Balanced Search Trees

- Why balanced search trees?
- Examples
  - AVL trees
  - 2-3 trees & red-black trees

Why balanced BSTs?

- Reminder: Definition of Binary Search Tree (BST)
- Search
- Insertion
- Deletion

Definition: Binary Search Tree (BST)

- A *binary search tree* (*BST*) is a binary tree where
  - each node has a (comparable) key and
  - satisfies the restriction that the key in any node is
    - larger than the keys in all nodes in that node's left subtree and
    - smaller than the keys in all nodes in that node's right subtree

Convention

- In a binary search tree, keys are stored in internal nodes only
- Every internal node in a binary search tree contains a key/an element with a key
- Every node as exactly two children (one or two of which can be leaves)

Search

- recursive
- follows structure of tree
- return the key's associated value if search successful; null otherwise

Insertion of new key

- Perform search (ends in leaf)
- replace leaf with new node containing the new key

Deletion

- Search key
  - key not found
  - key found

Deletion of existing key (key found)

- Three cases

  1. The node containing the key is parent of leaves (null) only

  2. The node containing the key is parent of one internal node only

  3. The node containing the key is parent of two internal nodes

Deletion of existing key

1. The node containing the key is parent of leaves (null) only
    - simply remove the node and replace it by a leaf

Deletion of existing key

2. The node containing the key is parent of one internal node only

    - Remove the internal node and replace it with the child that is an internal node

Deletion of existing key

3. The node x containing the key is parent of two internal nodes

- Identify the node y that is x's in-order successor
- Replace the content of x with the content of y
- Delete key of y in subtree rooted by node y
    - *Note:* node y will have at most one internal child node and thus case 1 or 2 will apply

Properties of binary search trees

- Height $O(n)$
- Worst-Case Time complexity
  - Search $O(n)$
  - Insertion $O(n)$
  - Deletion $O(n)$

Balanced Search Trees

- Why balanced search trees?
  - unbalanced search trees are not efficient due to height $O(n)$
- Examples
  - AVL trees
  - 2-3 trees & red-black trees

AVL Trees

- AVL trees are ***height balanced*** binary search trees.
- Idea: balance of height avoids linear running time of dictionary operations
- Inventors: 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, the heights of the children of v differ by at most 1.

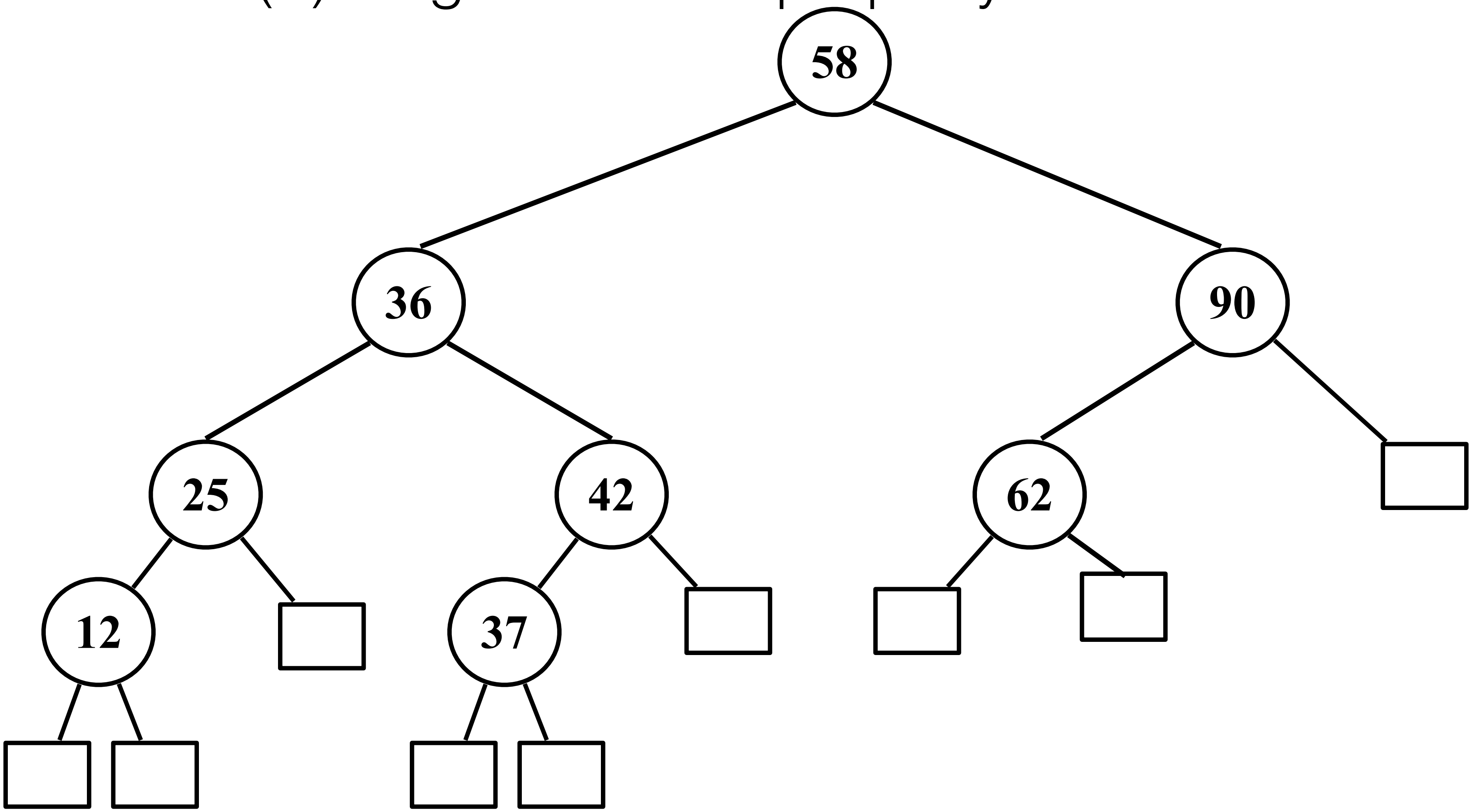
Balanced and unbalanced nodes

- 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 v's children is at most 1.
- Otherwise, we say v is *unbalanced*.

It follows: A binary search tree T is an AVL-tree iff every node in T is balanced.

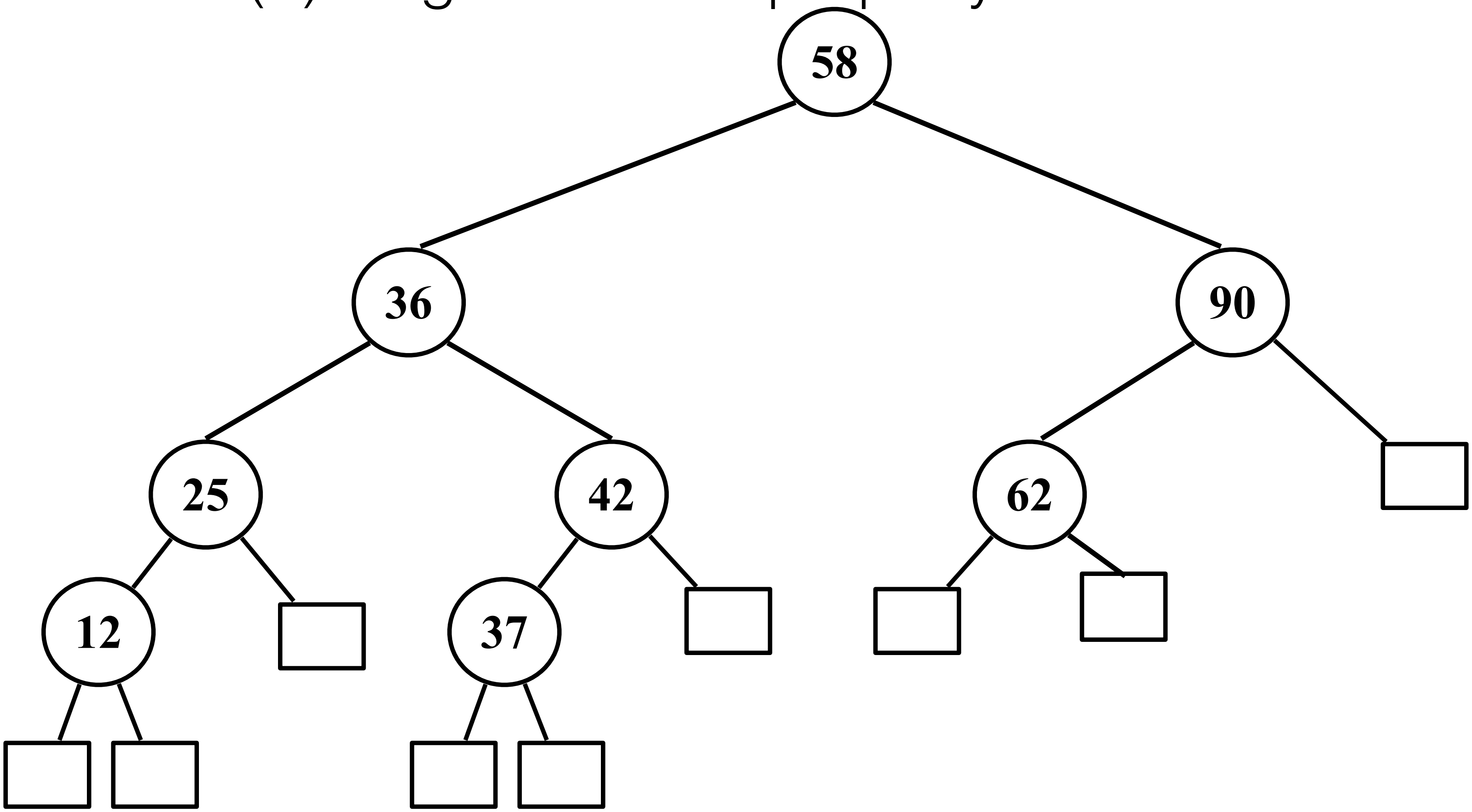
Is this an AVL-tree?

To verify: (1) binary search tree
(2) height-balance property



Is this an AVL-tree?

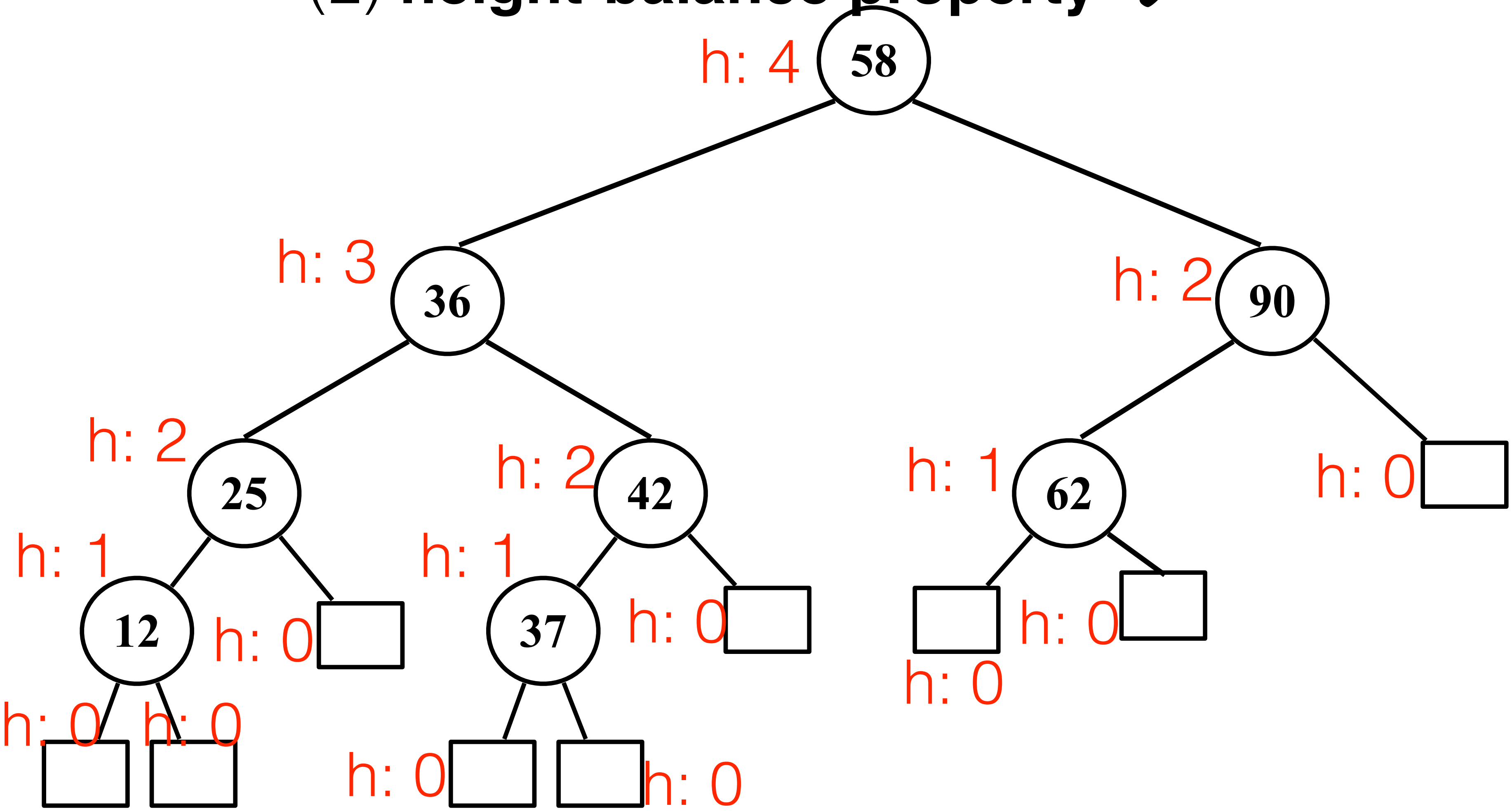
To verify: (1) **binary search tree** ✔
(2) height-balance property



Is this an AVL-tree?

