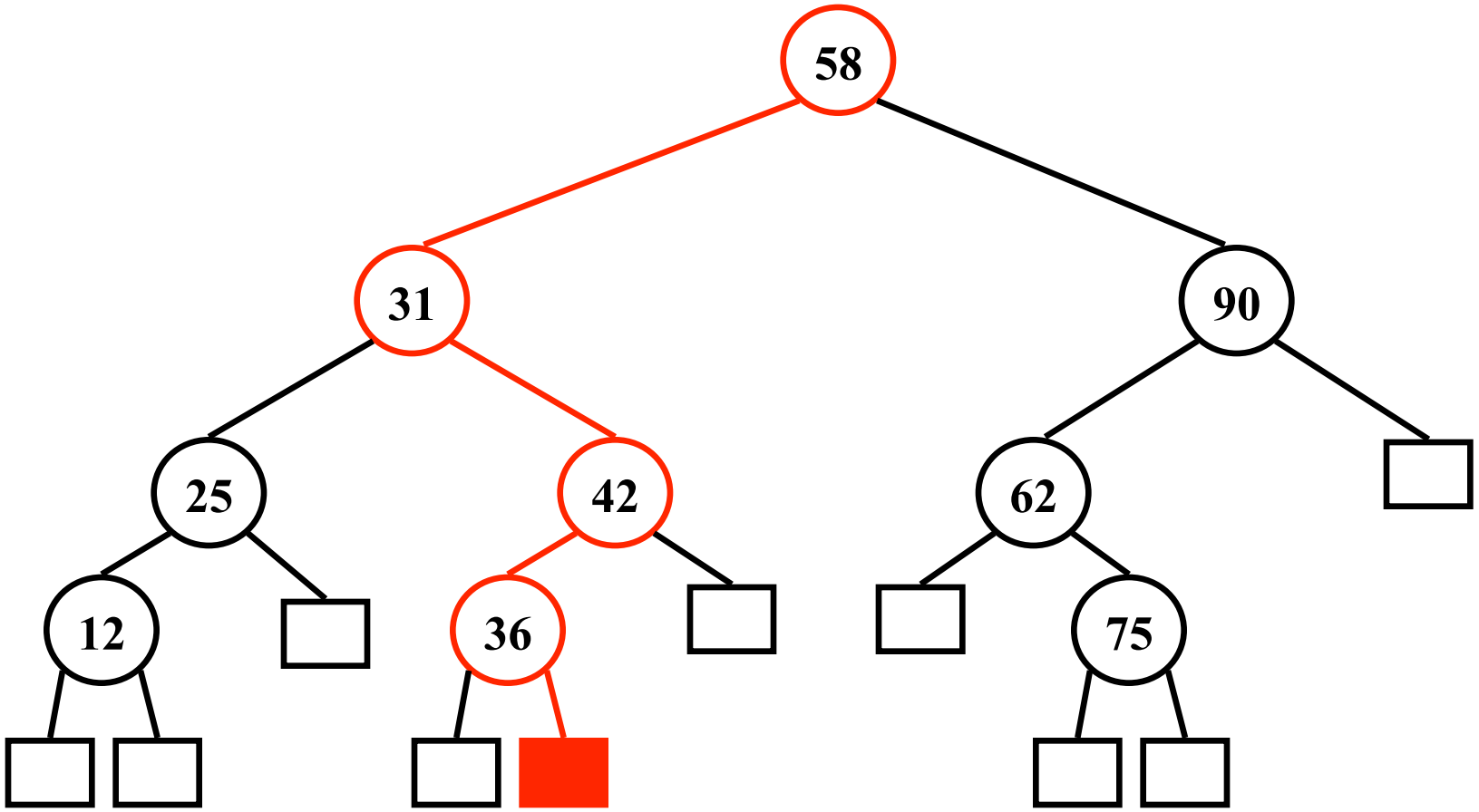
Removal in a Binary Search Tree

removeElement(37): $w \leftarrow$ TreeSearch(37,*T*.root())



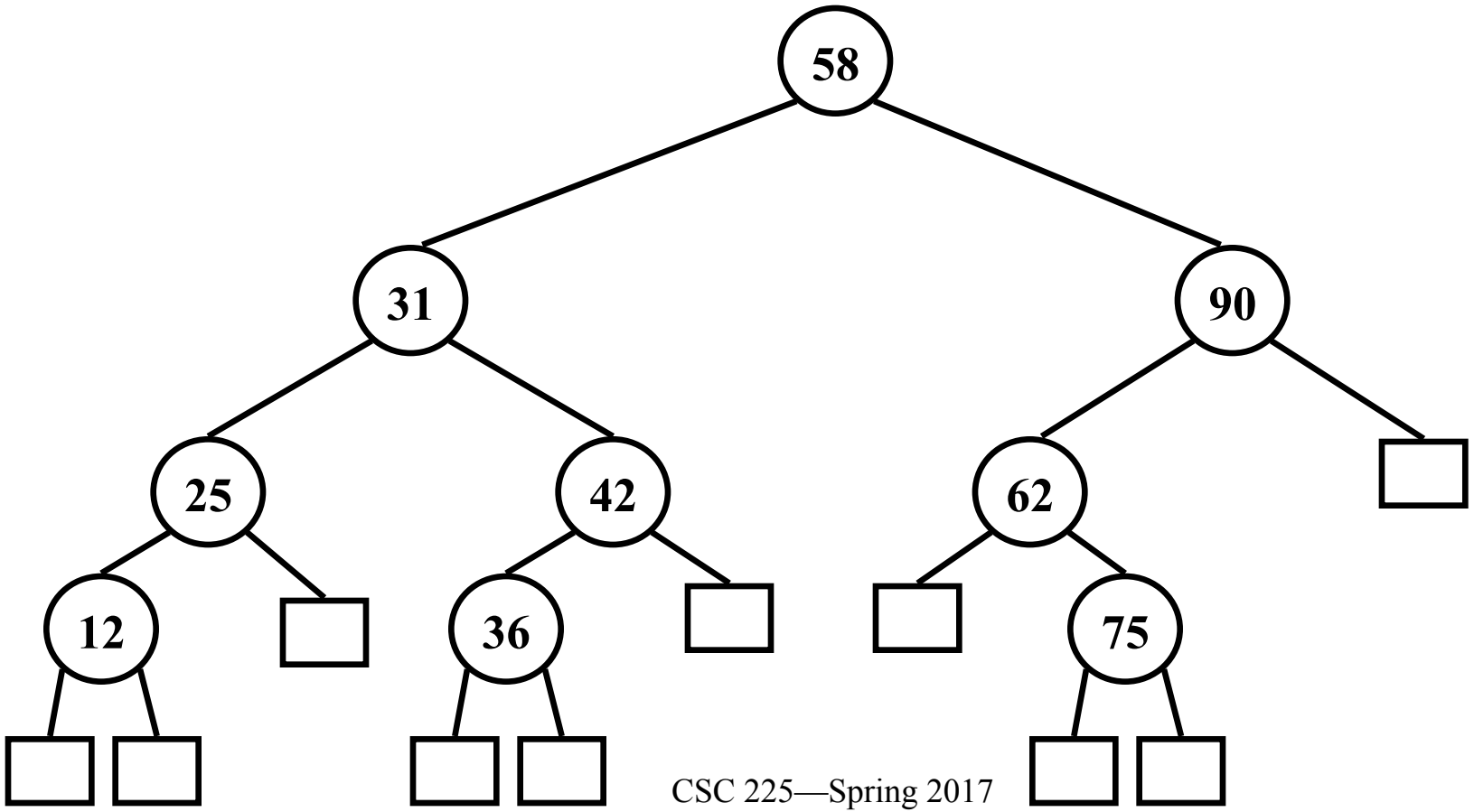
Removal in a Binary Search Tree

removeElement(37): $w \leftarrow$ **TreeSearch(37,***T***.root())**
if *w* is a leaf **then return** element NO_SUCH_KEY



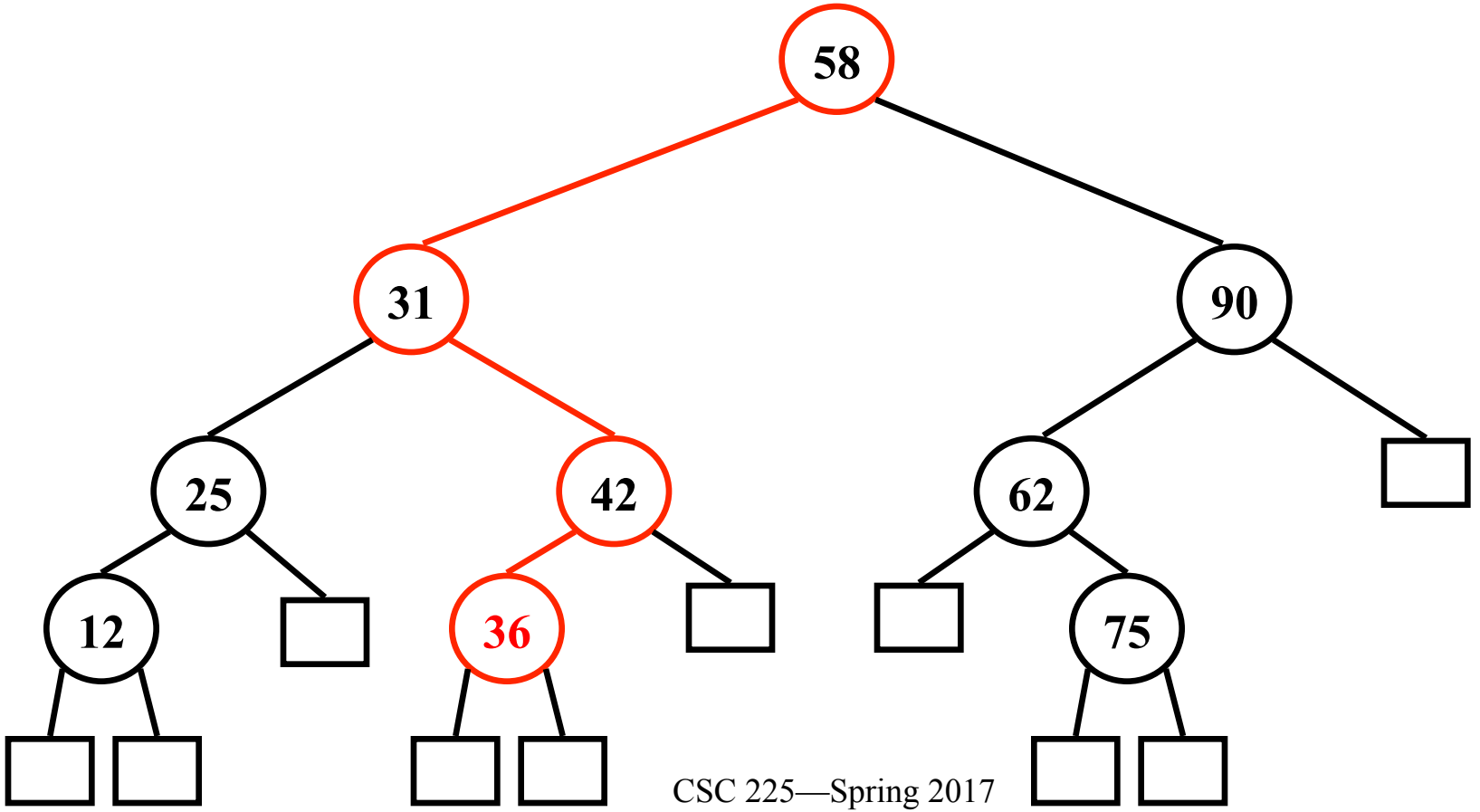
Removal in a Binary Search Tree

removeElement(36): $w \leftarrow$ TreeSearch(36,*T*.root())



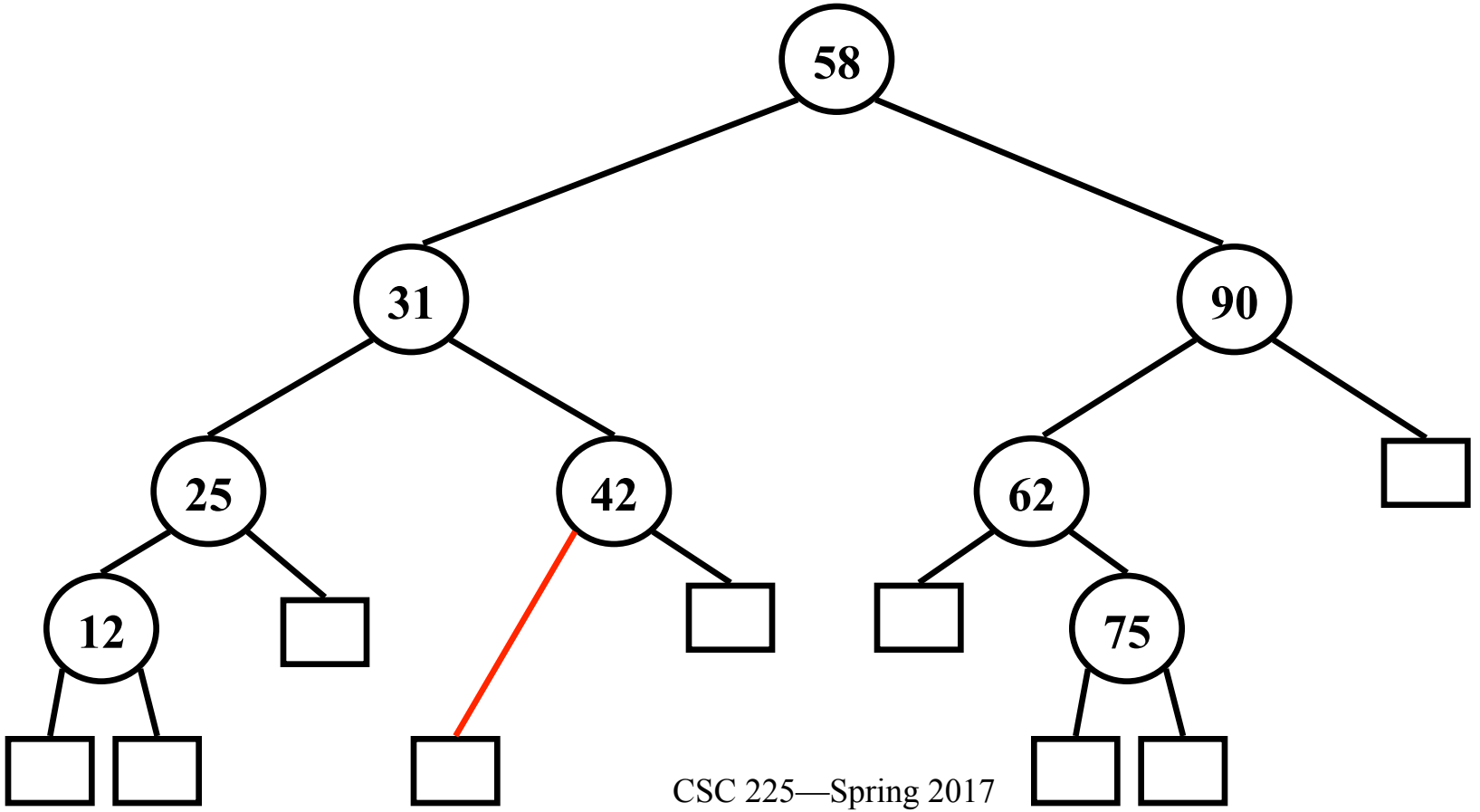
Removal in a Binary Search Tree

removeElement(36): $w \leftarrow$ **TreeSearch(36,***T***.root())**
if *w* is an internal node and *w* has a child *z* that is a leaf **then**
remove *w* and *z*.



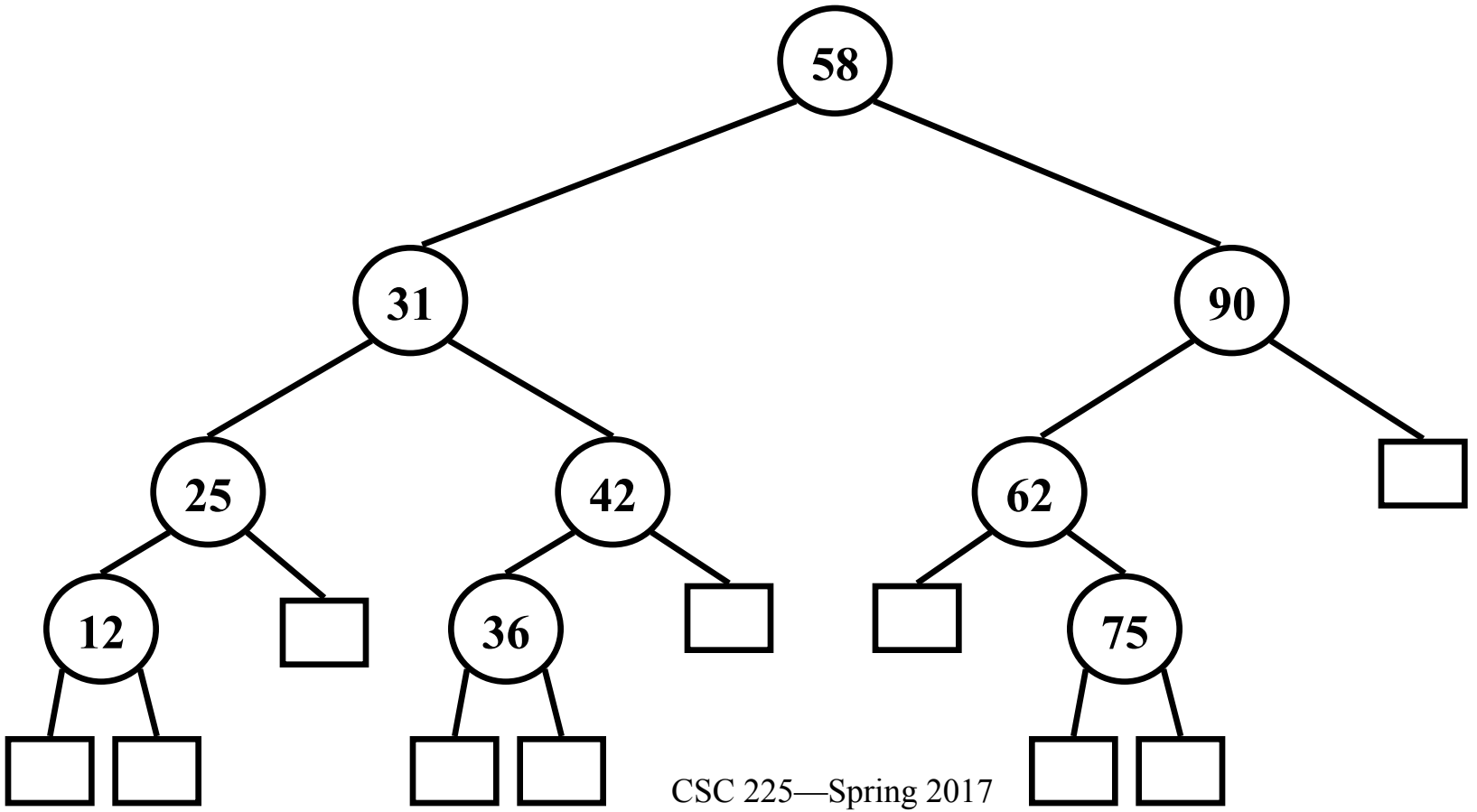
Removal in a Binary Search Tree

removeElement(36): $w \leftarrow$ TreeSearch(36,*T*.root())
if *w* is an internal node and *w* has a child *z* that is a leaf **then**
remove *w* and *z* (and at the same time replace *w* by *z*'s sibling).



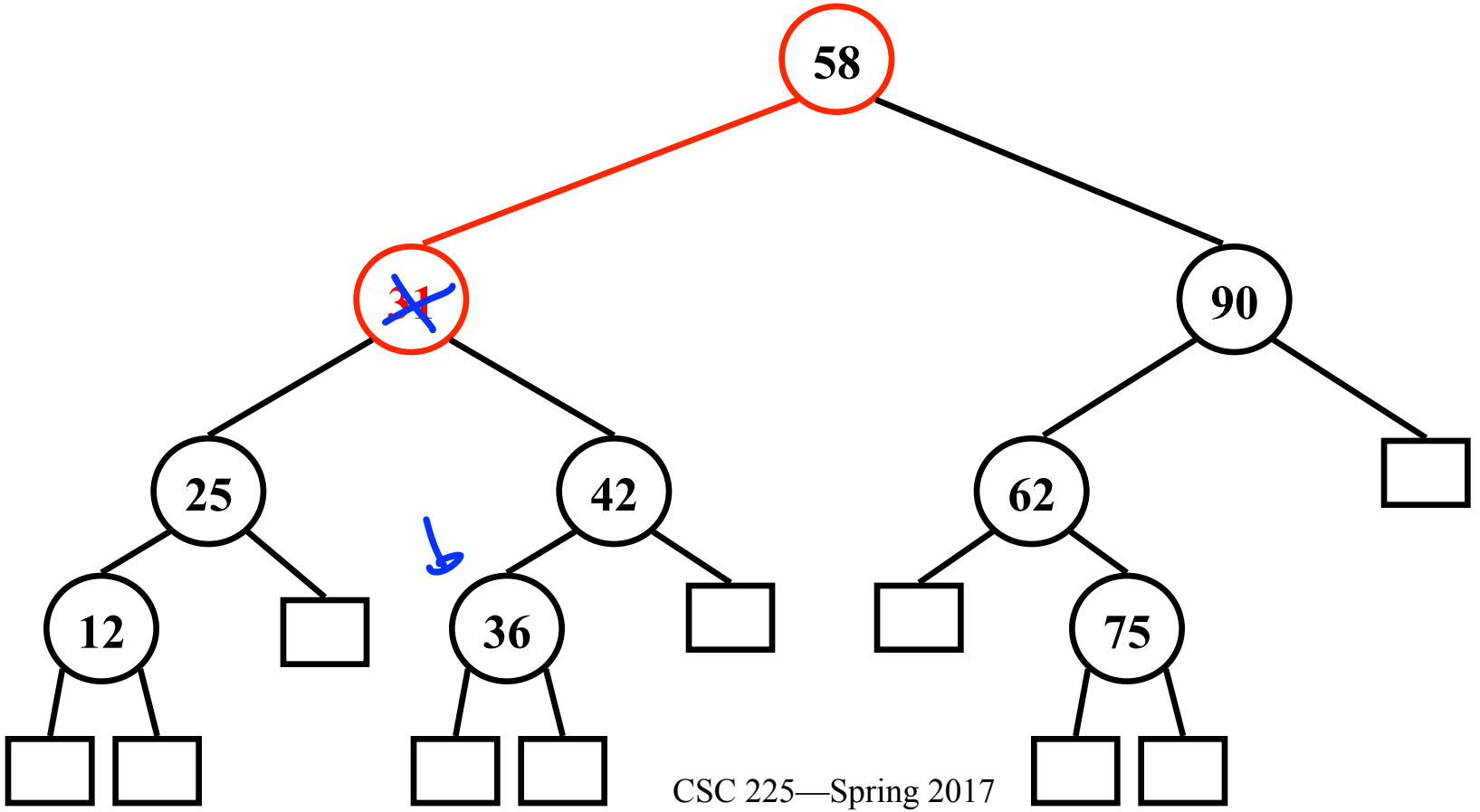
Removal in a Binary Search Tree

removeElement(31): $w \leftarrow$ TreeSearch(31,*T*.root())



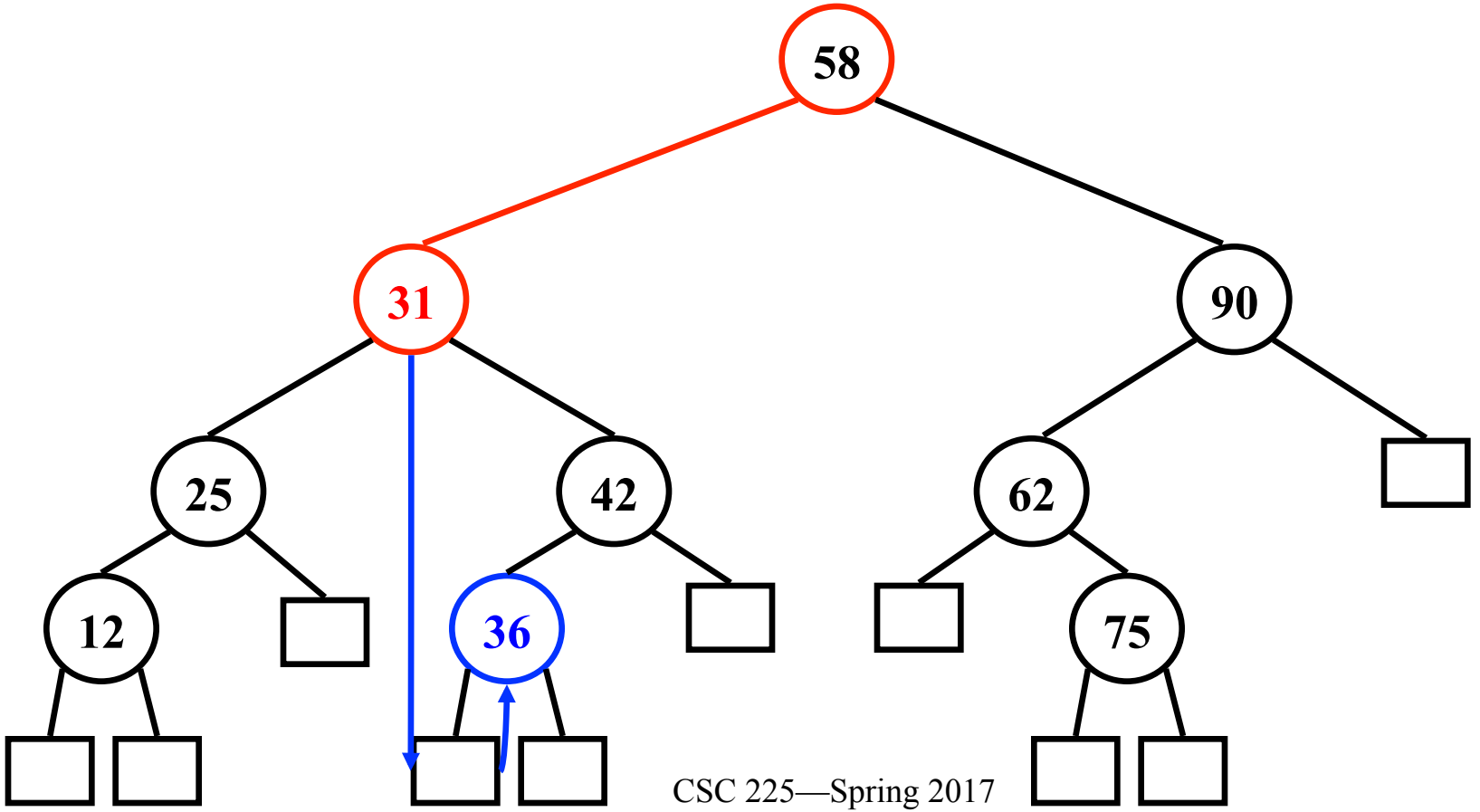
Removal in a Binary Search Tree

removeElement(31): $w \leftarrow$ **TreeSearch(31,***T***.root())**
if *w* is an internal node and *w* has no children that are leaves **then**
find the first internal node that follows *w* in an inorder traversal of *T*.



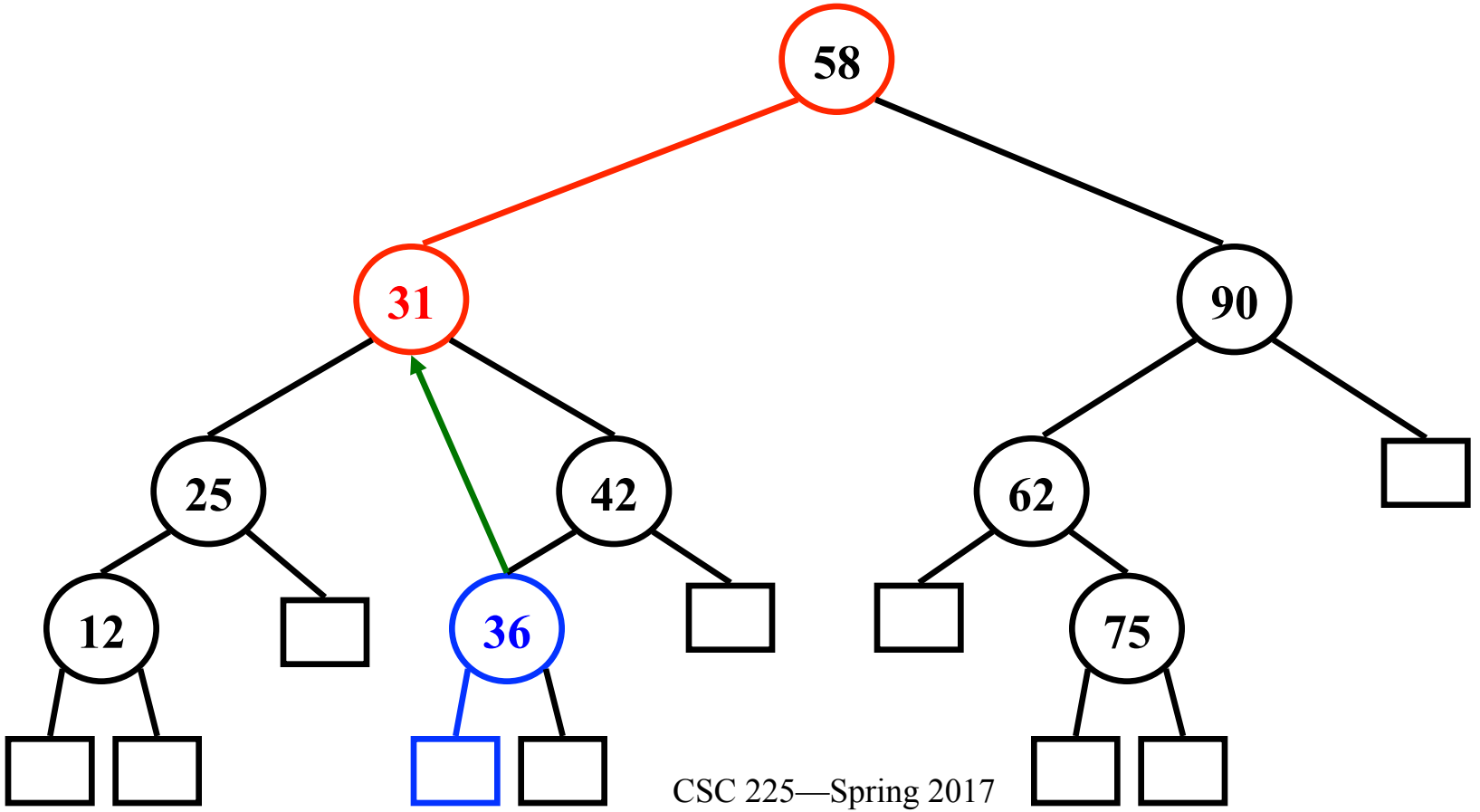
Removal in a Binary Search Tree

removeElement(31): $w \leftarrow$ **TreeSearch(31,***T***.root())**
if *w* is an internal node and *w* has no children that are leaves **then**
find the first internal node that follows *w* in an inorder traversal of *T*.



Removal in a Binary Search Tree

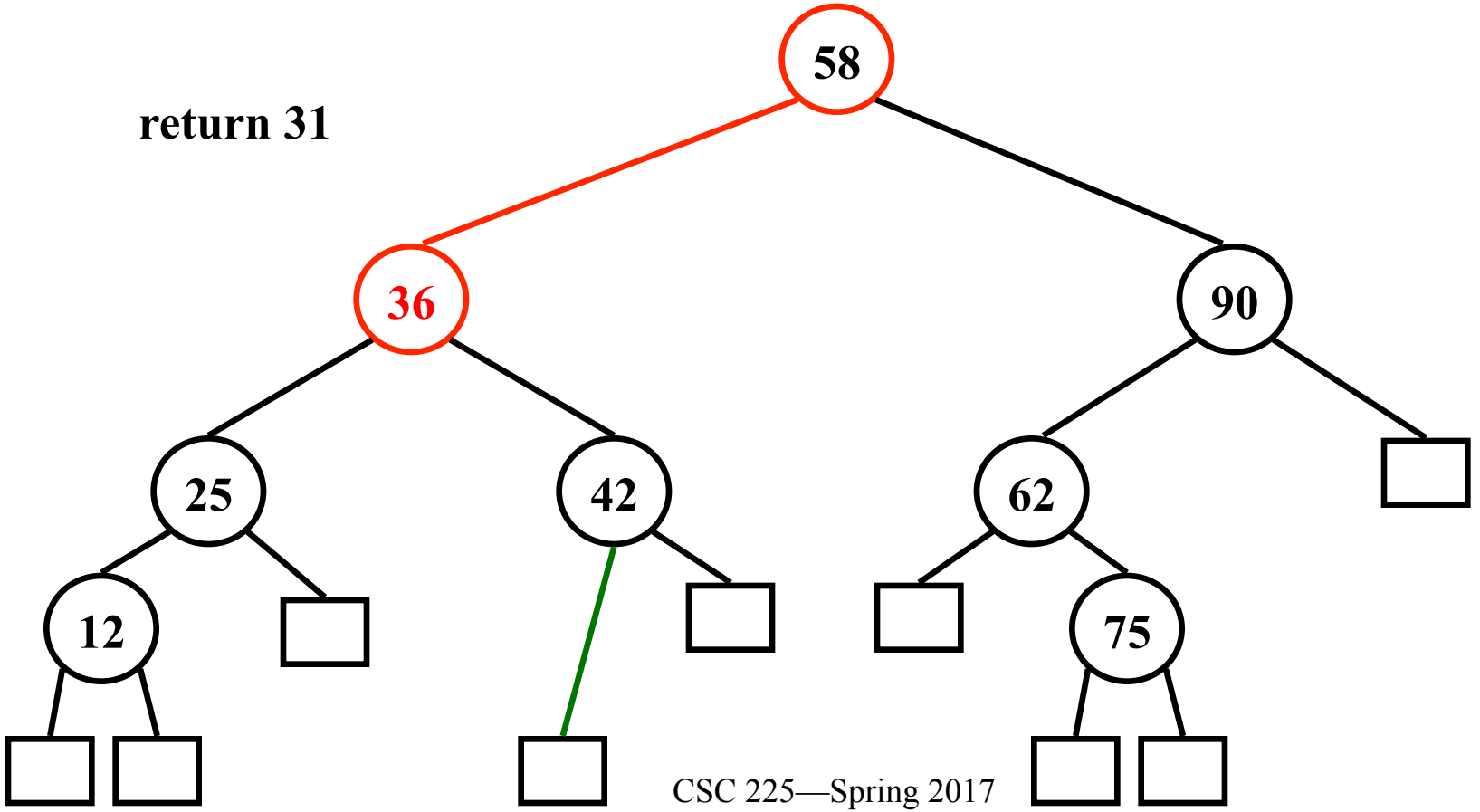
removeElement(31): $w \leftarrow$ **TreeSearch(31,***T***.root())**
if *w* is an internal node and *w* has no children that are leaves **then**
find the first internal node that follows *w* in an inorder traversal of *T*.



Removal in a Binary Search Tree

removeElement(31): $w \leftarrow$ **TreeSearch(31,***T***.root())**
if w is an internal node and w has no children that are leaves **then**
find the first internal node that follows w in an inorder traversal of T.

return 31



Performance of Binary Search Trees

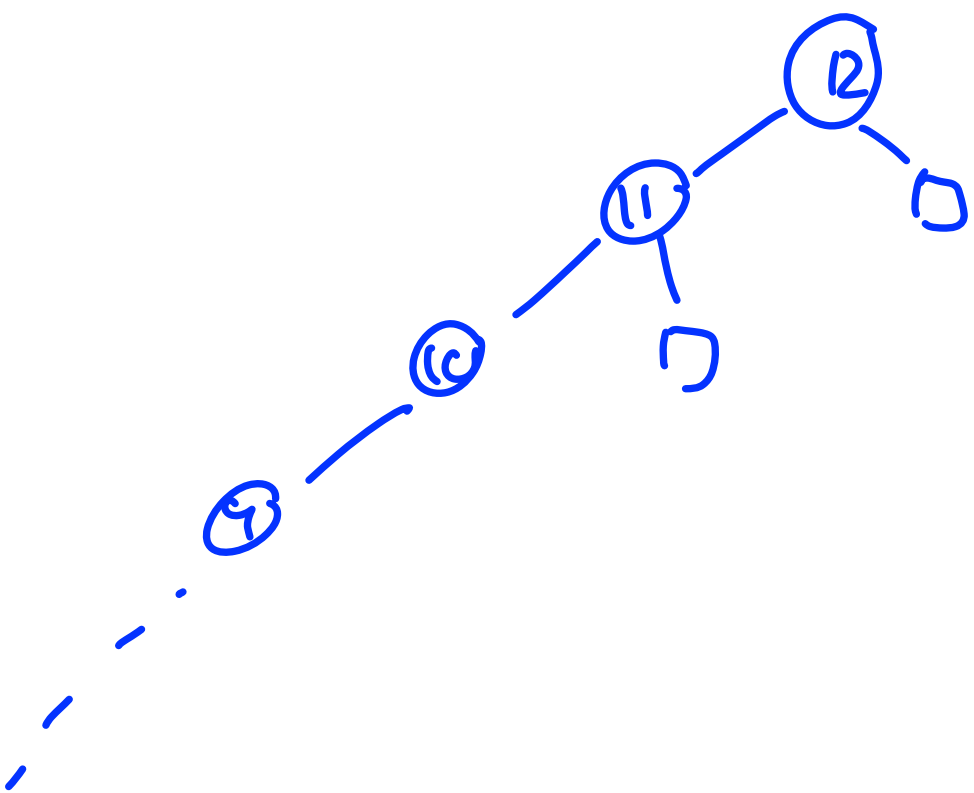
Let h be the height of a binary search tree T.

The following methods are performed on T:

	time complexity
• size, isEmpty	O(1)
• findElement, insertItem, removeElement	O(h)
• findAllElements, removeAllElements [s: size of the returned iterators]	O(h+s)

Space complexity: O(n)

Worst-case height of a binary search tree



$O(h)$

$O(n)$

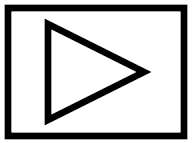
AVL Trees

- AVL trees are *height balanced* binary search trees.
- Idea with balance of height avoid linear running time of dictionary operations
- Inventors of AVL trees: Adel'son-Vel'skii and Landis

Definition of AVL trees

An *AVL tree* is a binary search tree satisfying the *height-balance property*.

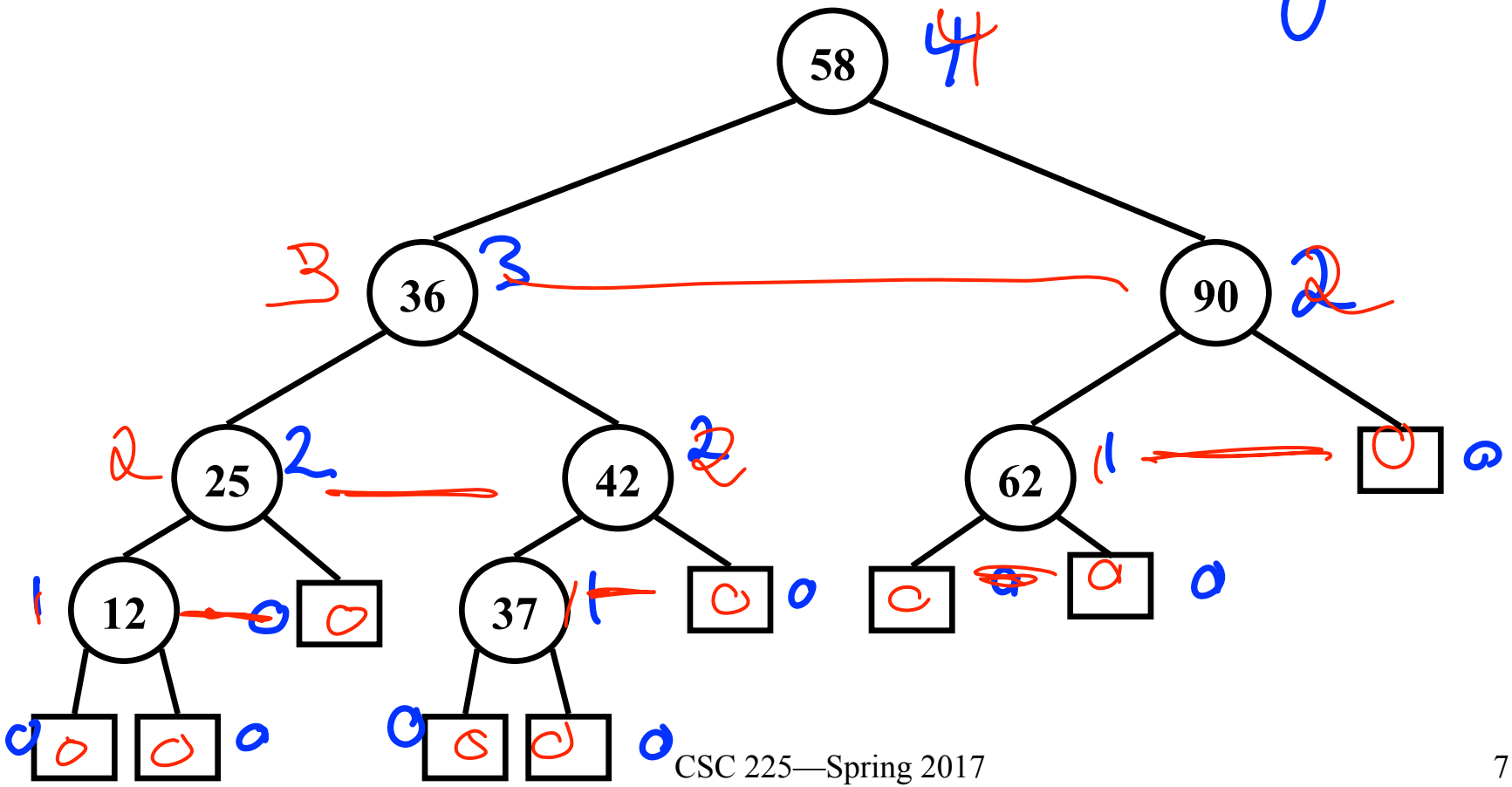
Height-Balance Property: For every internal node v of T, the heights of the children of v can differ by at most 1.



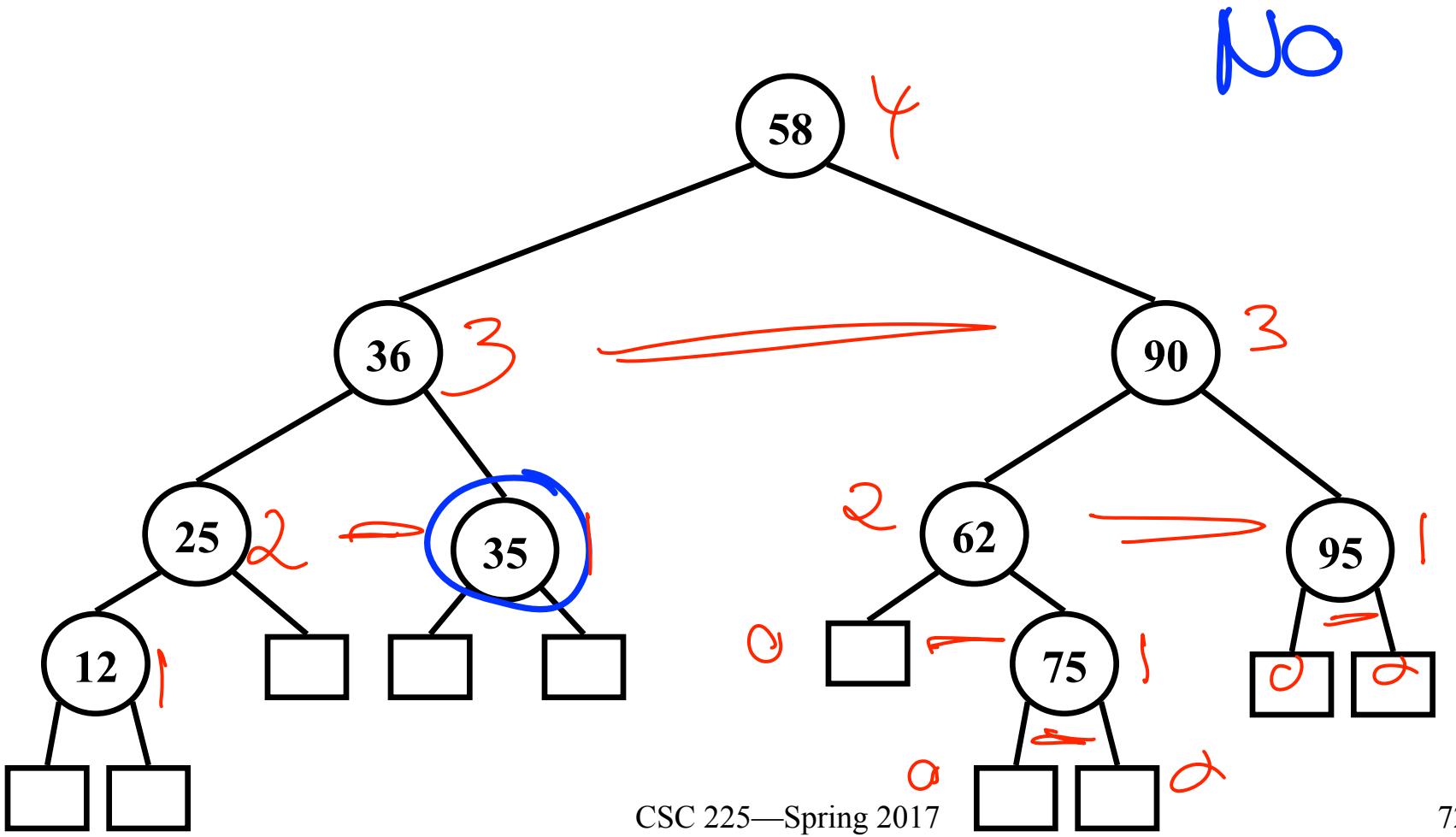
Is this an AVL tree?

- bin. search tree
- height balanced

Yes.



Is this an AVL tree?



The height of an AVL tree

Theorem: The height of an AVL tree T storing n items is $O(\log n)$.

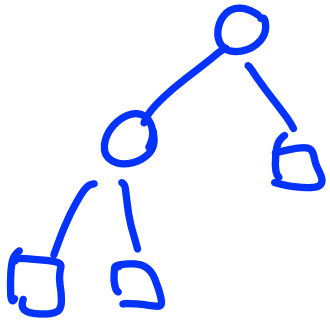
$n(h)$: minimum number of internal nodes in AVL tree of height h.

$n(h)$ grows exponentially

so h is $O(\log n)$

$h = 1$

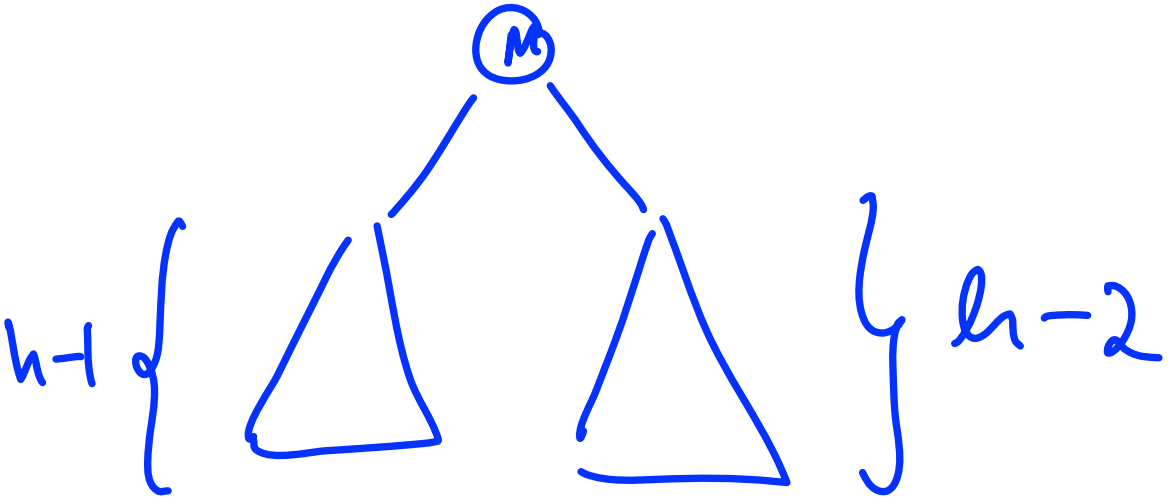
$h = 2$



$n(1) = 1$

$n(2) = 2$

$$n(h) = 1 + n(h-1) + n(h-2)$$



Proof. Let n(h) be the *minimum* number of internal nodes of an AVL tree with height h.

Then $n(1) = 1$,

$n(2) = 2$

and $n(h) = 1 + n(h-1) + n(h-2)$ for $h \geq 2$.

We show:

(1) n(h) grows exponentially

(2) (1) implies that the height of an AVL tree storing n keys is O(log n).

(1) We show that

Induction hypothesis:

$n(h) > 2^i\, n(h - 2i)$ for any integer i such that $h - 2i \geq 1$.

We show the claim using induction on i.

Base case: $i = 1$

$$n(h) = n(h-1) + n(h-2) + 1$$

$$= n(h-2) + n(h-3) + 1 + n(h-2) + 1$$

$n(h-1) = n(h-2) + n(h-3) + 1$

$$= 2n(h-2) + n(h-3) + 2$$

$$> 2n(h-2)$$

Induction step: $i \rightarrow i + 1$

We know that $n(h) > 2^i\, n(h - 2i)$ (hypothesis).

Therefore,

$$n(h) > 2^i(n(h-2i - 1) + n(h-2i - 2) +1)$$

$n(h - 2i) = n(h - 2i - 1) + n(h - 2i - 2) + 1$

and further

$$n(h) > 2^i(n(h - 2i - 2) + n(h - 2i - 3) +1 + n(h - 2i - 2) +1)$$

$n(h - 2i - 1) = n(h - 2i - 2) + n(h - 2i - 3) + 1$

$$= 2^i(2n(h - 2i - 2) + n(h - 2i - 3) +2)$$

which yields

$$n(h) > 2^i\,(2\, n(h-2i - 2))$$

$$= 2^{i+1}\,(2\, n(h-2(i + 1)).$$

This proves the claim!

(2) We pick $i = \lceil h/2 \rceil - 1$. Then

$$\begin{aligned} n(h) &> 2^{\lceil h/2 \rceil - 1}\, n(h - 2(\lceil h/2 \rceil - 1)) \\ &\geq 2^{\lceil h/2 \rceil - 1}\, n(h - h + 2)) \\ &= 2^{\lceil h/2 \rceil - 1}\, n(2) \\ &\geq 2^{\lceil h/2 \rceil} \end{aligned}$$

Therefore $\log(n(h)) > h/2$.

Thus $2 \log(n(h)) > h$.

We conclude: if $n(h) = n$

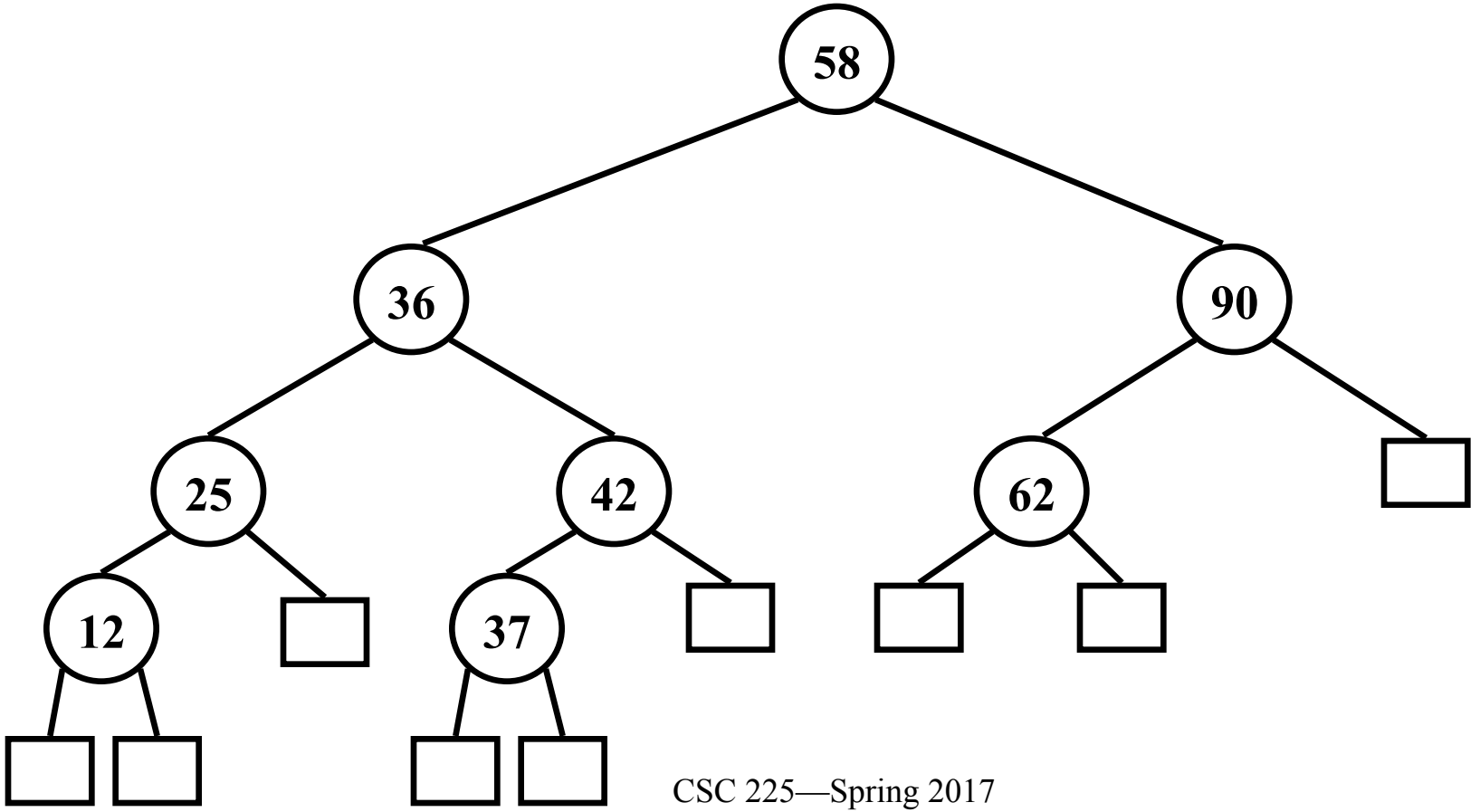
then $h < 2 \log n$, that is h is $O(\log n)$.

Updating AVL trees

- Height balance property must always hold, that is it must hold even after operations such as *insertion* and *removal*.

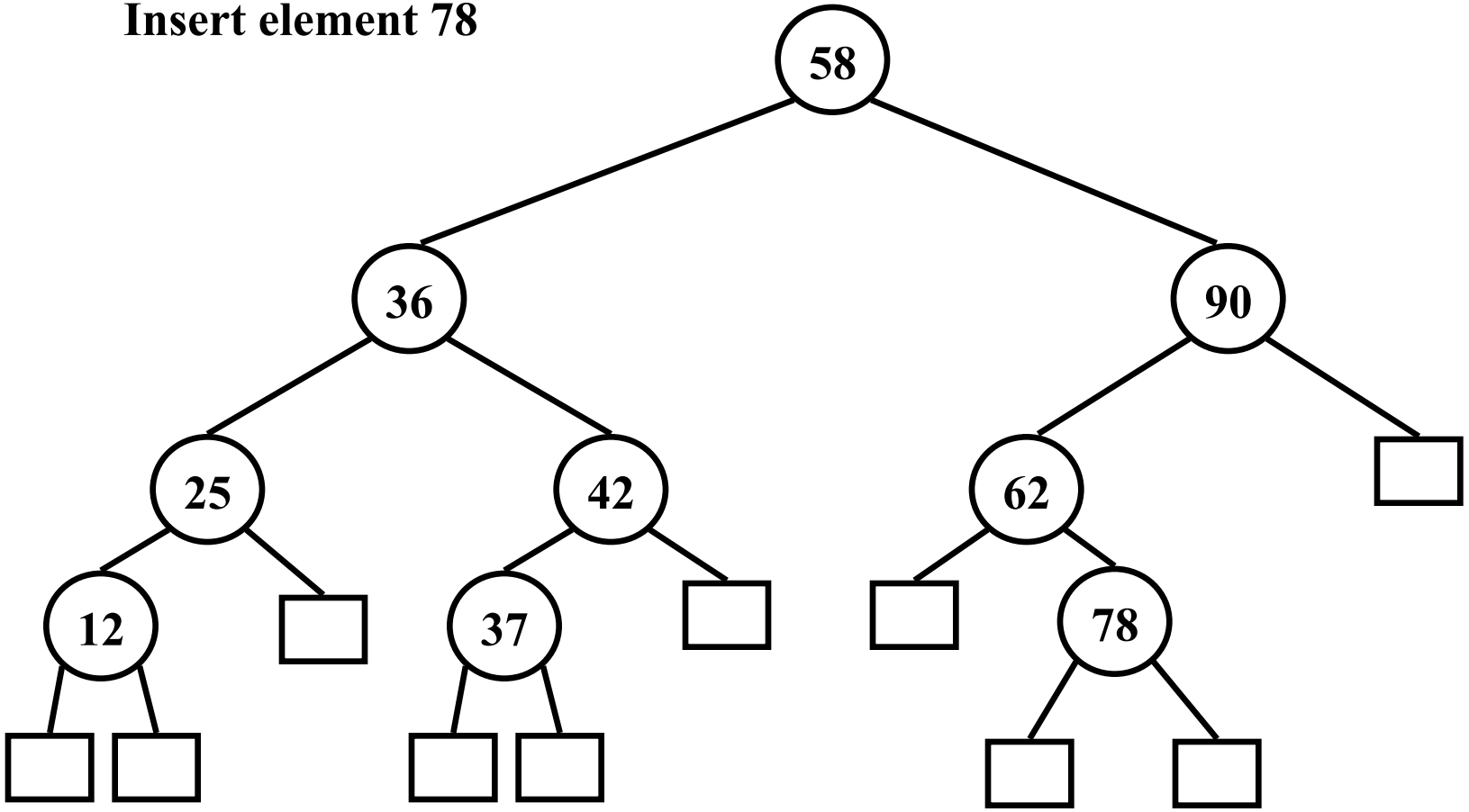
Insertion

Insert element 78

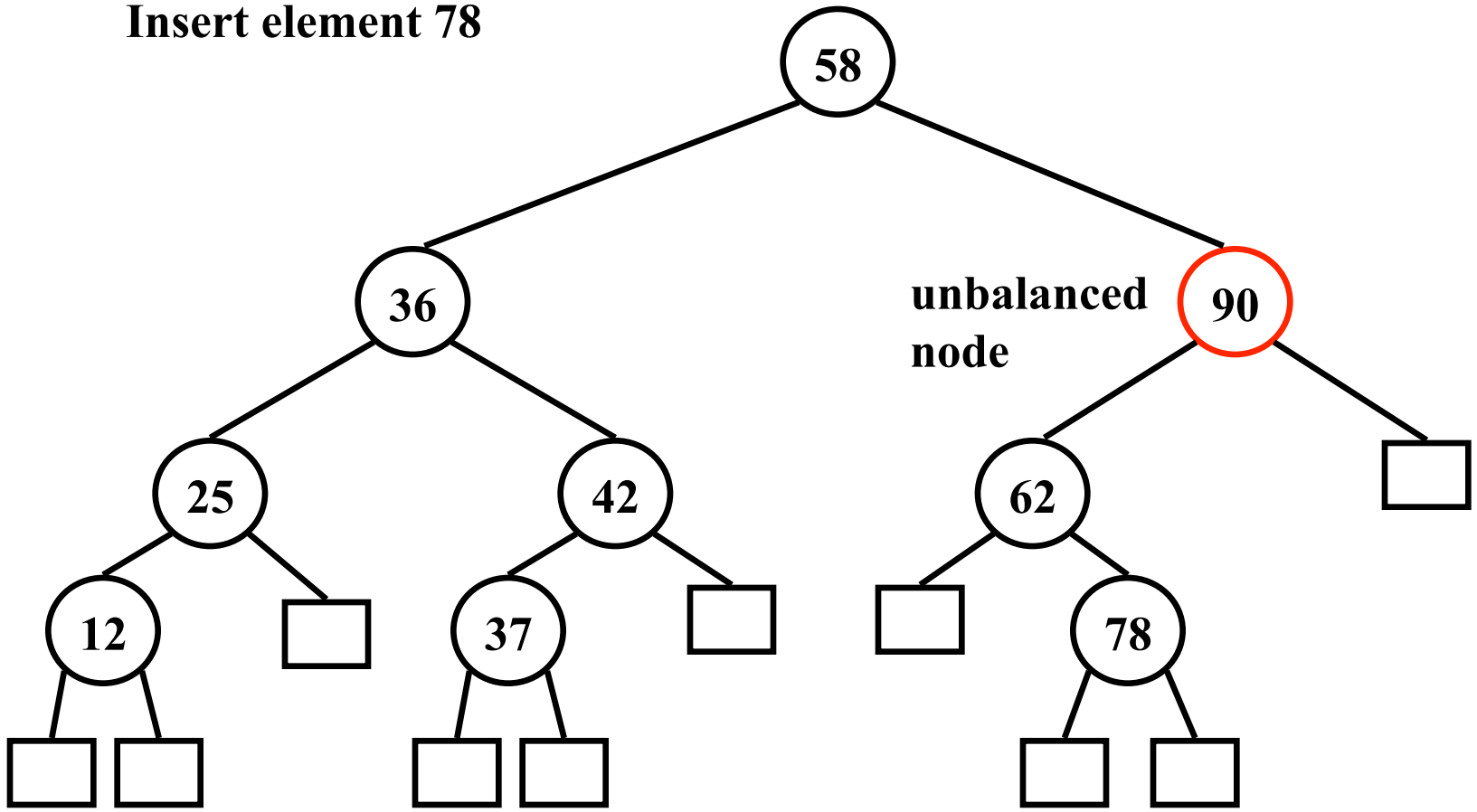


Insertion – Step 1: Insert as in binary search trees

Insert element 78



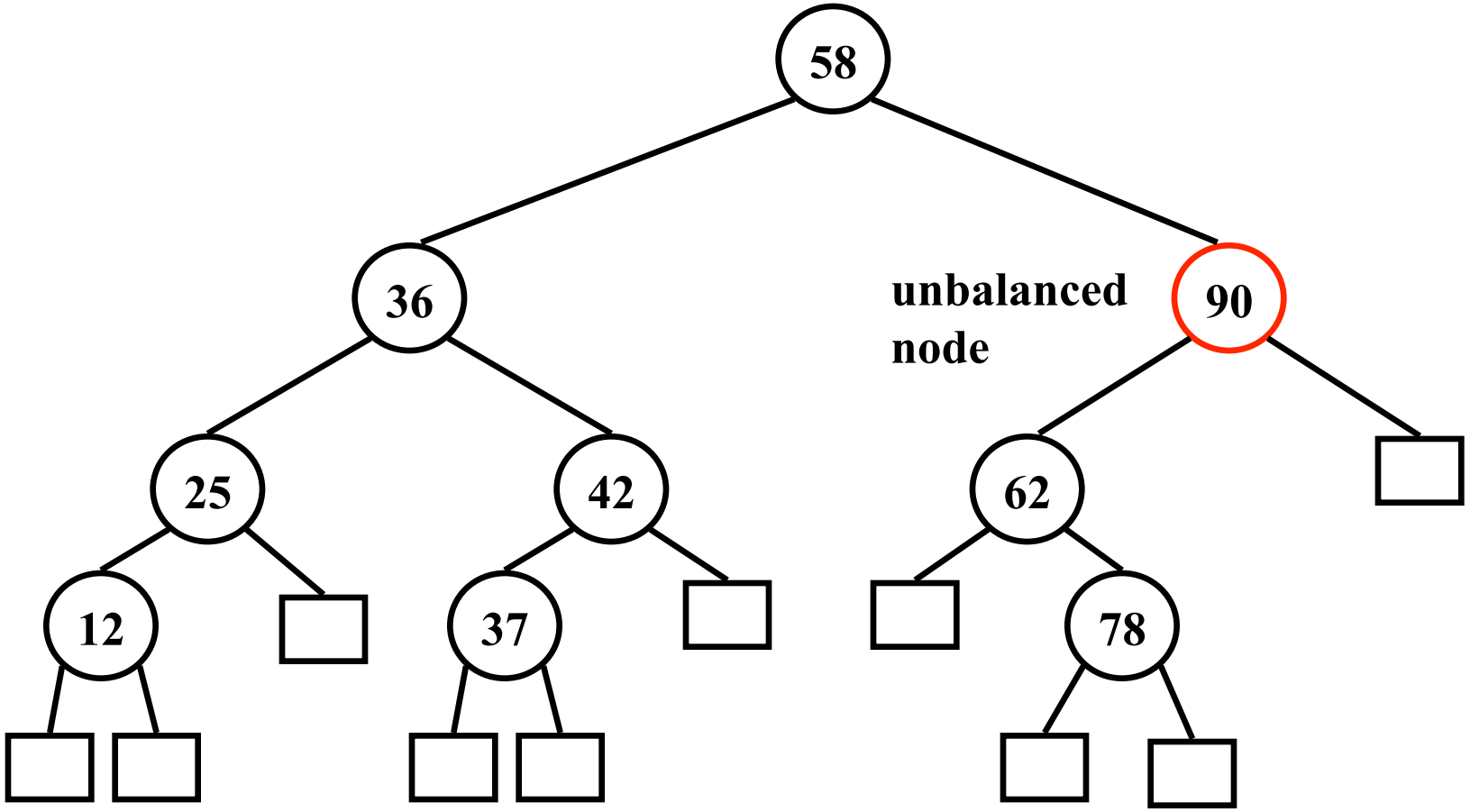
Insertion



Balanced and unbalanced nodes

- Given a binary search tree *T,* we say a node *v* in *T* is *balanced* if the absolute value of the difference between the height of its children is at most 1.
- Otherwise, we say *v* is *unbalanced*.
- *Theorem*: A binary search tree *T* is an AVL tree iff every node in *T* is balanced.

After Insertion: Fix the Unbalance!

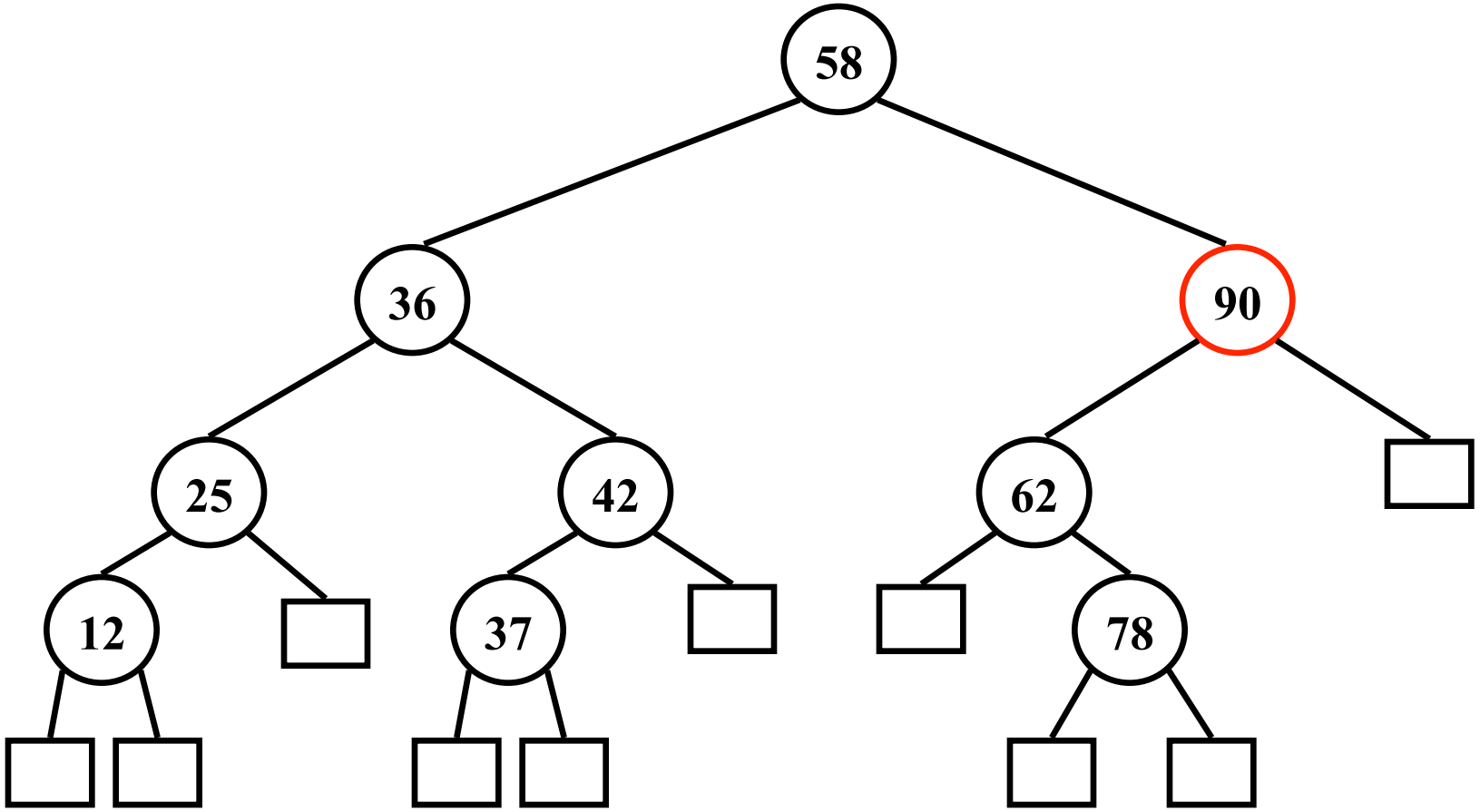


Algorithm restructure(x)

Input: A node x of a binary search tree T that has both a parent y and a grandparent z

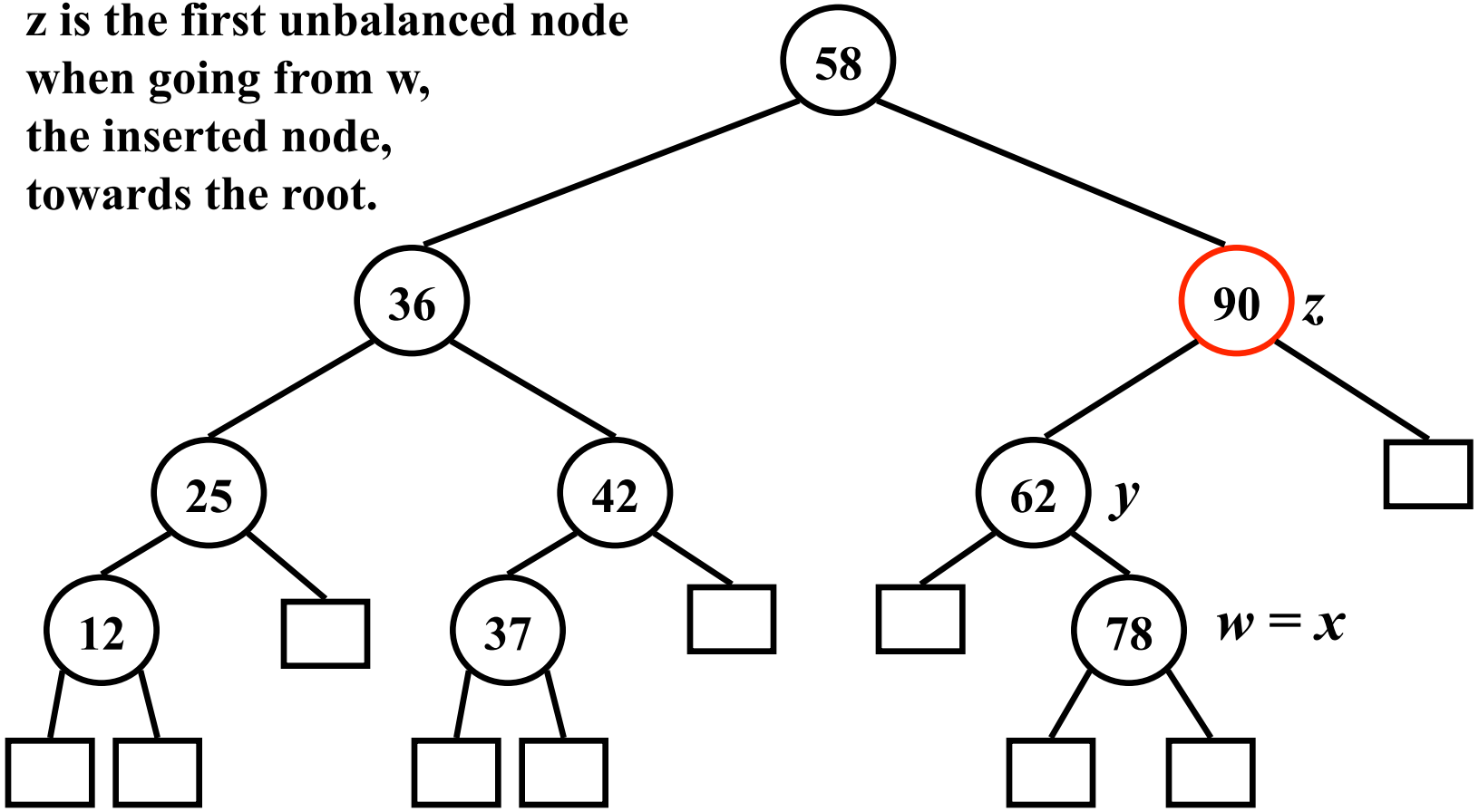
Output: Tree T after *trinode-restructuring* (which corresponds to a single or double rotation) involving x, y, and z.

After Insertion: Fix the Unbalance!



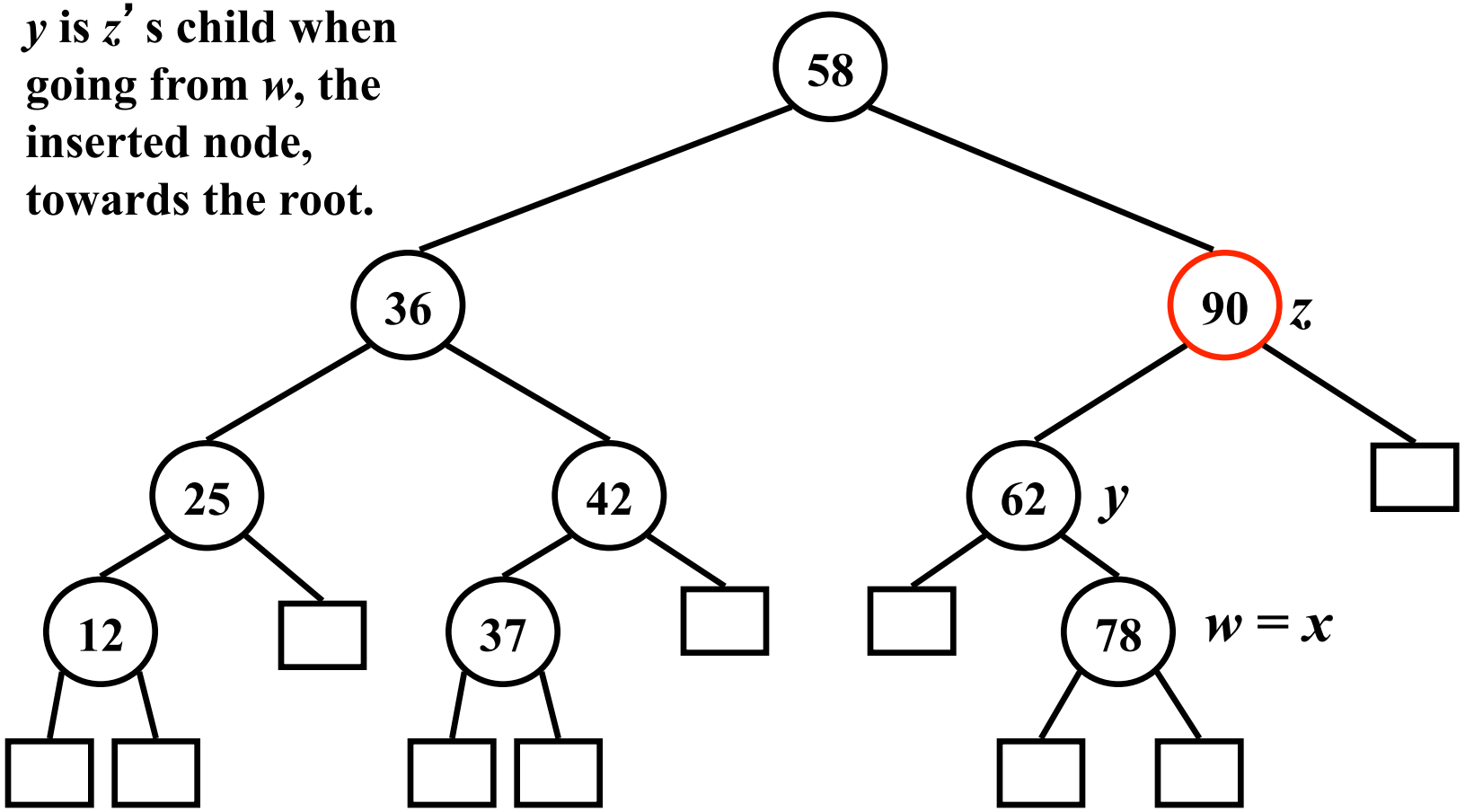
restructure(x)

z is the first unbalanced node when going from w, the inserted node, towards the root.



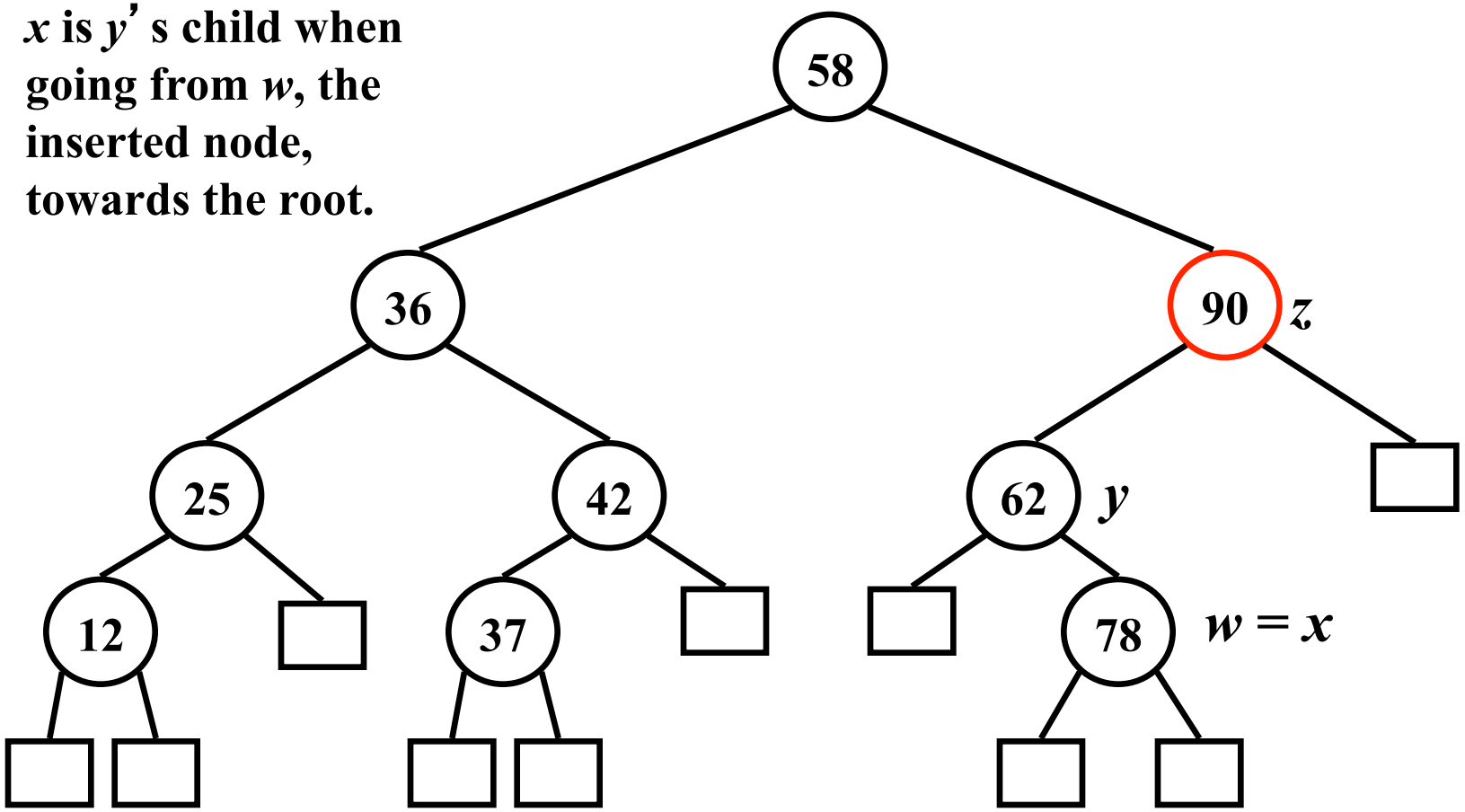
restructure(x)

***y* is *z*' s child when going from *w*, the inserted node, towards the root.**



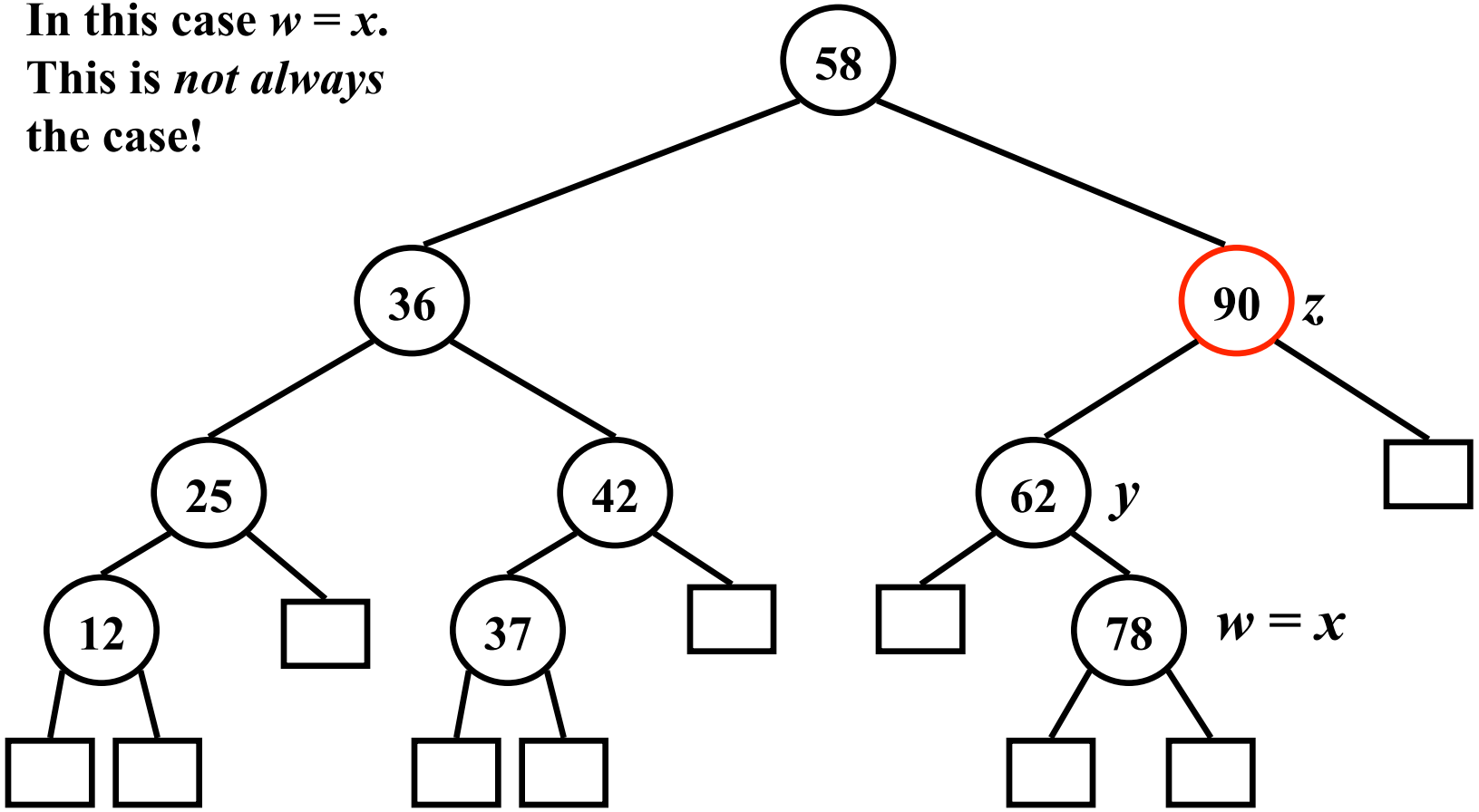
restructure(x)

x is y's child when going from w, the inserted node, towards the root.