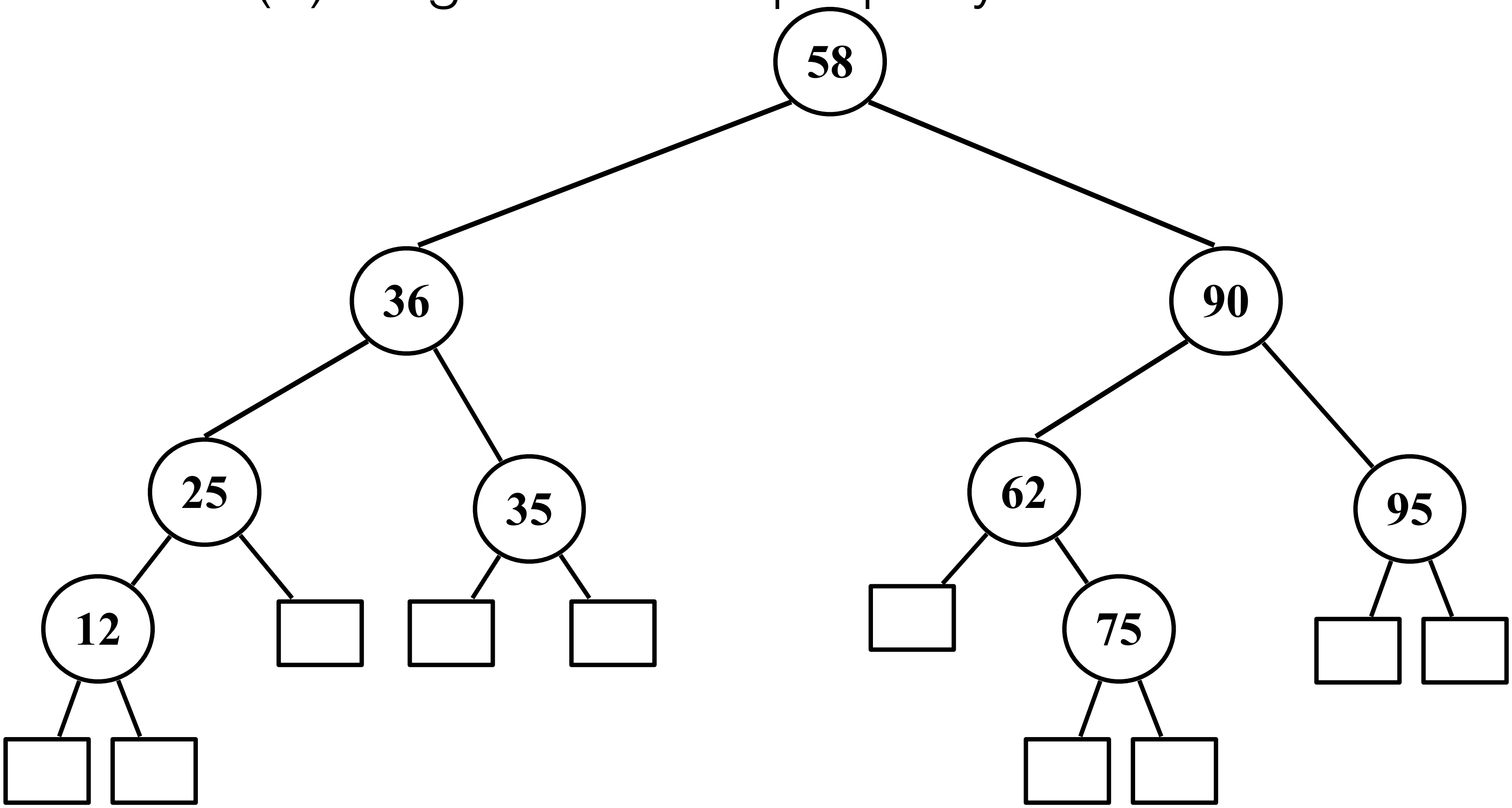
To verify: (1) binary search tree ✔

(2) **height-balance property** ✔



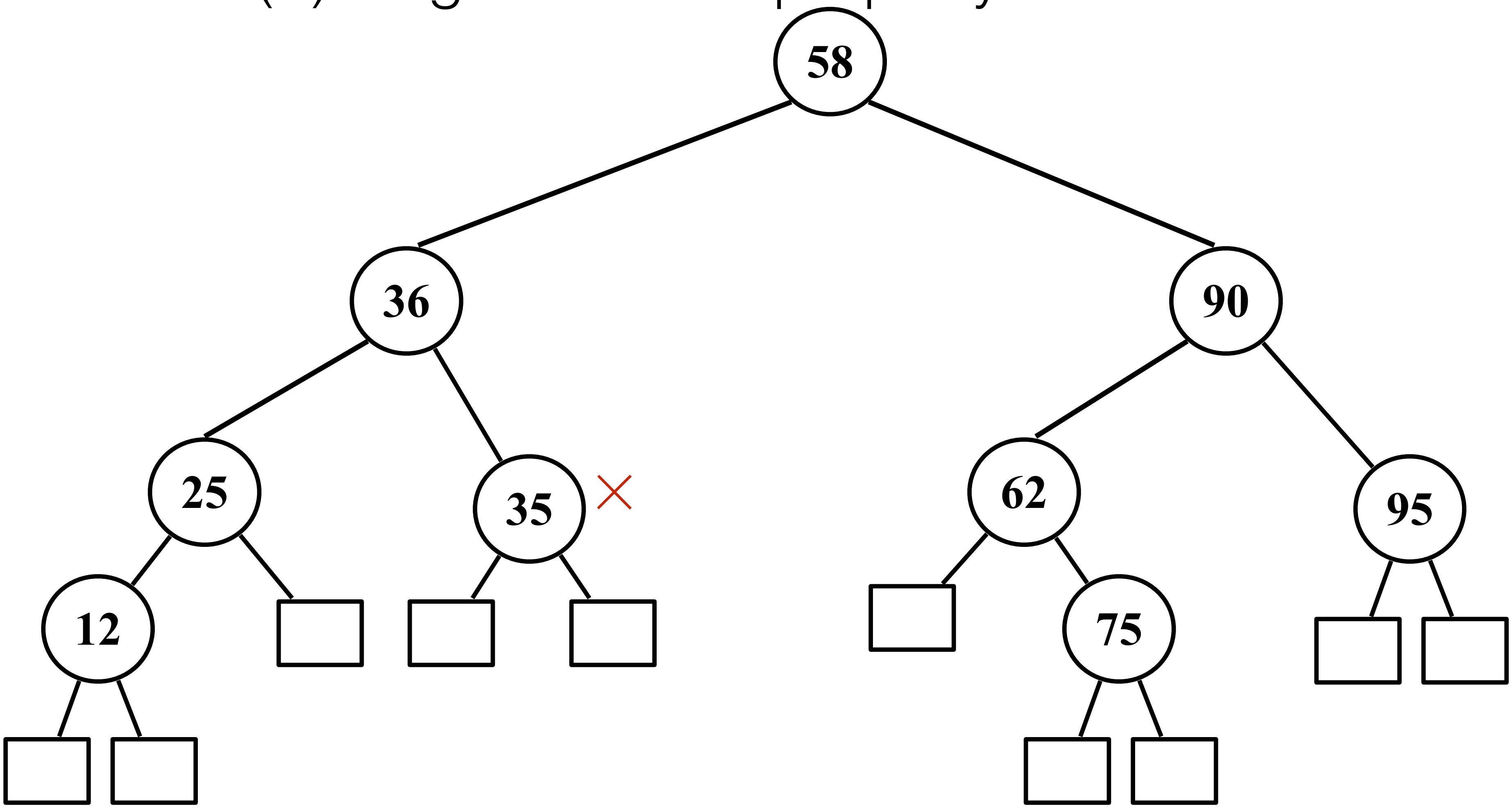
Is this an AVL-tree?

To verify: (1) binary search tree
(2) height-balance property



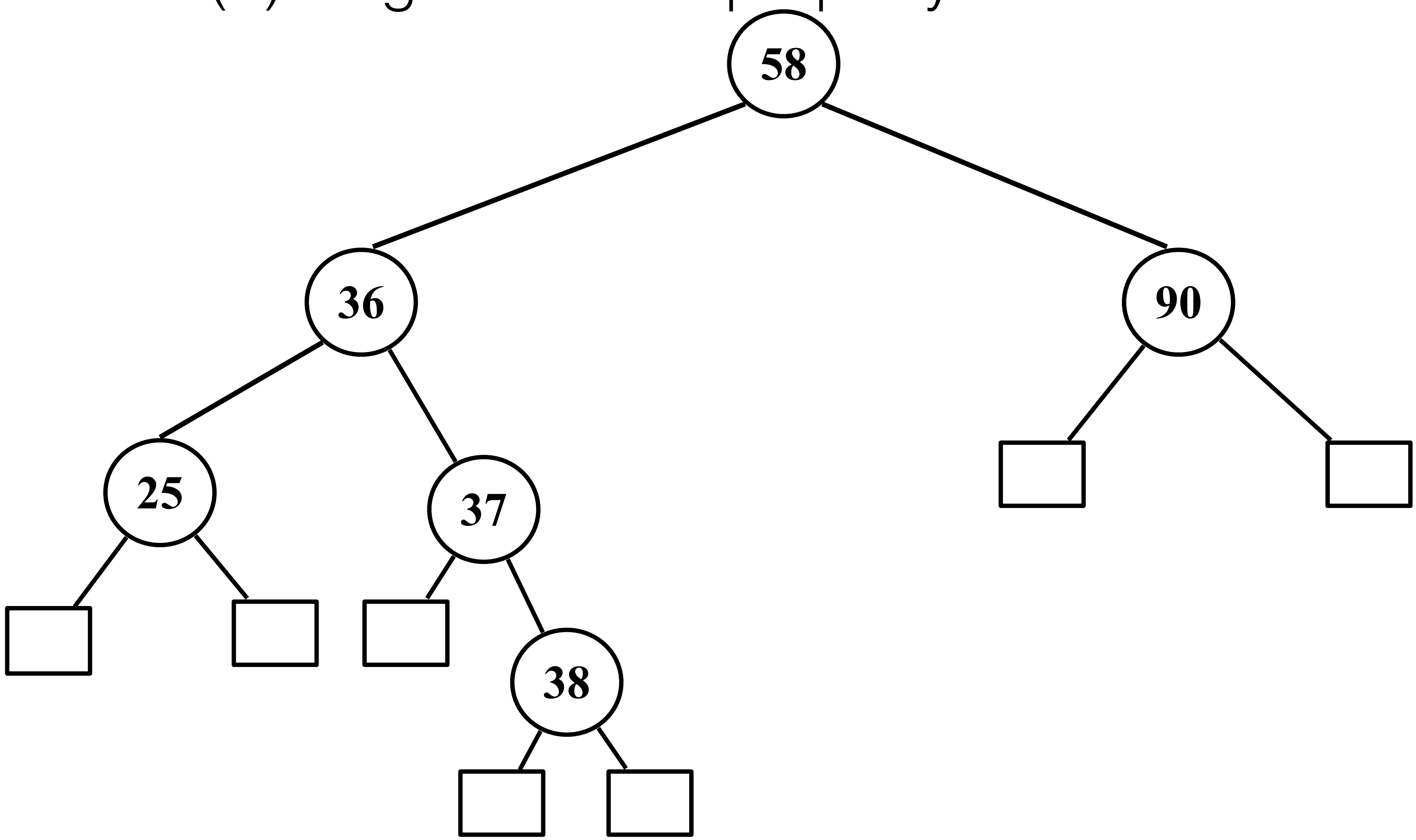
Is this an AVL-tree?

To verify: (1) **binary search tree** ×
(2) height-balance property



Is this an AVL-tree?

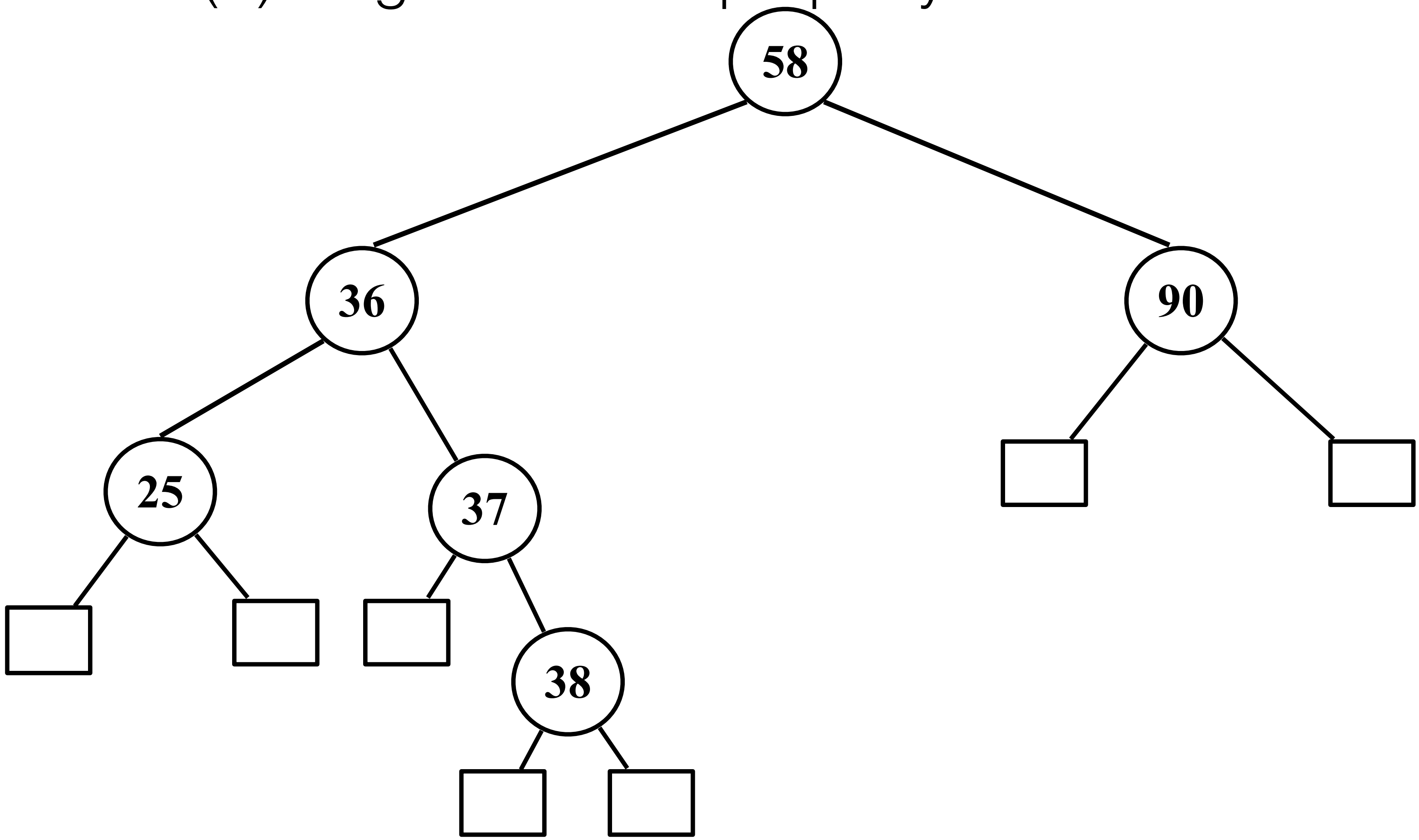
To verify: (1) binary search tree
(2) height-balance property



Is this an AVL-tree?

To verify: (1) **binary search tree** ✔

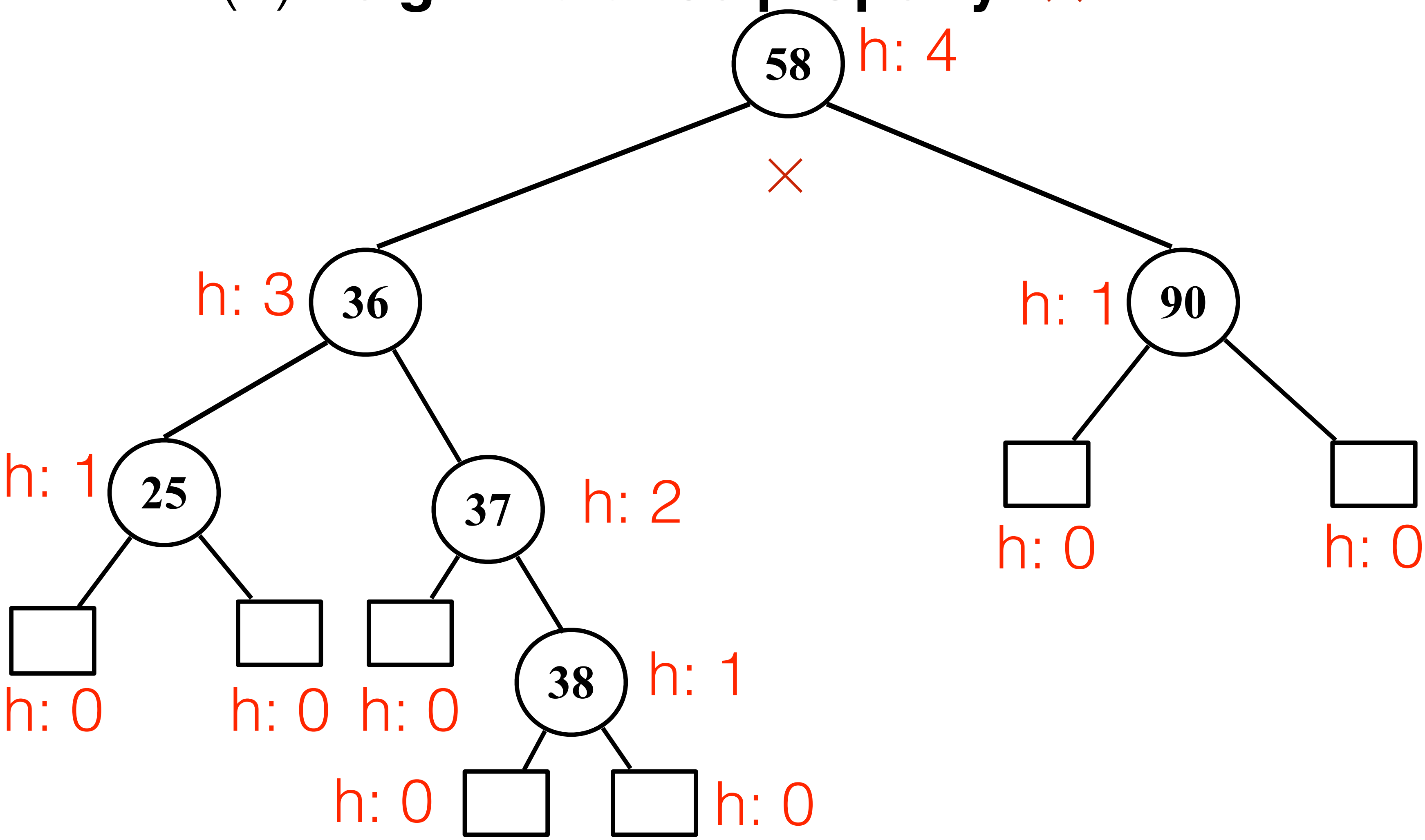
(2) height-balance property



Is this an AVL-tree?

To verify: (1) binary search tree ✔

(2) **height-balance property** ×



The height of an AVL-tree

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

Proof. $\mathrm{n}(h)$: minimum number of internal nodes of AVL-tree of height h

Then: $\mathrm{n}(1) = 1, \ \mathrm{n}(2) = 2$ and

$$\mathrm{n}(h) = 1 + \mathrm{n}(h-1) + \mathrm{n}(h-2) \text{ for } h \geq 2$$

We show:

(1) $\mathrm{n}(h)$ grows exponentially

(2) (1) implies: height of an AVL-tree storing n keys is $\mathrm{O}(\log(n))$

(1)We show that

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

We show the claim using induction on i.

Induction hypothesis:

$$\boxed{\mathrm{n}(h) > 2^i\, \mathrm{n}(h-2i) \text{ for any integer } i \text{ such that } h - 2i \geq 1}$$

Base case: $i = 1$

$\mathrm{n}(h-1) = \mathrm{n}(h-2) + \mathrm{n}(h-3) + 1$

$$\begin{aligned}
\mathrm{n}(h) &= \mathrm{n}(h-1) + \mathrm{n}(h-2) + 1 \\
&= \mathrm{n}(h-2) + \mathrm{n}(h-3) + 1 + \mathrm{n}(h-2) + 1 \\
&= 2\mathrm{n}(h-2) + \mathrm{n}(h-3) + 2 \\
&> 2\mathrm{n}(h-2)
\end{aligned}$$

Induction step: $i \longrightarrow i + 1$

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

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

Therefore,

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

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

and further

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

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

which yields

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

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

This proves the claim!

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

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

and therefore $\mathrm{n(h)}$ grows exponentially

(2) To show: (1) implies that height of an AVL-tree storing n keys is $\mathrm{O}(\log(n))$

Since $\mathrm{n}(h) \geq 2^{\lceil h/2 \rceil}$: $\log(\mathrm{n}(h)) > h/2$.

Thus $2 \log(\mathrm{n}(h)) > h$.

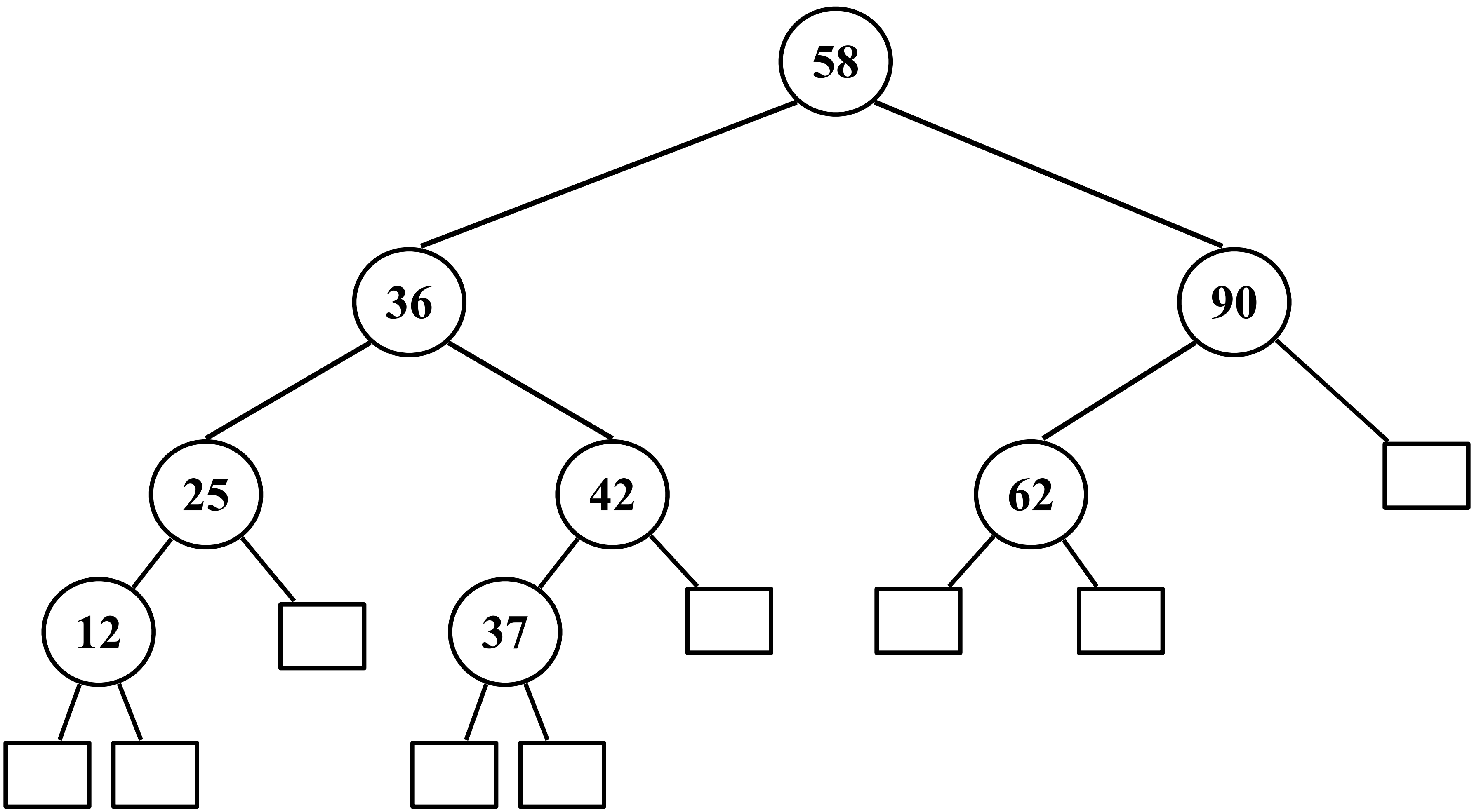
We conclude: if $\mathrm{n}(h) = n$ then $h < 2 \log(n)$, that is h is $\mathrm{O}(\log(n))$.\

Updating AVL trees

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

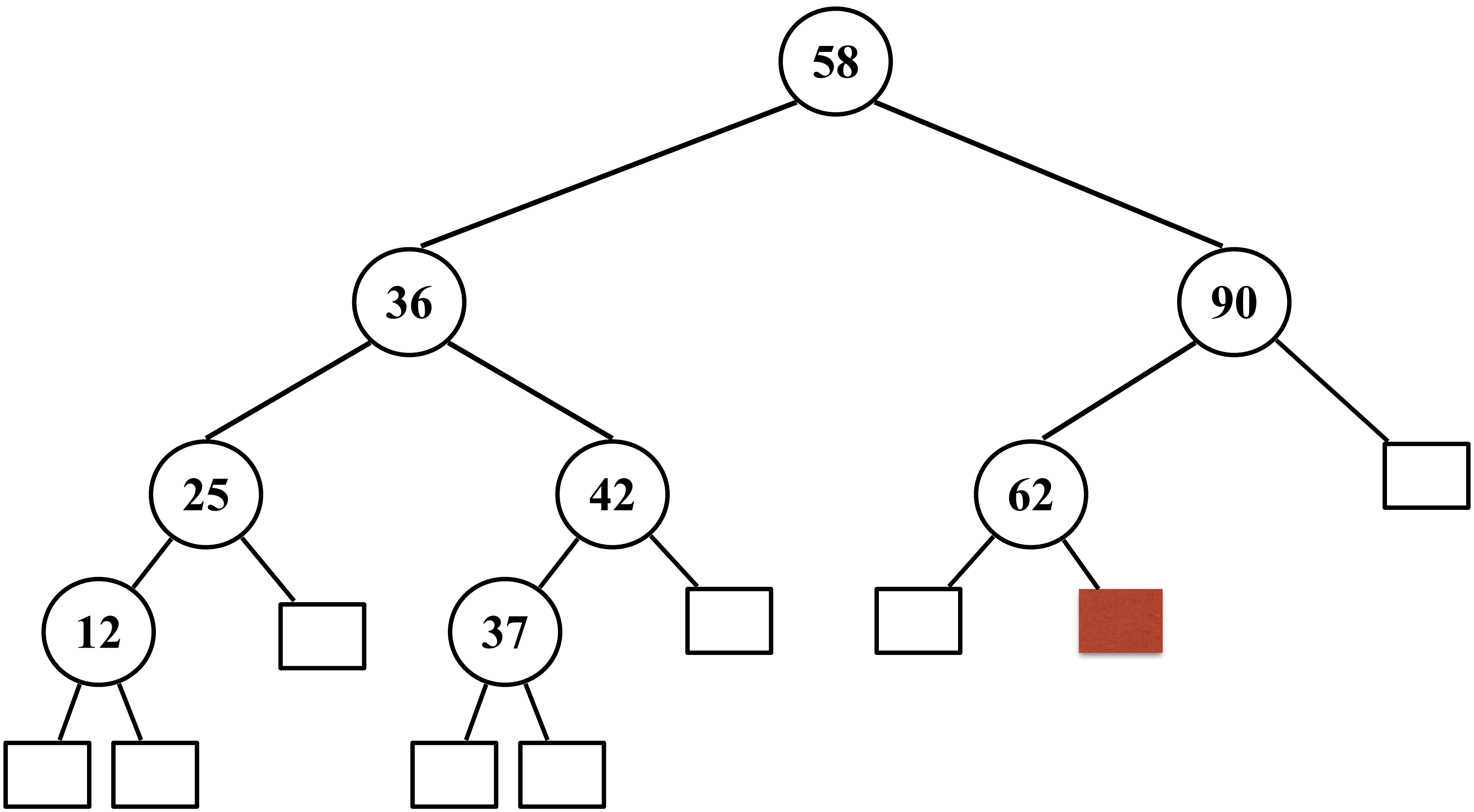
Insertion

Insert element 78



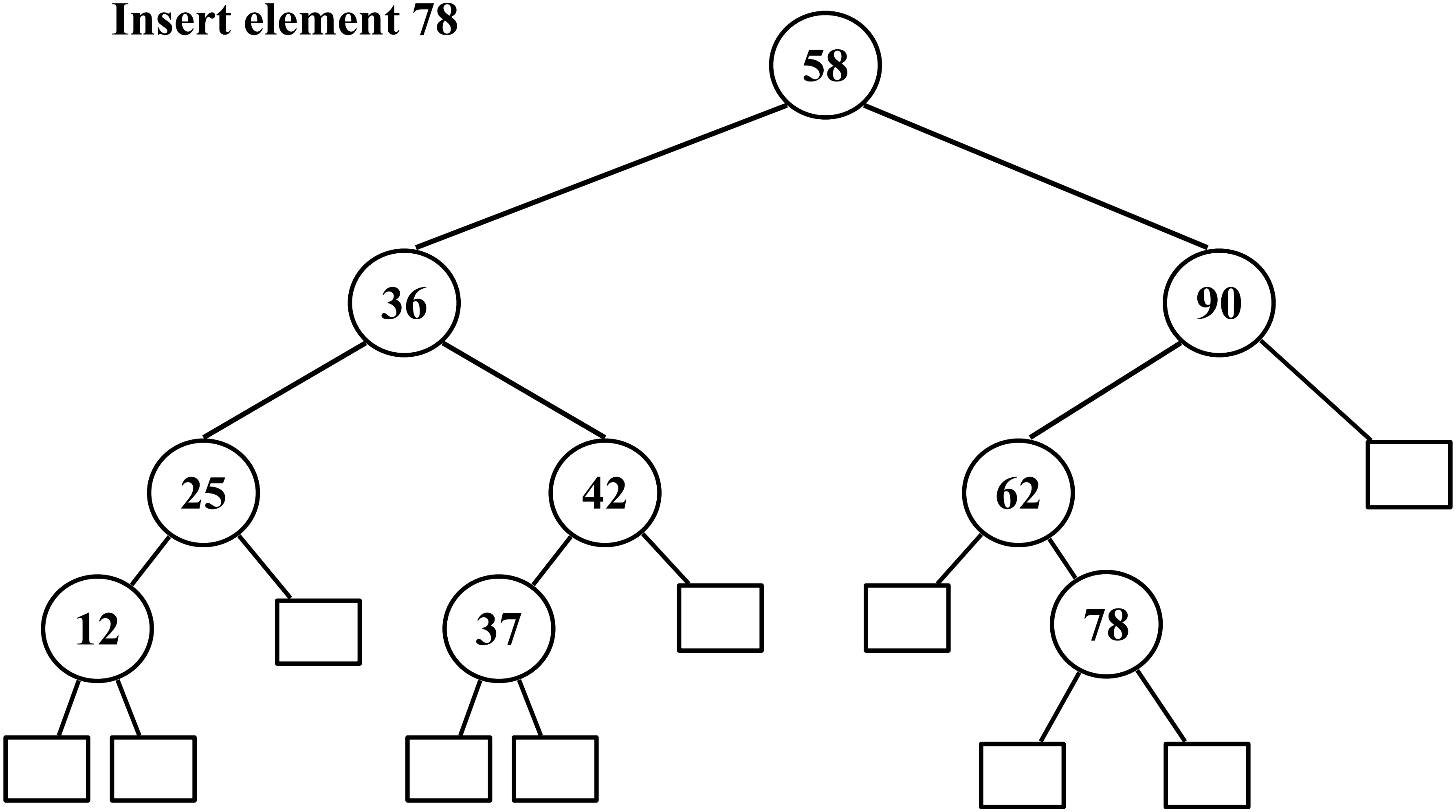
Insertion

Insert element 78



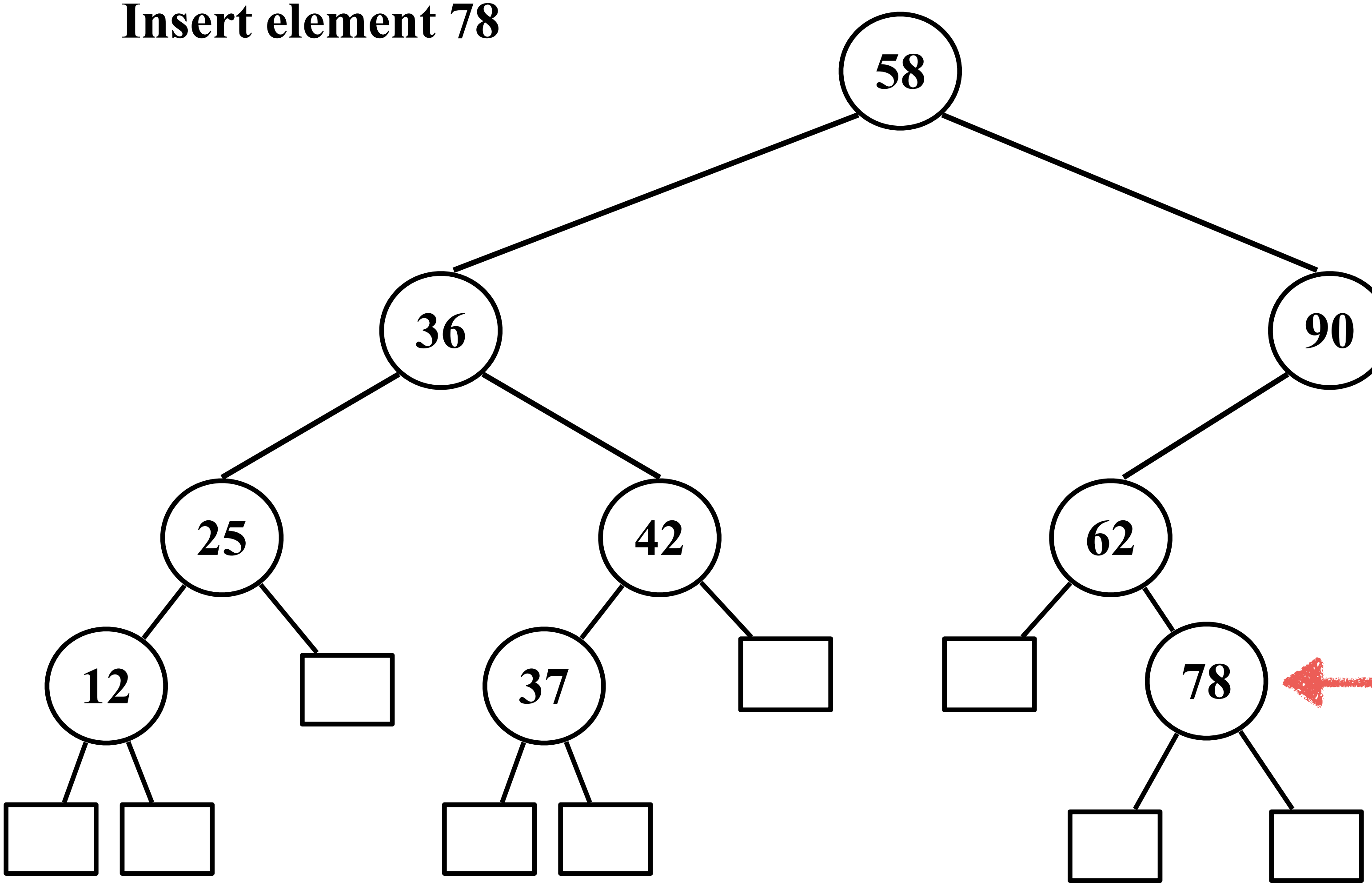
Insertion – Step 1: Insert as in binary search trees