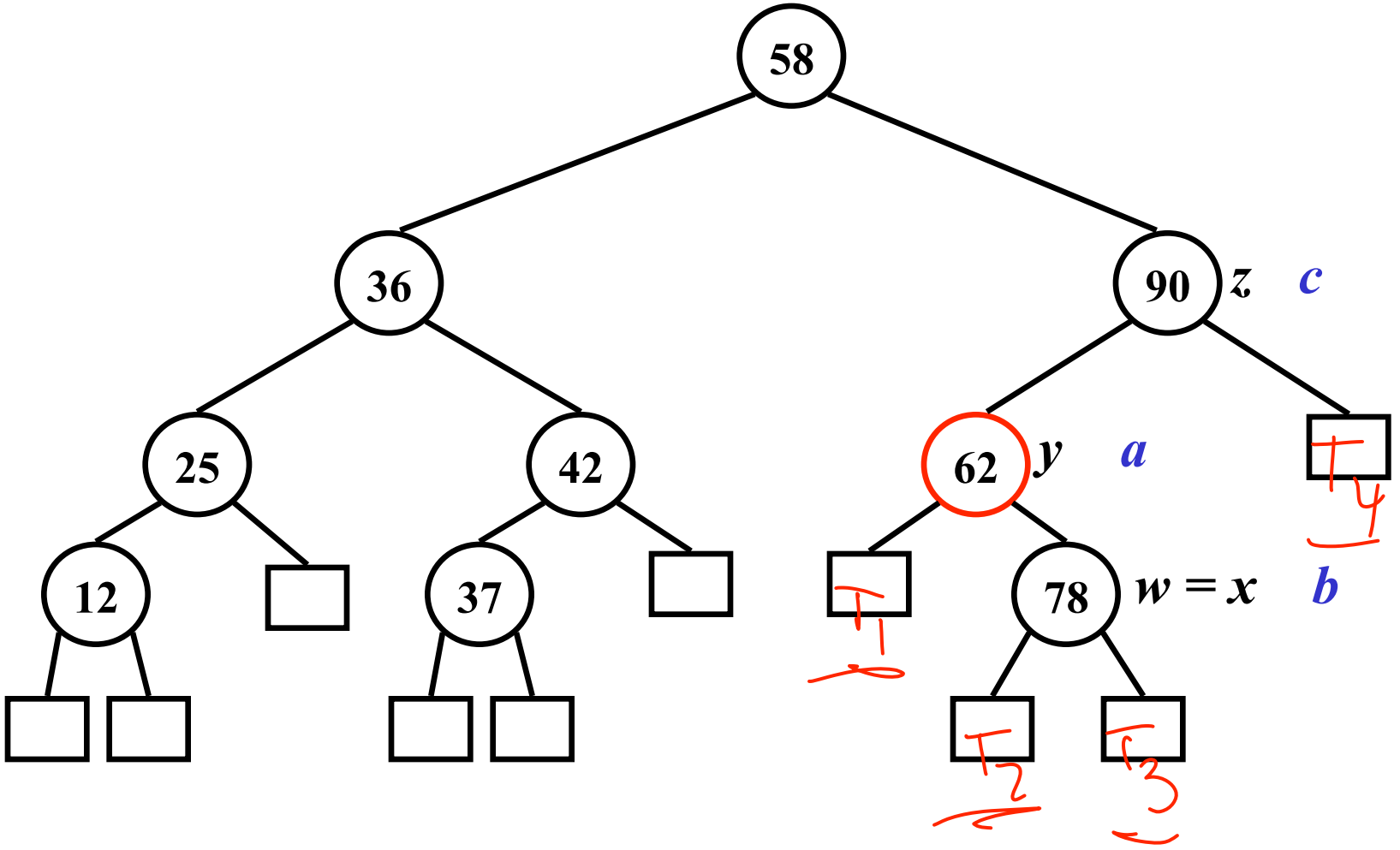


restructure(x)

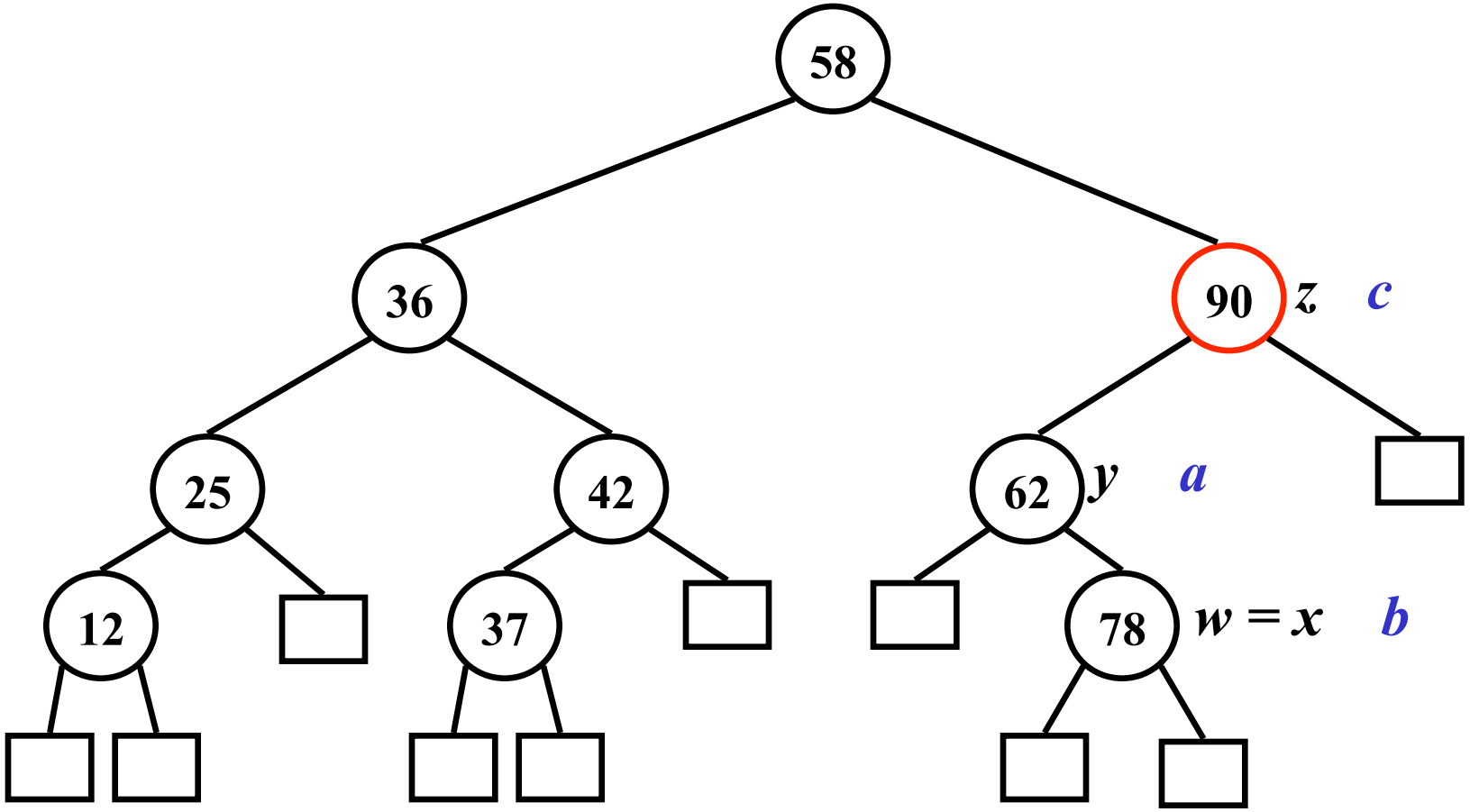
In this case $w = x$.
This is *not always* the case!



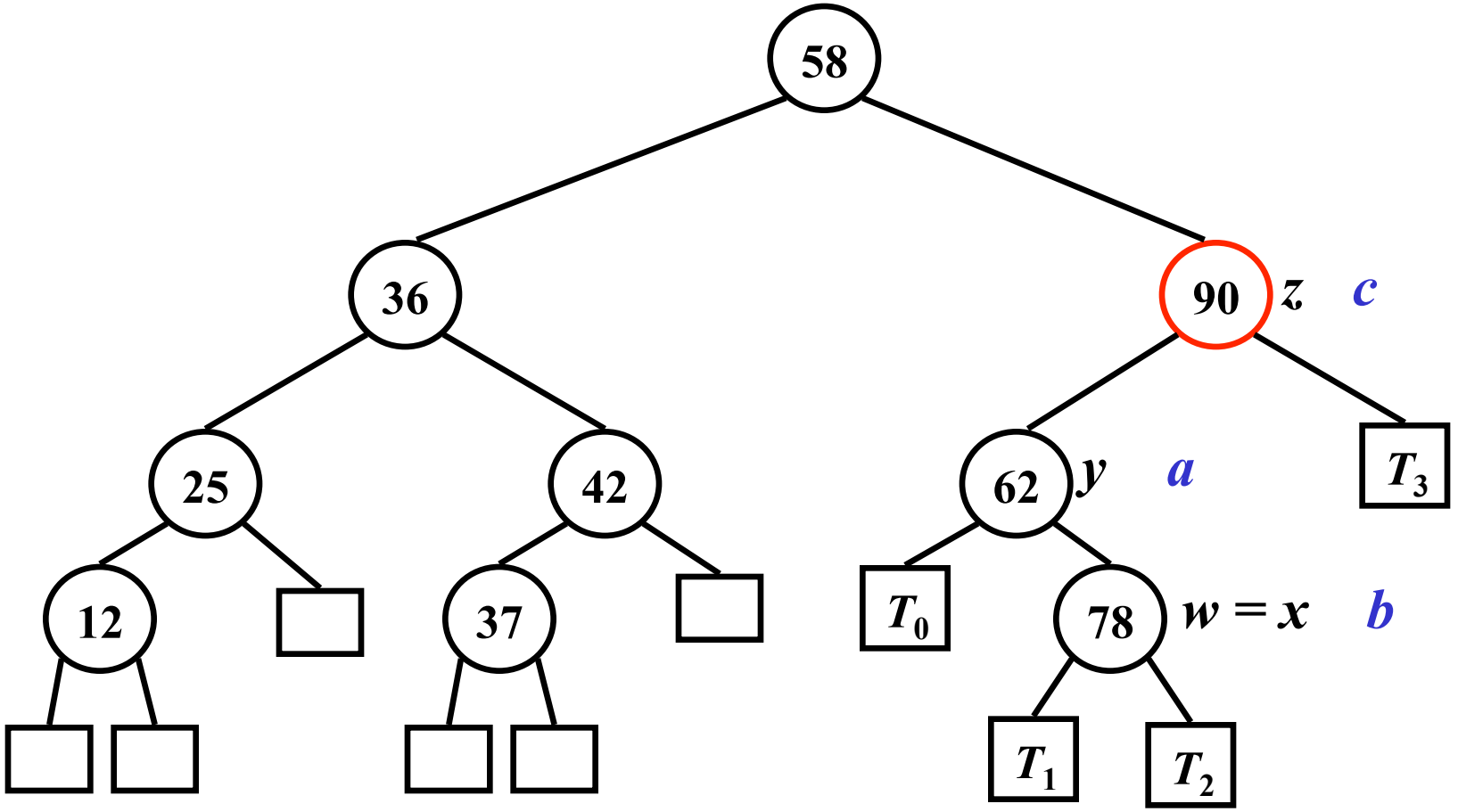
0. Let *a*, *b*, and *c* be a left to right inorder listing of *x*, *y*, and *z*



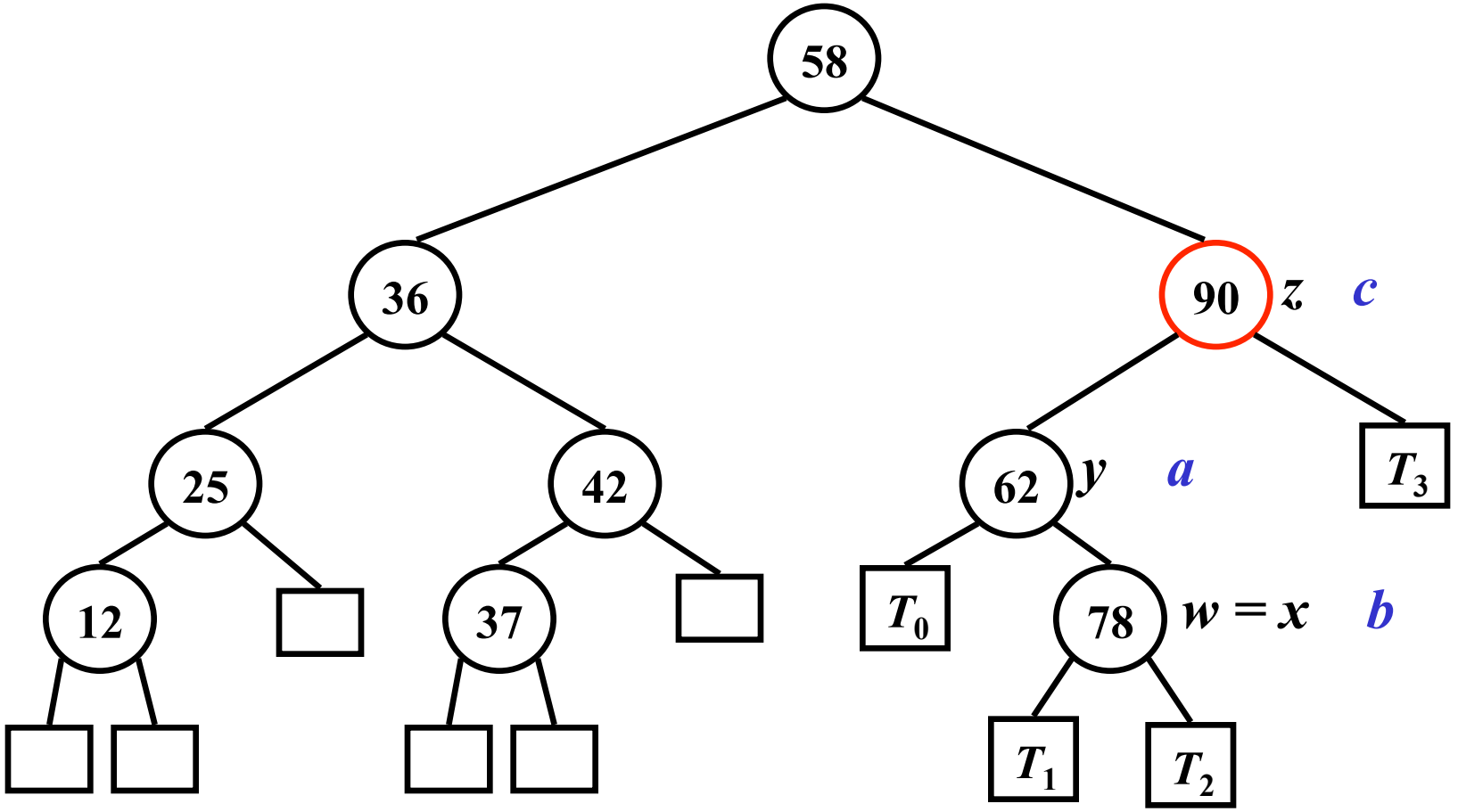
1. Let T_0, T_1, T_2, T_3 be subtrees rooted at x, y, and z



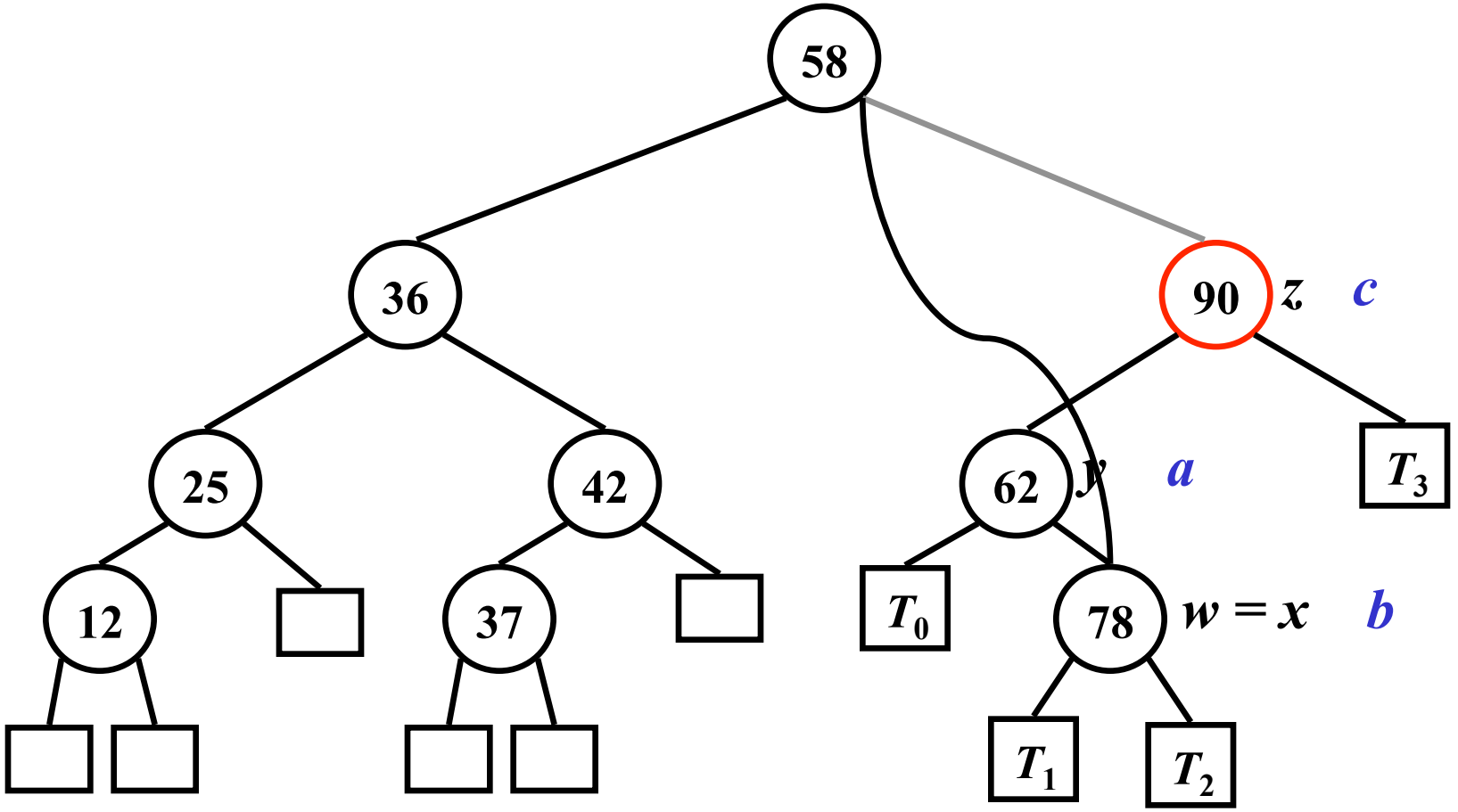
1. Let T_0, T_1, T_2, T_3 be subtrees rooted at x, y, and z



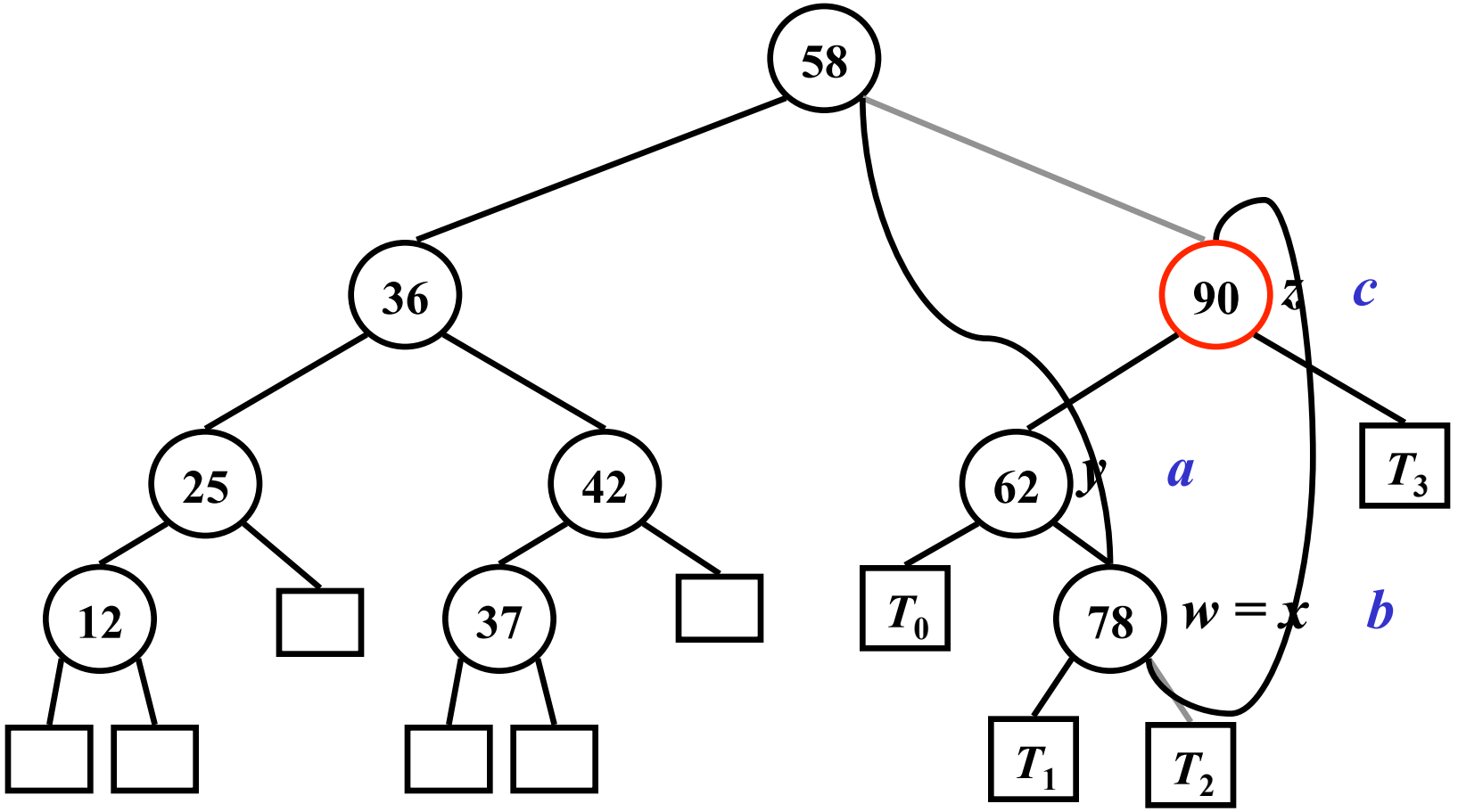
2. Restructure such that *b* becomes the new root



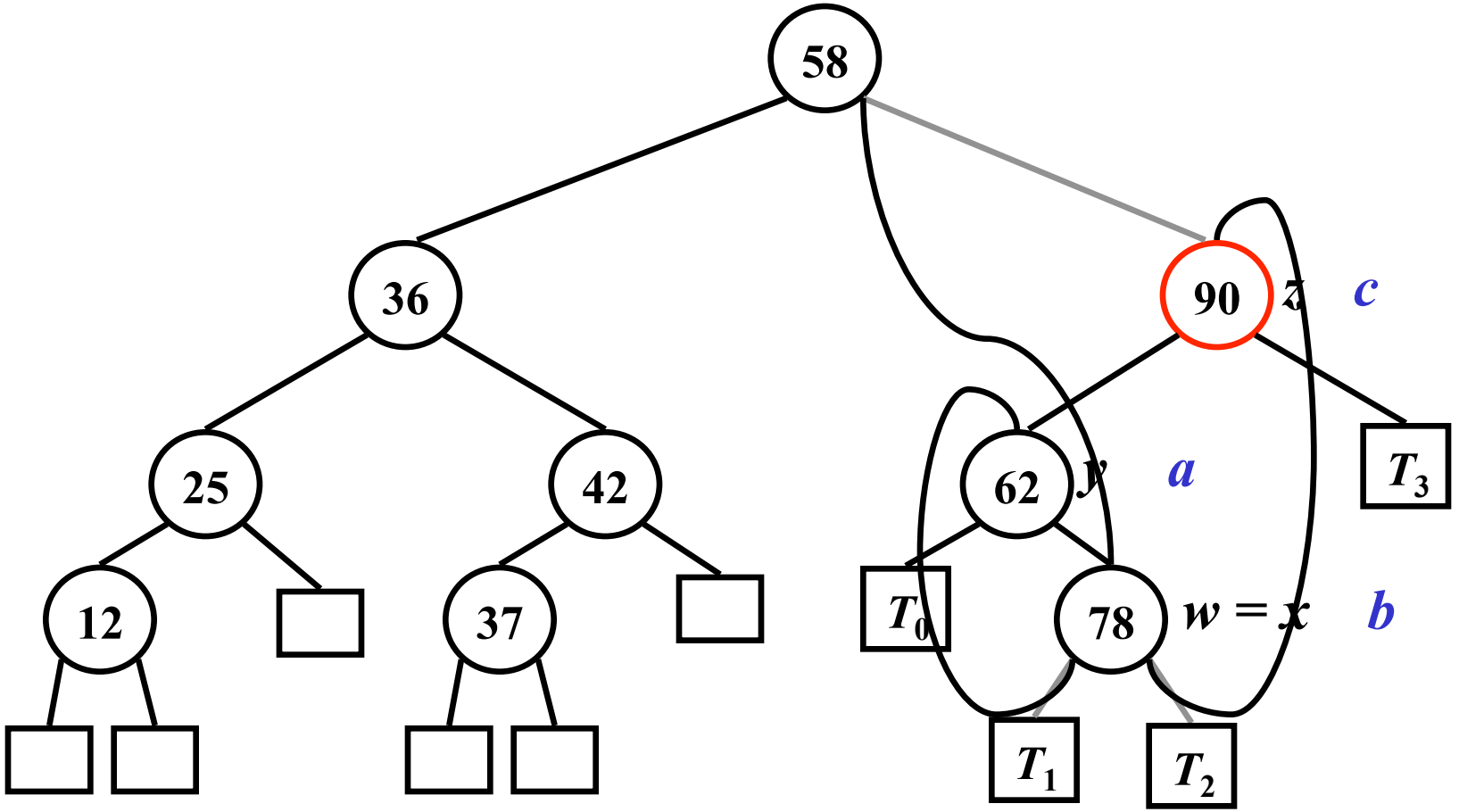
2. Restructure such that *b* becomes the new root



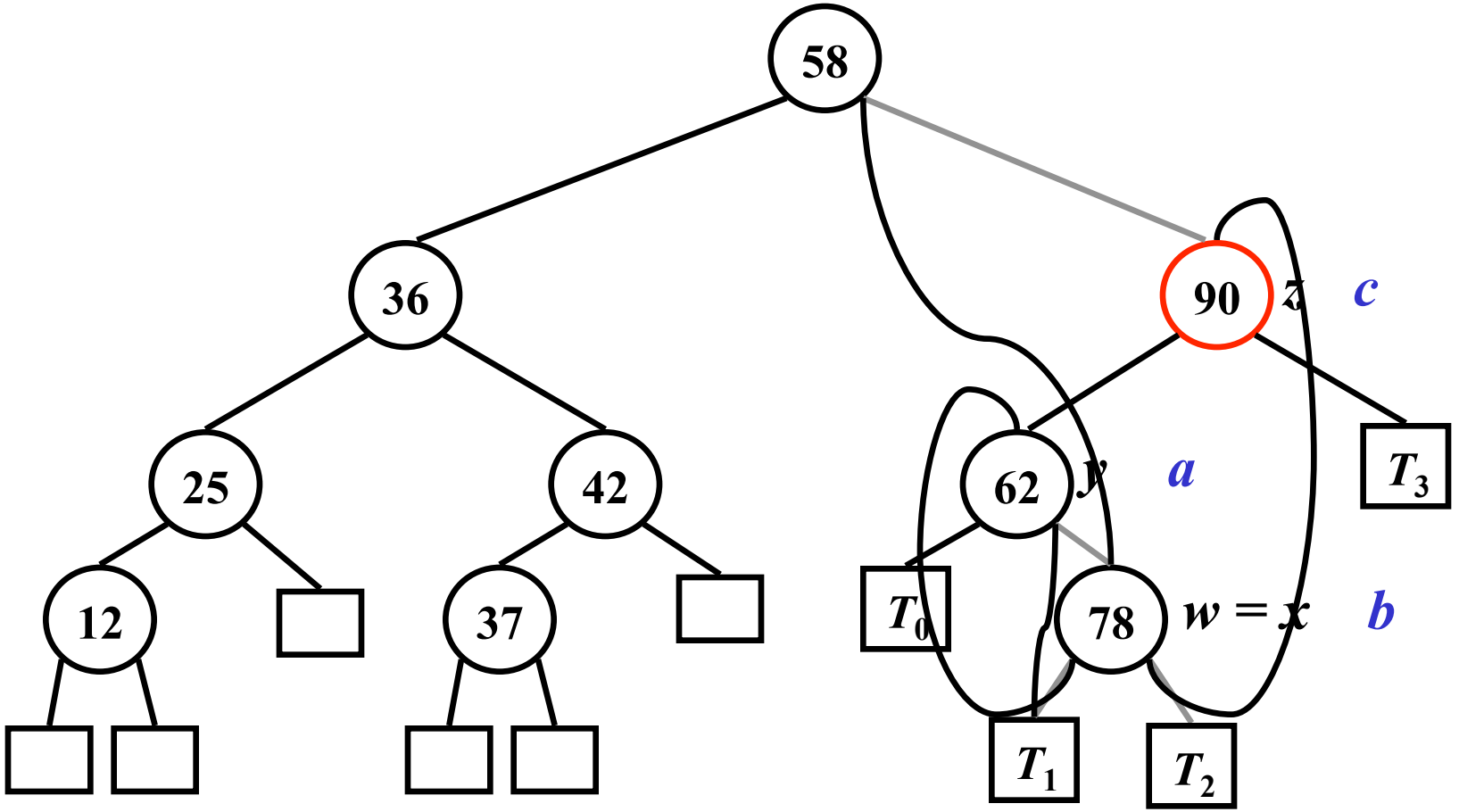
2. Restructure such that *b* becomes the new root



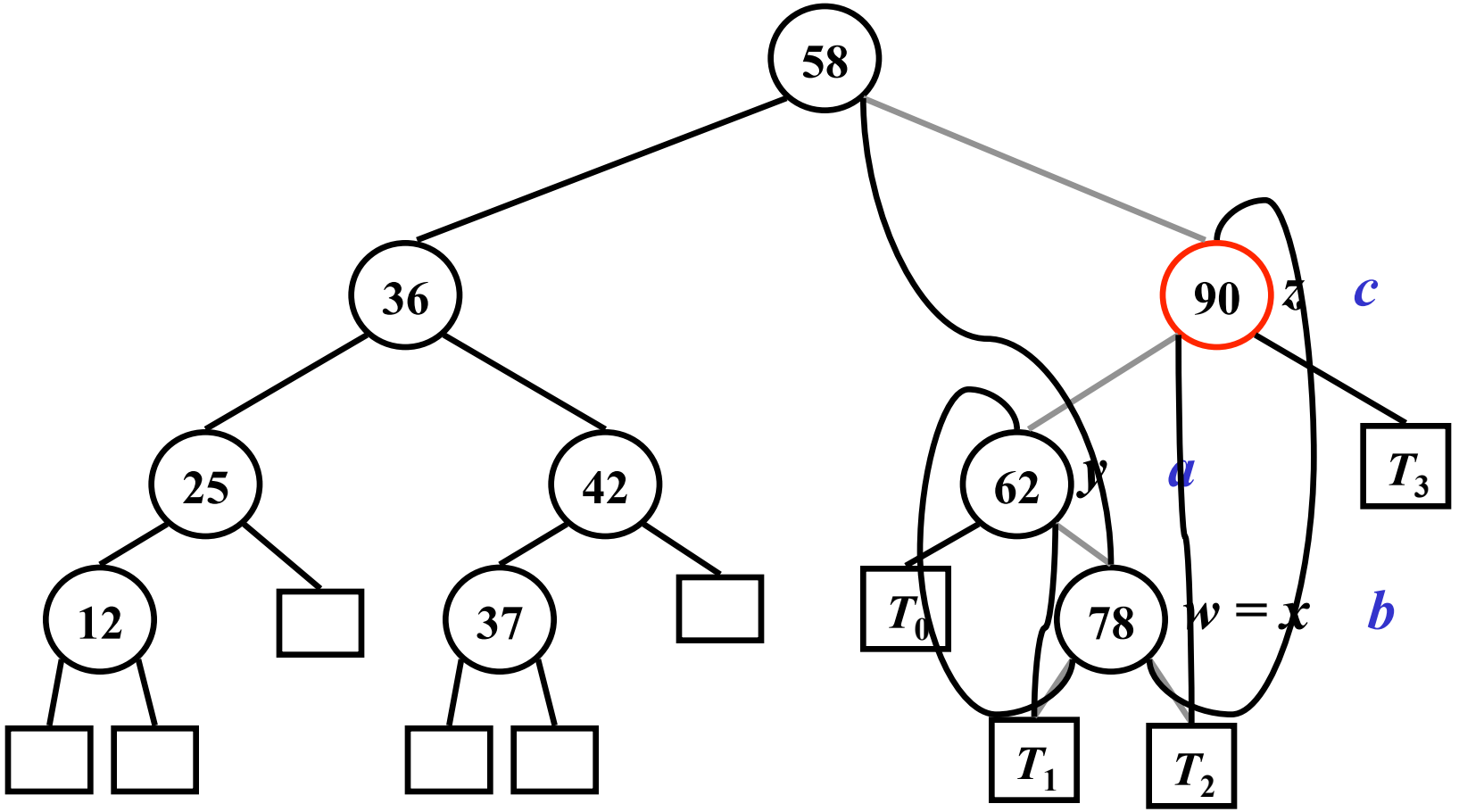
2. Restructure such that *b* becomes the new root



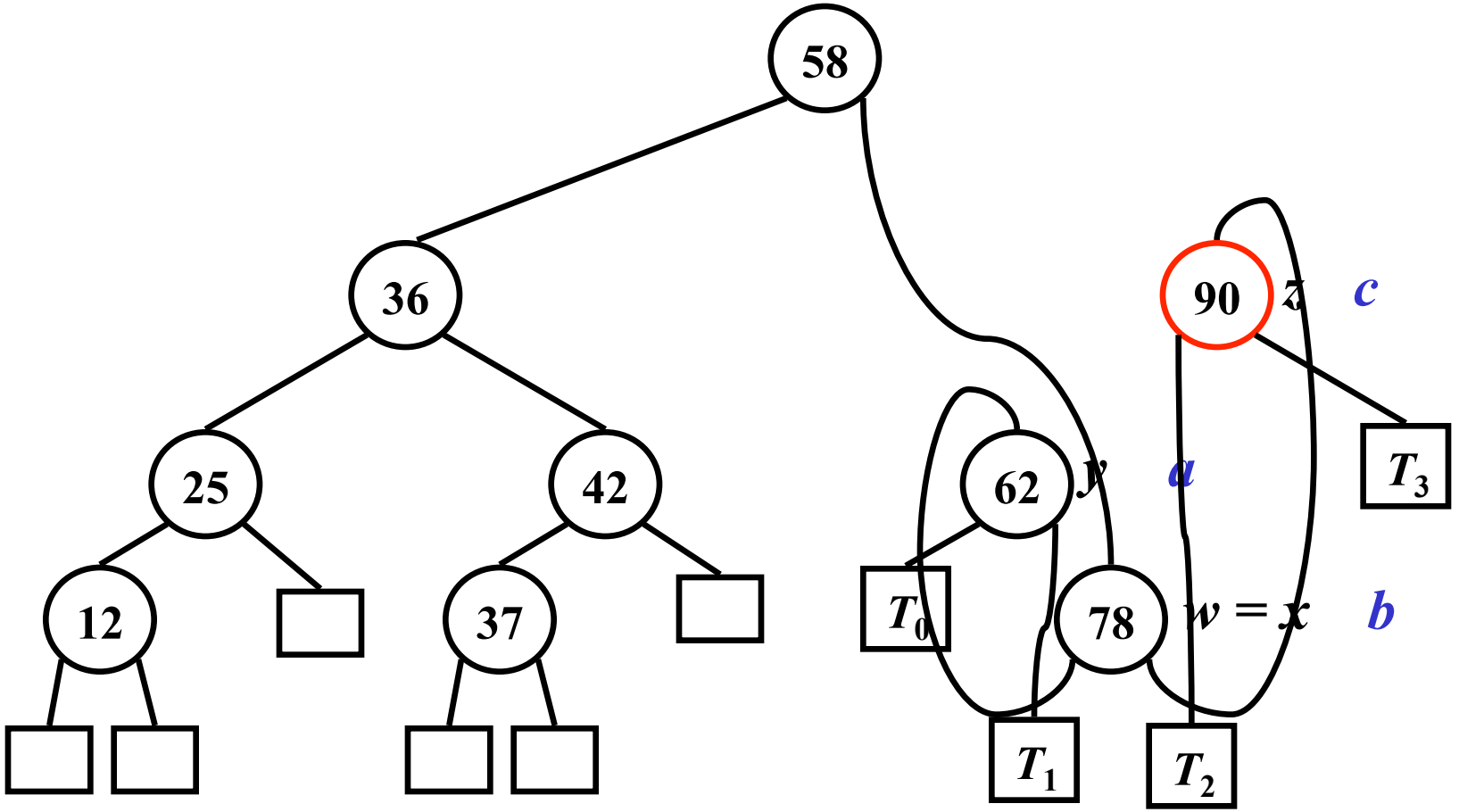
2. Restructure such that *b* becomes the new root