Insert element 78



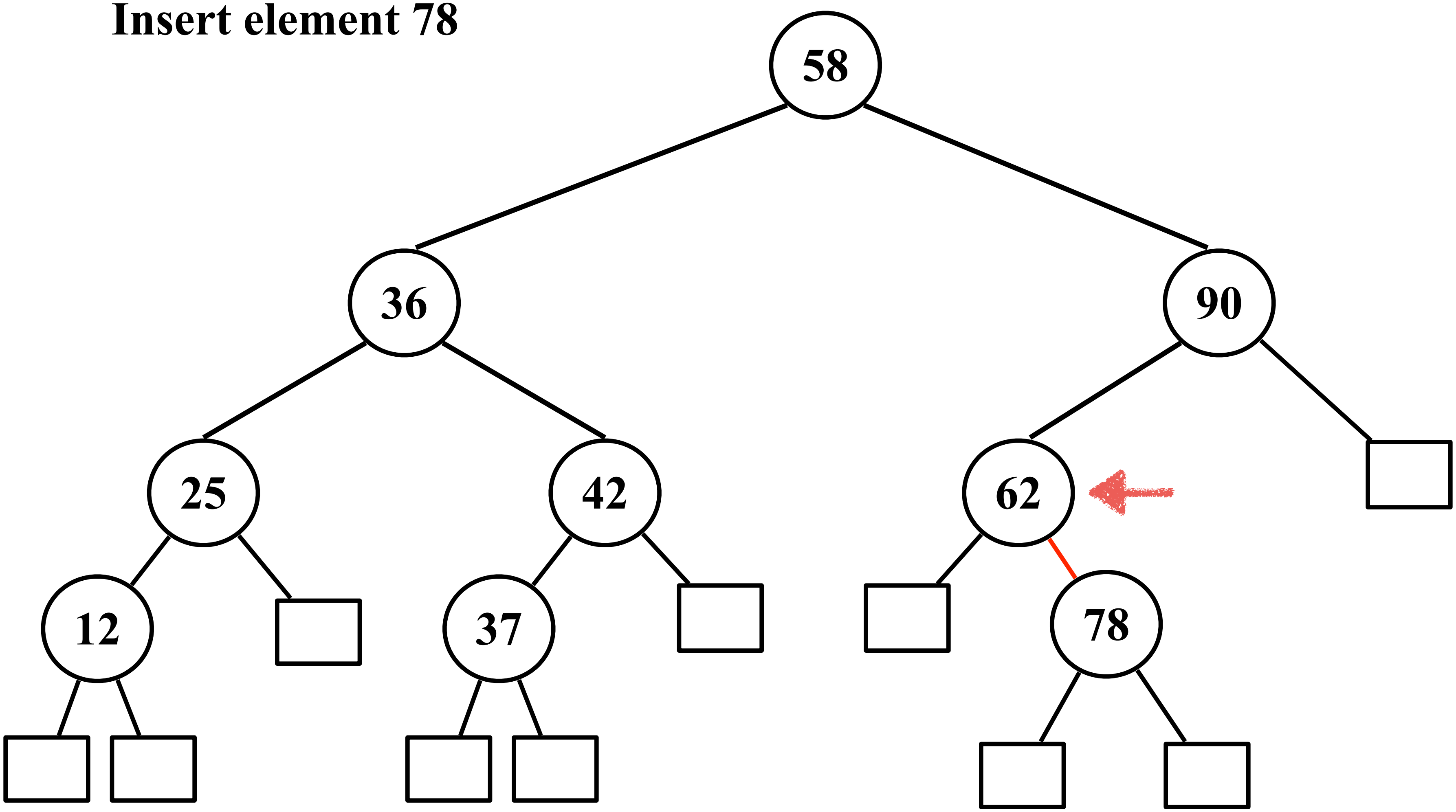
Insertion – Step 2: Check balance

Insert element 78



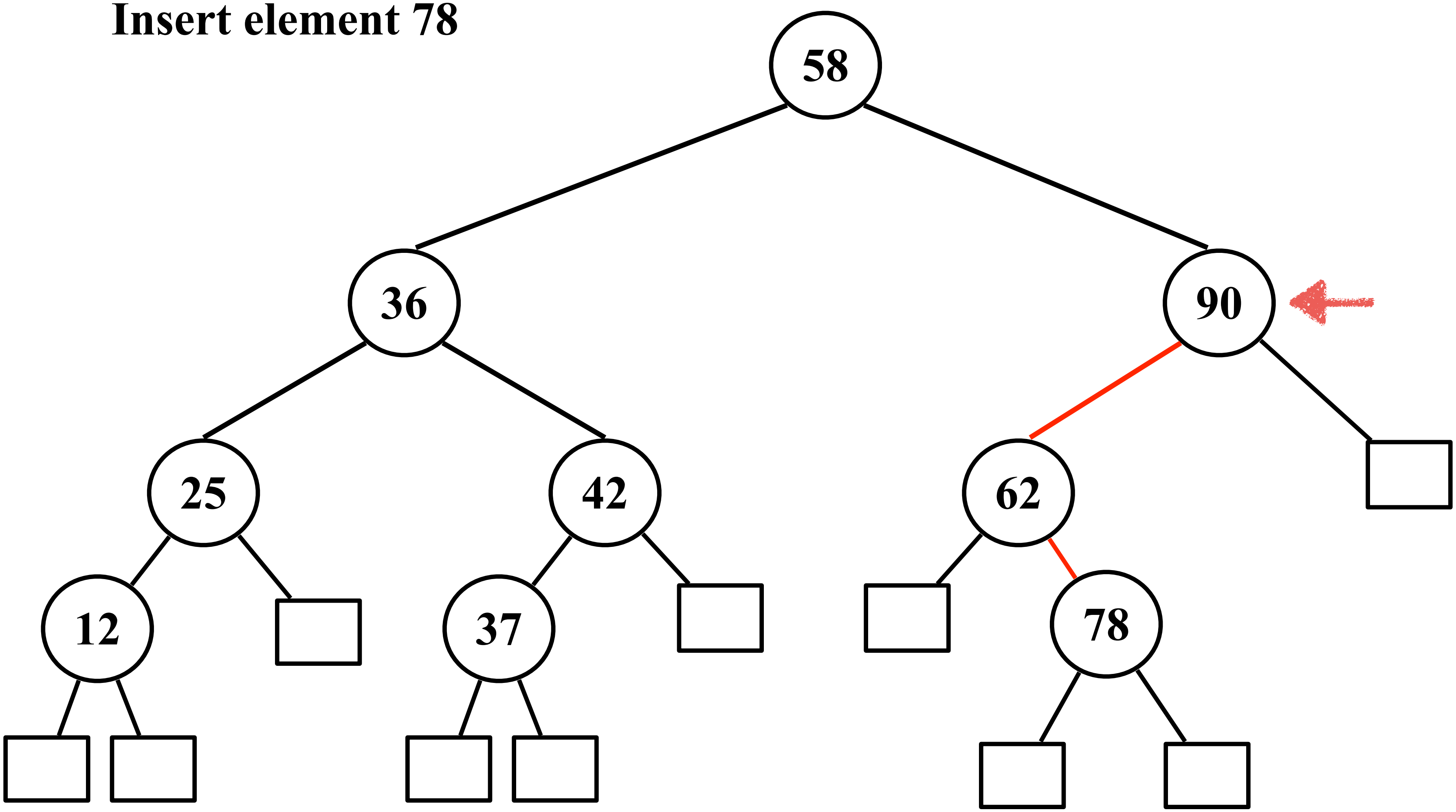
Insertion – Step 2: Check balance

Insert element 78



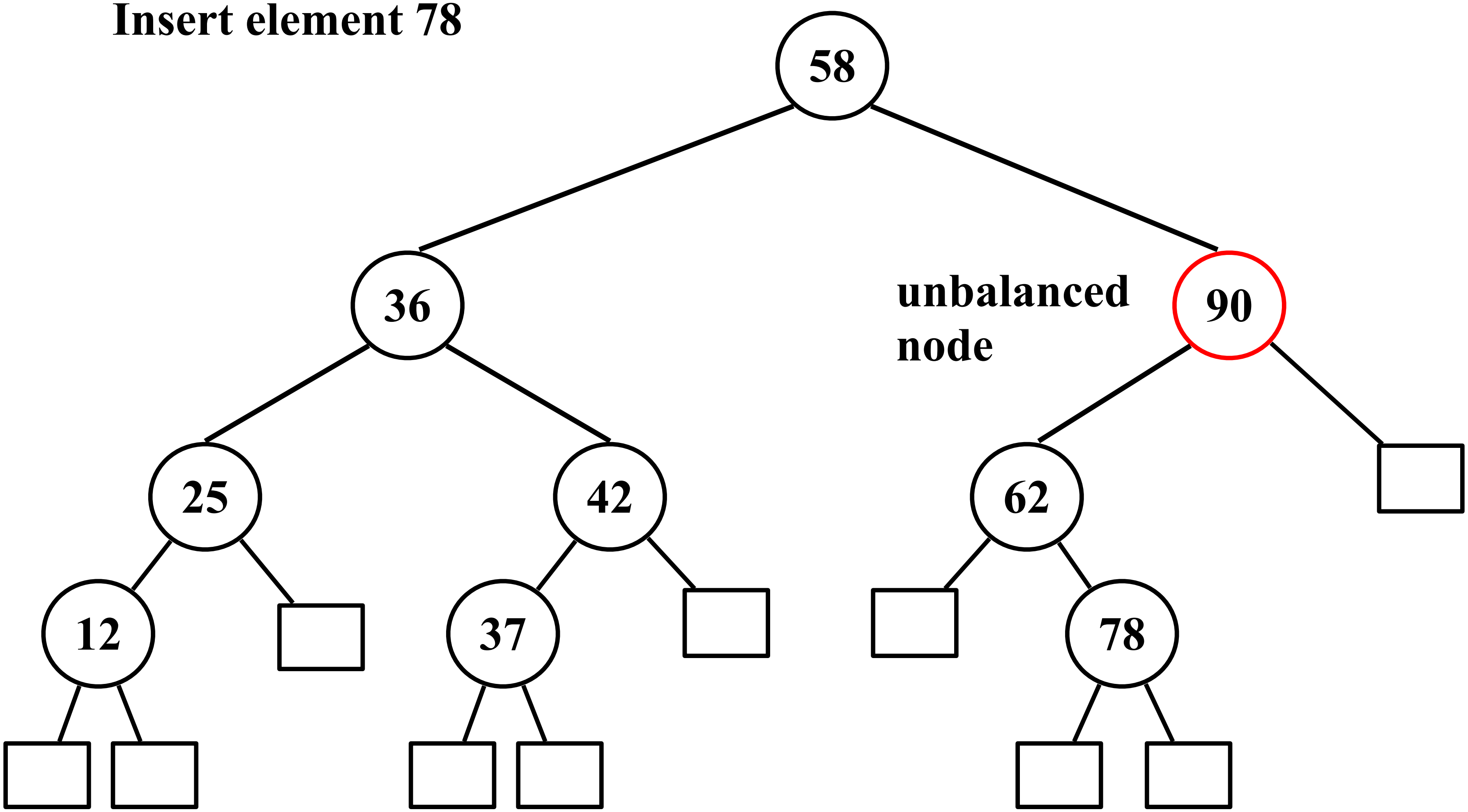
Insertion – Step 2: Check balance

Insert element 78

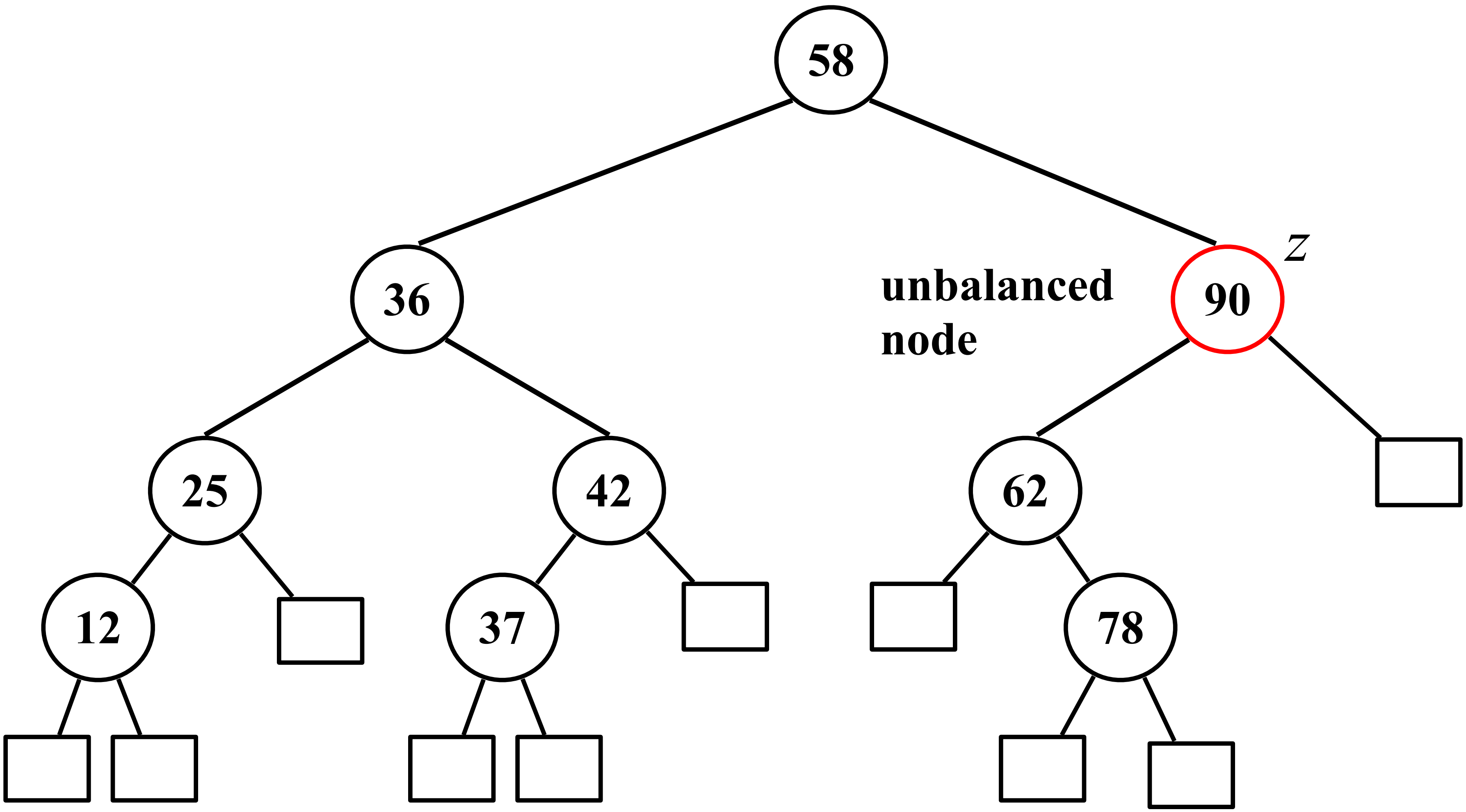


Insertion – Step 2: Check balance

Insert element 78



Insertion – Step 3: Fix the Unbalance!

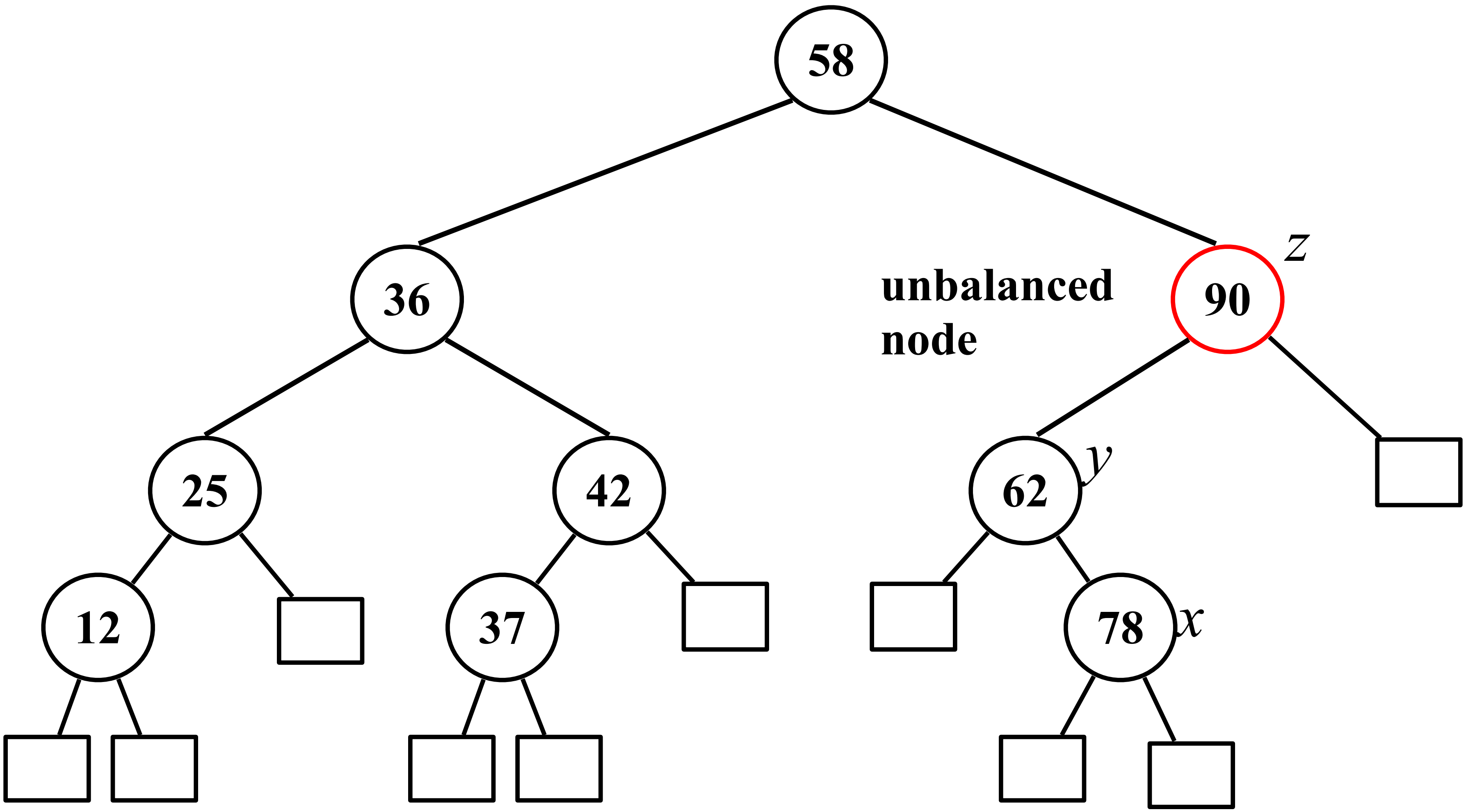


Algorithm restructure(x)

Input: A node x of a binary search tree T that has both a parent y and a grandparent z (z is the unbalanced node, y its child in its higher subtree, and x is y's child in y's higher subtree)

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

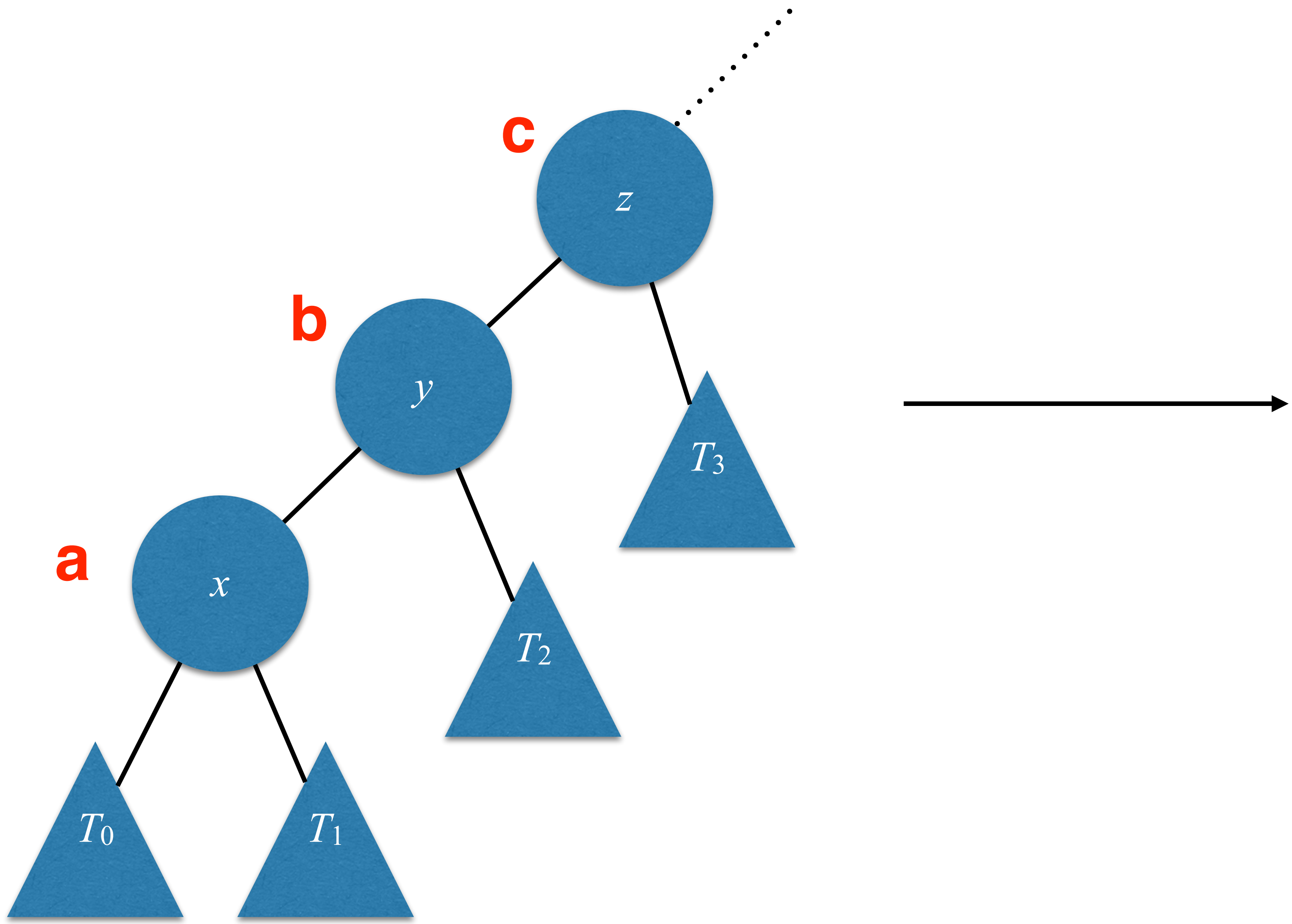
Fix the Unbalance!



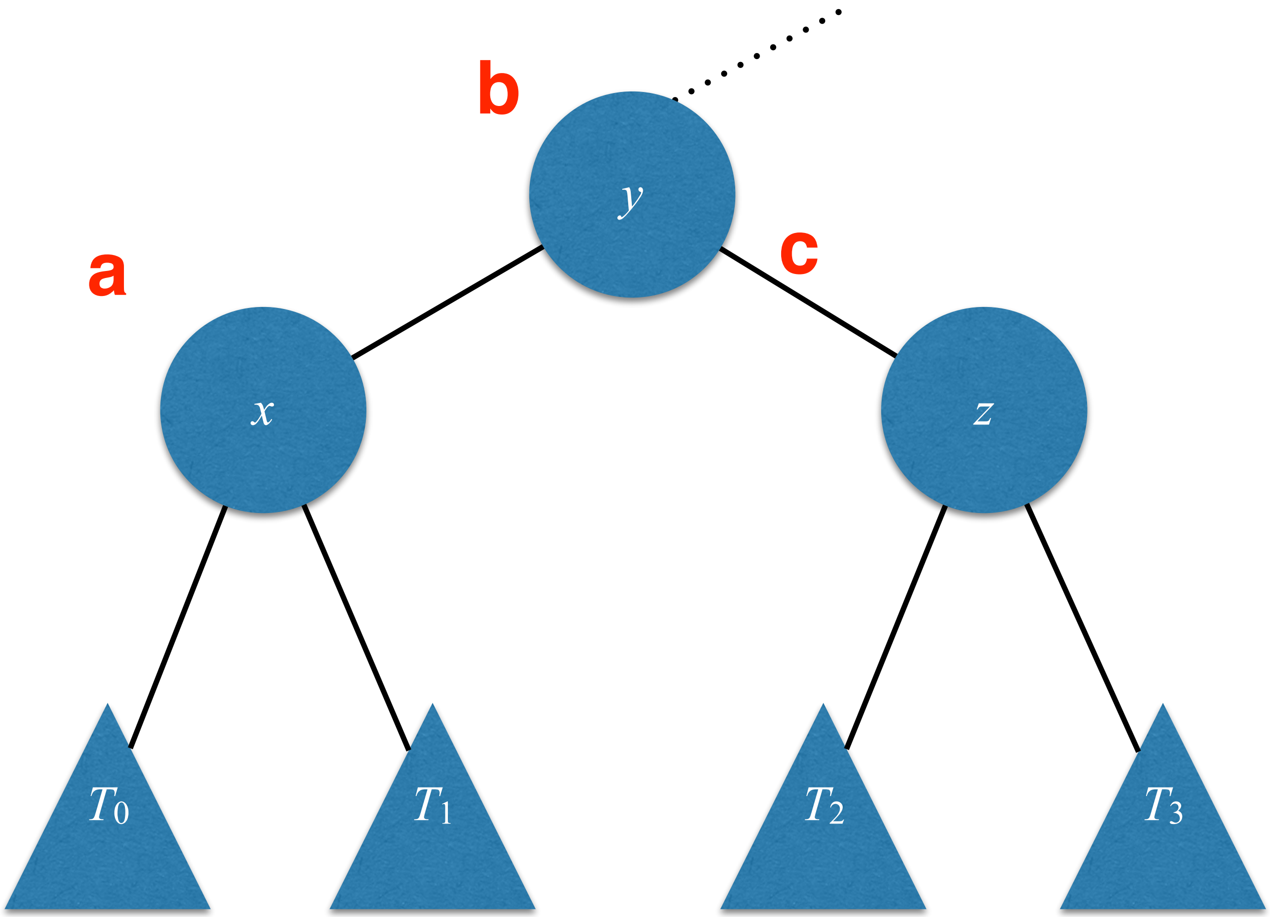
Algorithm restructure(x)

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

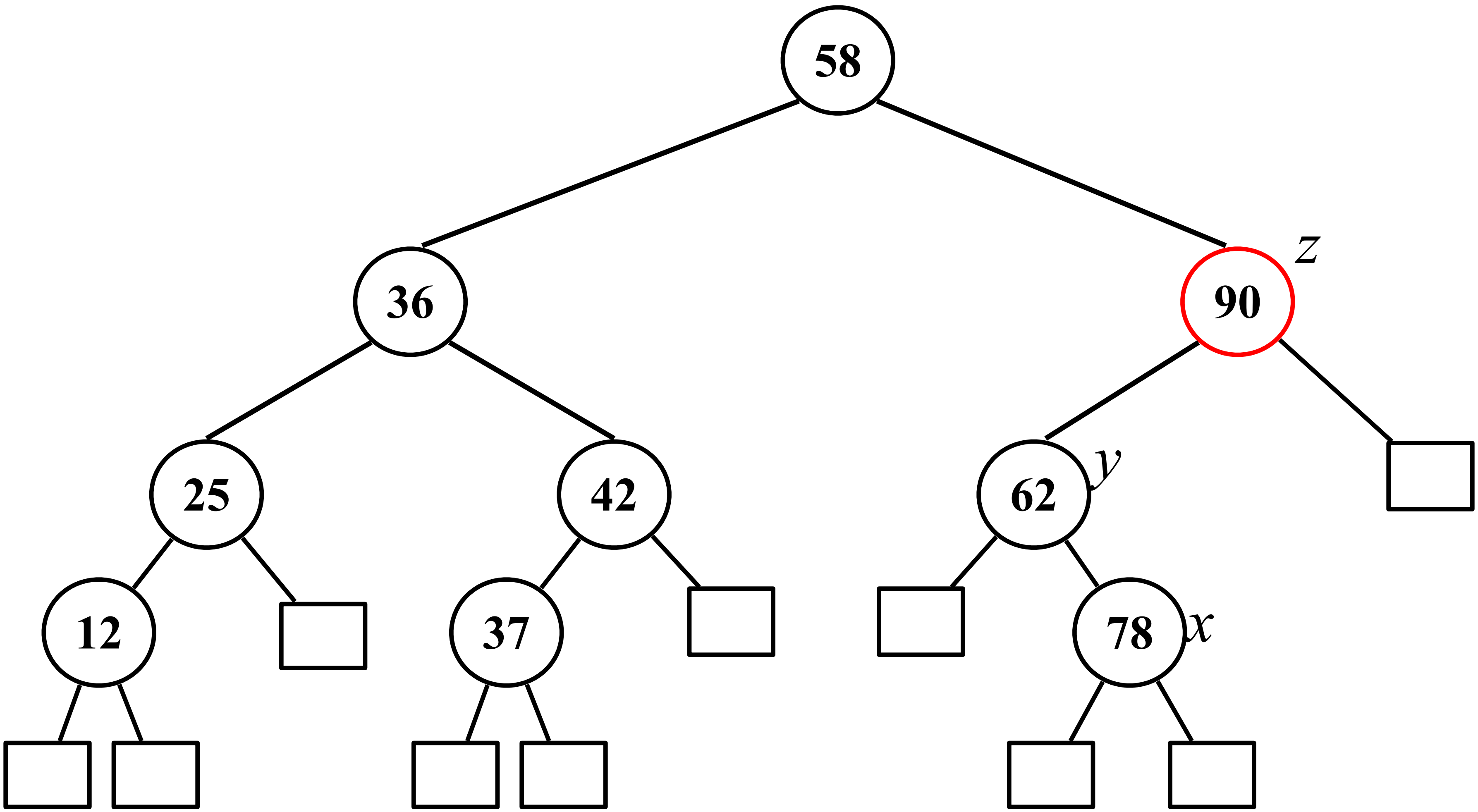
A case of **Algorithm** restructure(x)



A case of **Algorithm** restructure(x)

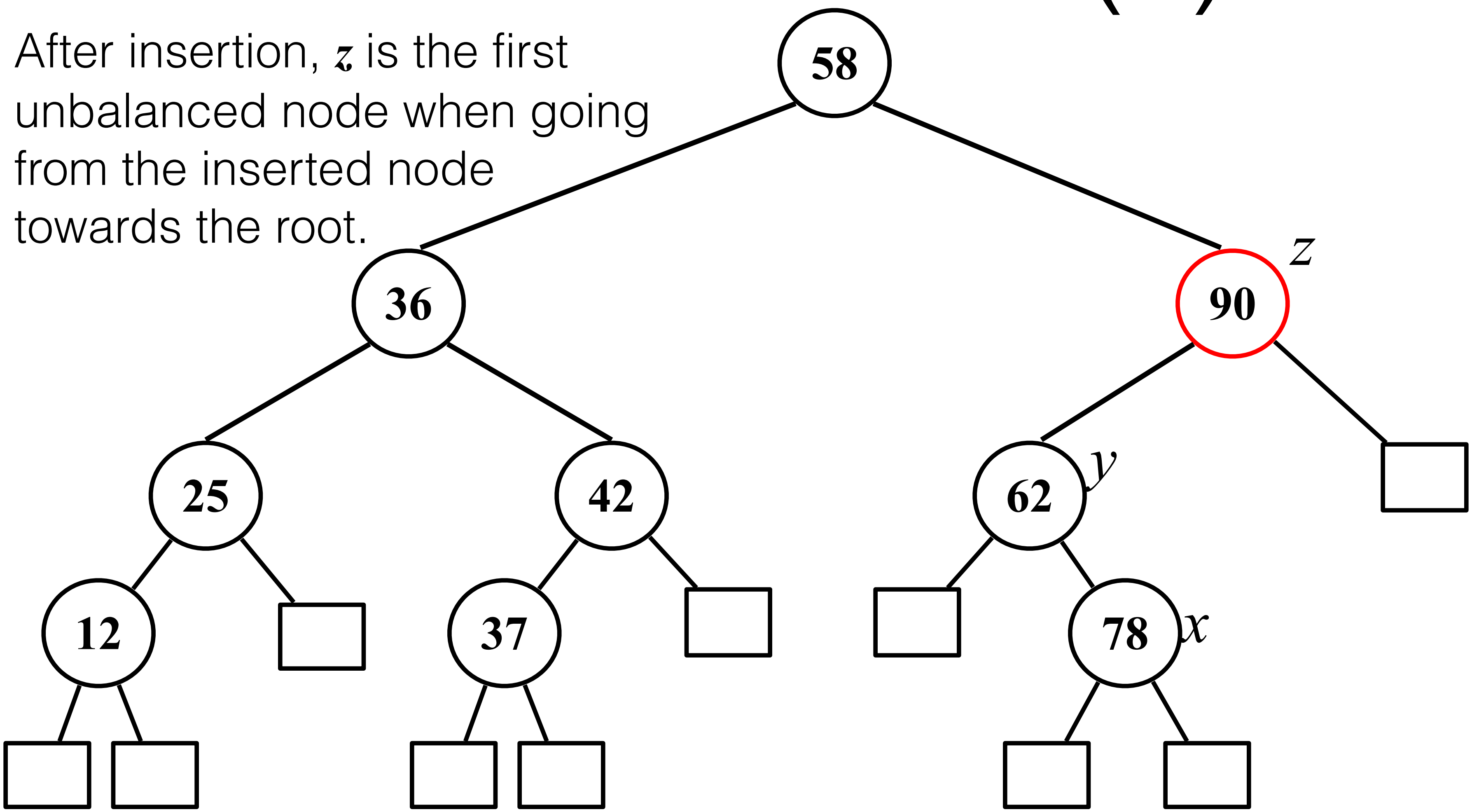


Fix the Unbalance!



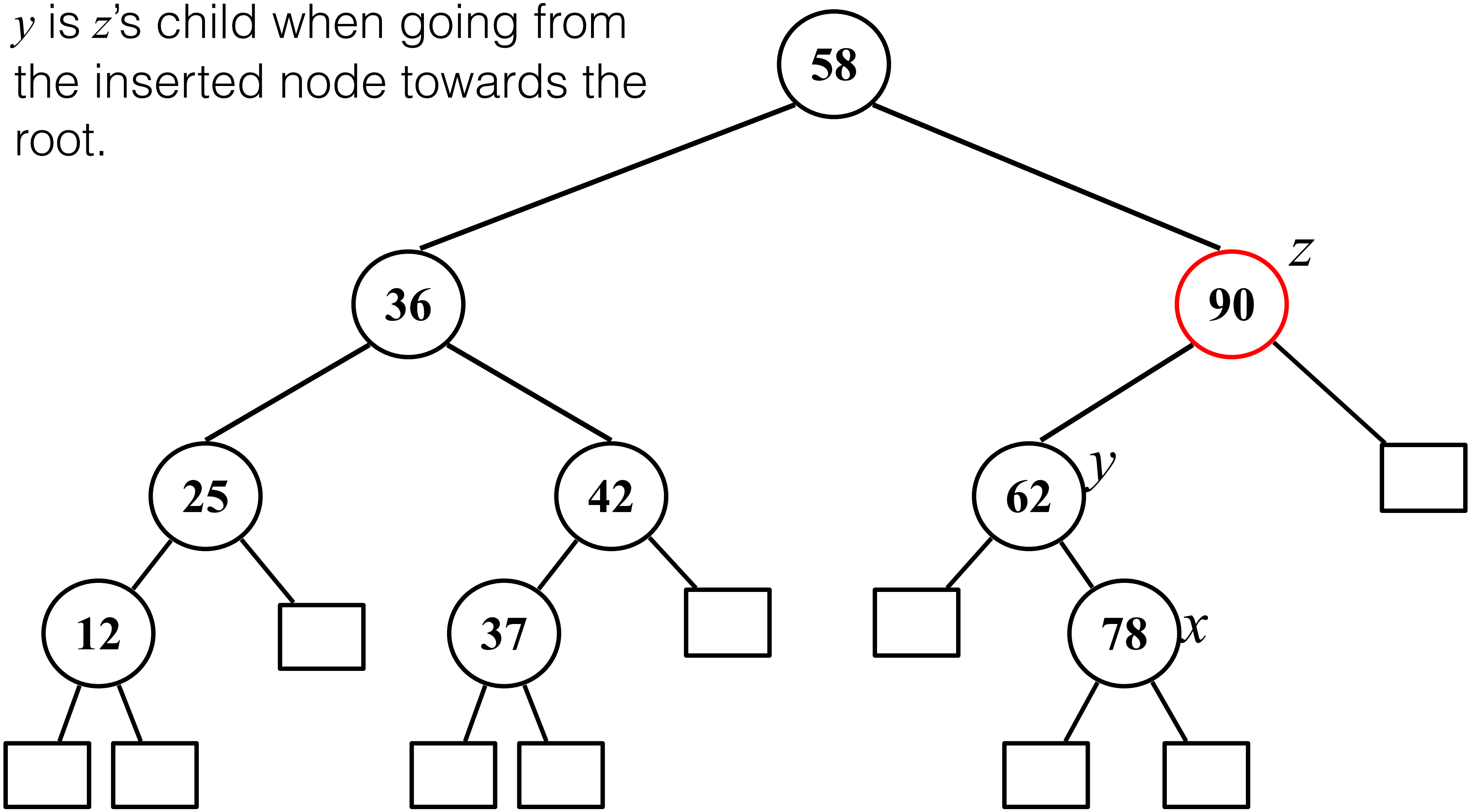
restructure(x)