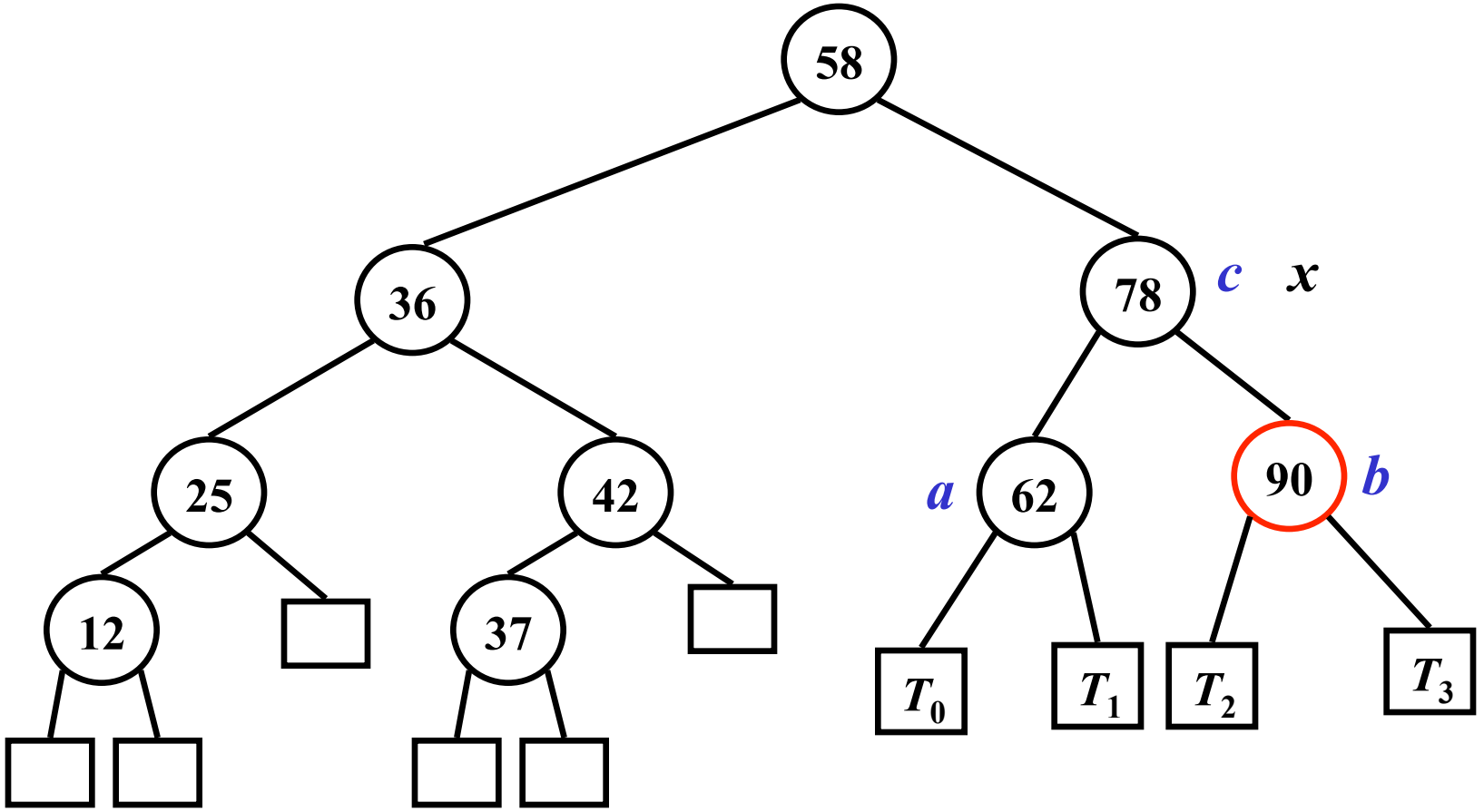
2. Restructure such that *b* becomes the new root



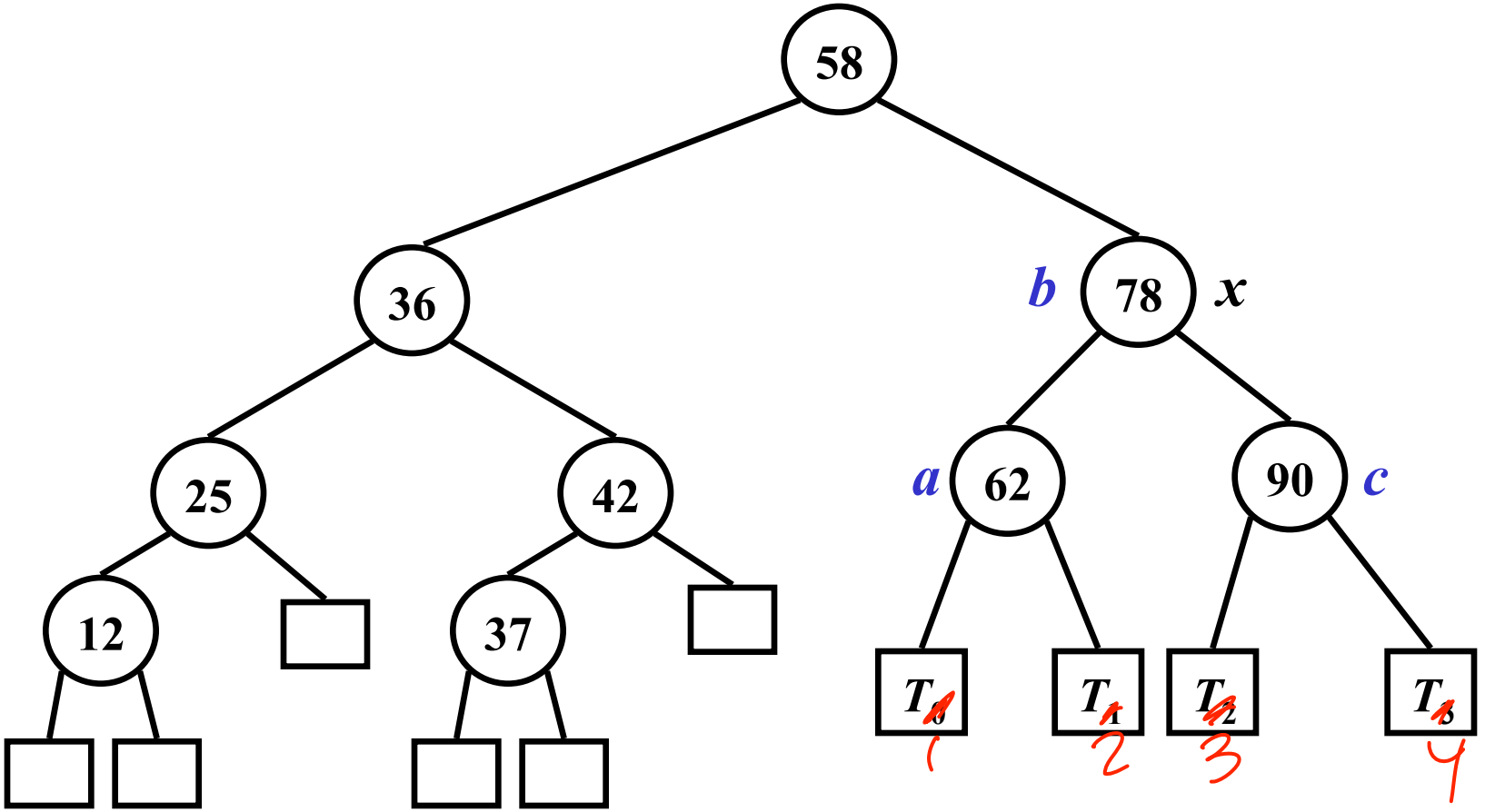
2. Restructure such that *b* becomes the new root



2. Restructure such that *b* becomes the new root



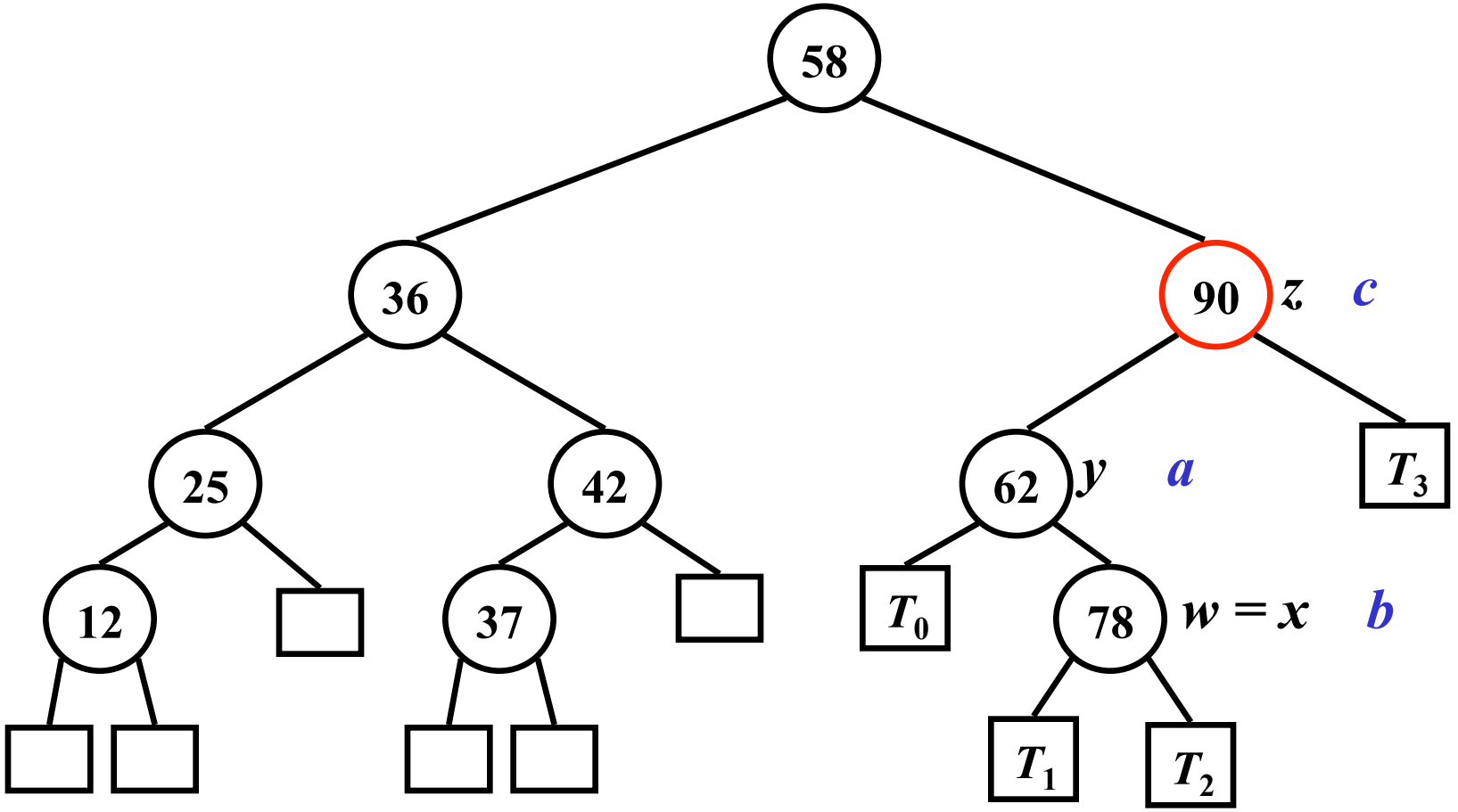
2. Restructure such that *b* becomes the new root



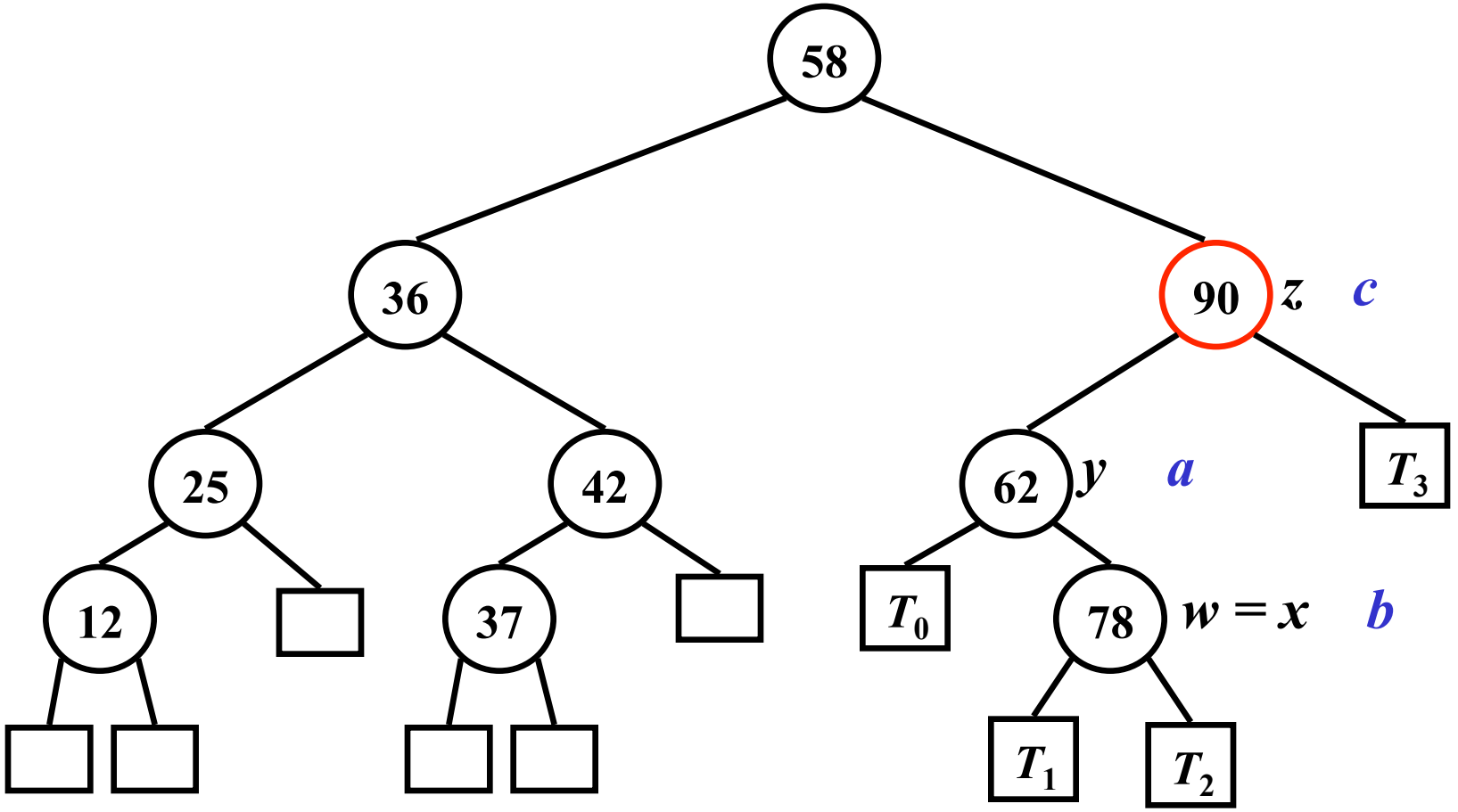
Algorithm restructure(x)

0. Let a, b, and c be a left to right inorder listing of x, y, and z.
1. Let T_0, T_1, T_2, T_3 be subtrees rooted at x, y, and z.
2. Replace the subtree rooted at z with a new subtree rooted at b.
3. Now a is the left child of b, T_0 and T_1 are the left and right subtrees of a. Node c is the right child of b, and T_2 and T_3 are the left and right subtrees of c.

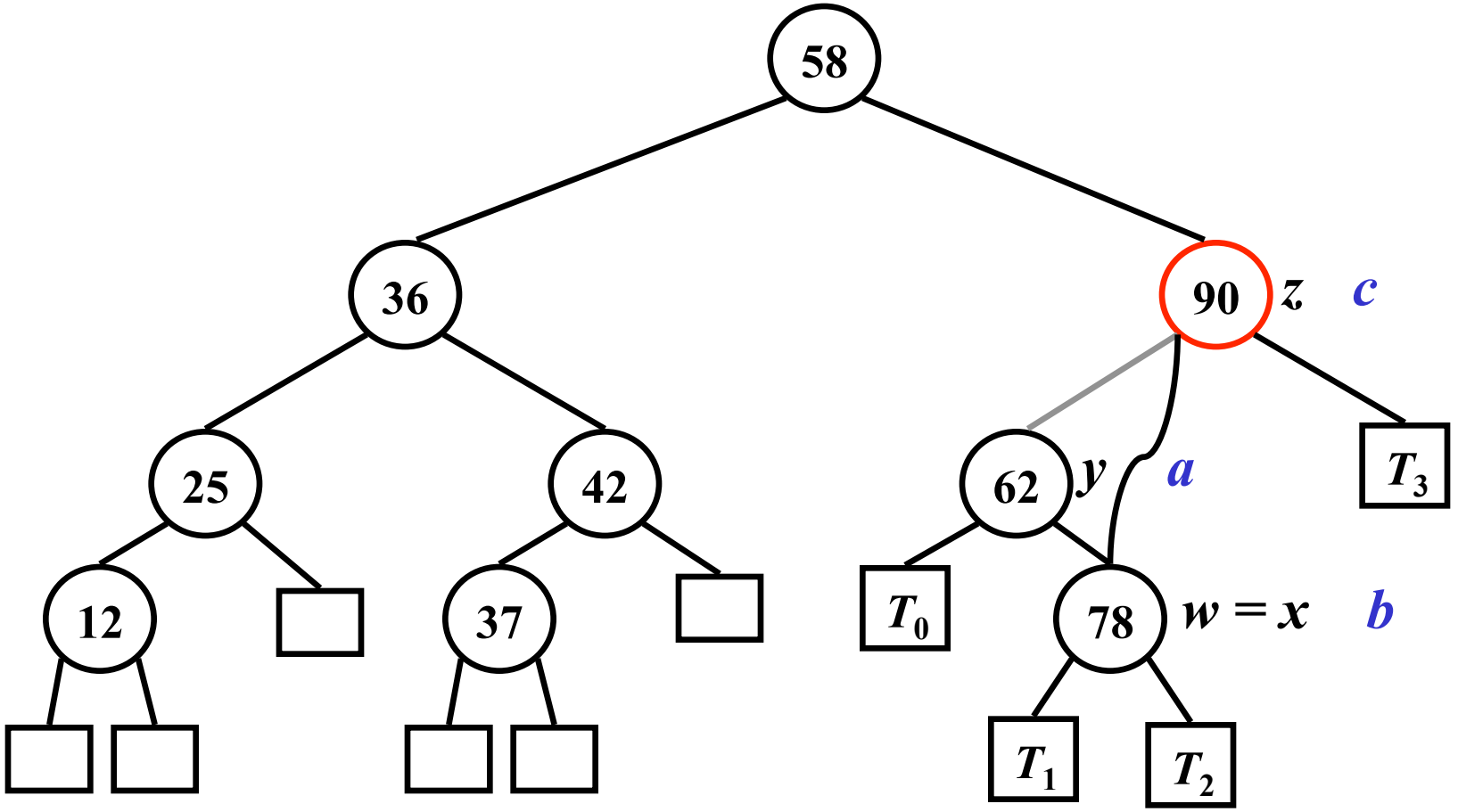
Another view: *Rotate until balanced.*



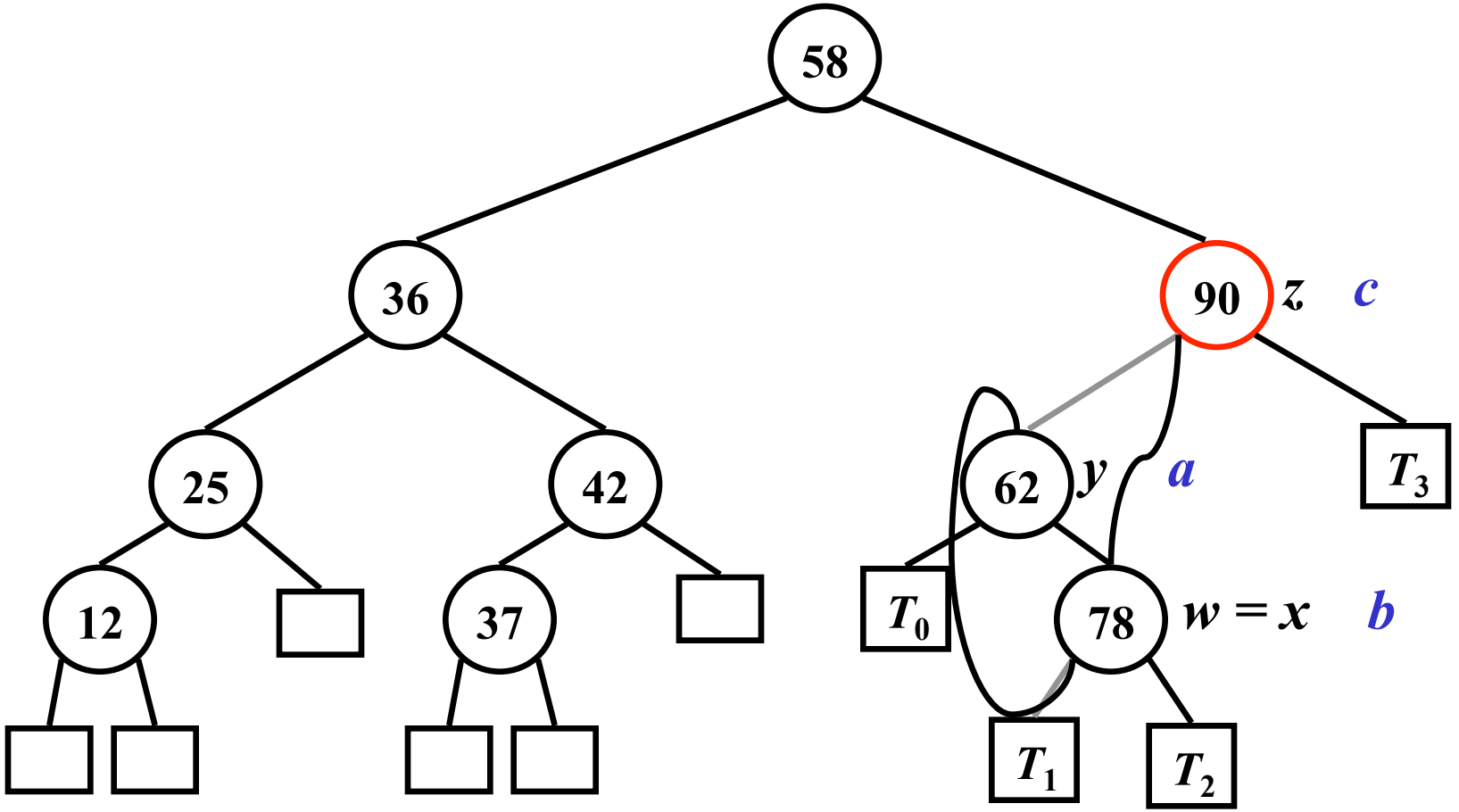
Rotation 1: Rotate such that *b* moves up by a level



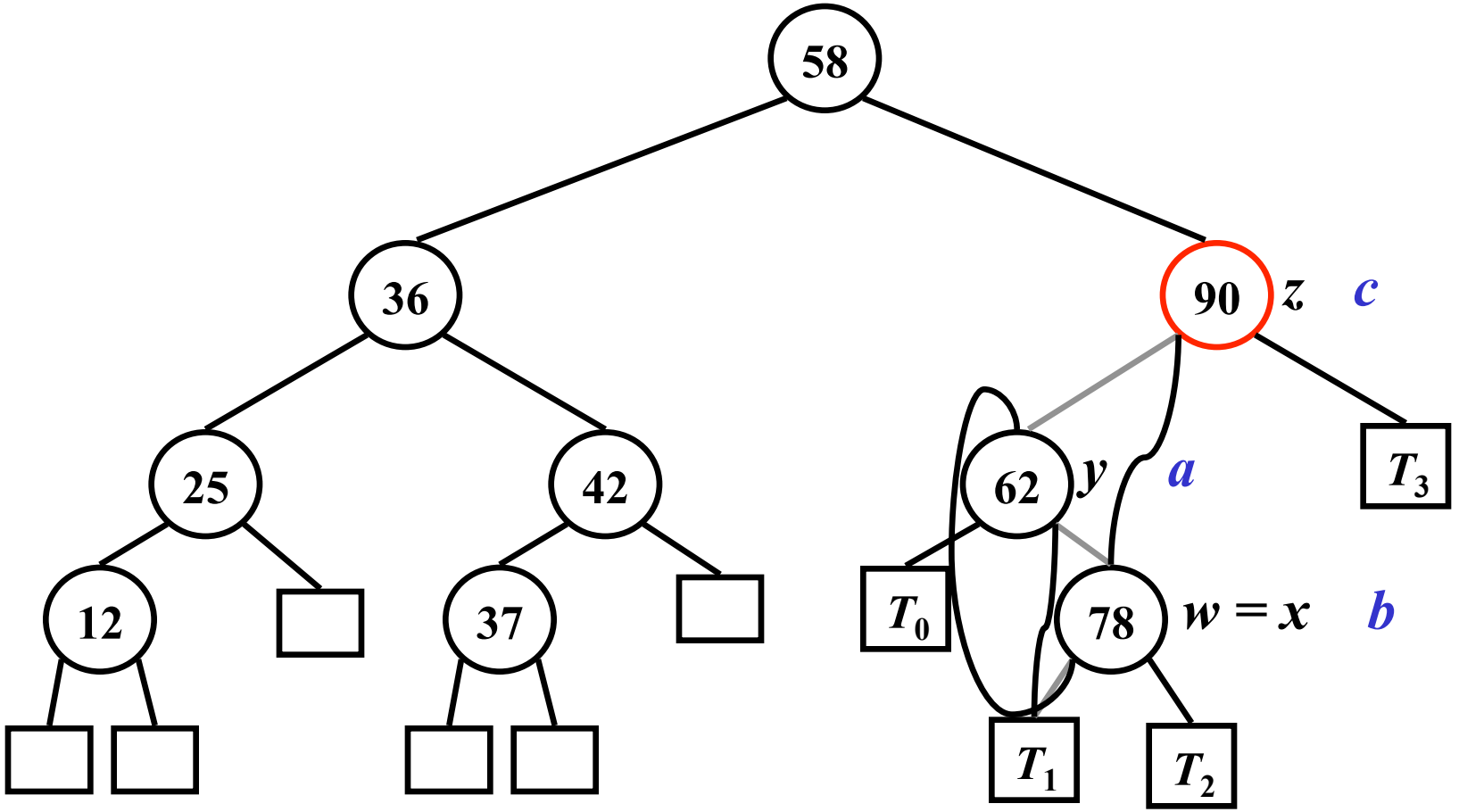
Rotation 1: Rotate such that *b* moves up by a level



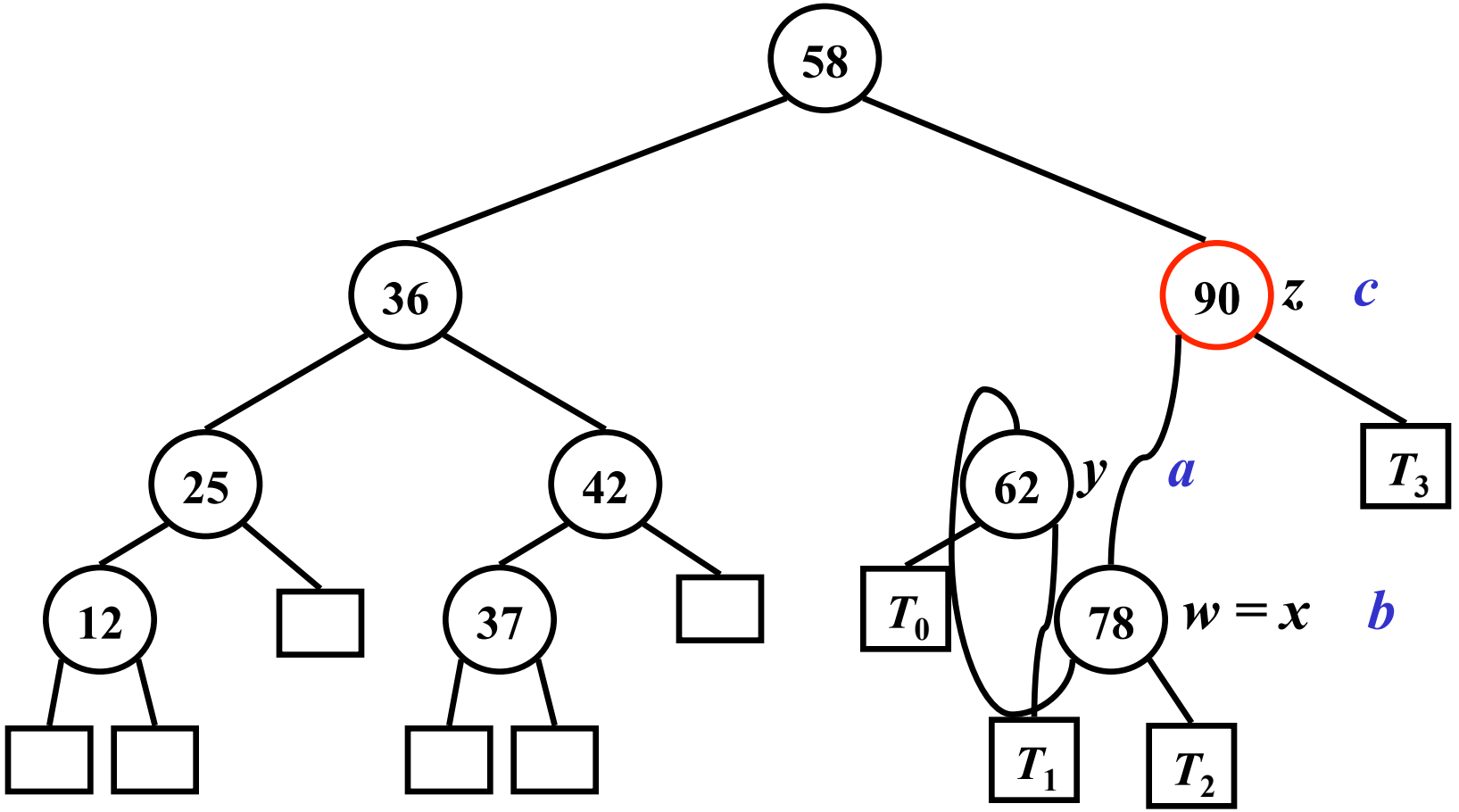
Rotation 1: Rotate such that *b* moves up by a level



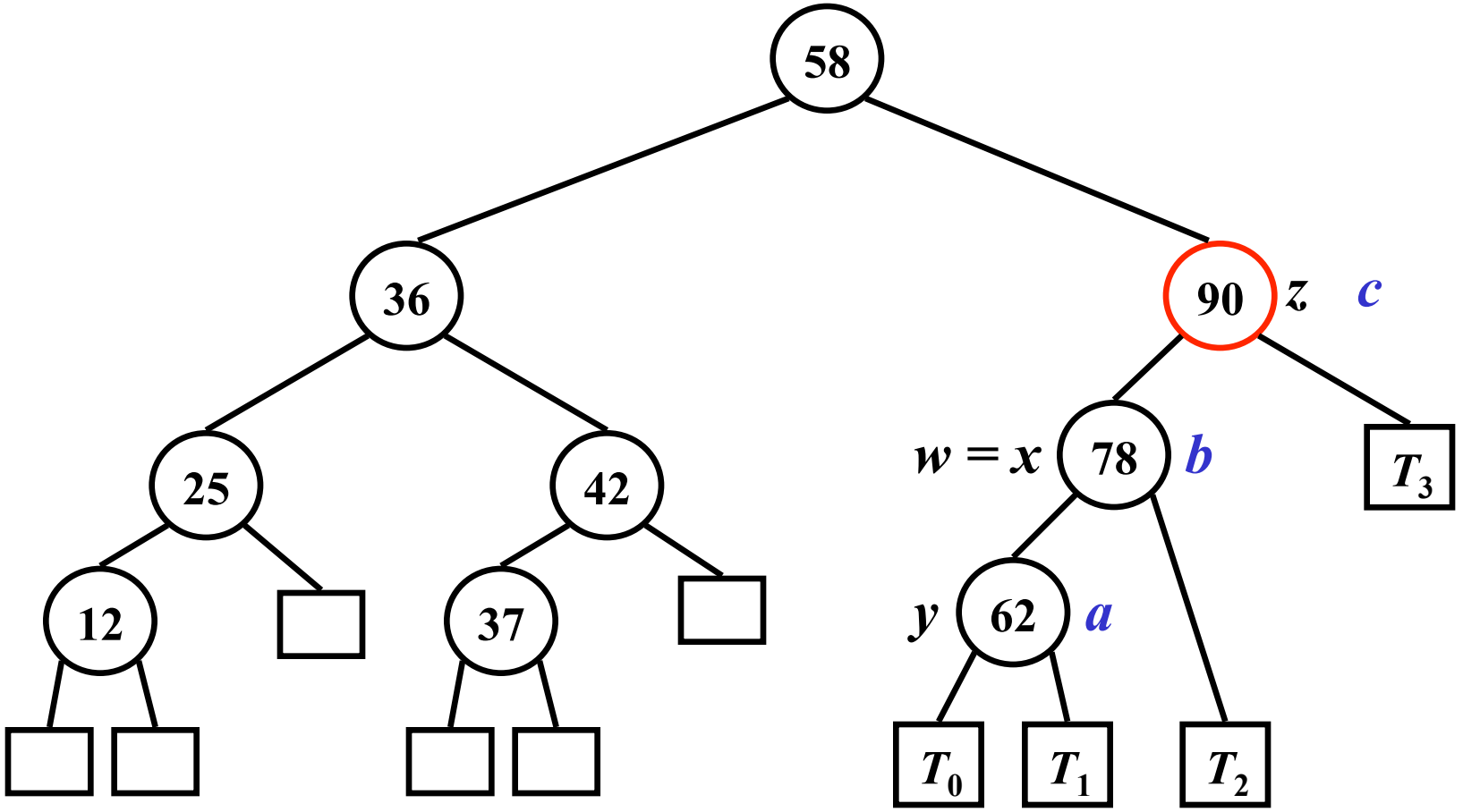
Rotation 1: Rotate such that *b* moves up by a level



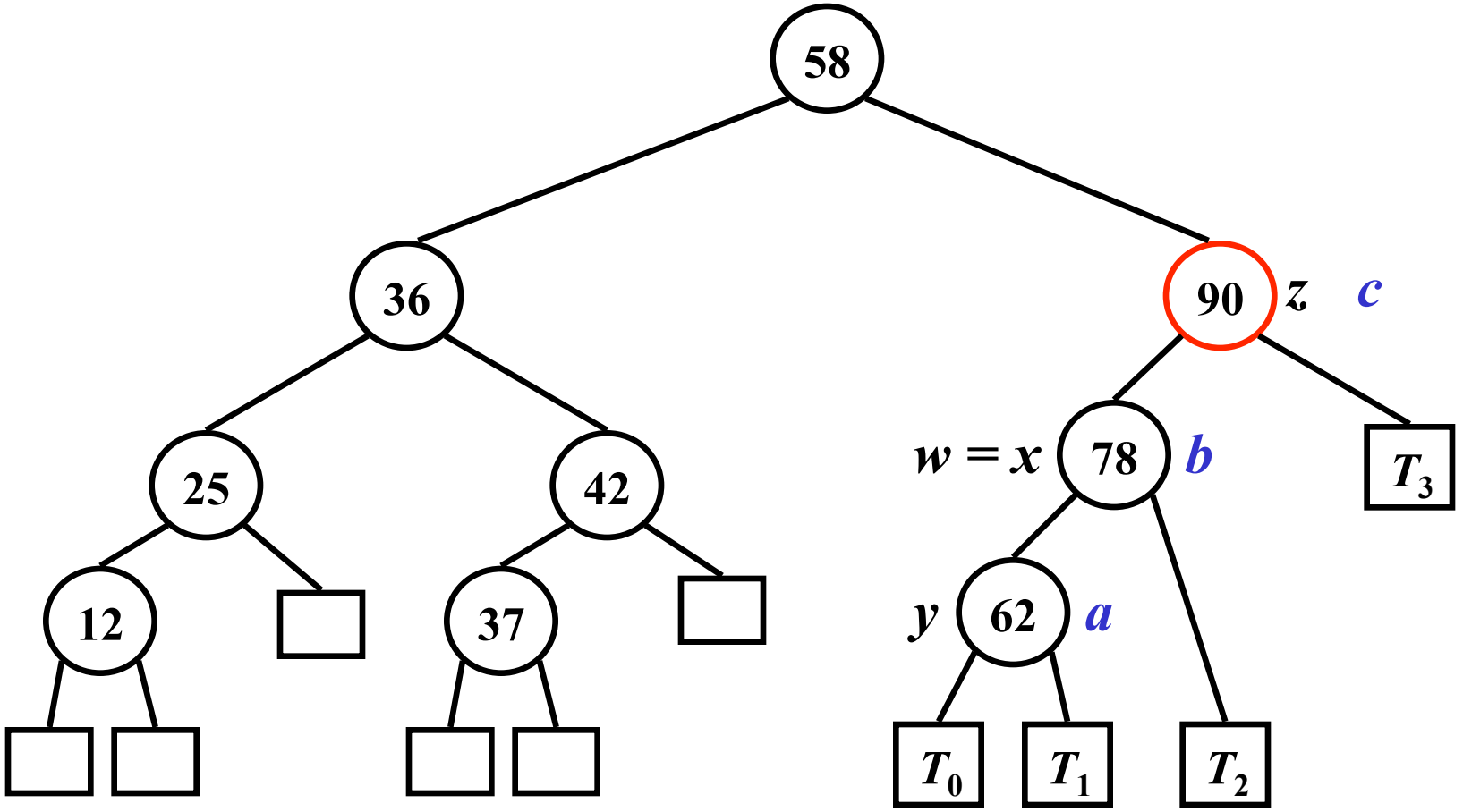
Rotation 1: Rotate such that *b* moves up by a level



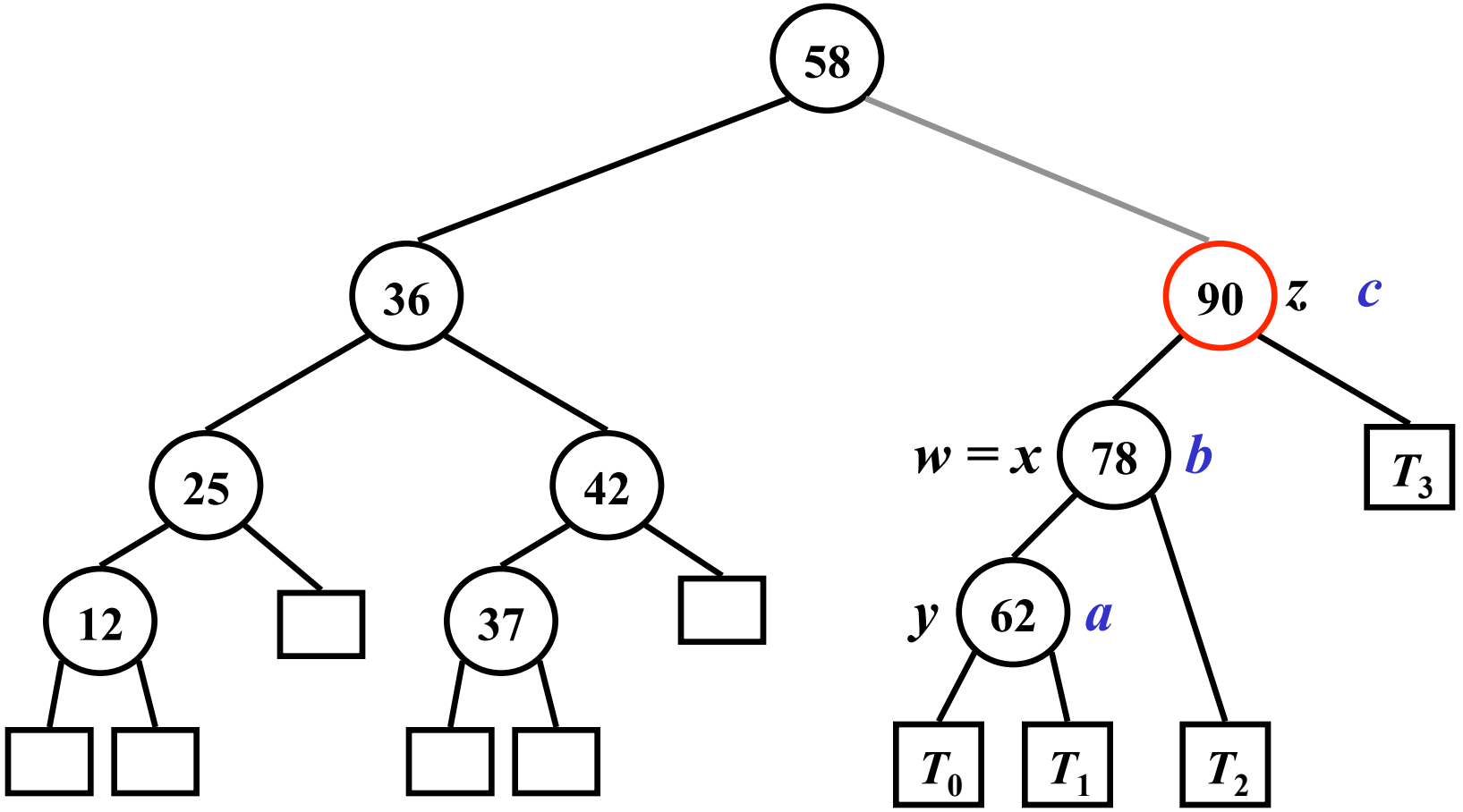
Rotation 1: Rotate such that *b* moves up by a level