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



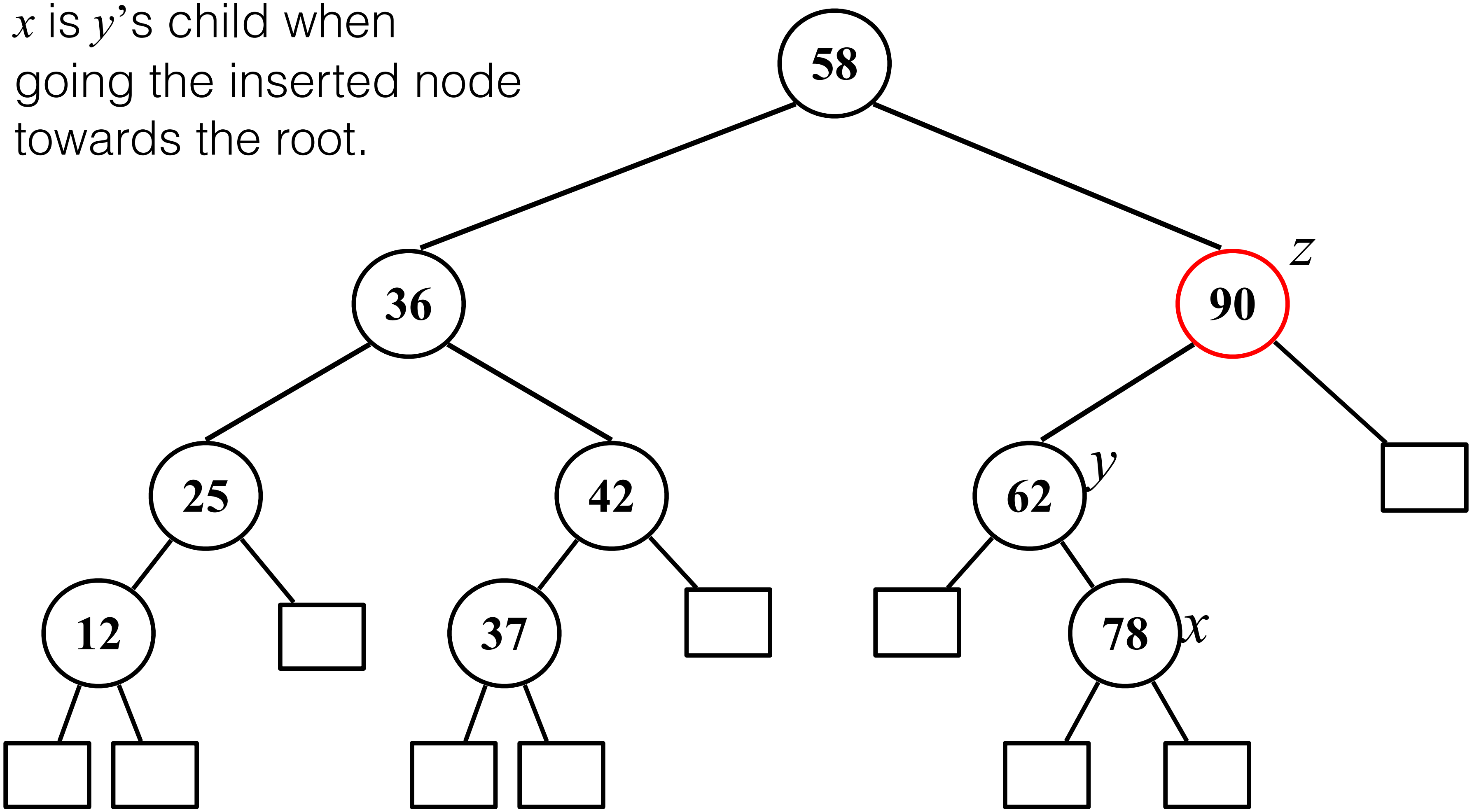
restructure(x)

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



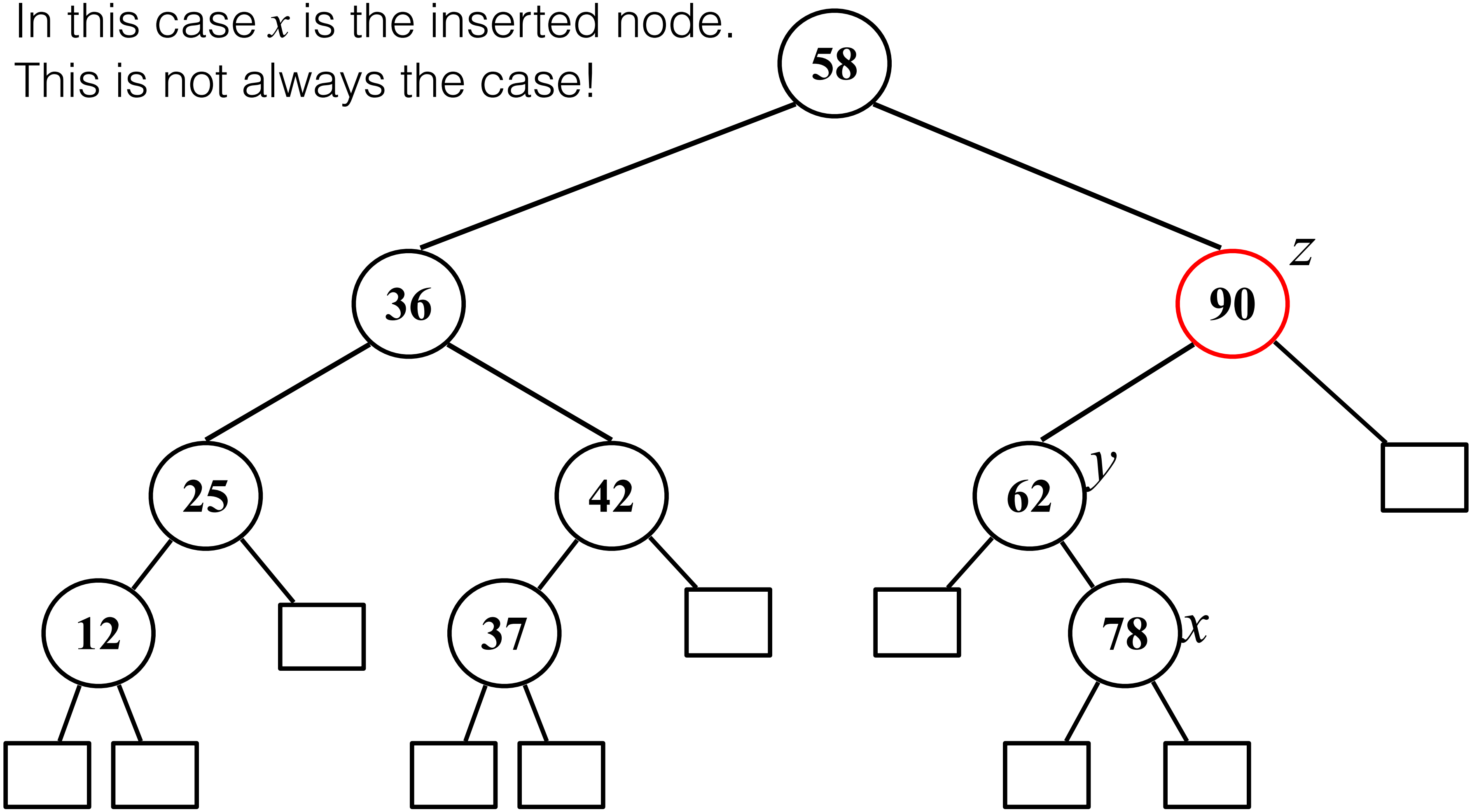
restructure(x)

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

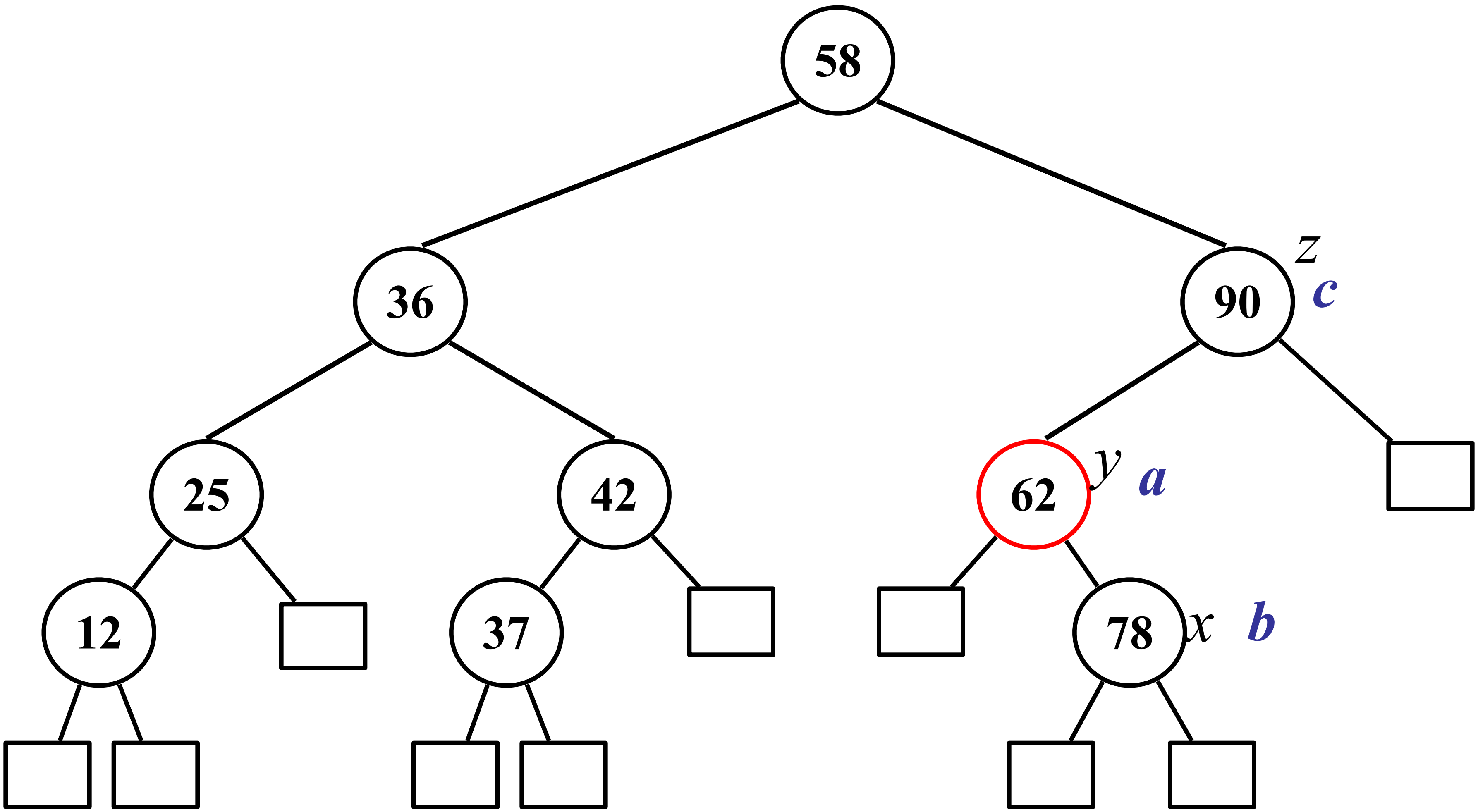


restructure(x)

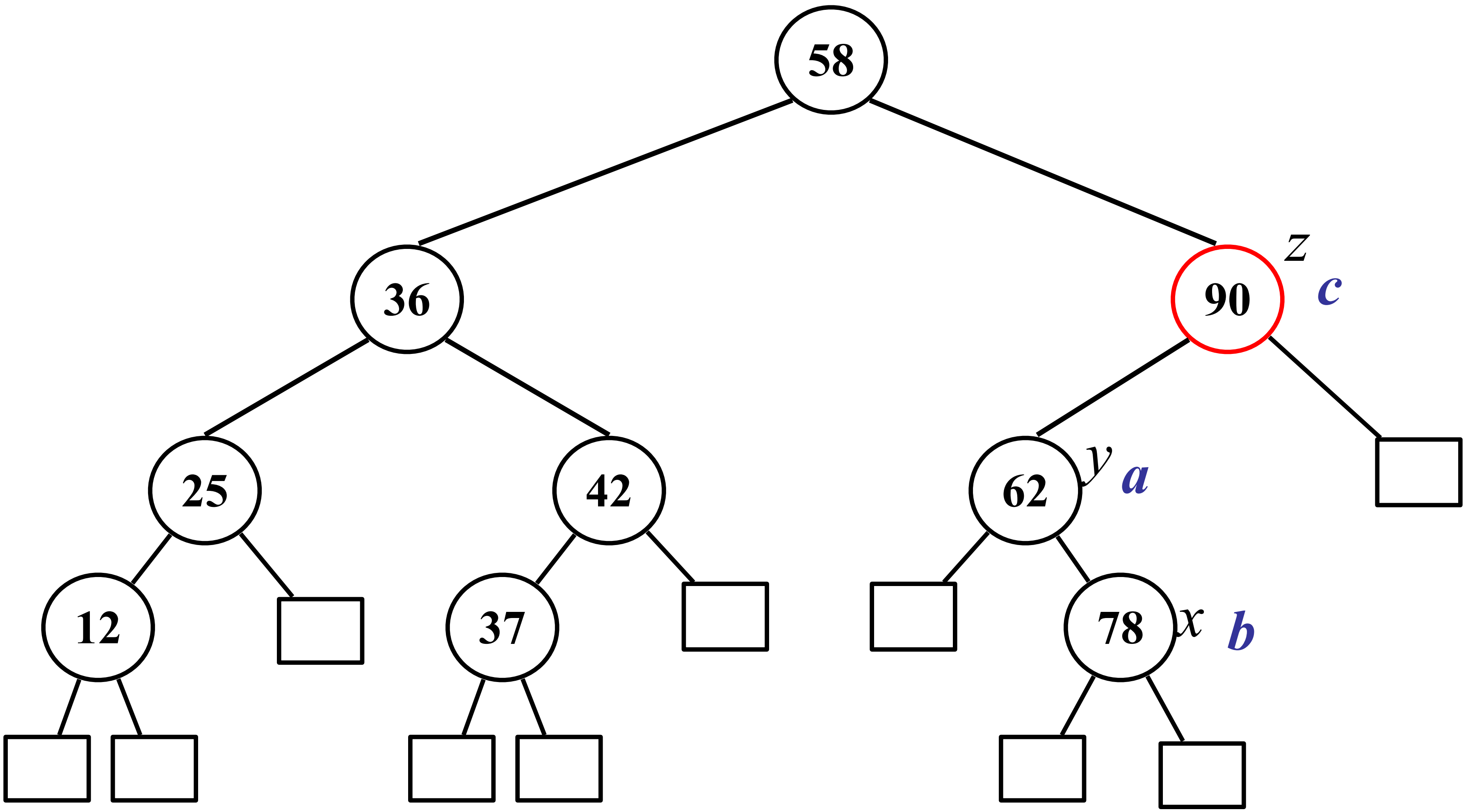
In this case x is the inserted node.
This is not always the case!



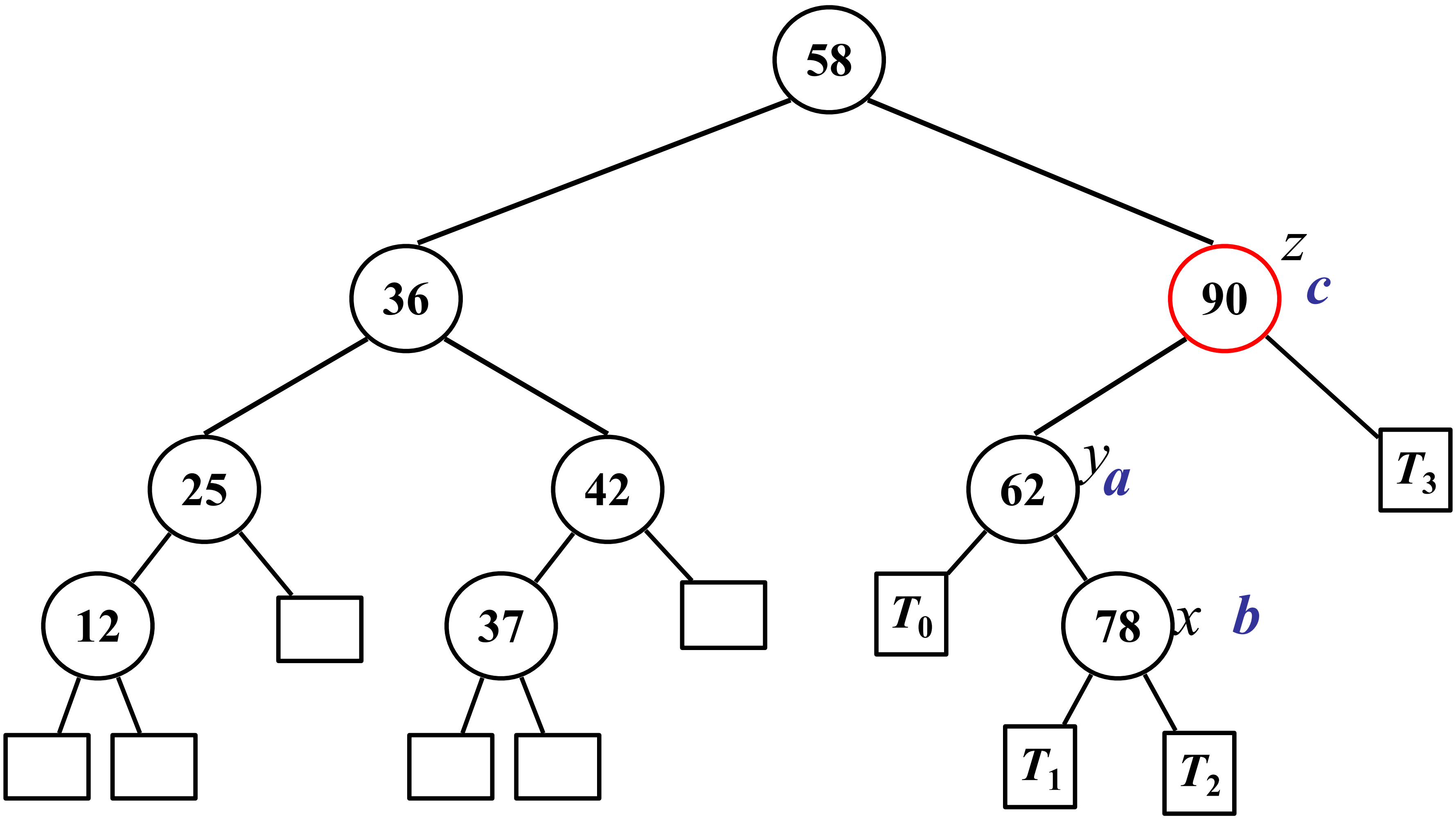
0. Let a, b, and c be a left to right in-order listing of x, y, and z



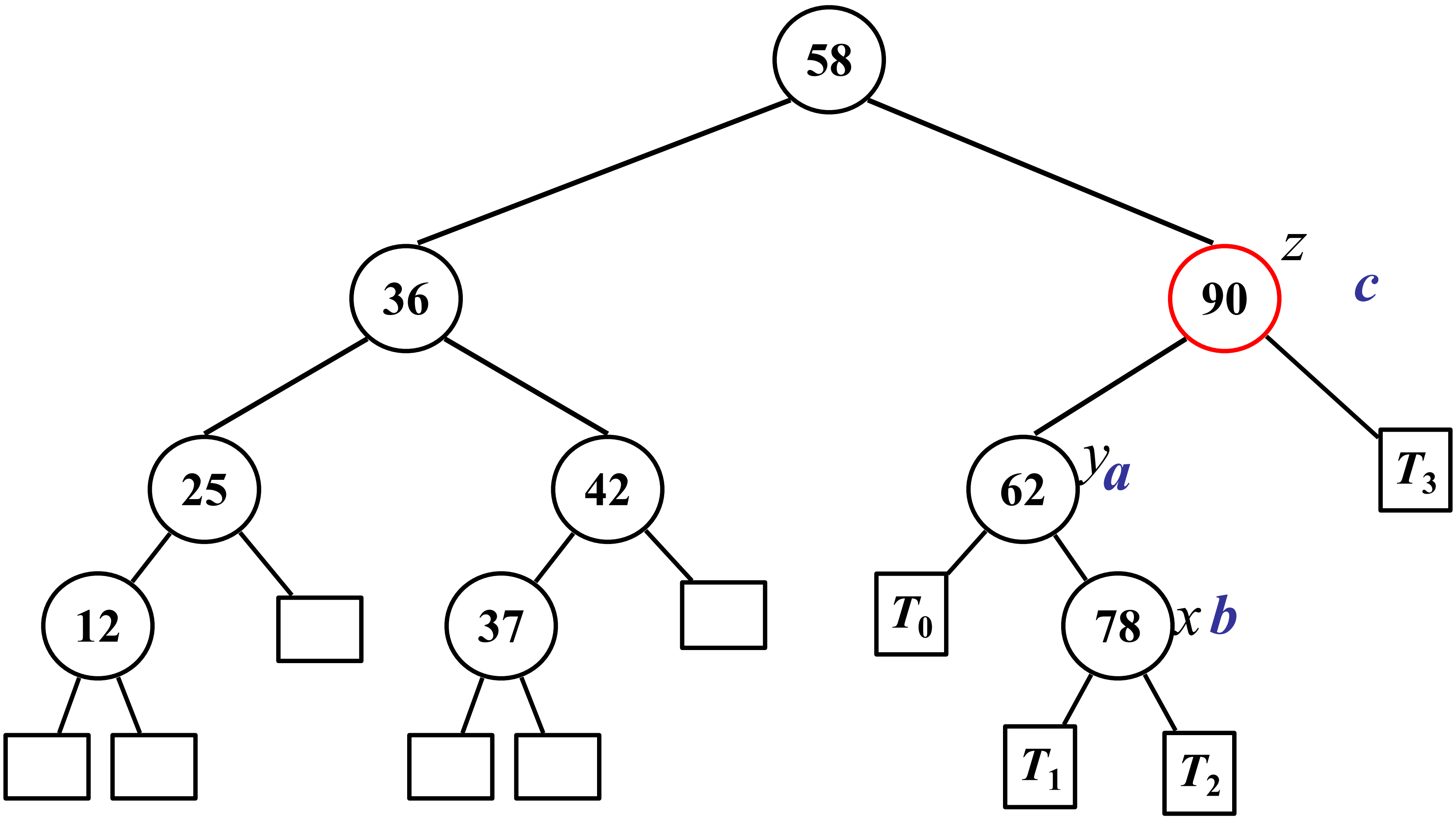
1. Let T_0, T_1, T_2, T_3 be subtrees rooted at x, y, and z



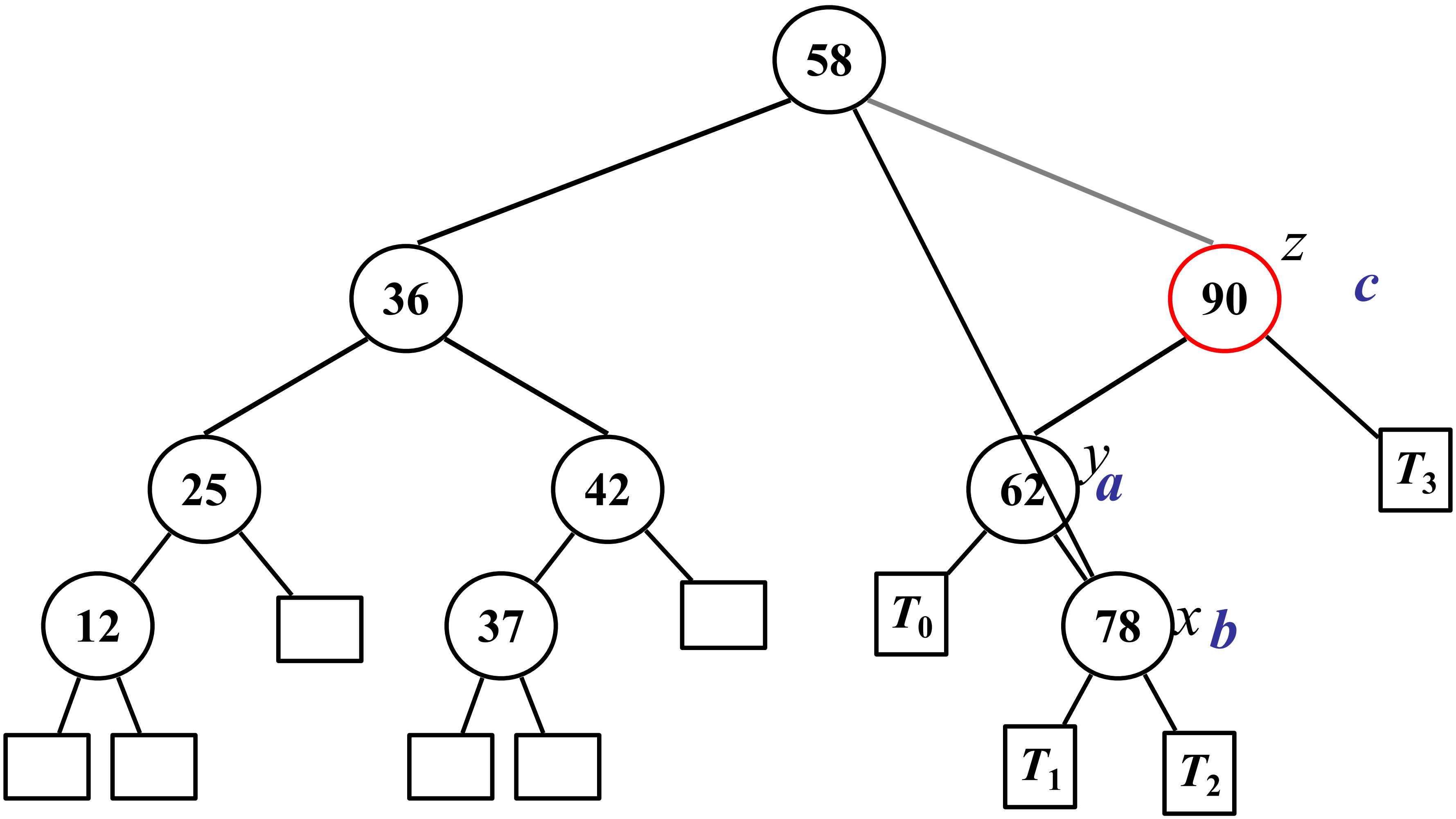
1. Let T_0, T_1, T_2, T_3 be subtrees rooted at x, y, and z



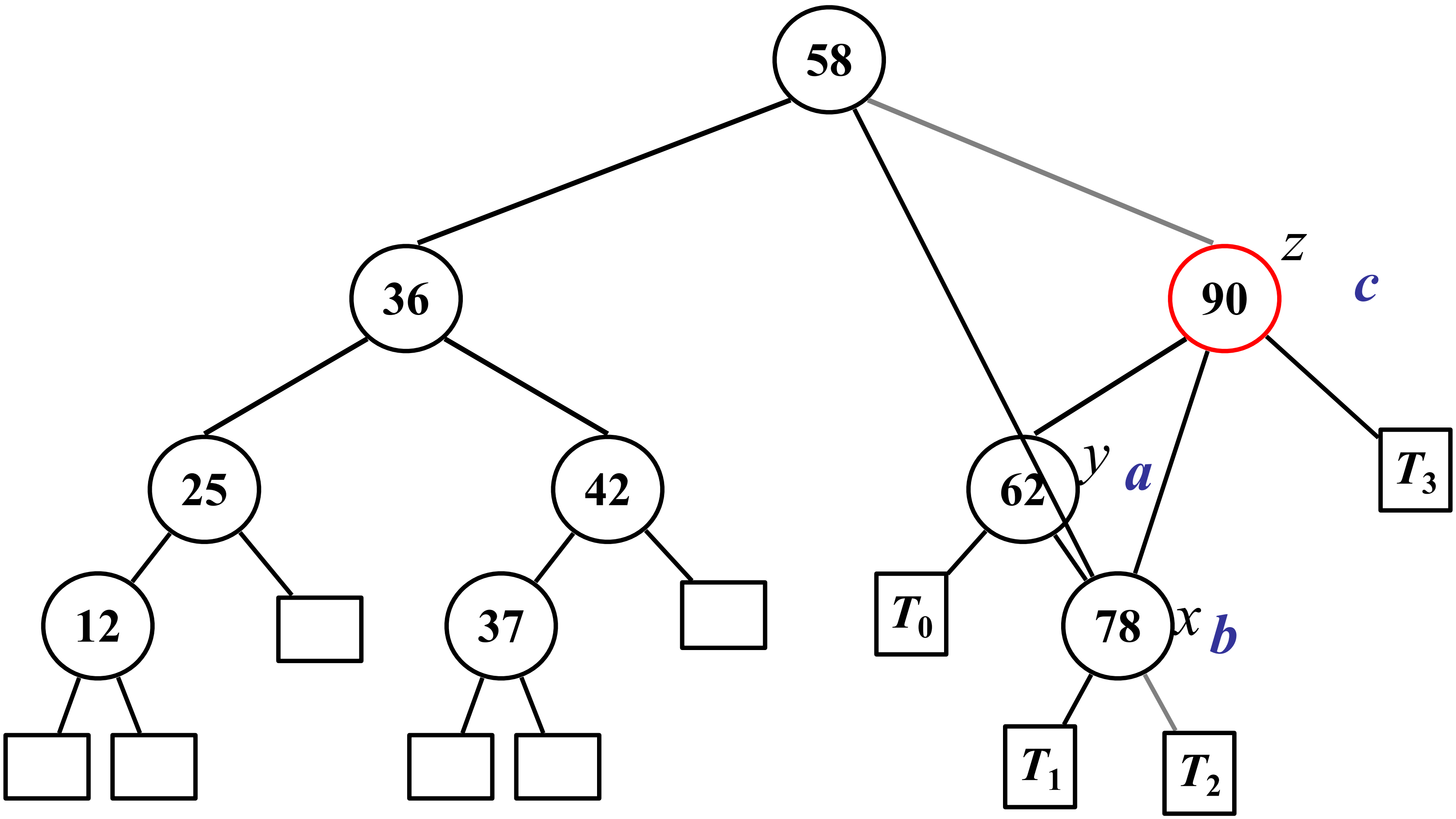
2. Restructure such that *b* becomes the new root



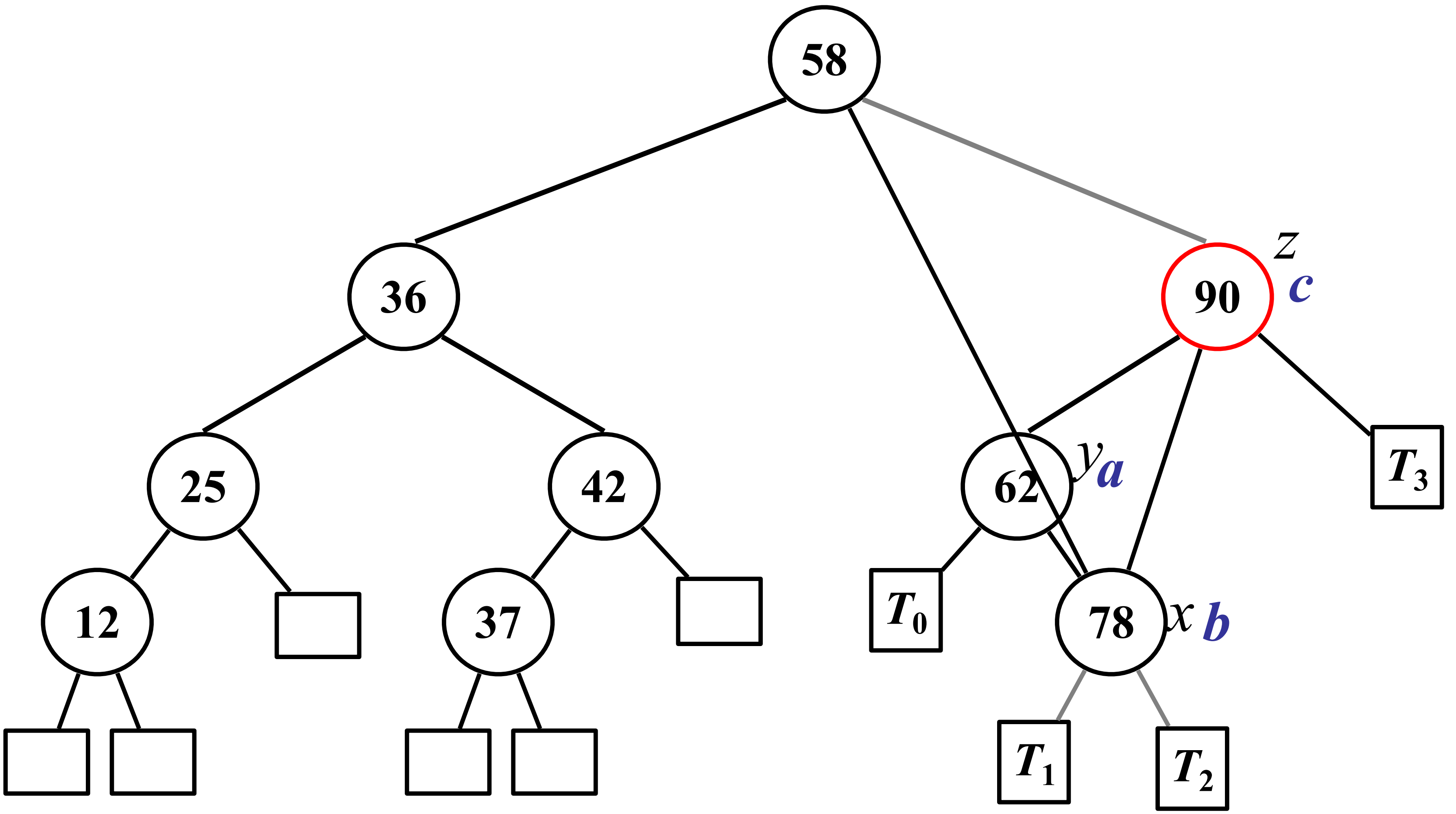
2. Restructure such that *b* becomes the new root