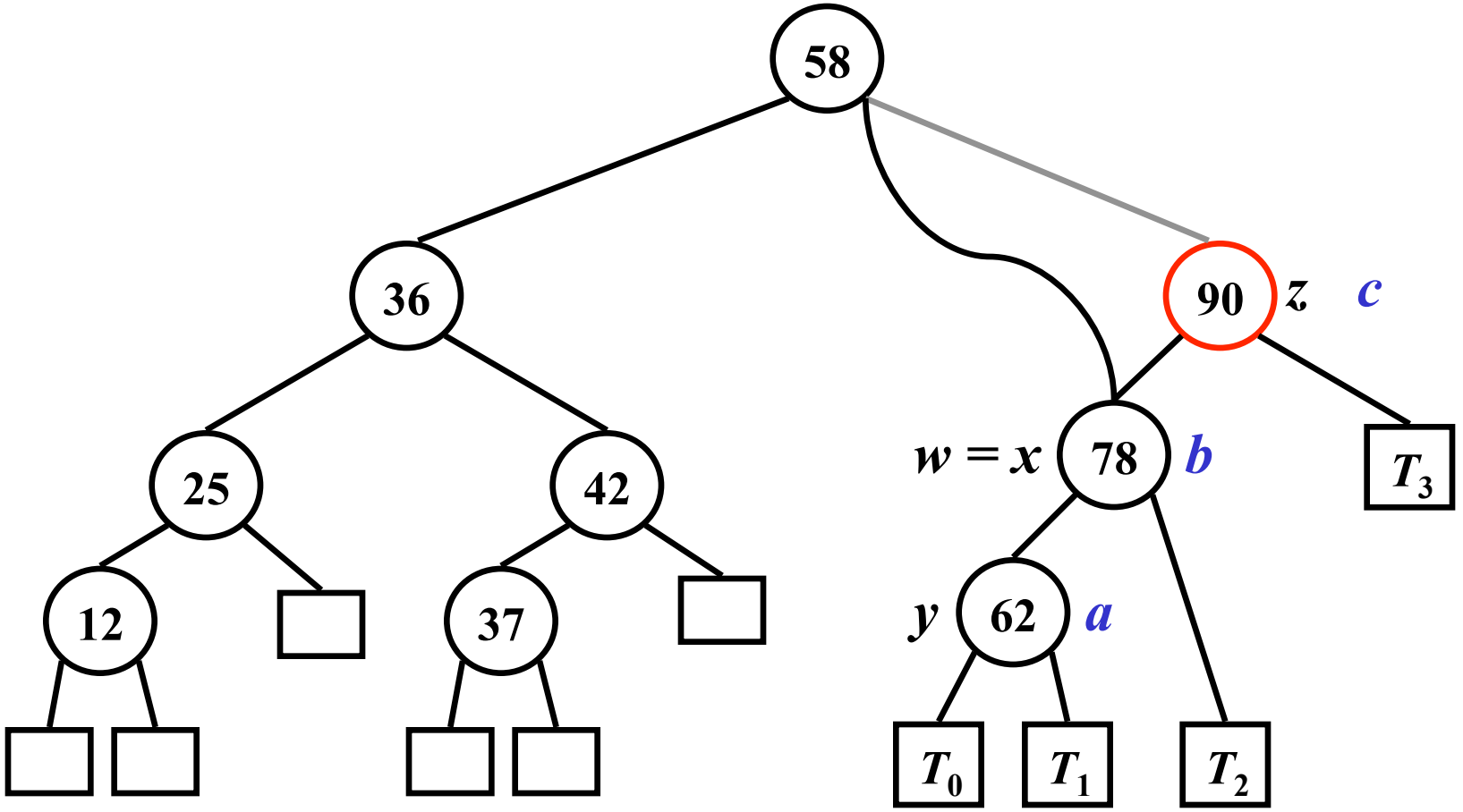
Rotation 2: Rotate such that *b* moves up by another level



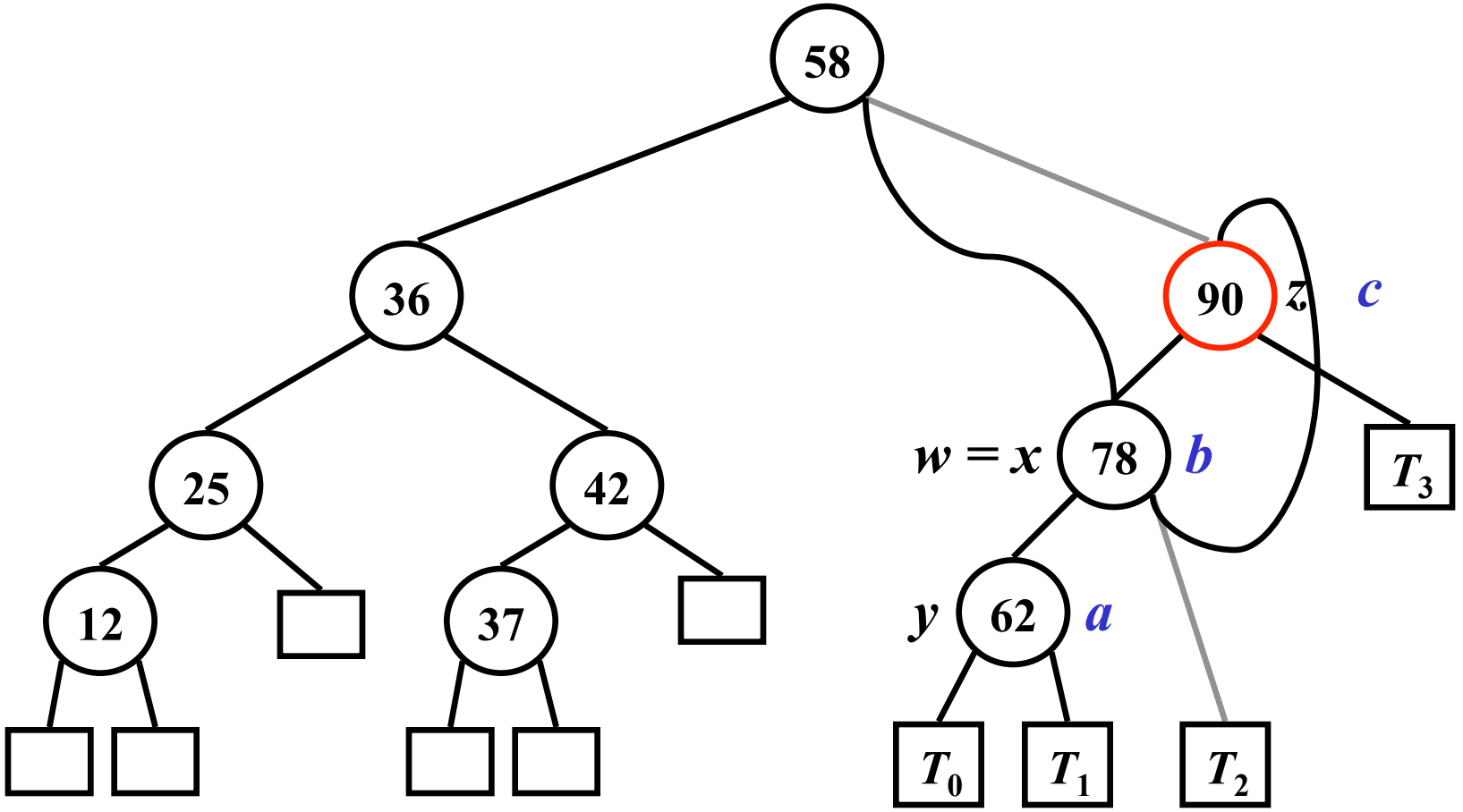
Rotation 2: Rotate such that *b* moves up by another level



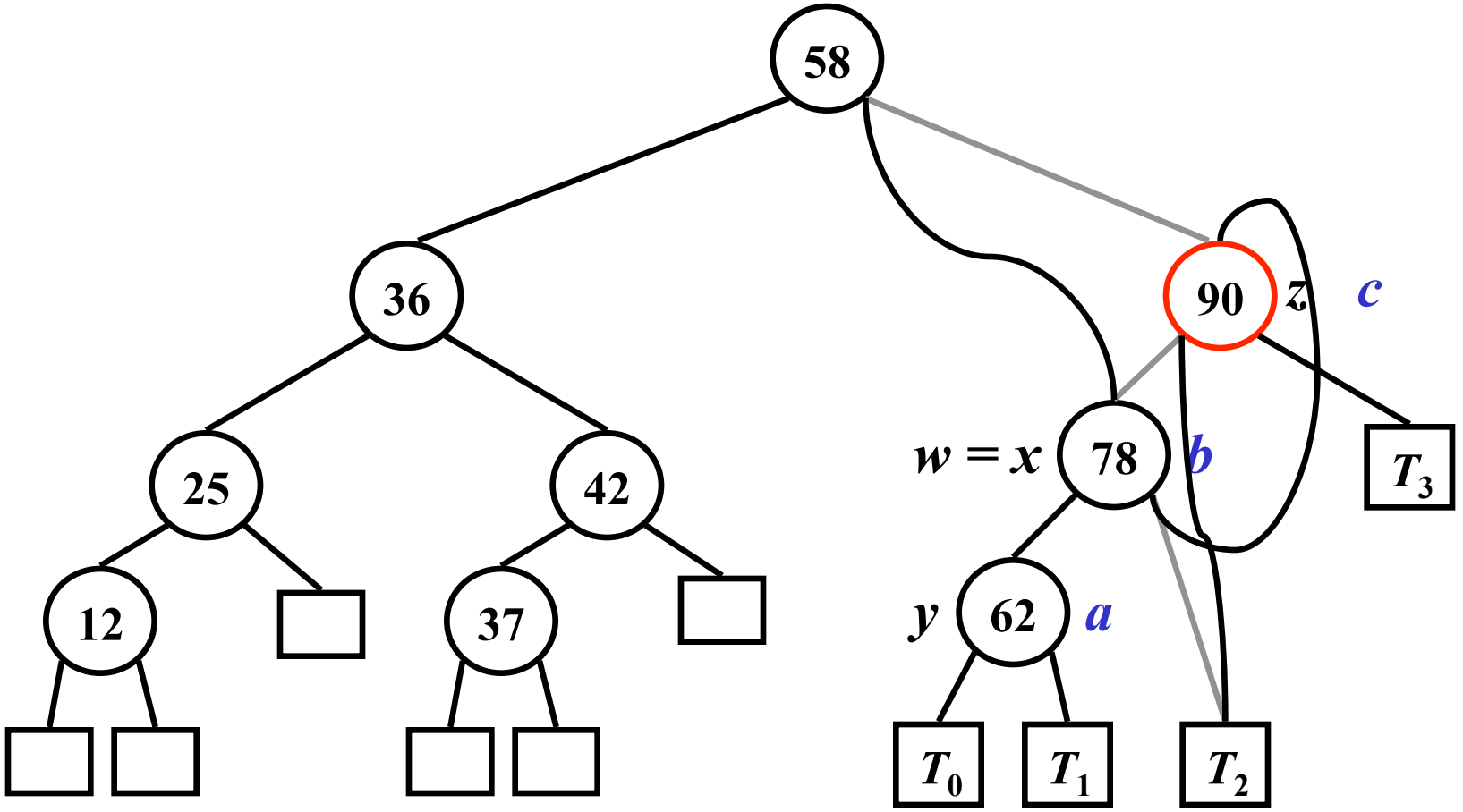
Rotation 2: Rotate such that *b* moves up by another level



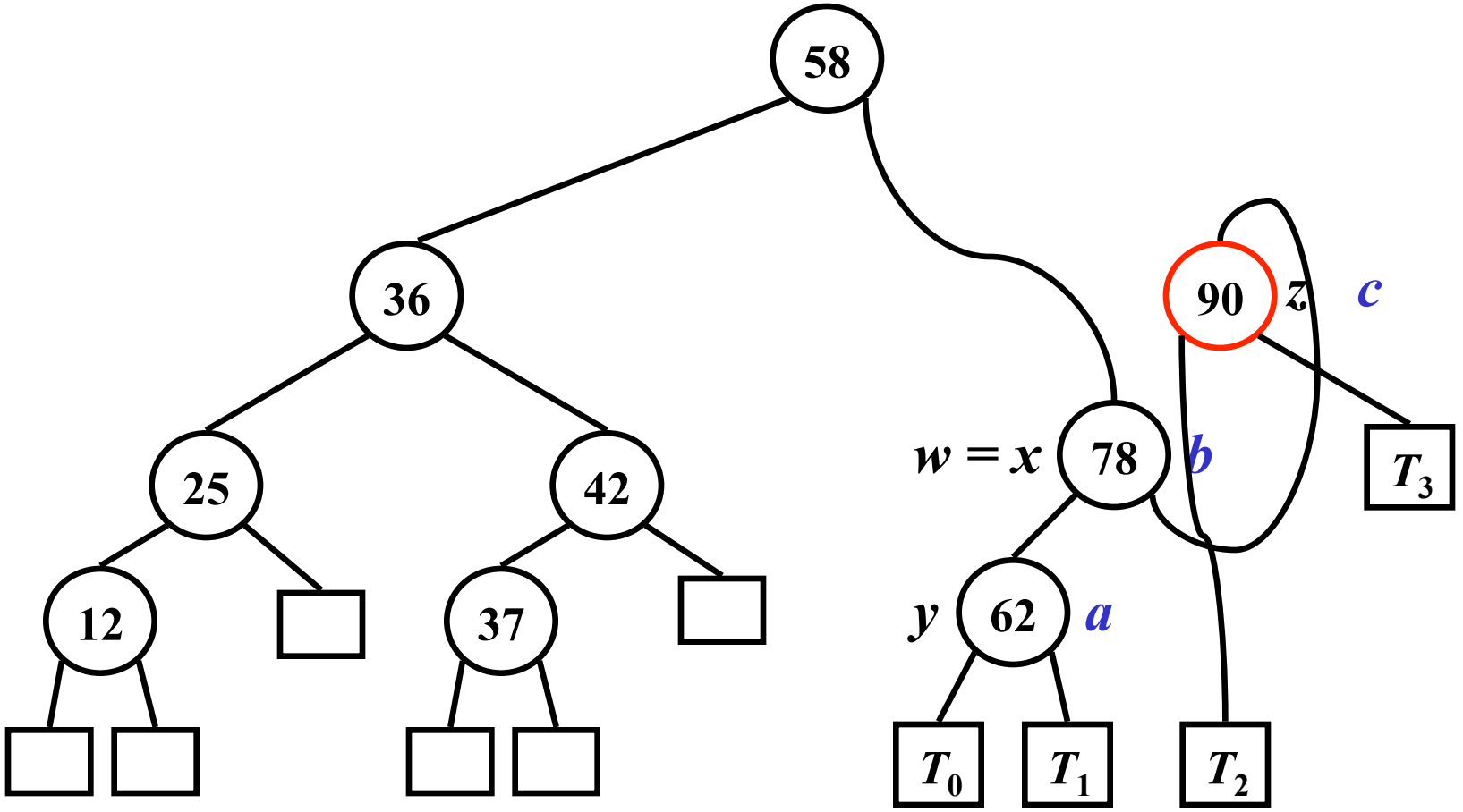
Rotation 2: Rotate such that *b* moves up by another level



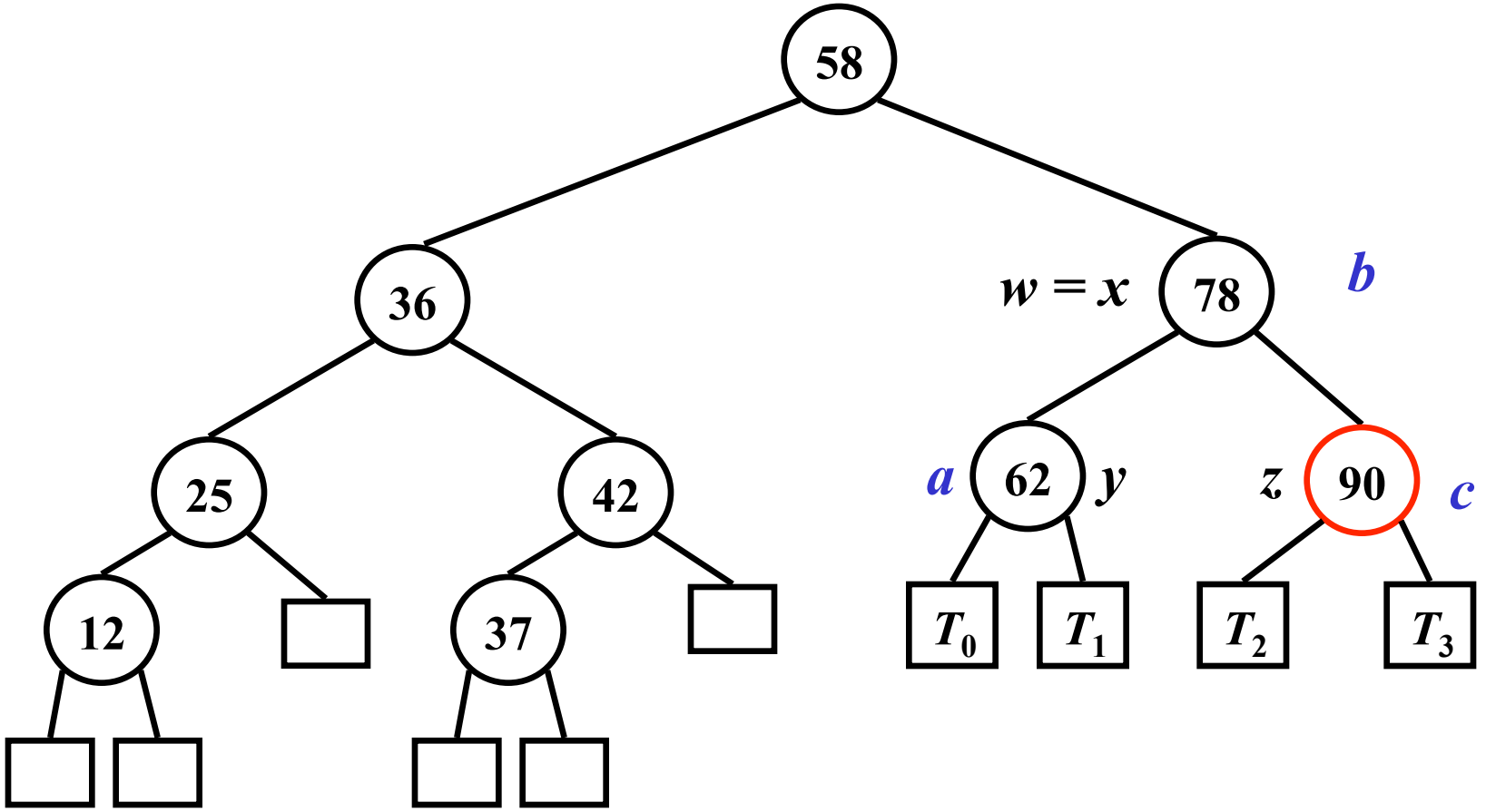
Rotation 2: Rotate such that *b* moves up by another level



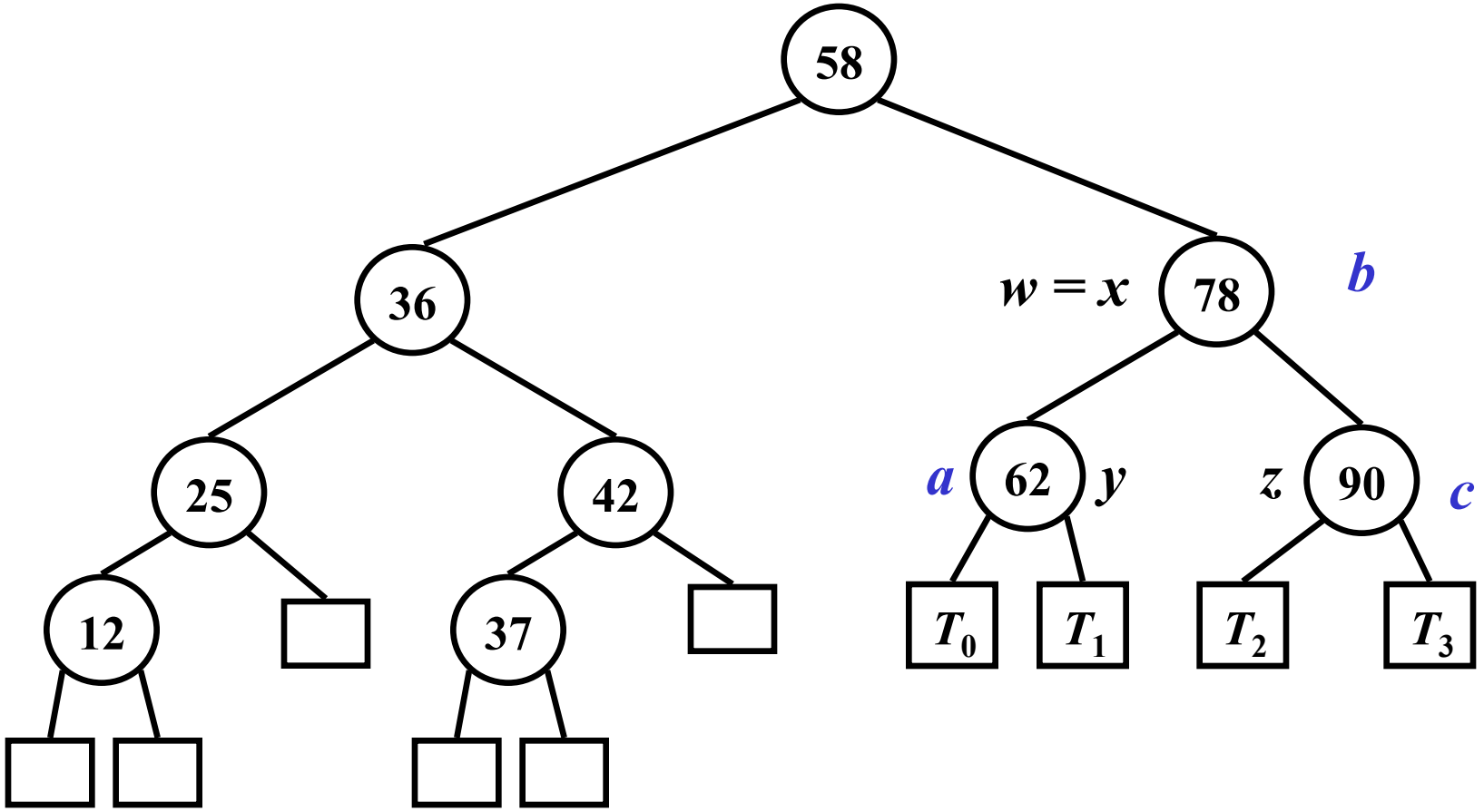
Rotation 2: Rotate such that *b* moves up by another level



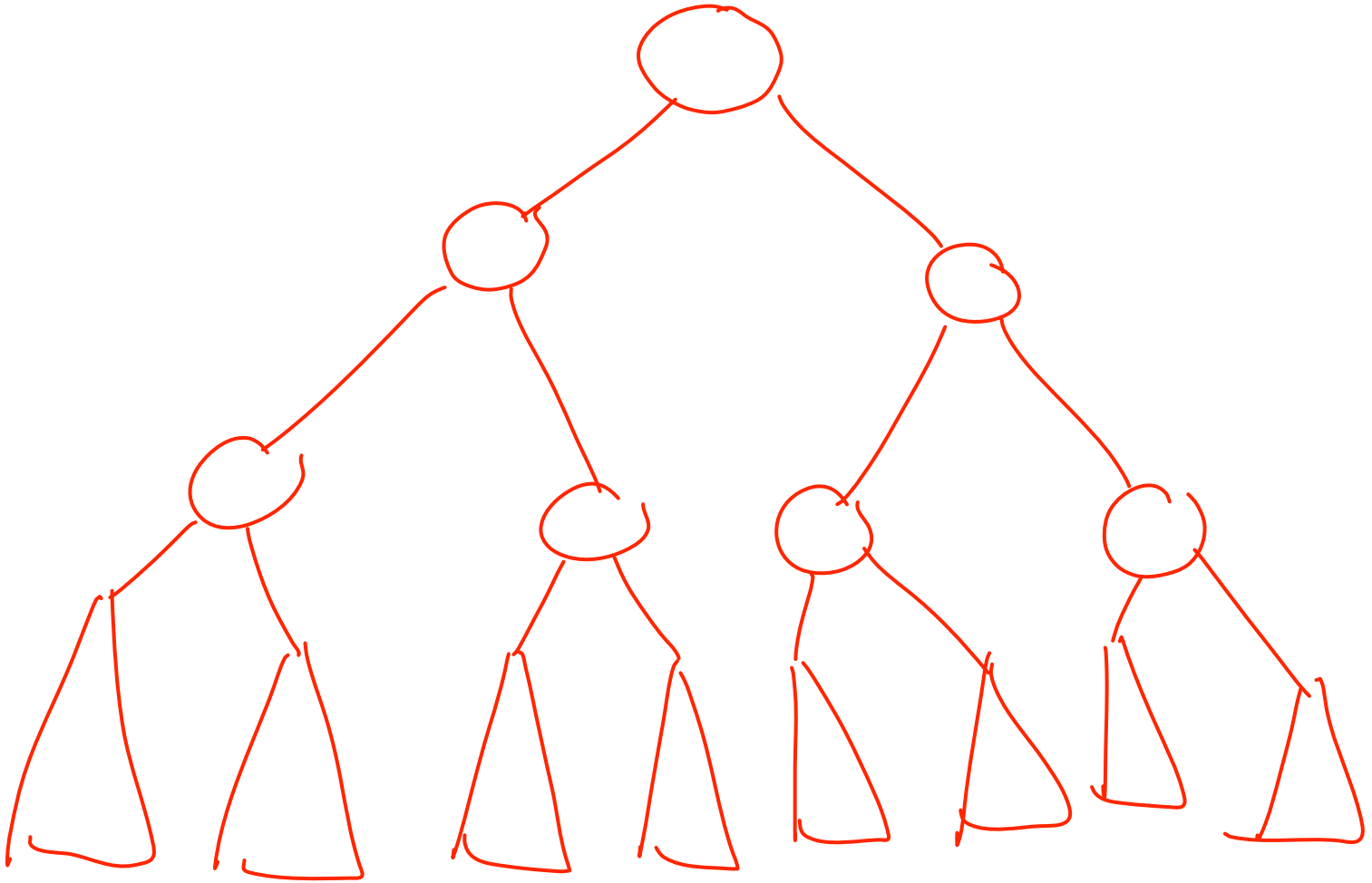
Rotation 2: Rotate such that *b* moves up by another level



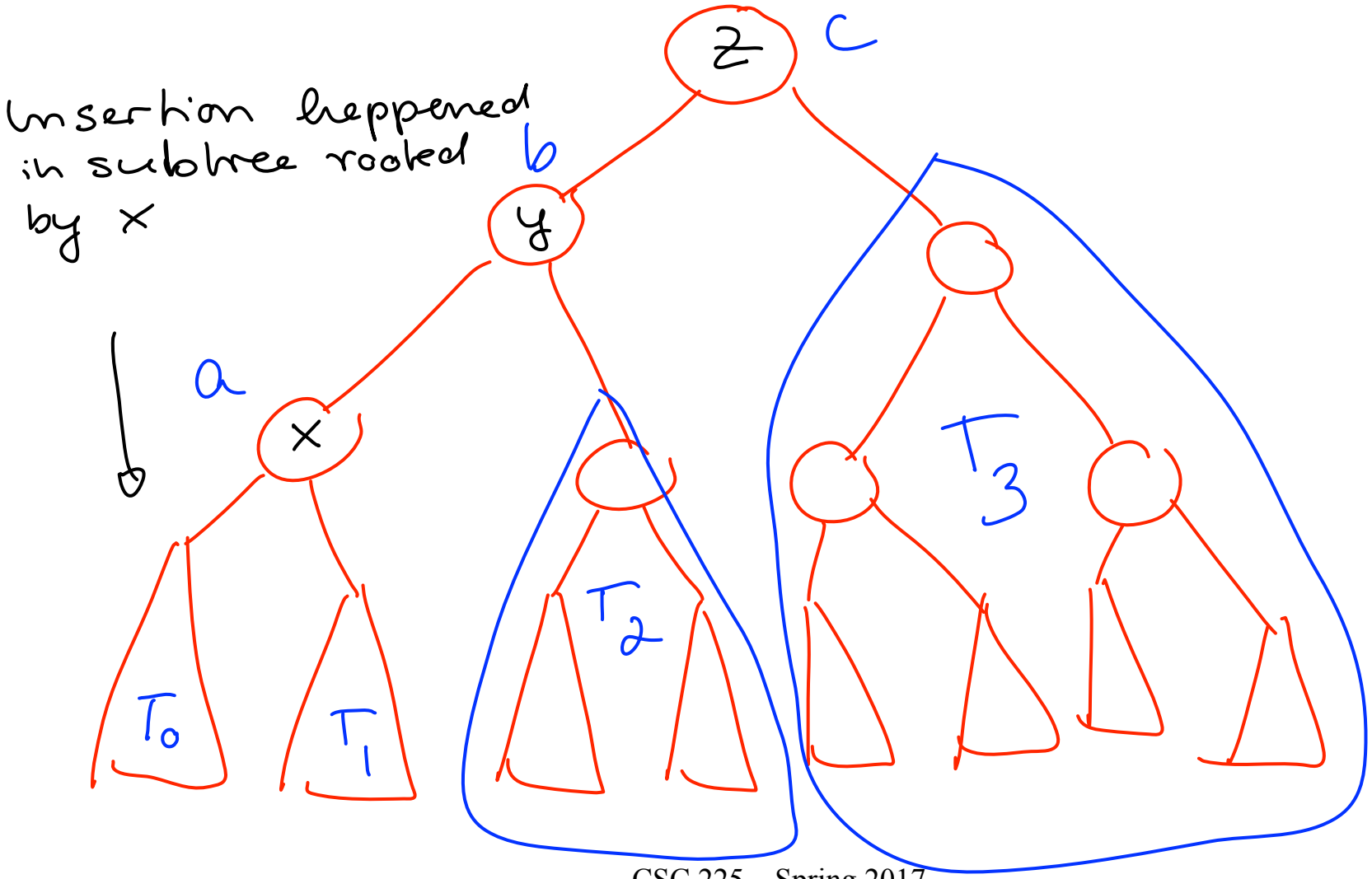
Rotation 2: Rotate such that *b* moves up by another level



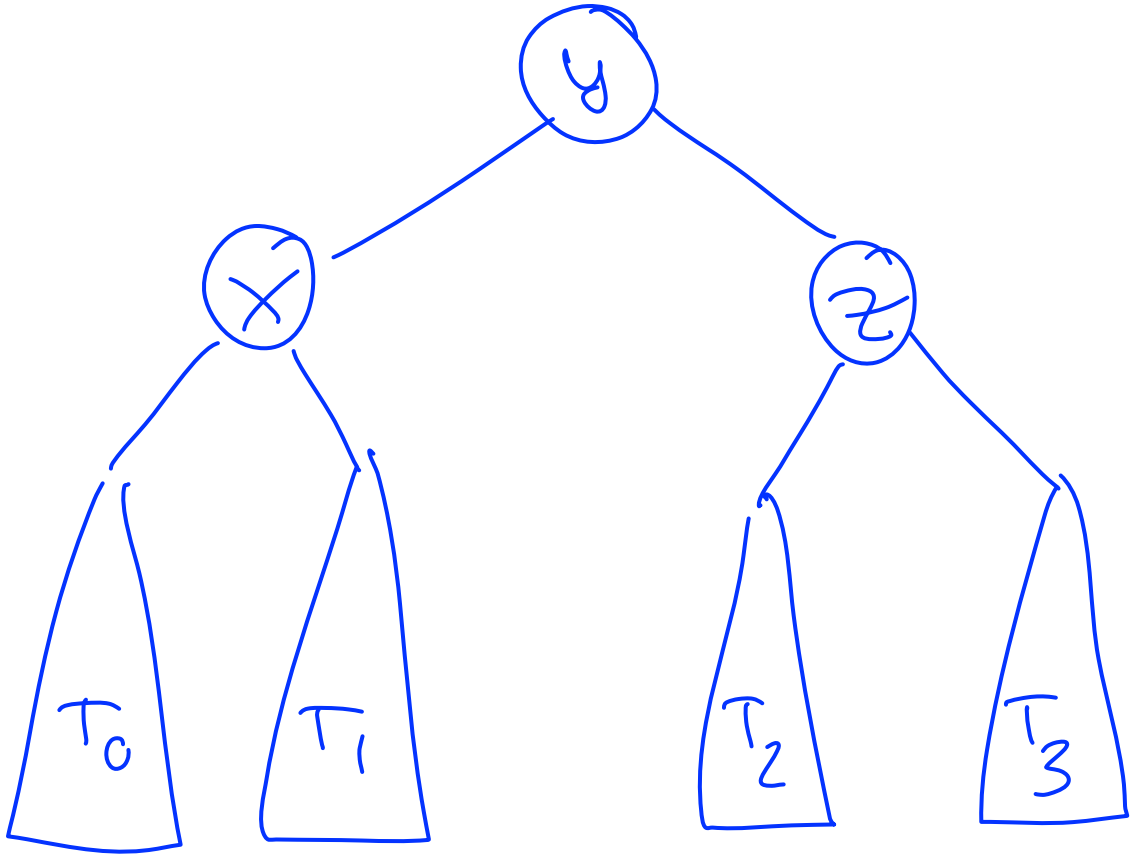
Restructuring: the 4 cases



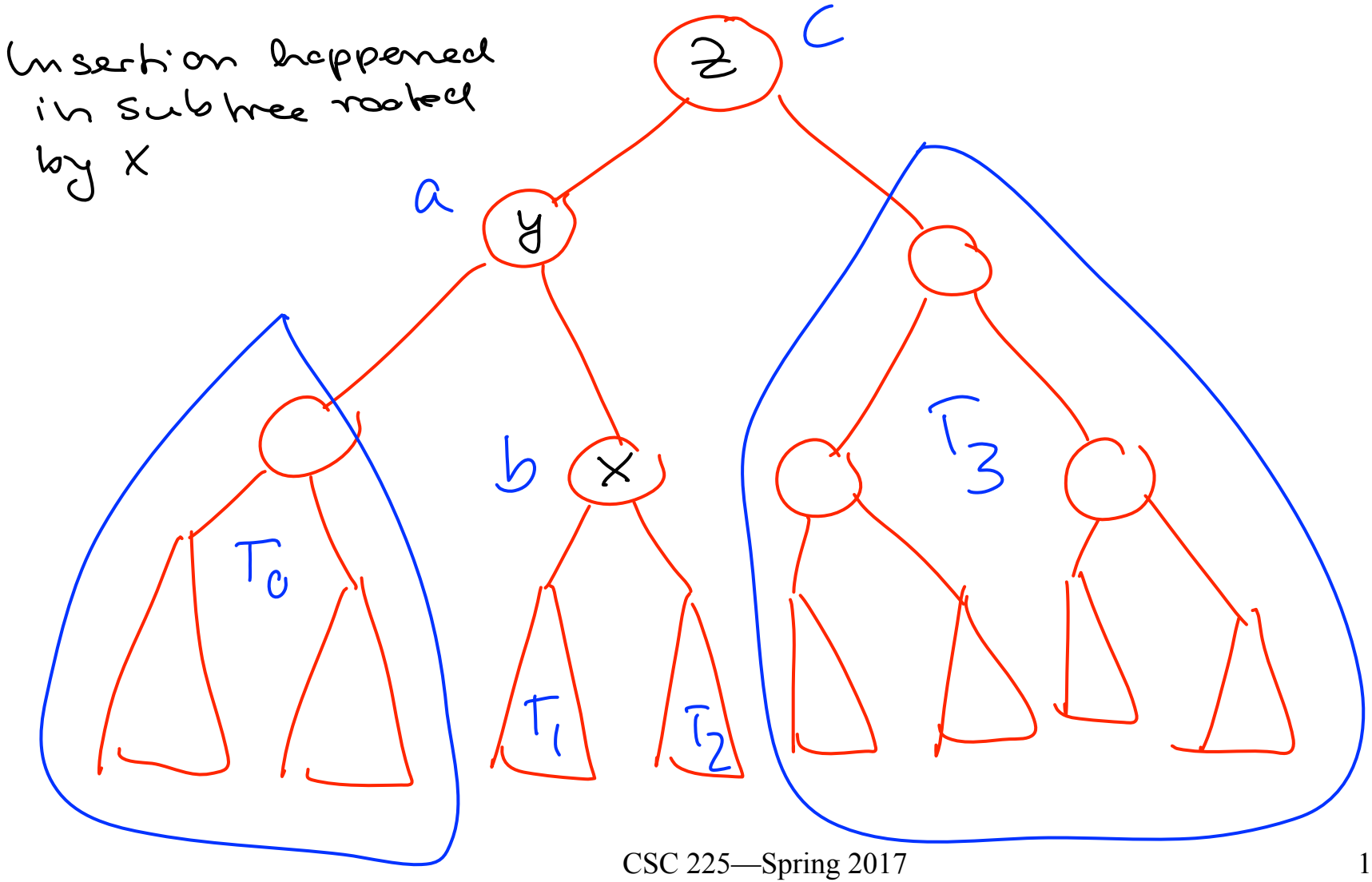
Restructuring: the 4 cases



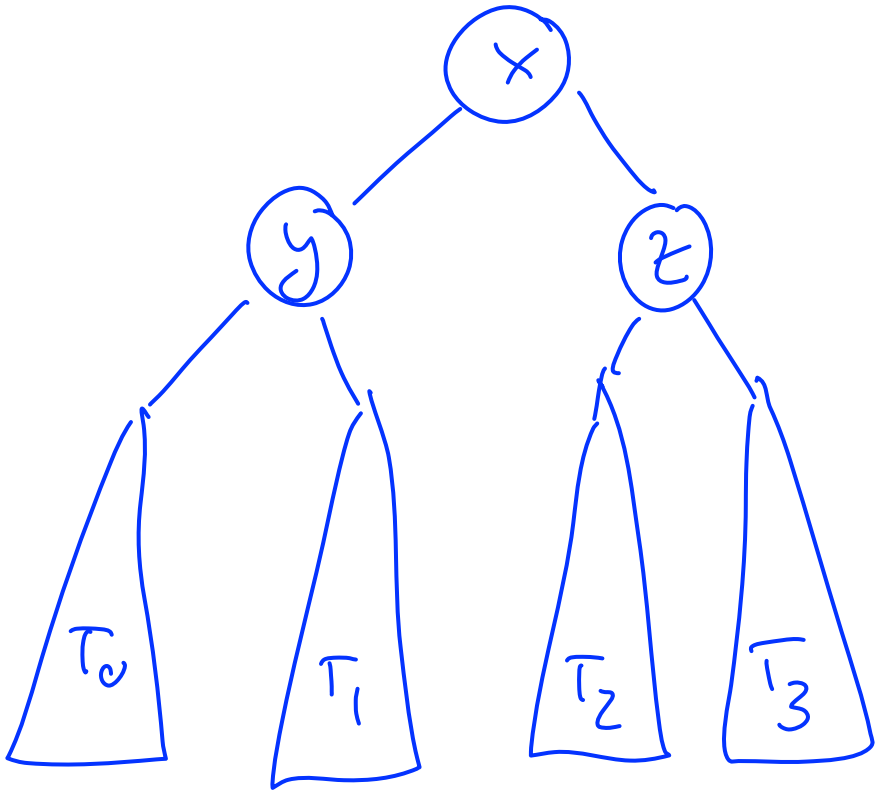
Restructuring: the 4 cases



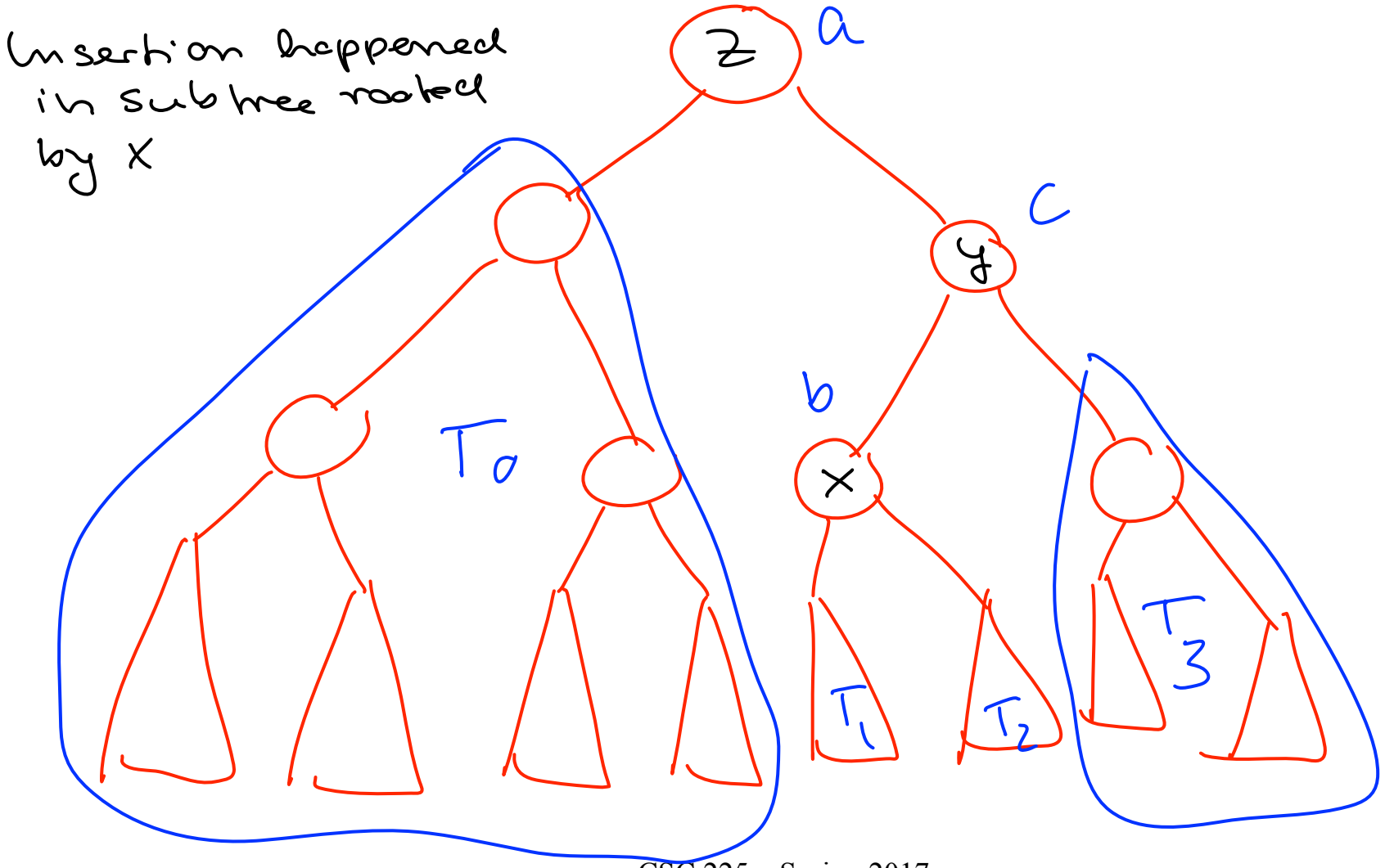
Restructuring: the 4 cases



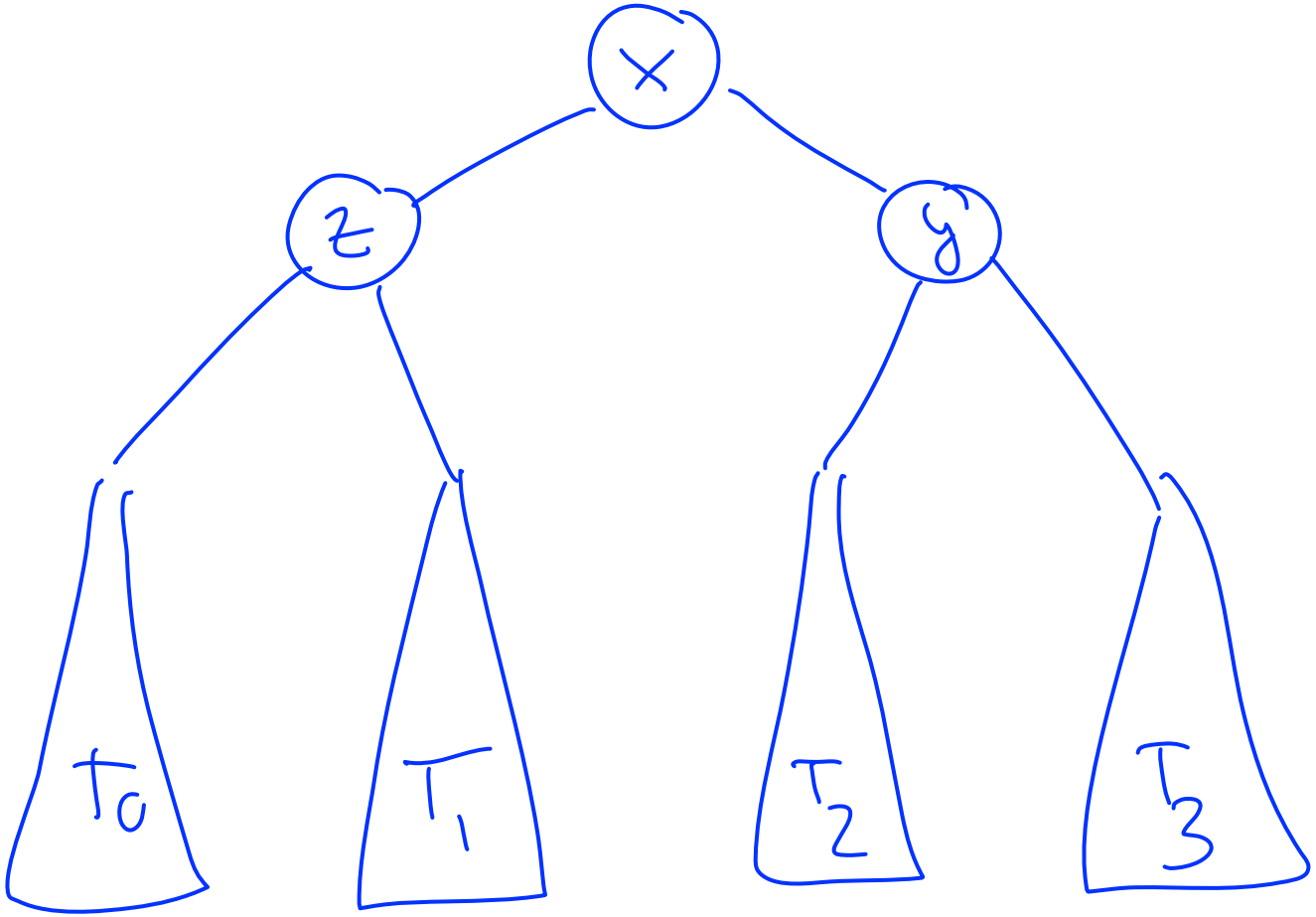
Restructuring: the 4 cases



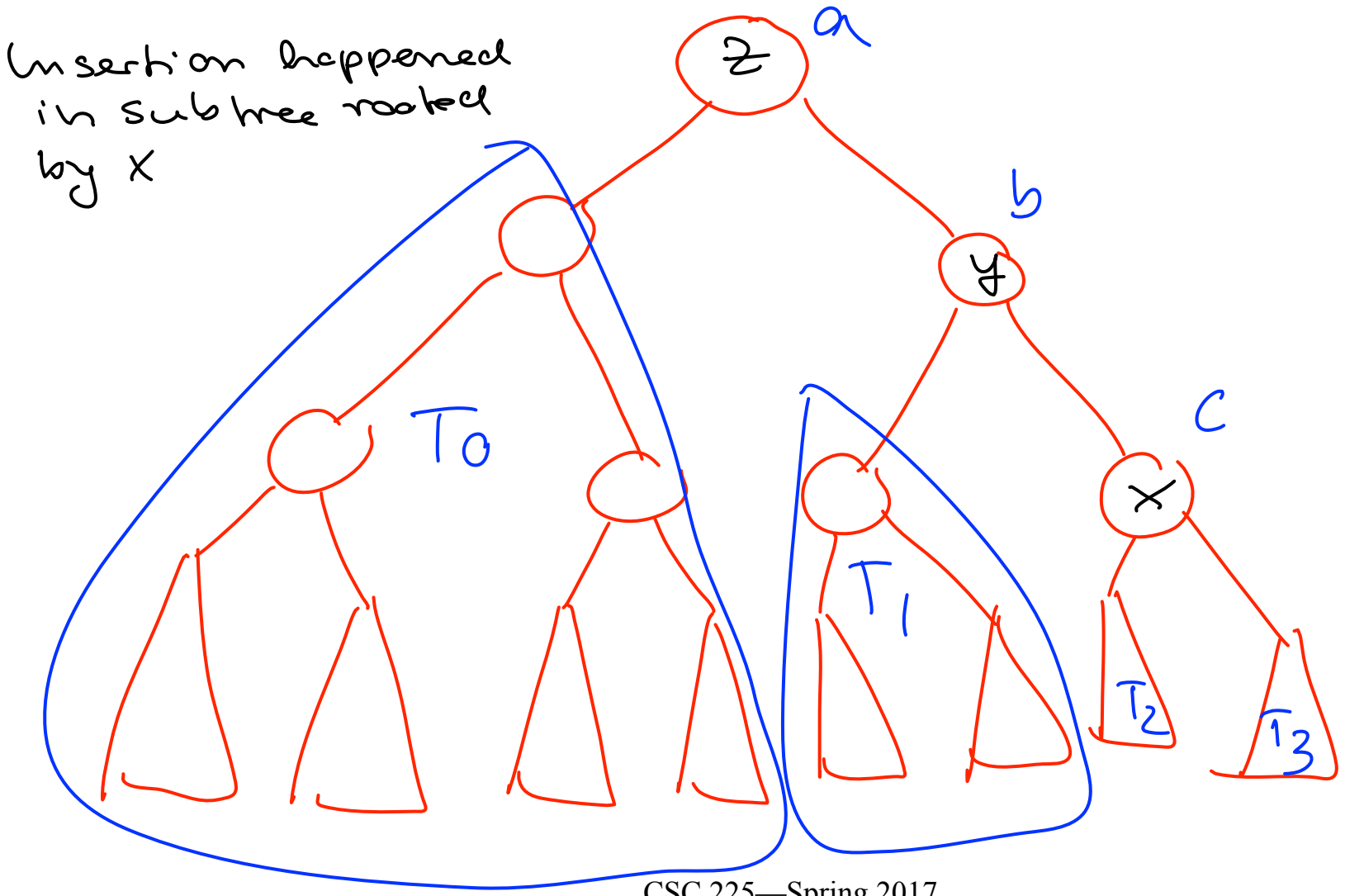
Restructuring: the 4 cases



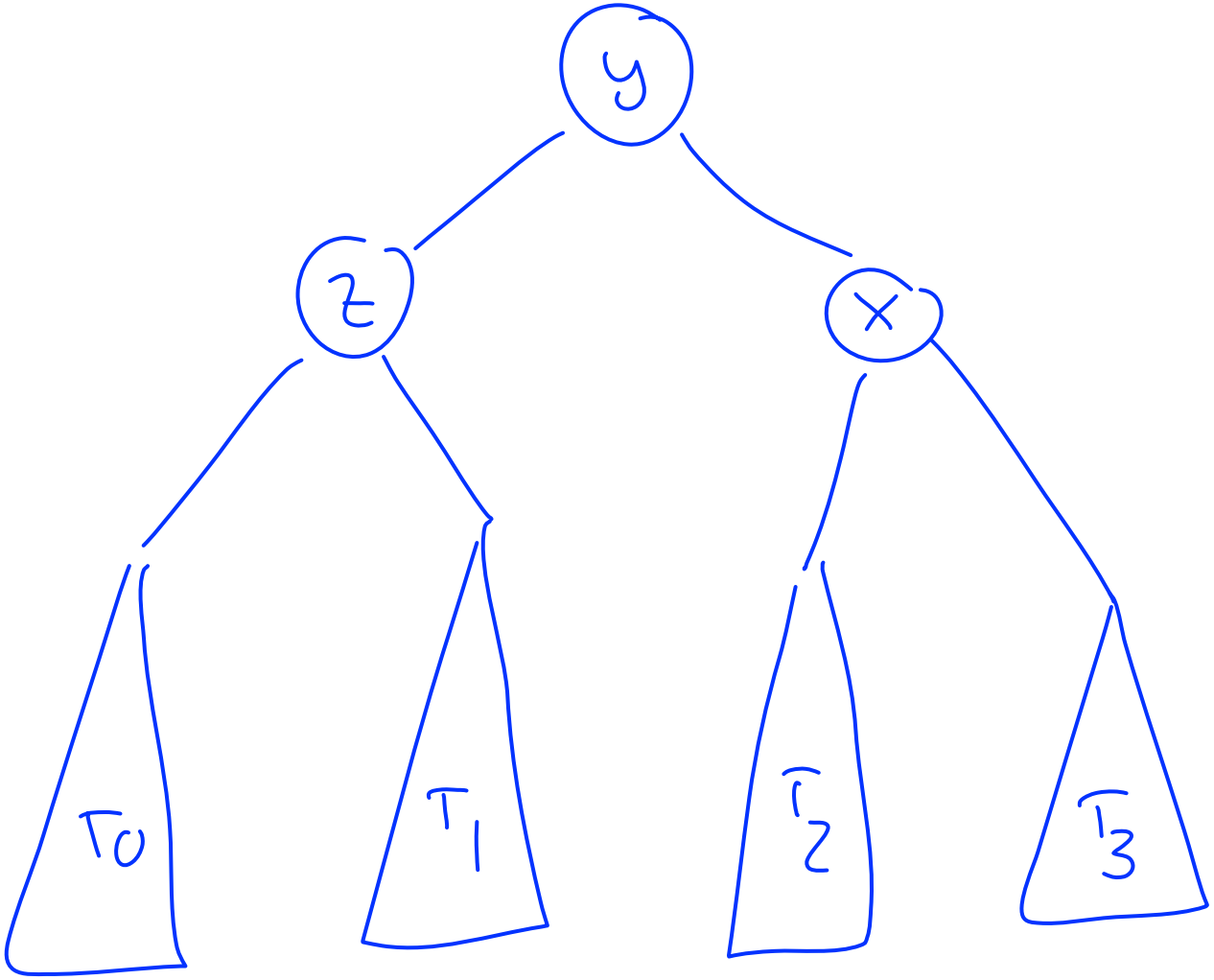
Restructuring: the 4 cases



Restructuring: the 4 cases



Restructuring: the 4 cases



Insertion: One restructuring operation balances the tree!

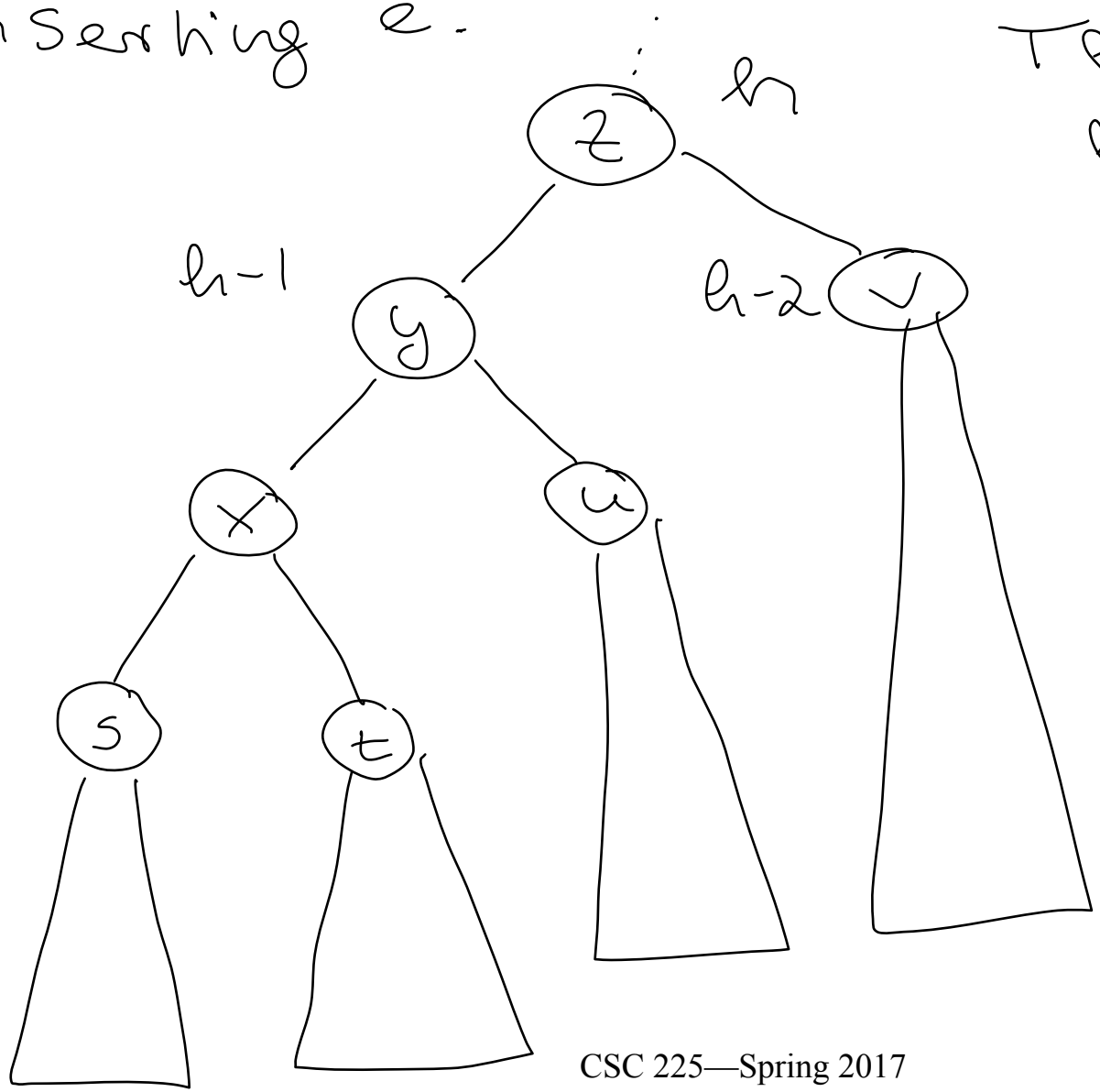
Consider an AVL-tree T and an element e to be inserted into the tree. Further assume, after inserting e into T using binary search tree insertion, the new tree has to be rebalanced.

Let z be the unbalanced node in T after inserting e (call it T').
Then T' can look like each of the 4 cases studied just before for the reconstruction process.

We prove the claim for case 1.

That we show that height(z) in T is identical to the height of the balanced subtree in T* after restructuring.

Assume height(z) = h before inserting e.



Then height(y) = h-1 and height(v) = h-2.

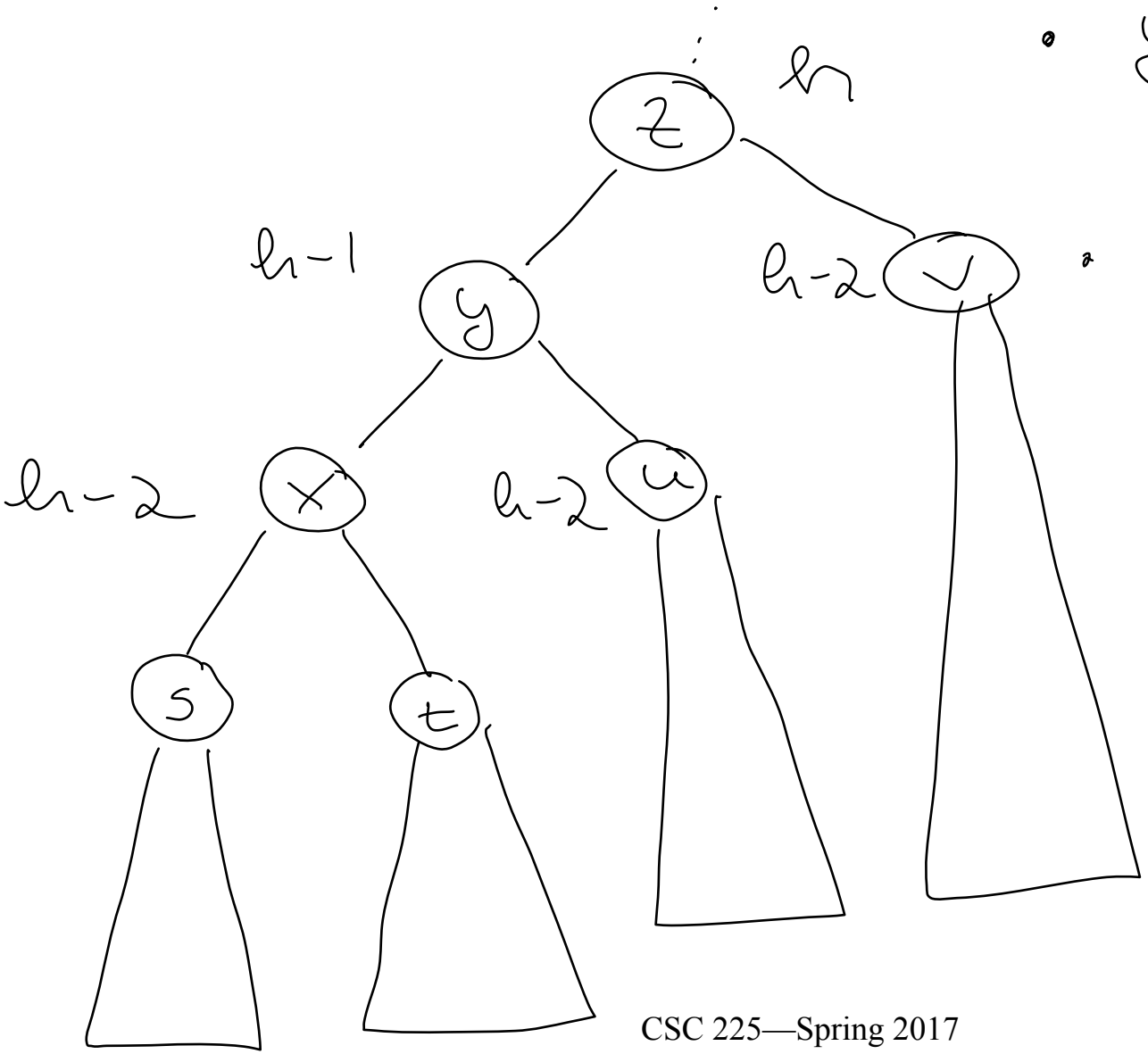
Why?

Since

- T is balanced
- T' is unbalanced

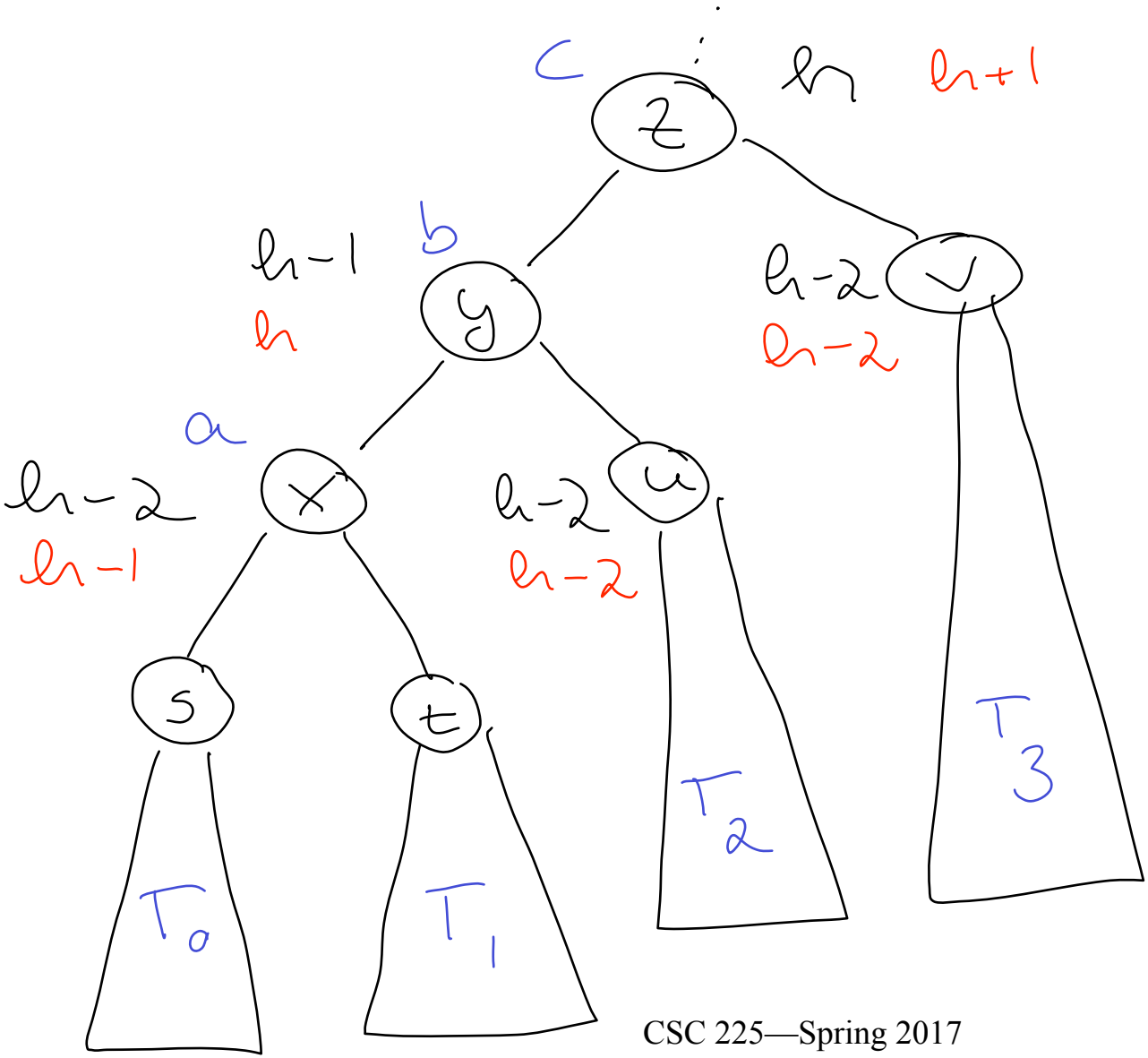
$\Rightarrow$ height(y) $>$ height(z)

height(x) = height(u) = h-2 since

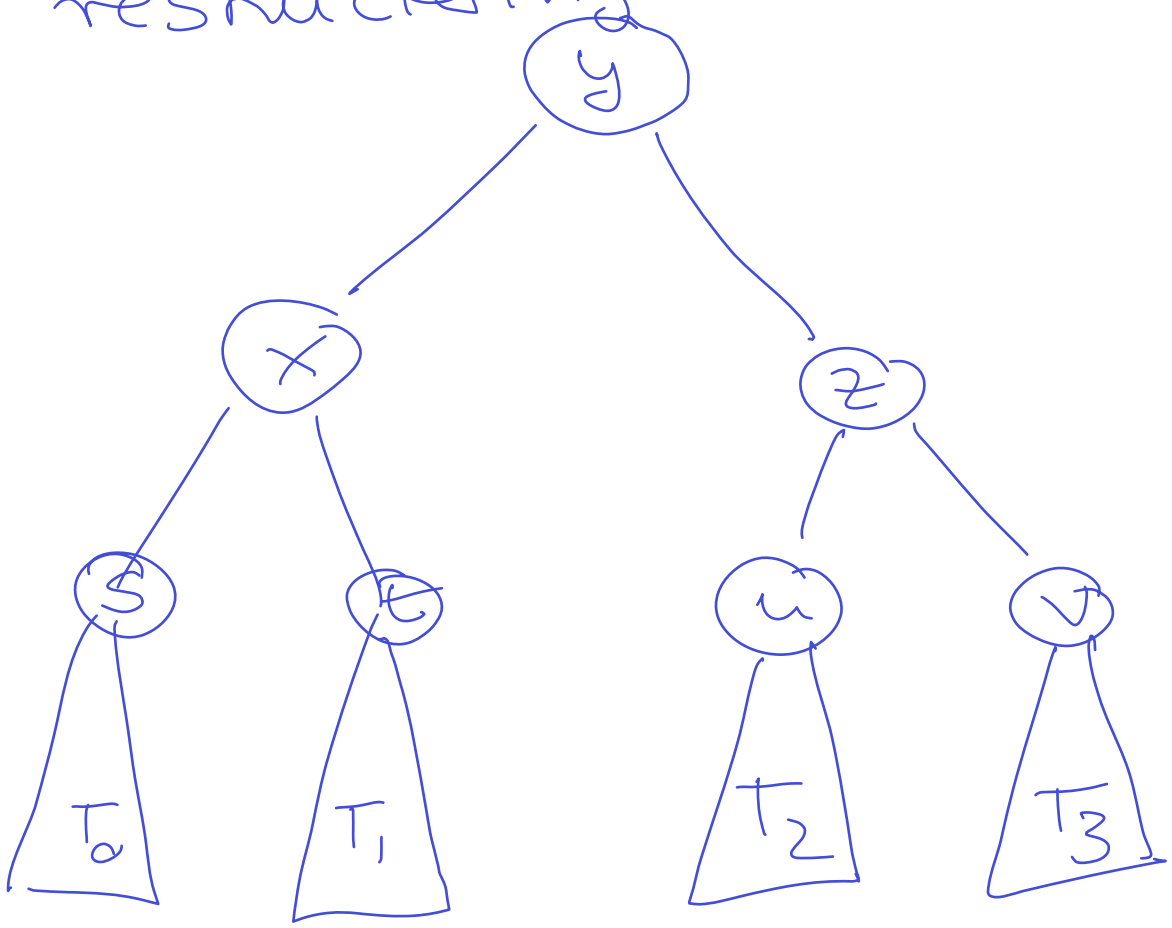


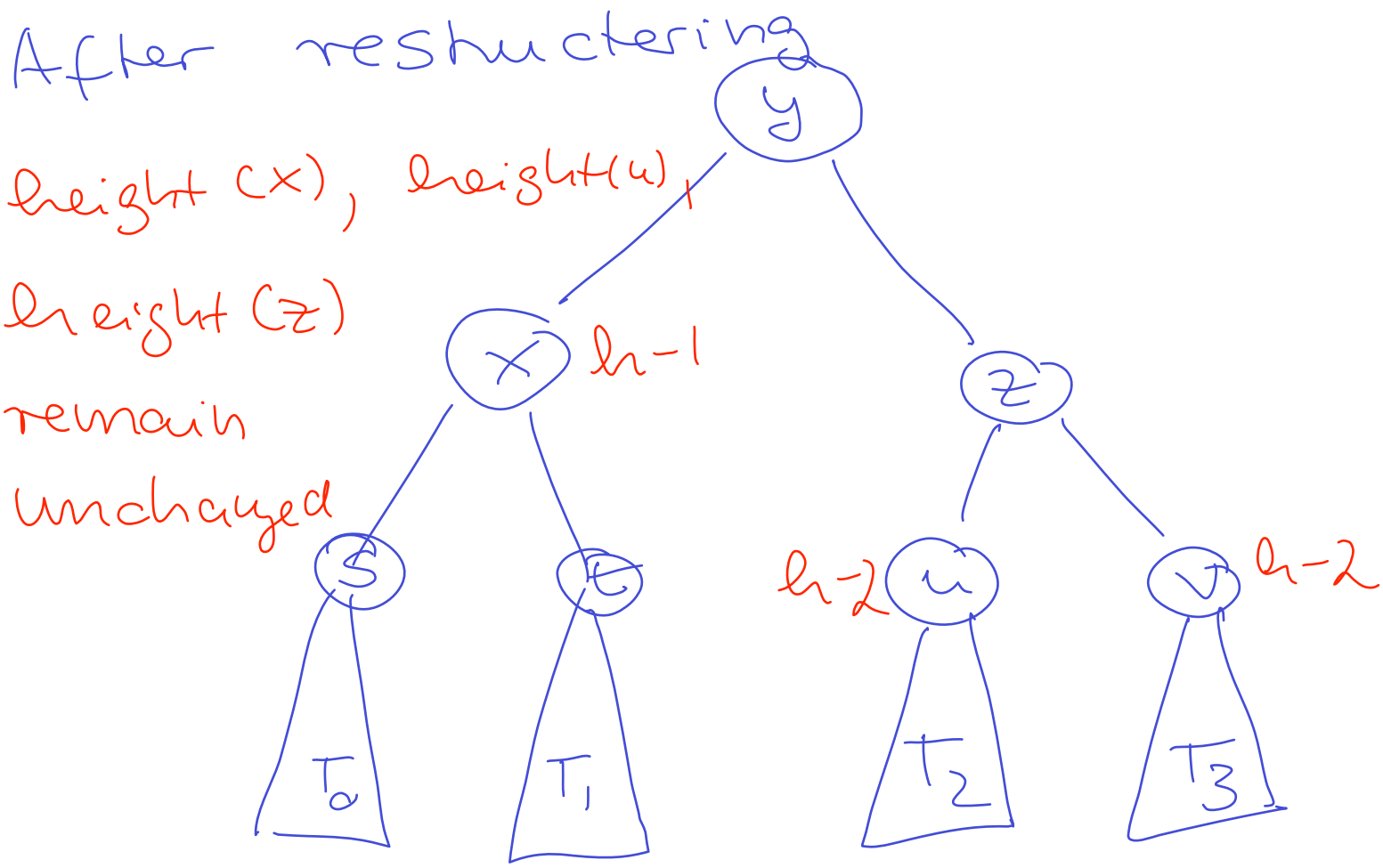
- y is balanced in T & T*
- z is balanced in T but unbalanced in T*

After insertion:

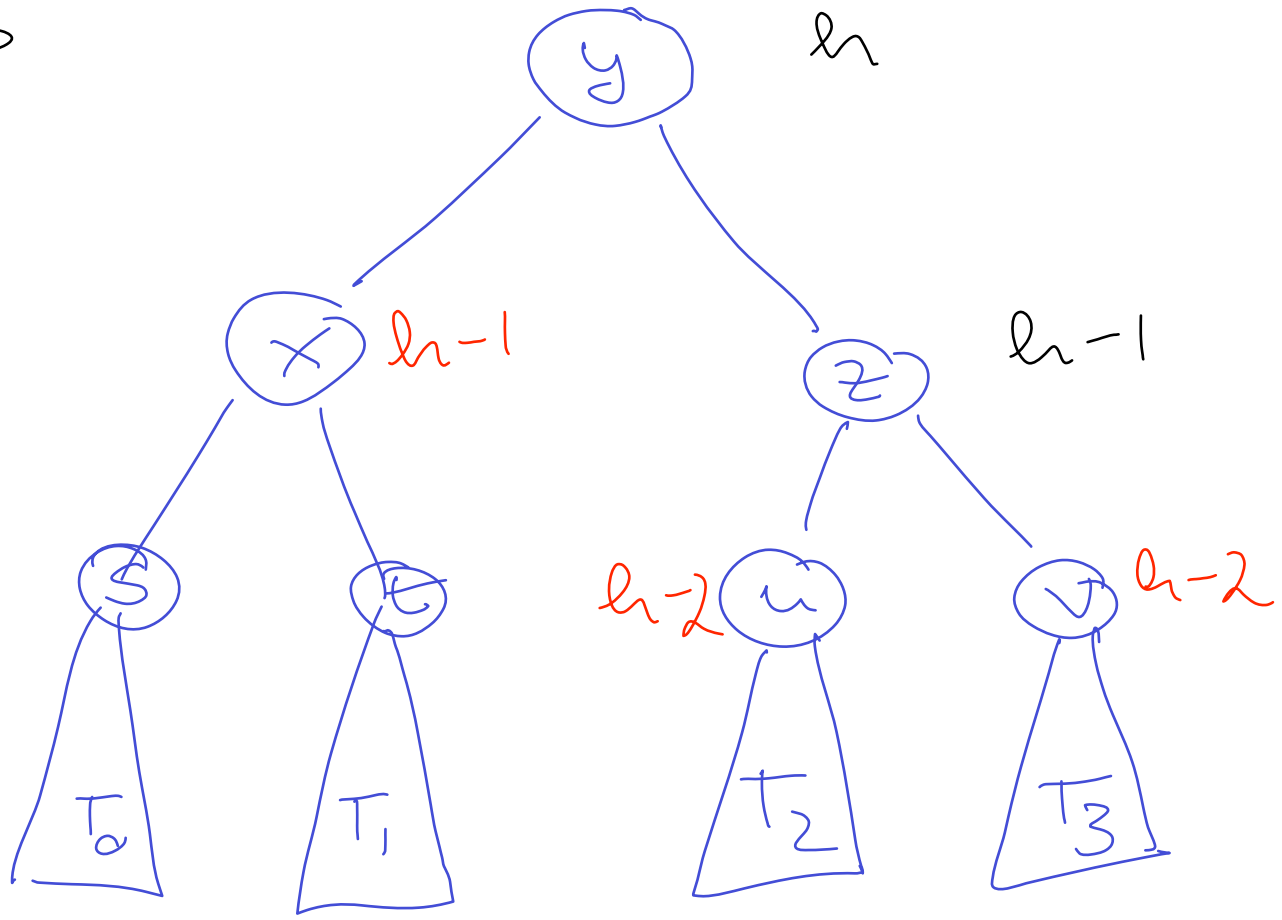


After restructering



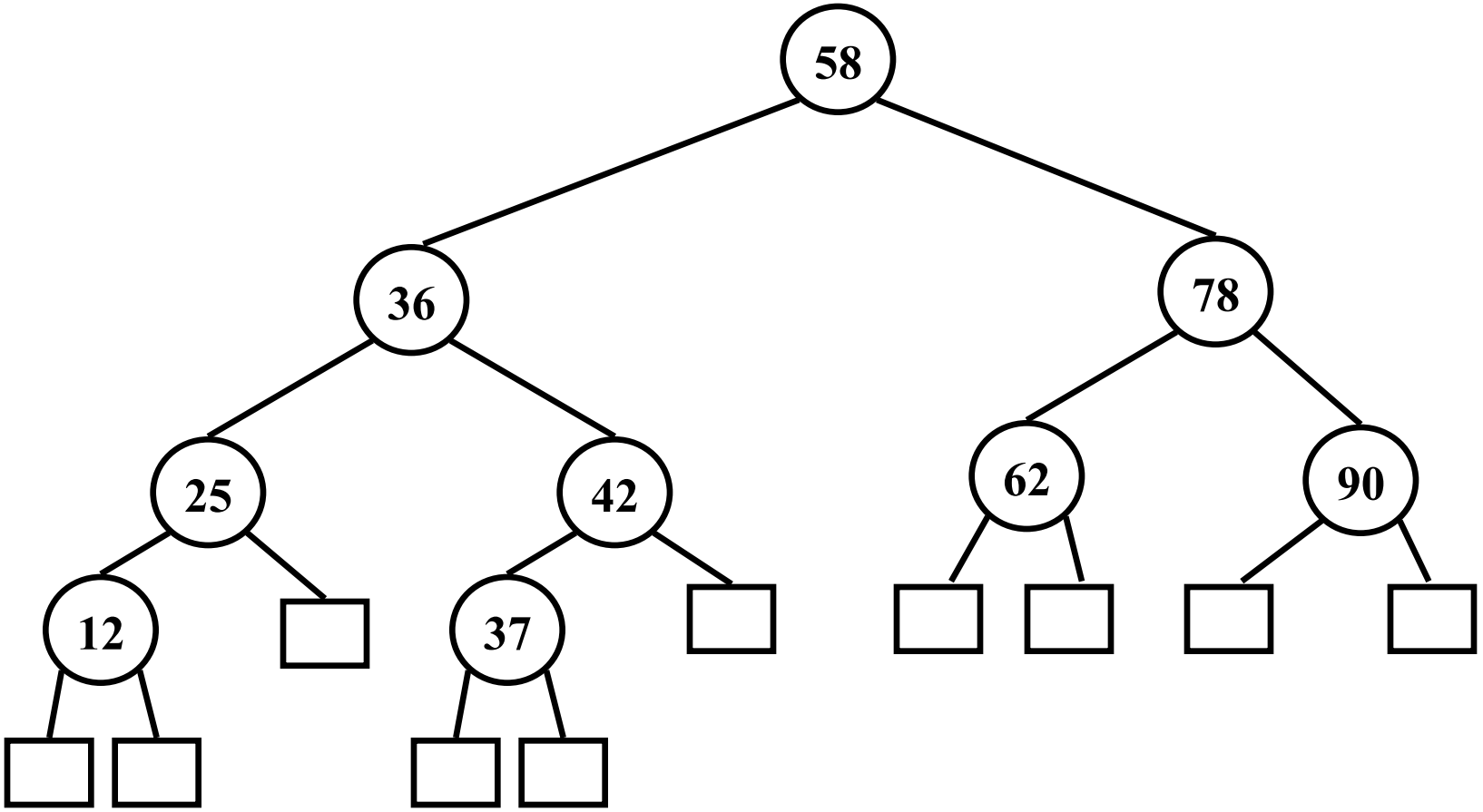

After restructering
height(x), height(u),
height(z)
remain
unchanged
y
x
h-1
z
s
t
h-2
u
v
h-2
T0
T1
T2
T3

Thus



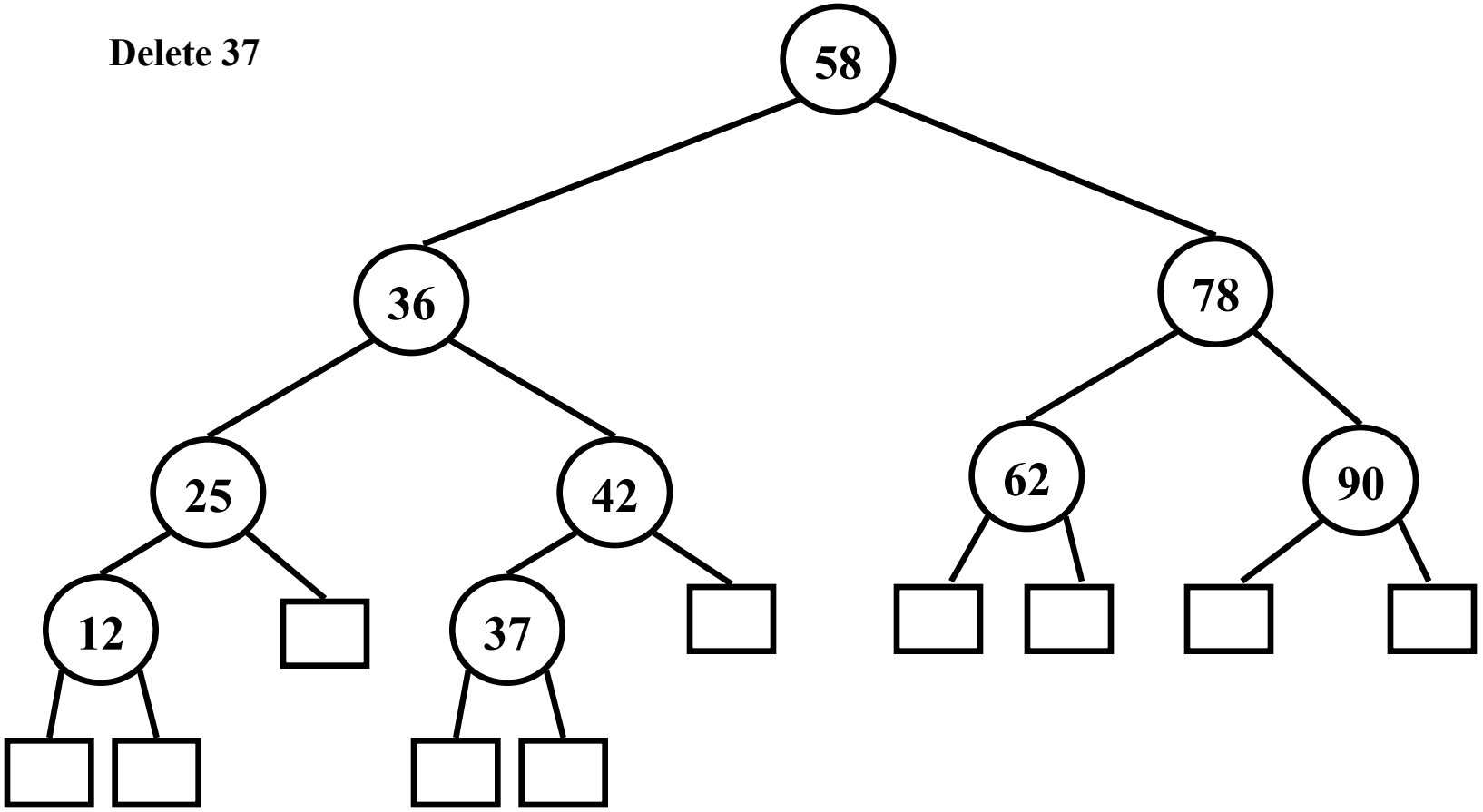
$\Rightarrow$ The restructured subtree is of the same height as before insertion!

Deletion

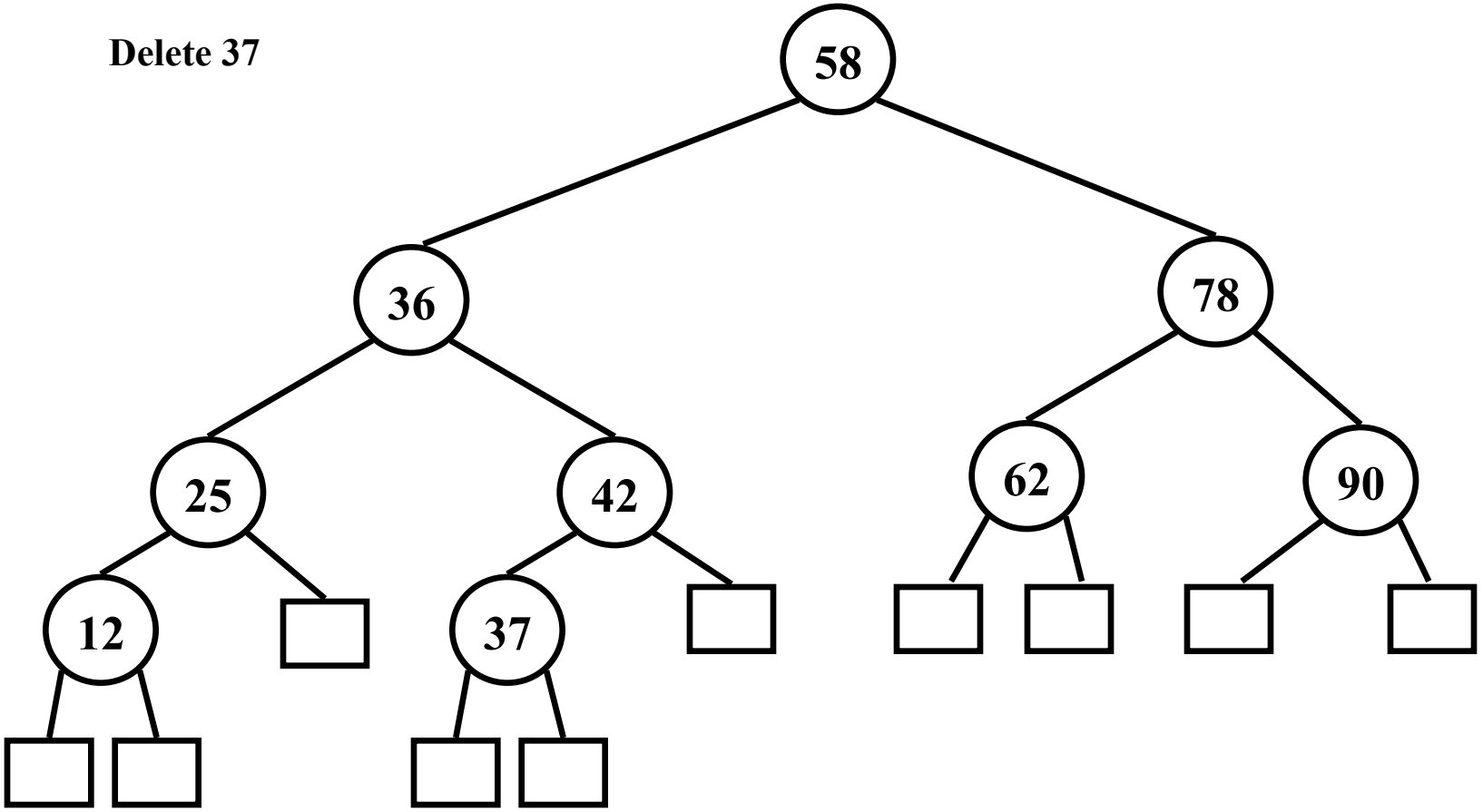


Deletion

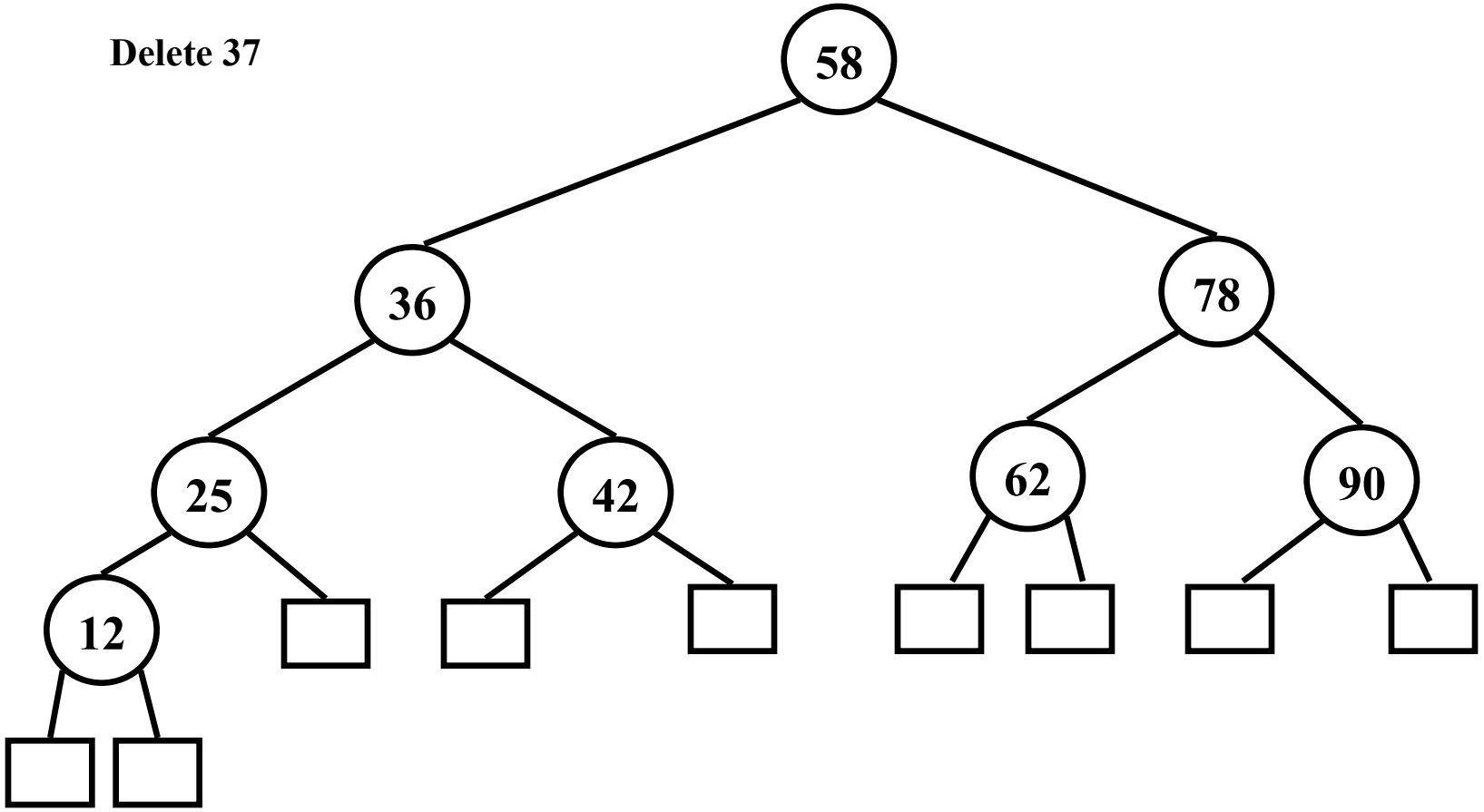
Delete 37



Step 1. Proceed as in binary search trees.

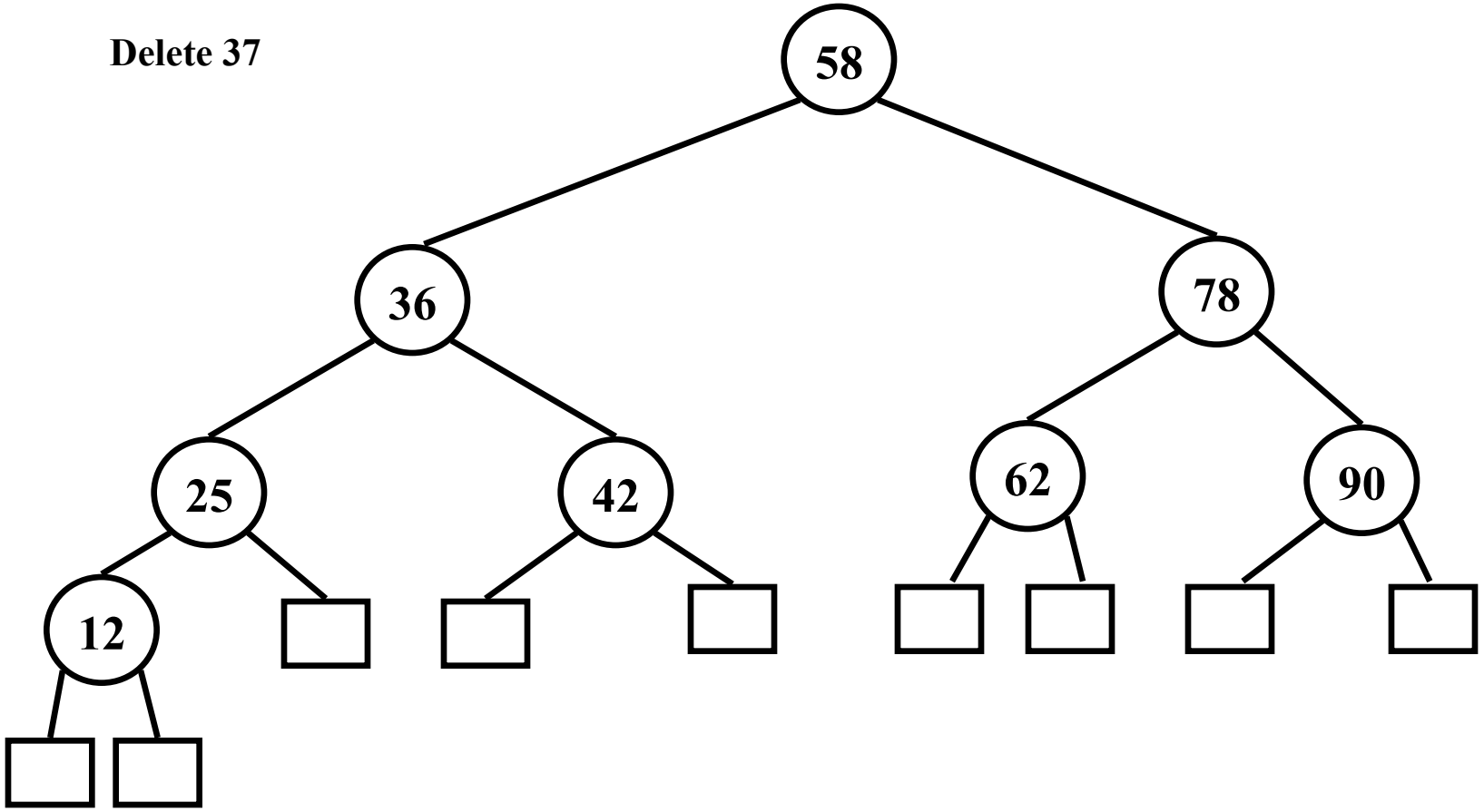


Step 1. Proceed as in binary search trees.



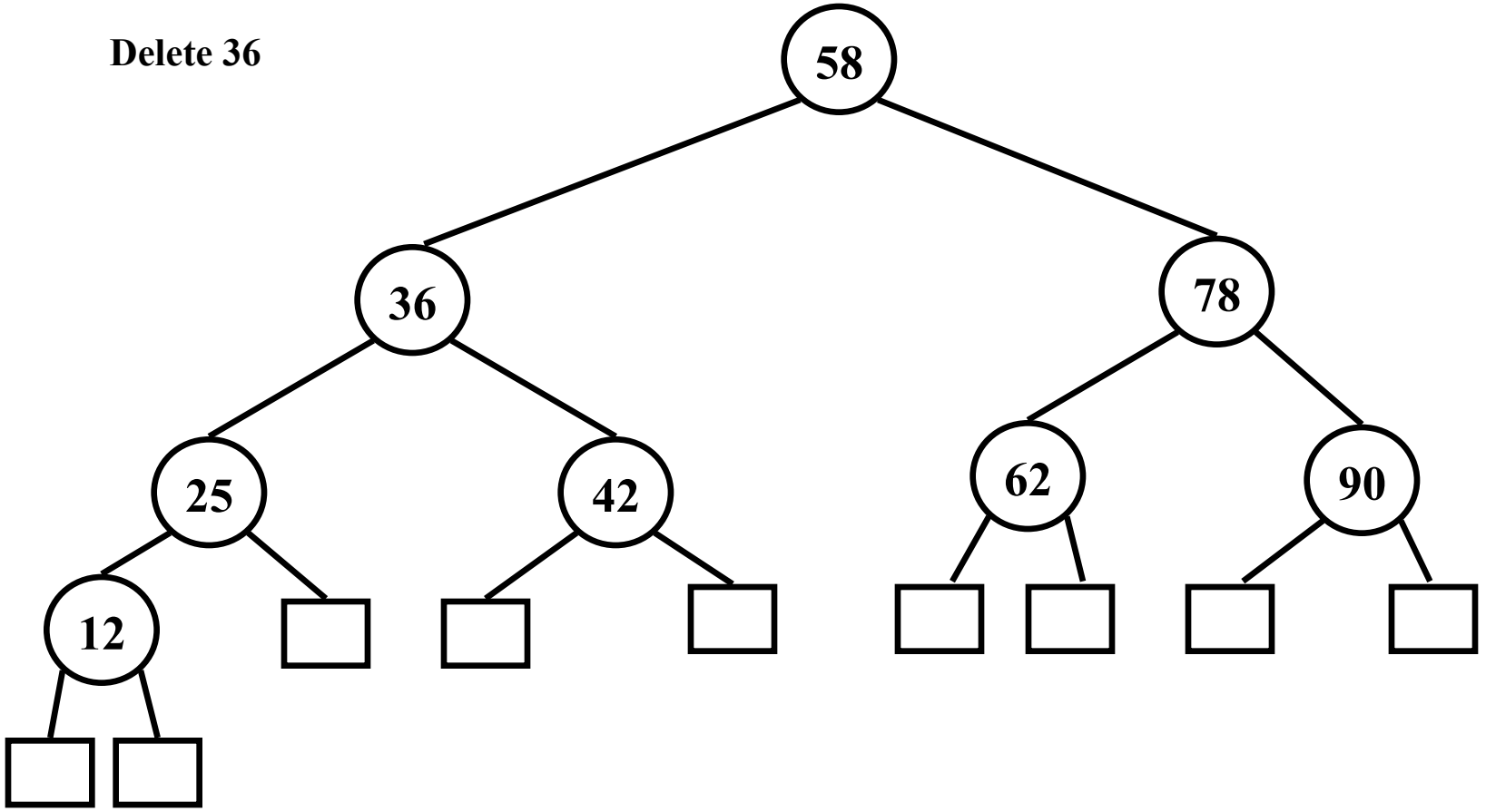
Step 2. Check height balance property

Delete 37

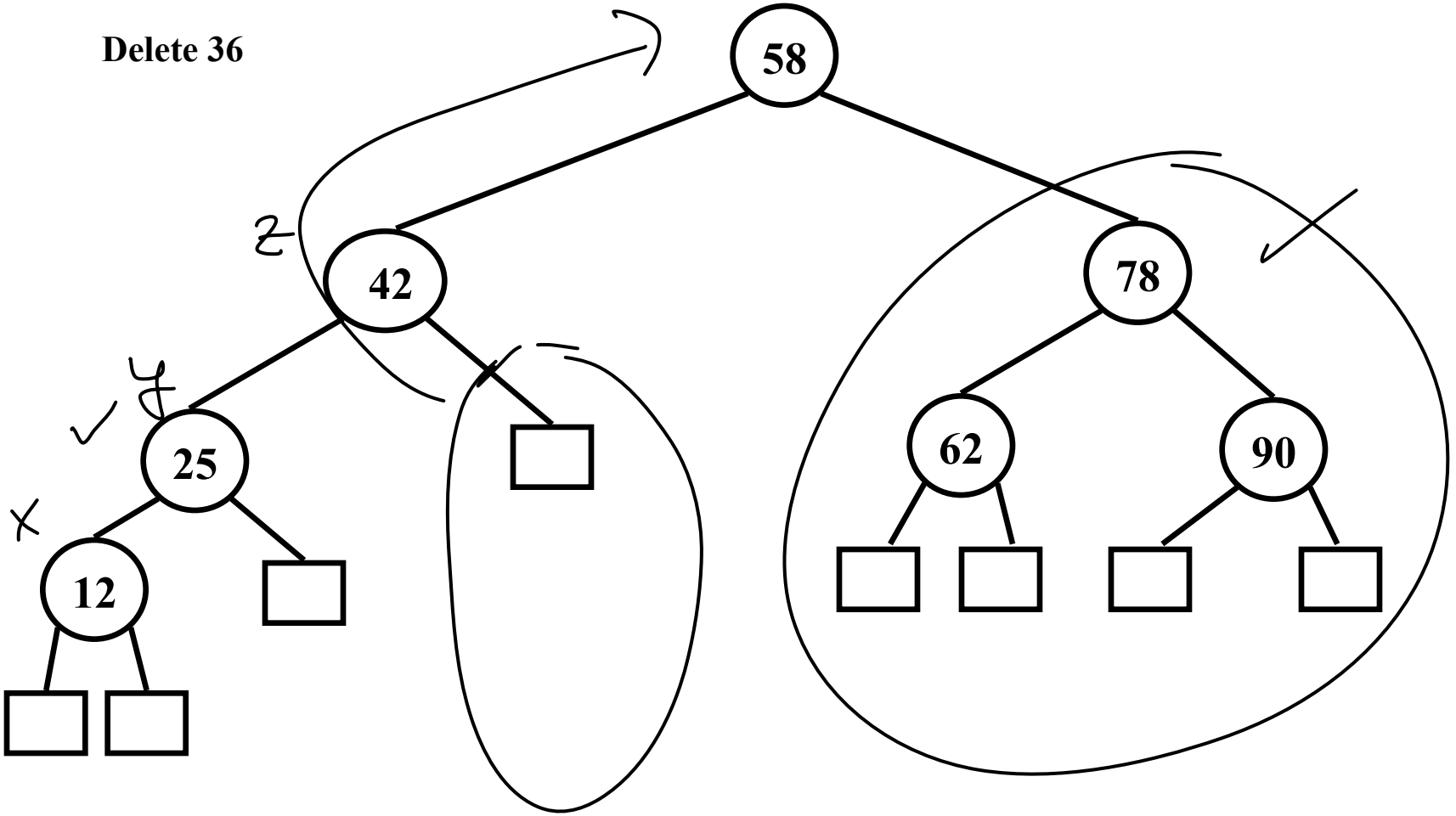


Another Deletion: Step 1

Delete 36

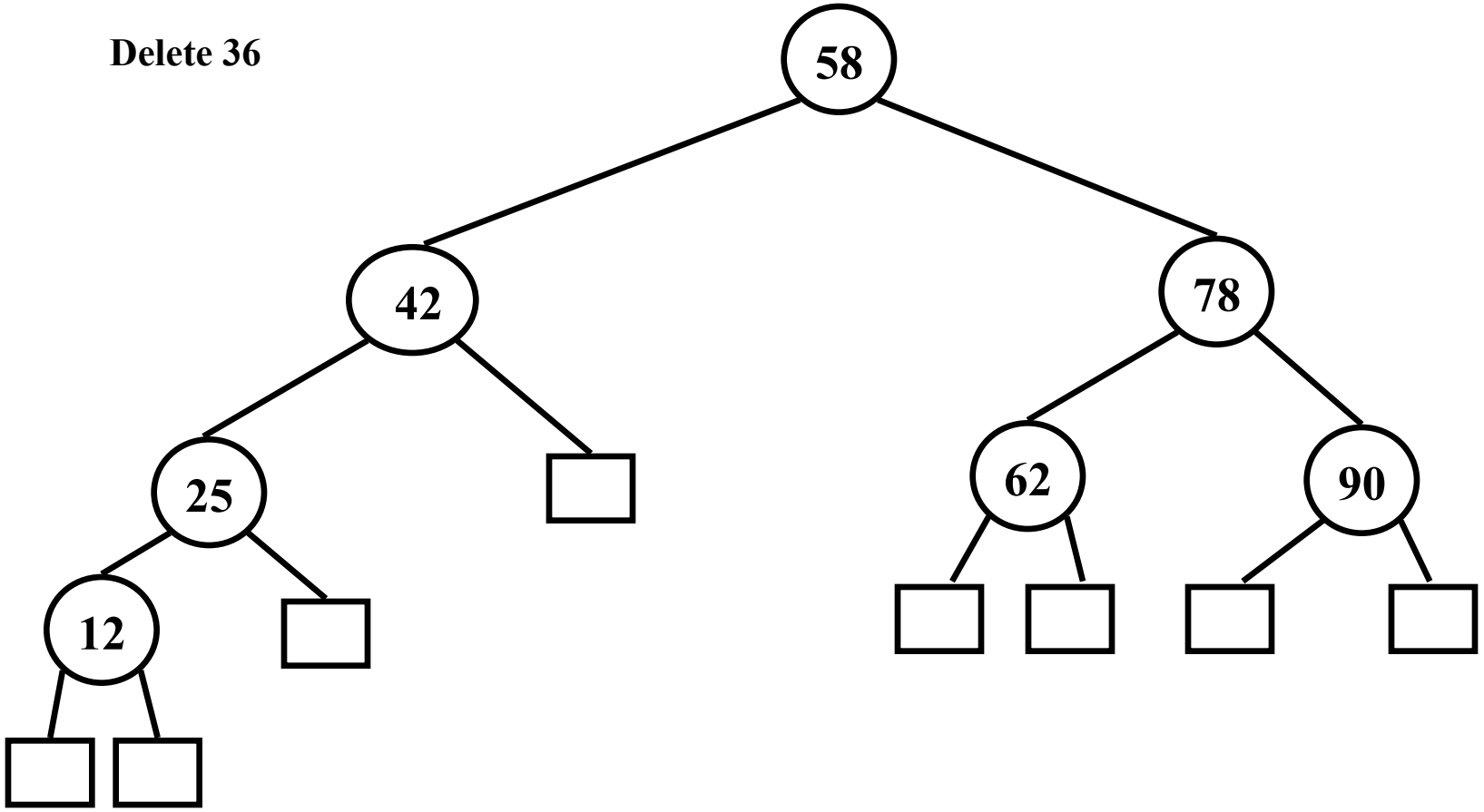


Step 1



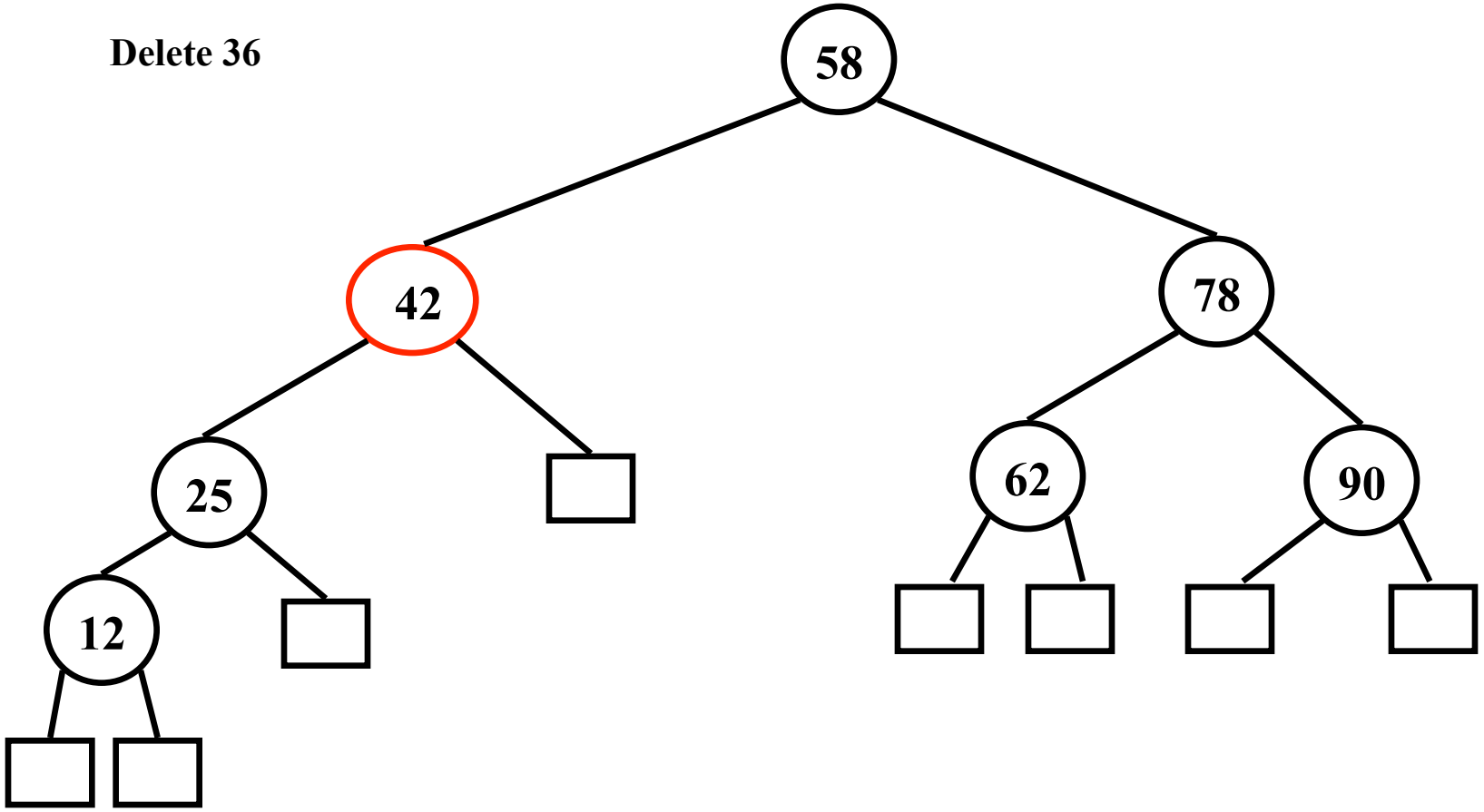
Step 2

Delete 36

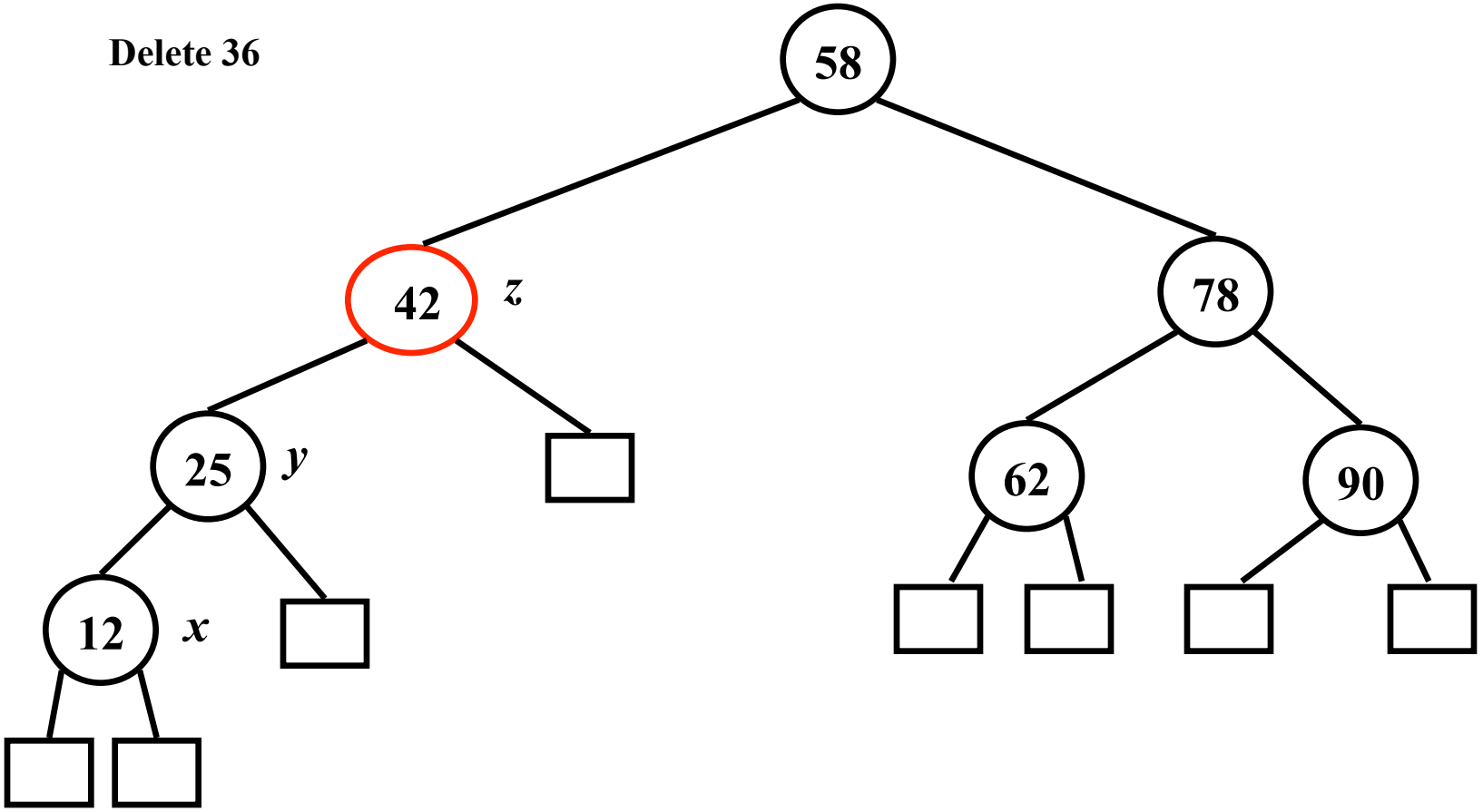


Restructure!

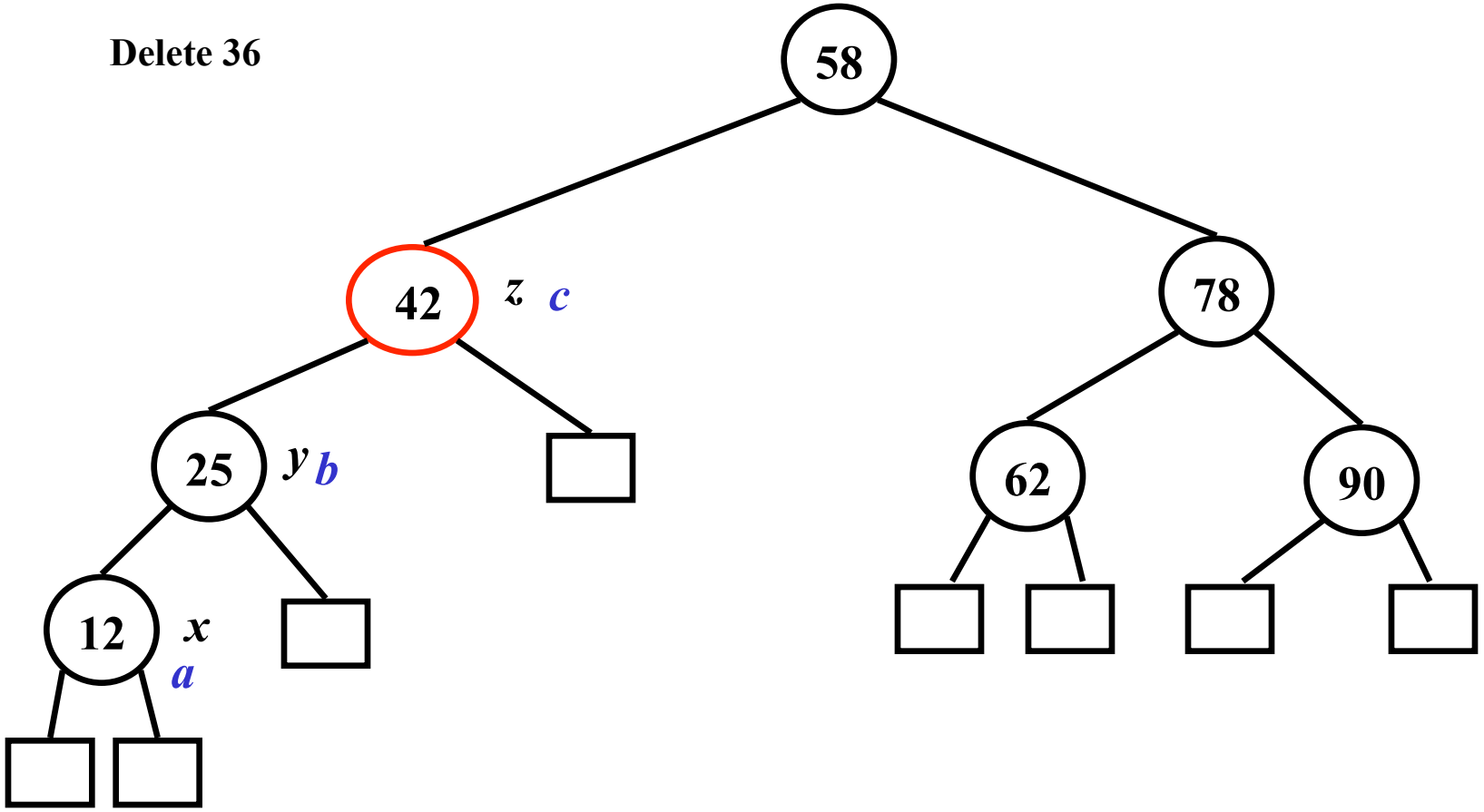
Delete 36



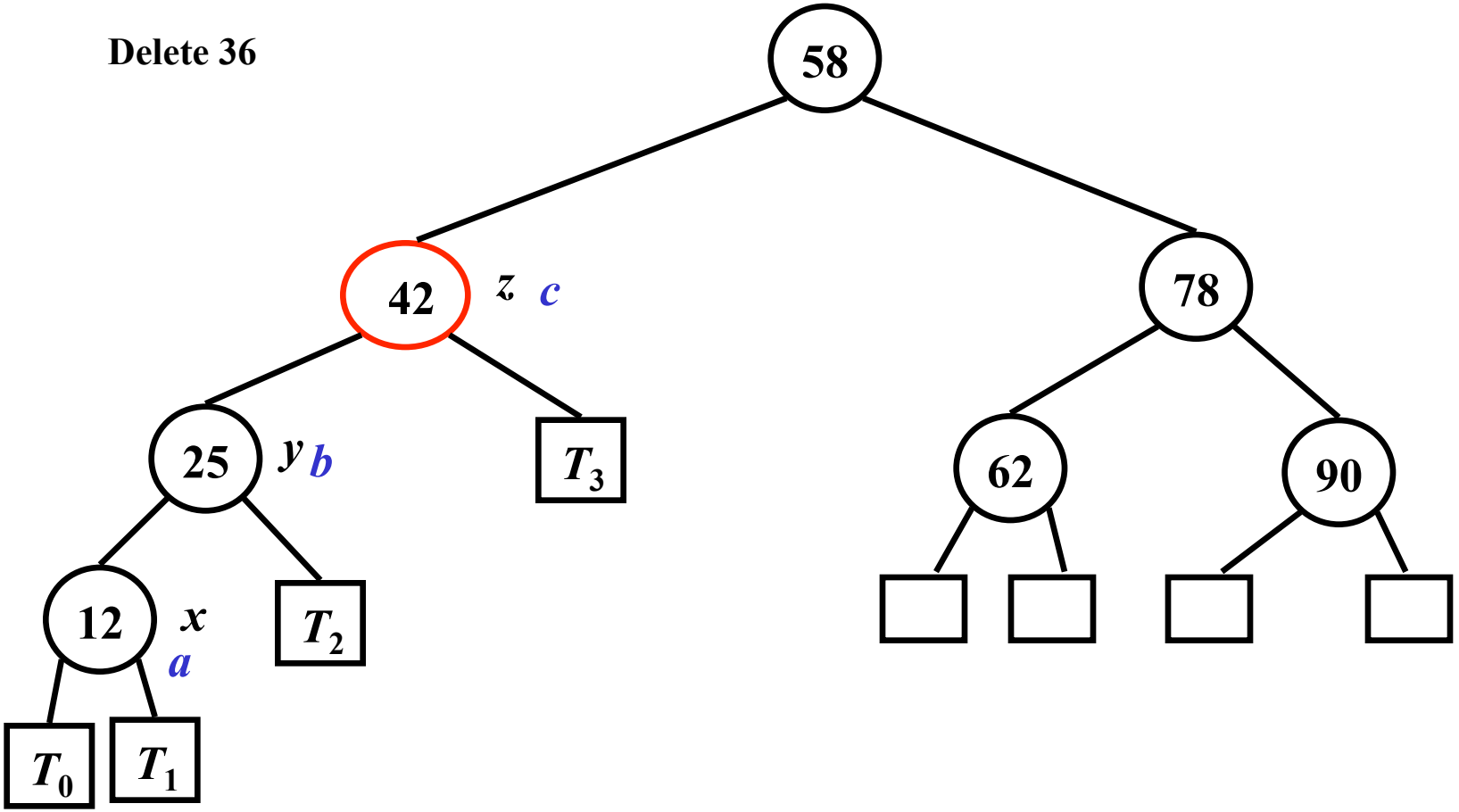
Restructure!



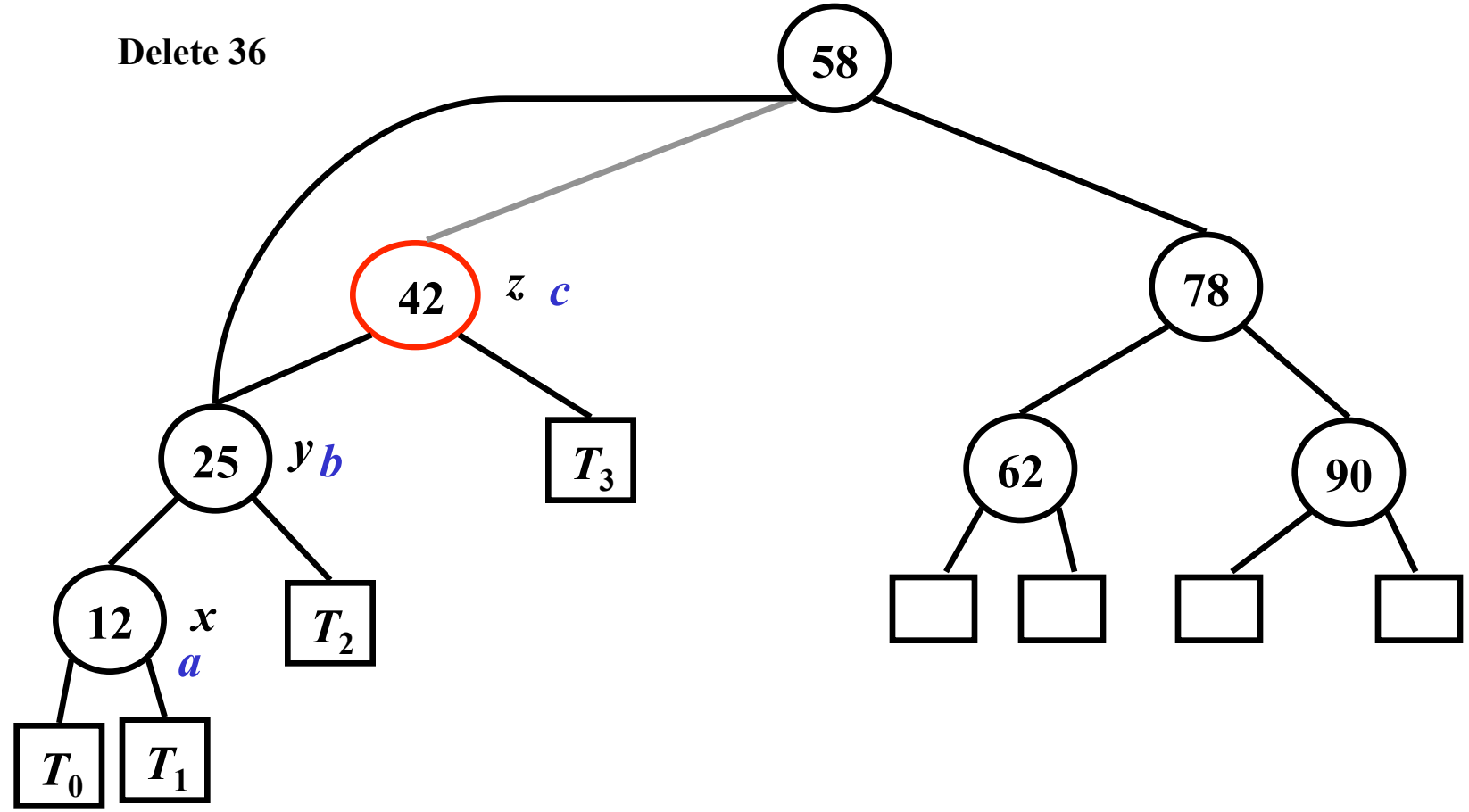
Restructure!



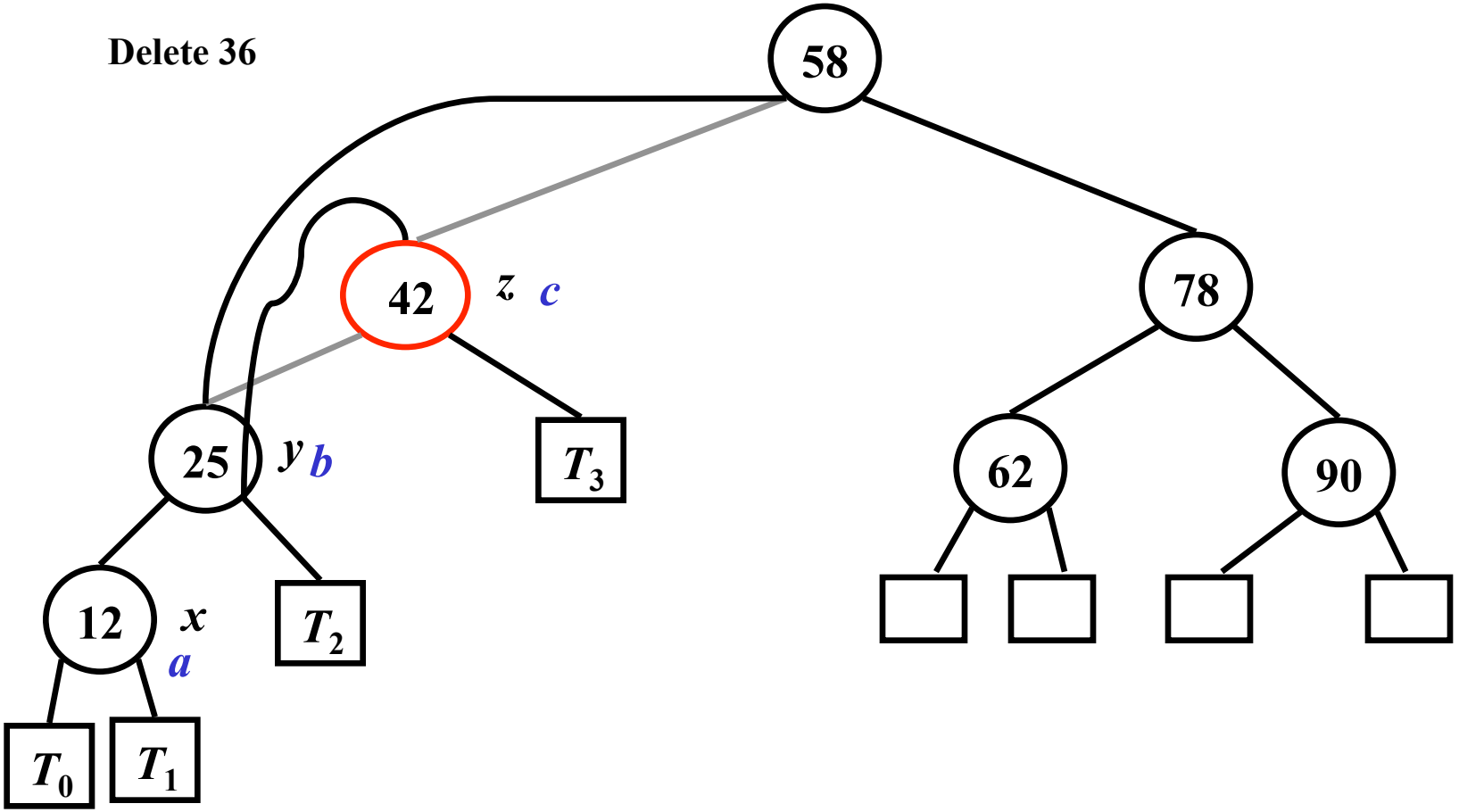
Restructure!



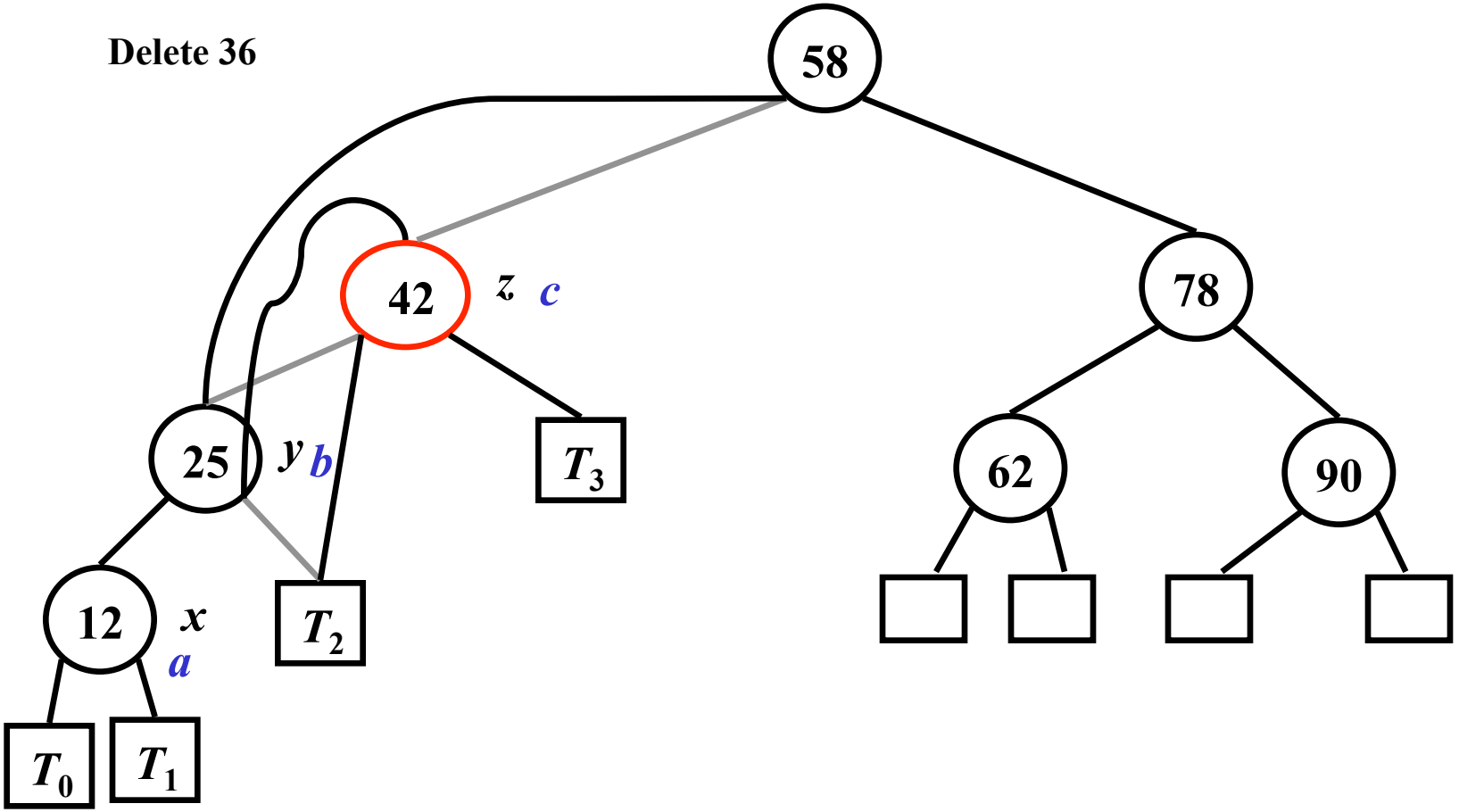
Restructure!



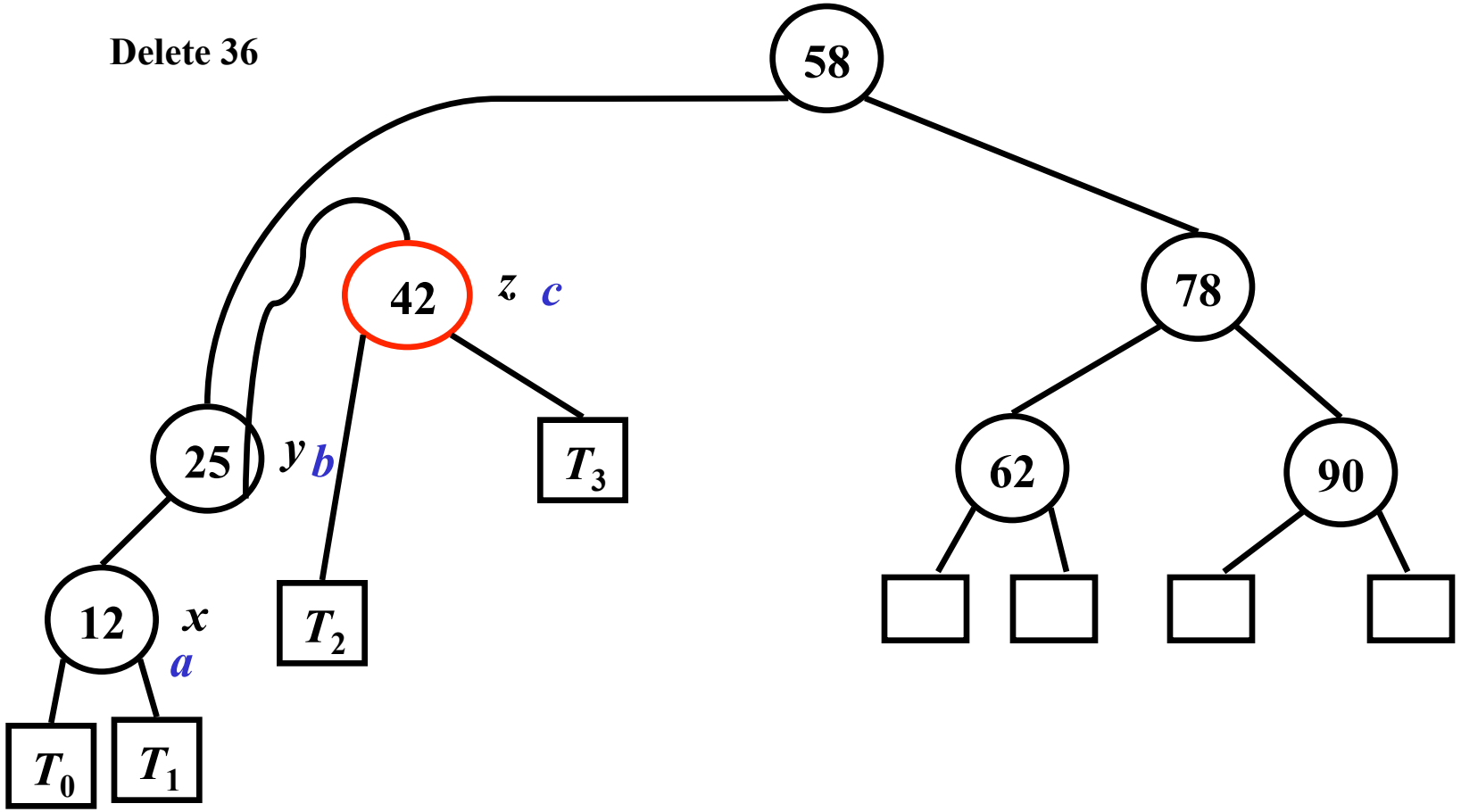
Restructure!



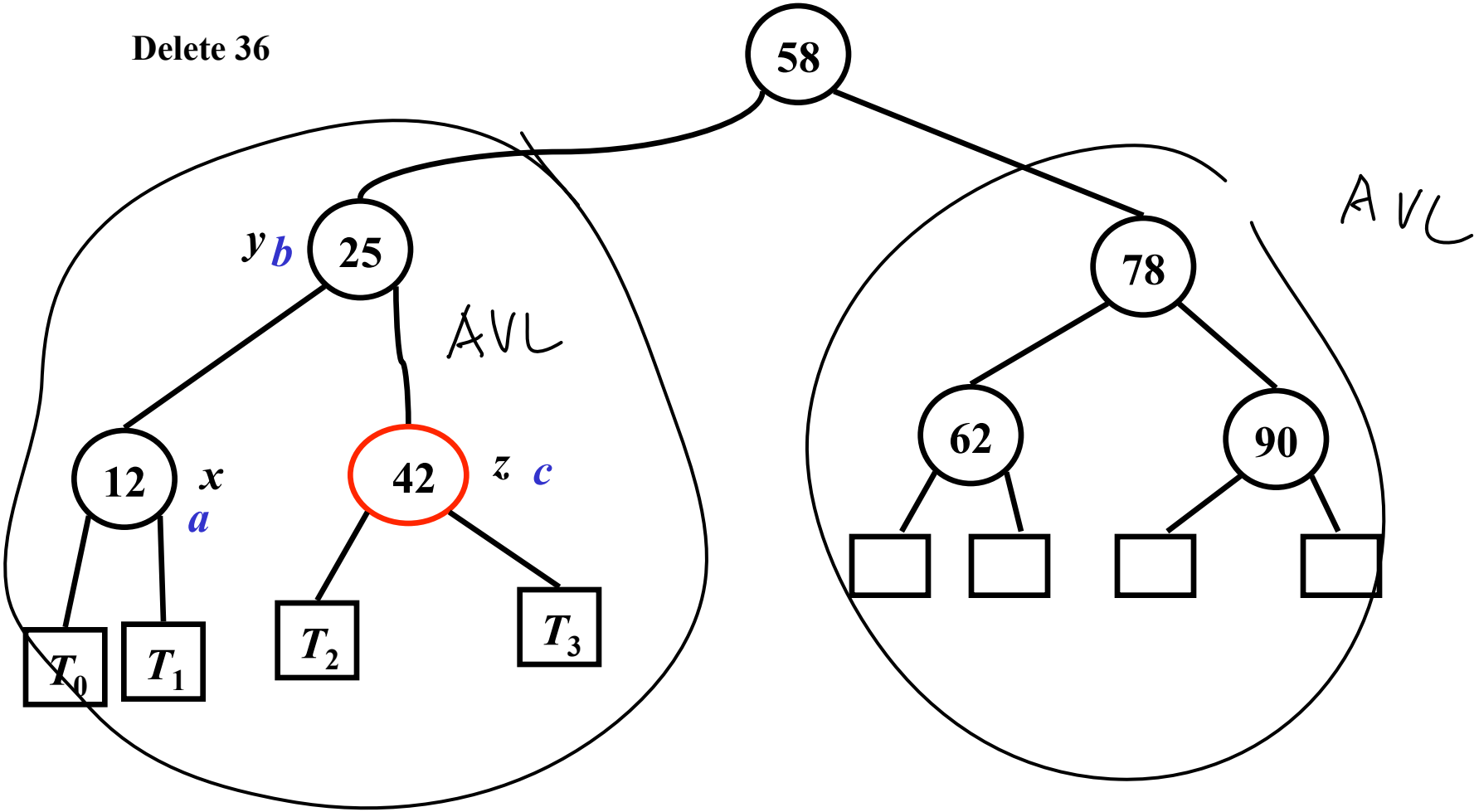
Restructure!



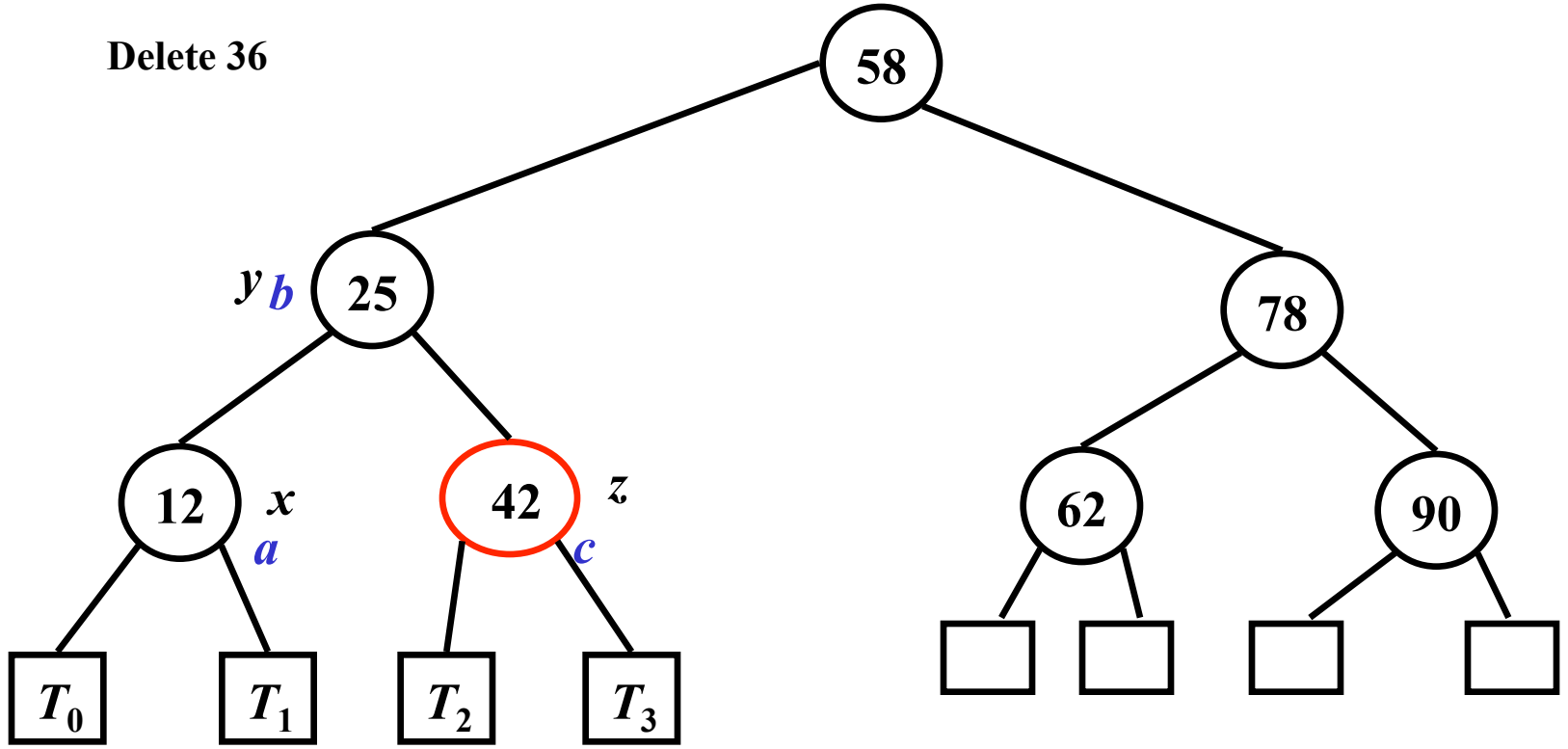
Restructure!



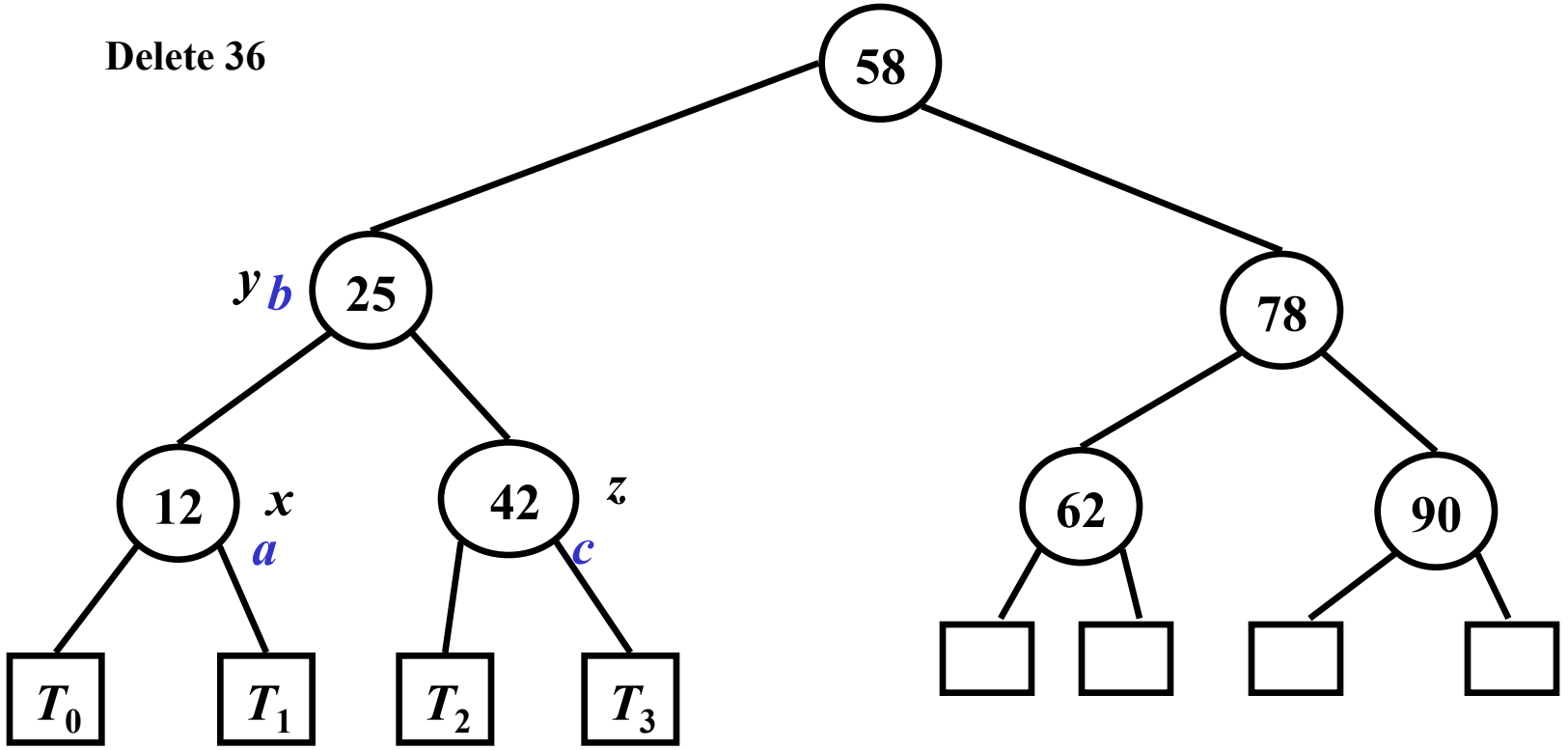
Restructure!



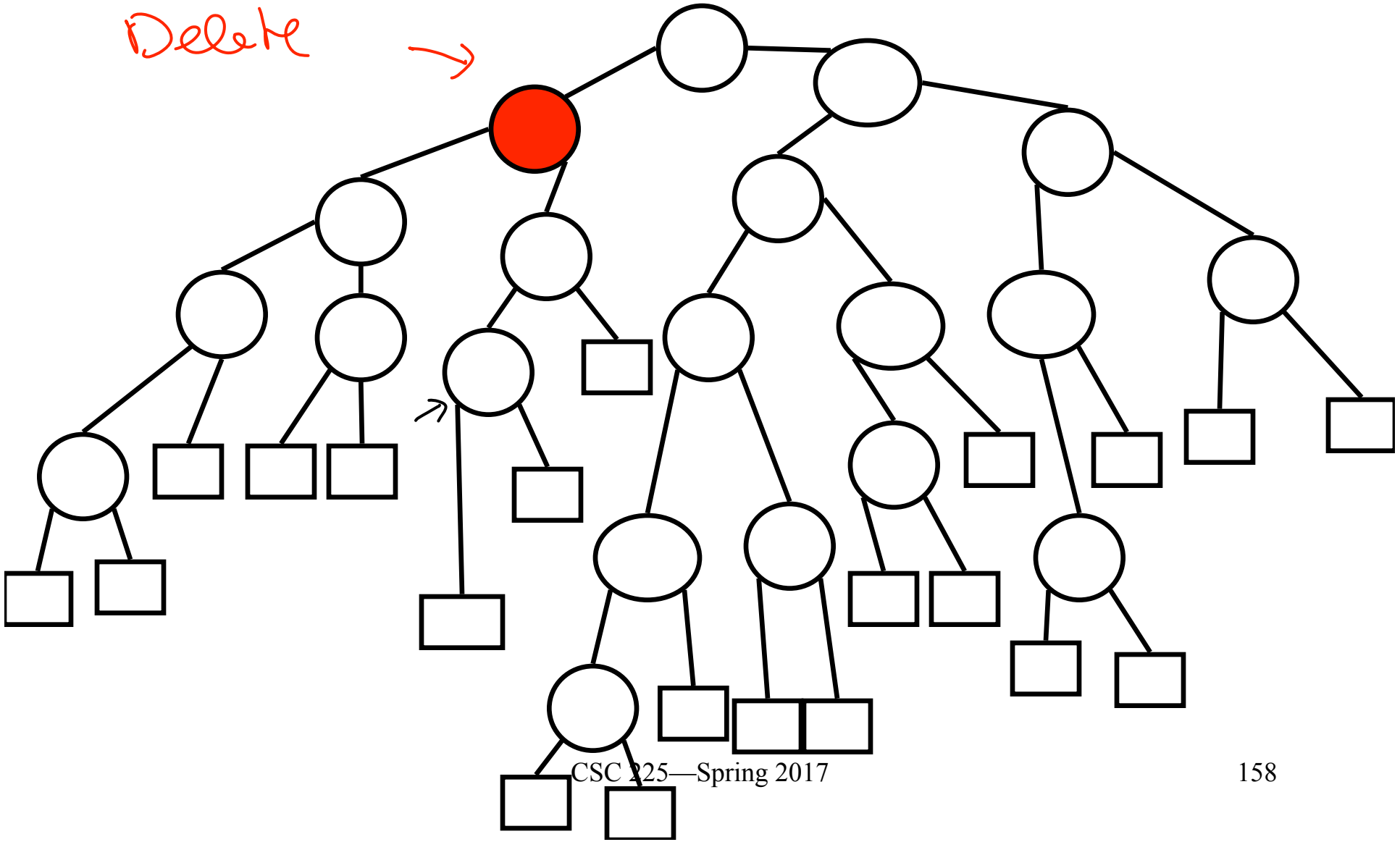
Restructure!



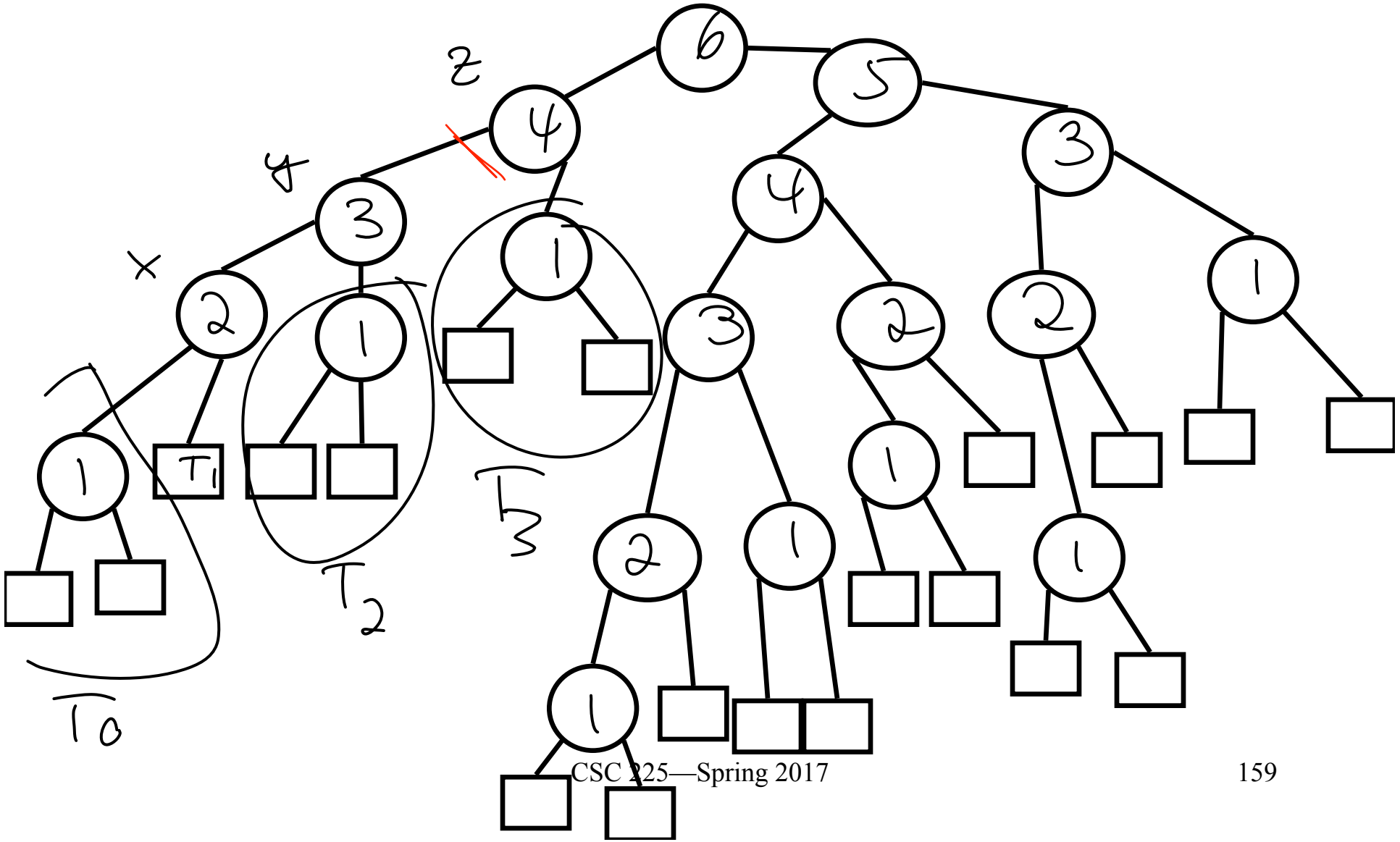
Restructure!



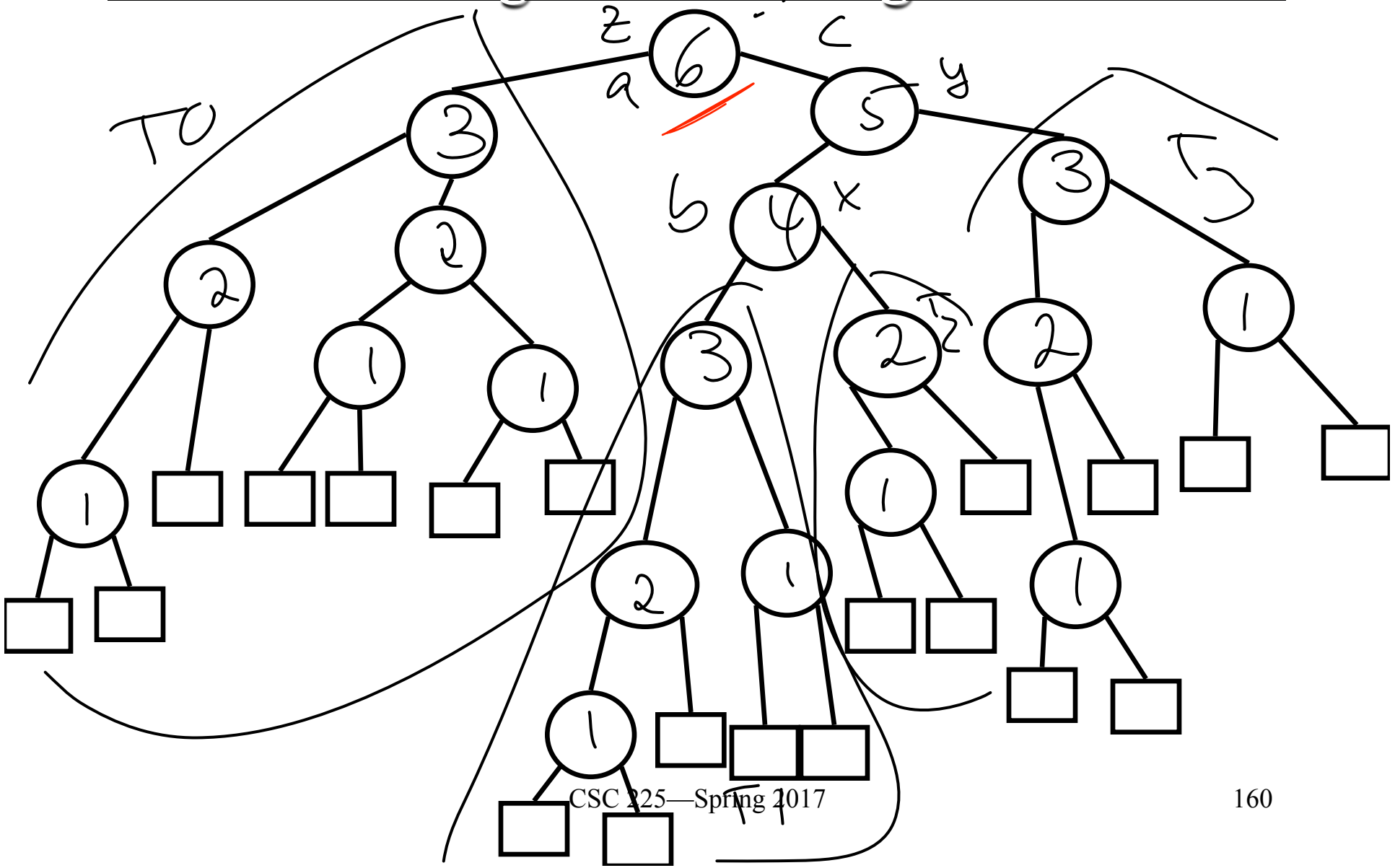
After Deletion: One Restructure-Operation might not be enough!



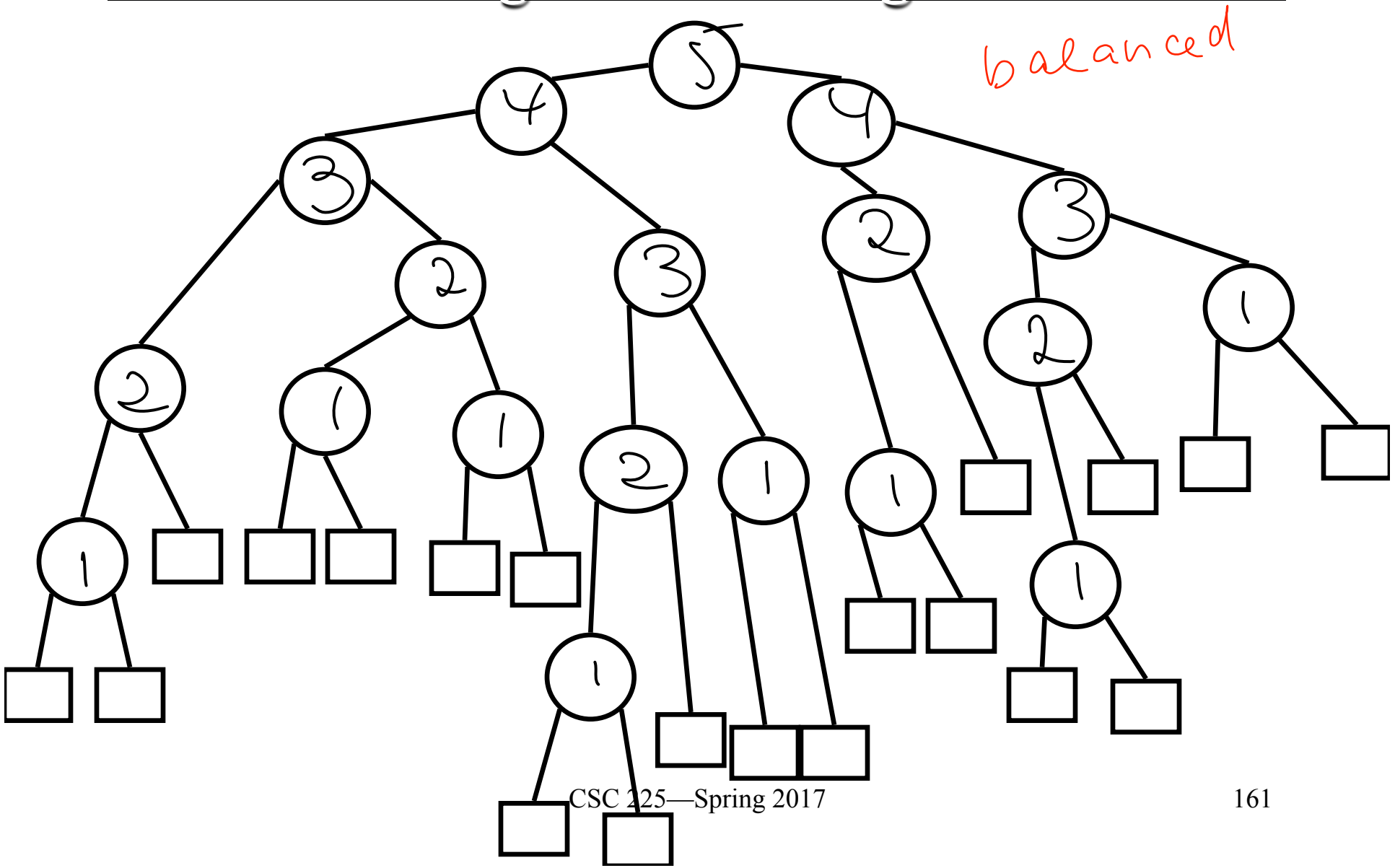
After Deletion: One Restructure-Operation might not be enough!



After Deletion: One Restructure-Operation might not be enough!



After Deletion: One Restructure-Operation might not be enough!



There can be as many as $O(\log n)$ many restructuring operations necessary to delete an element in an AVL tree.