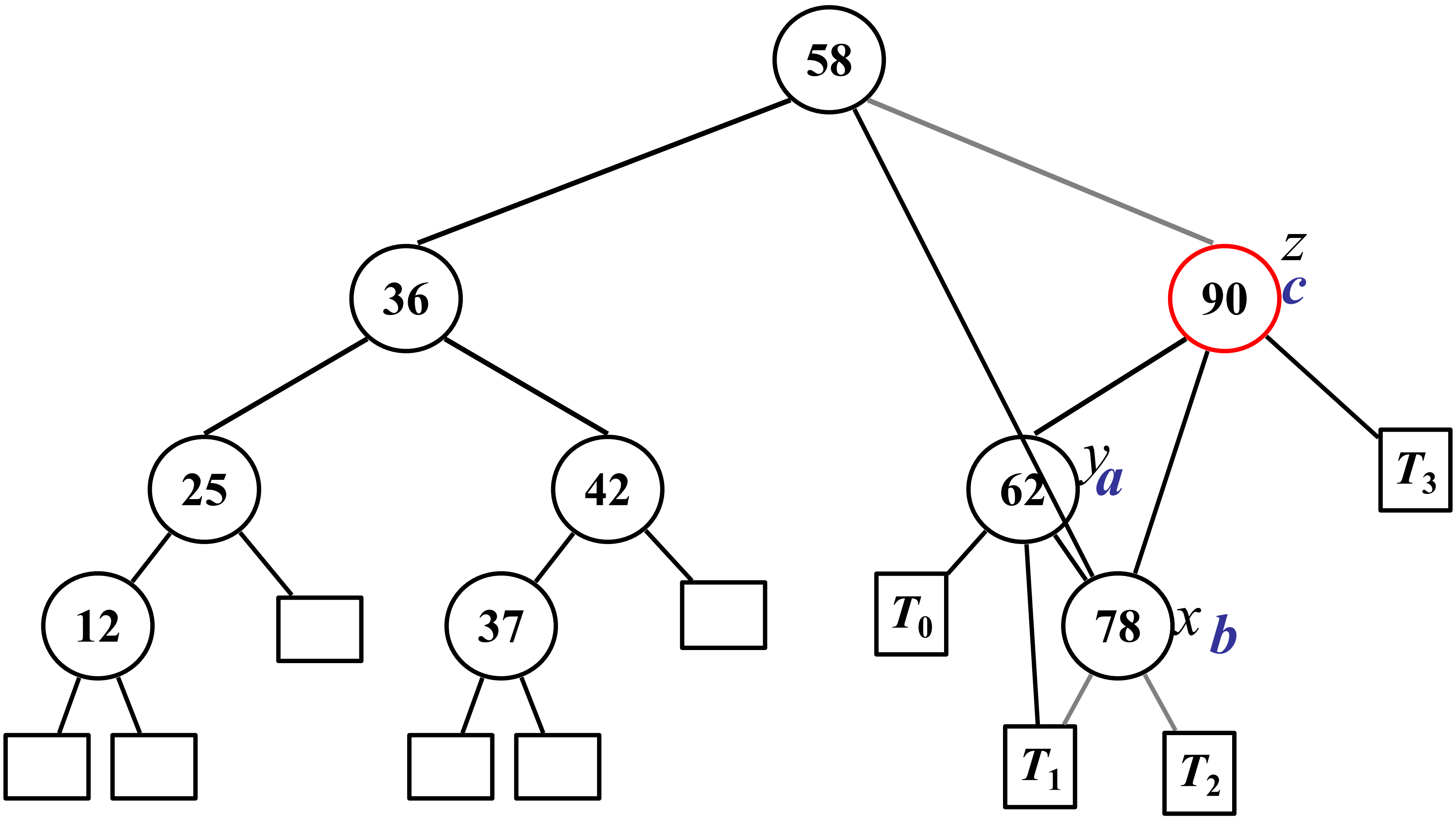
2. Restructure such that *b* becomes the new root



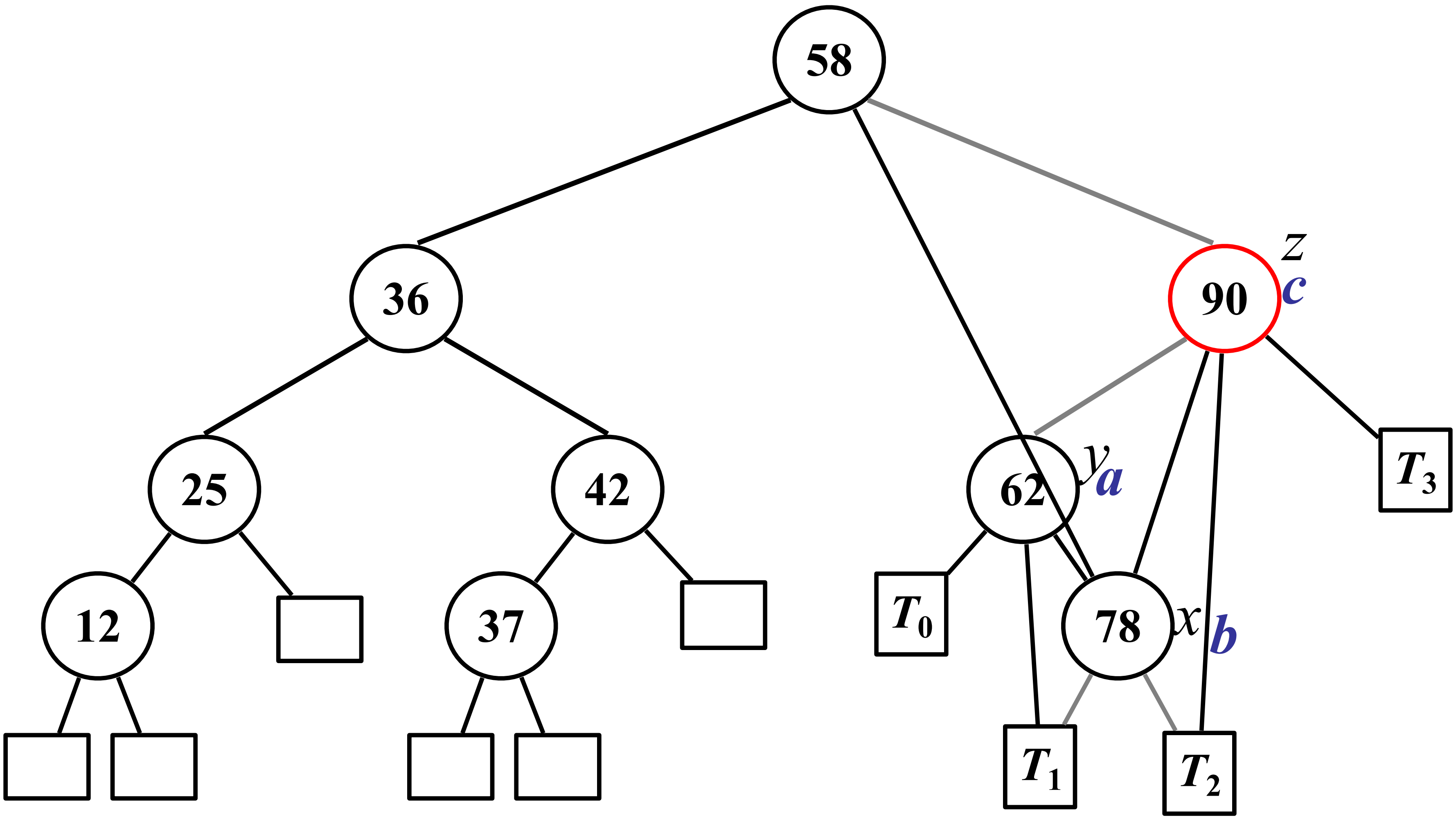
2. Restructure such that *b* becomes the new root



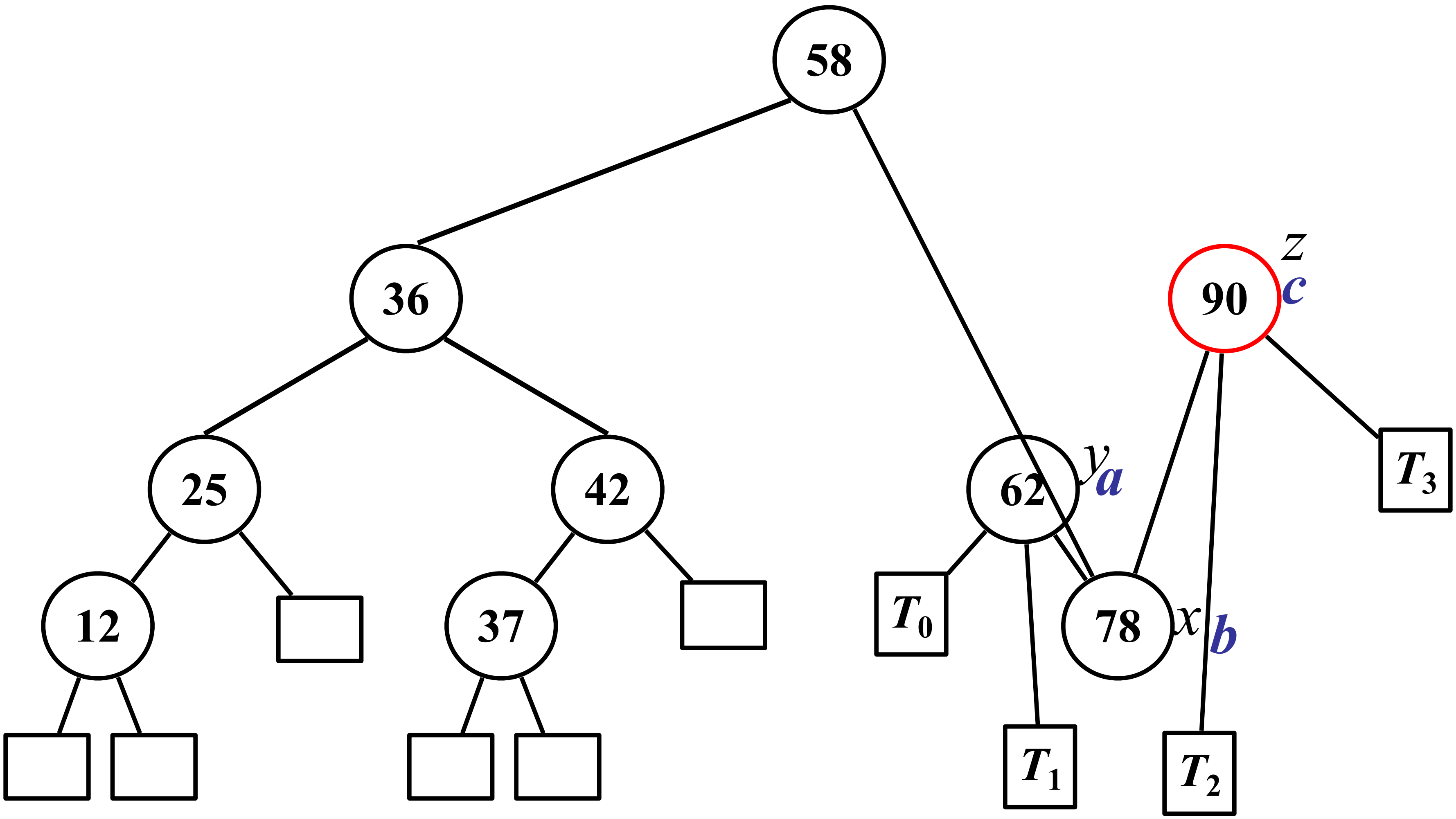
2. Restructure such that *b* becomes the new root



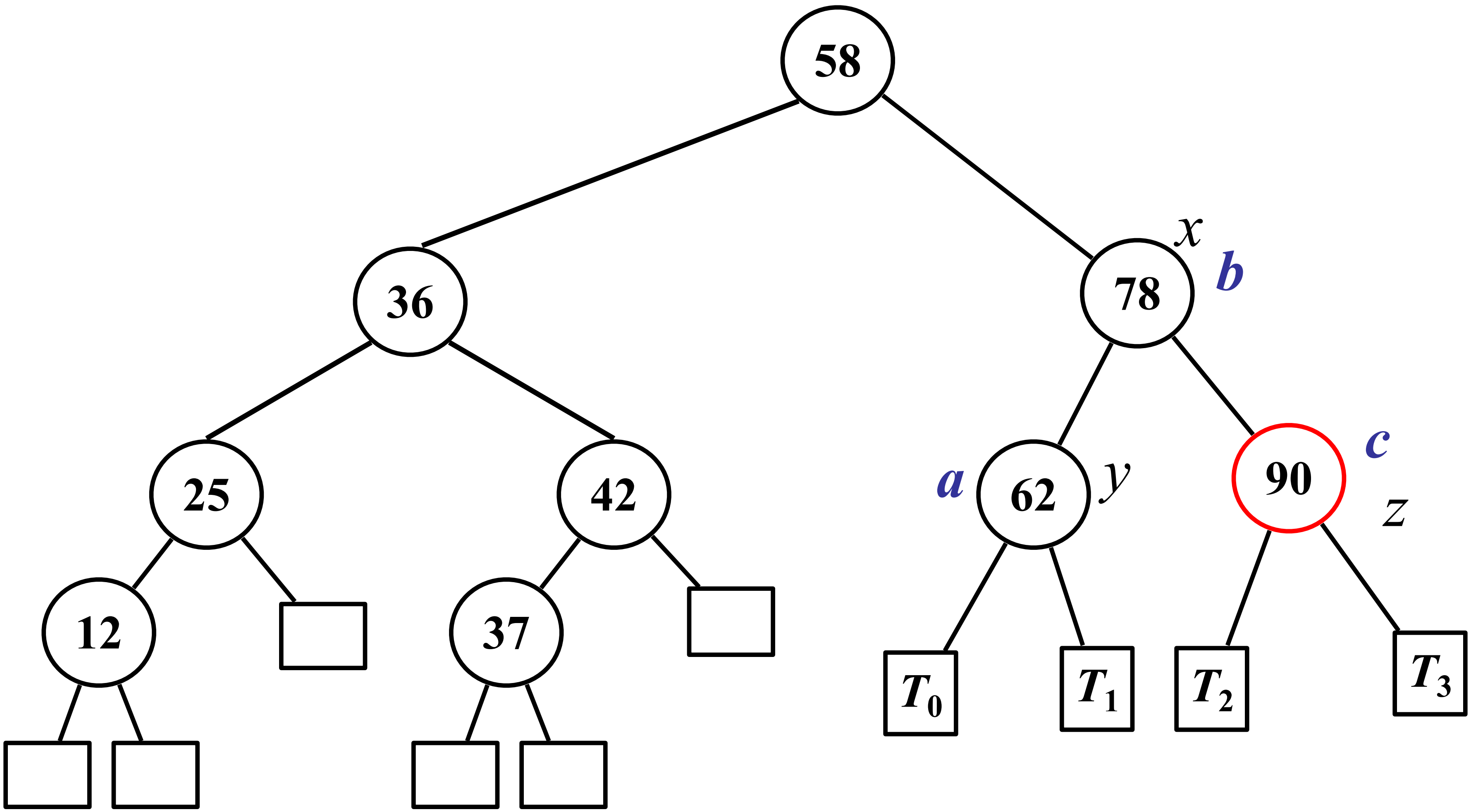
2. Restructure such that *b* becomes the new root



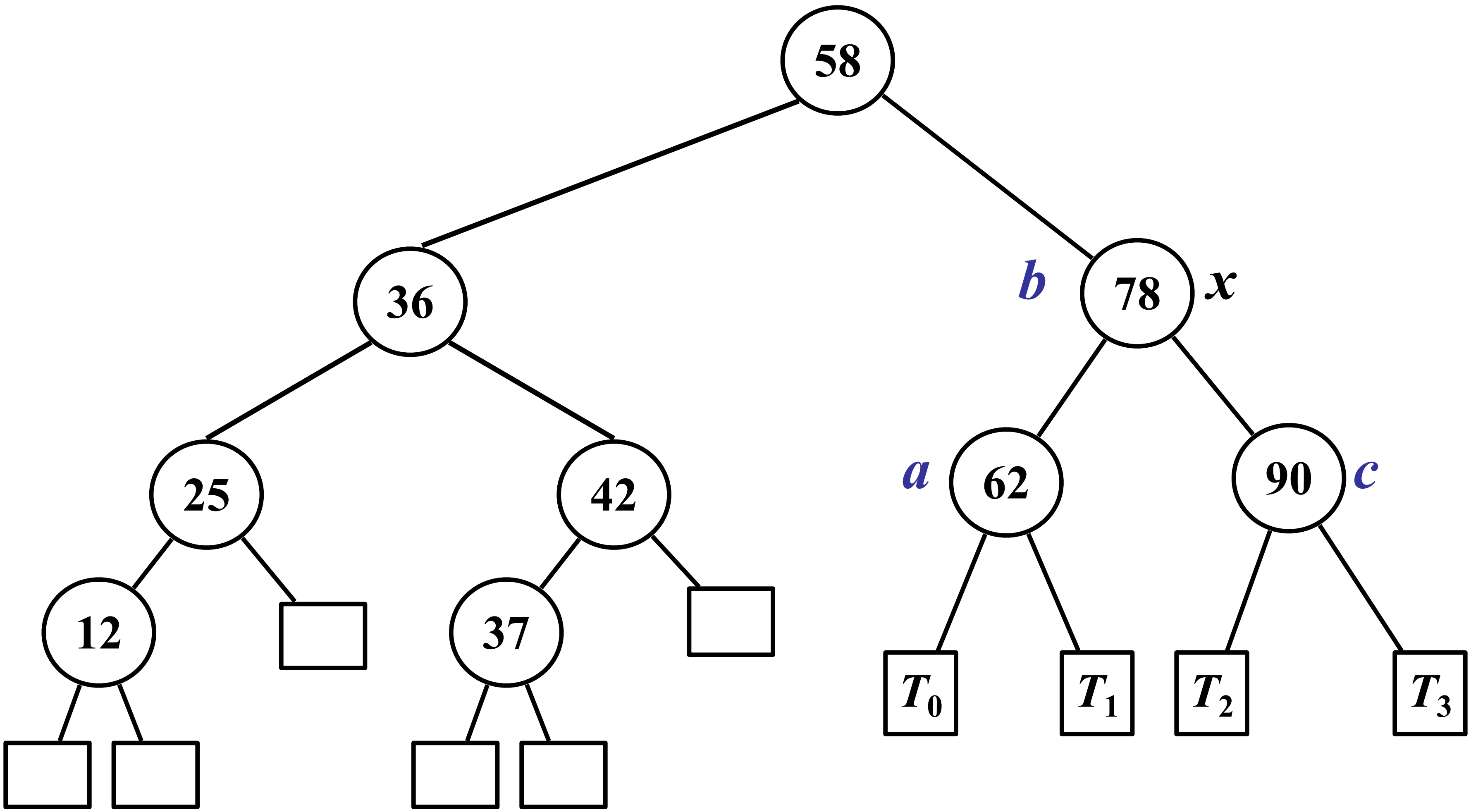
2. Restructure such that *b* becomes the new root



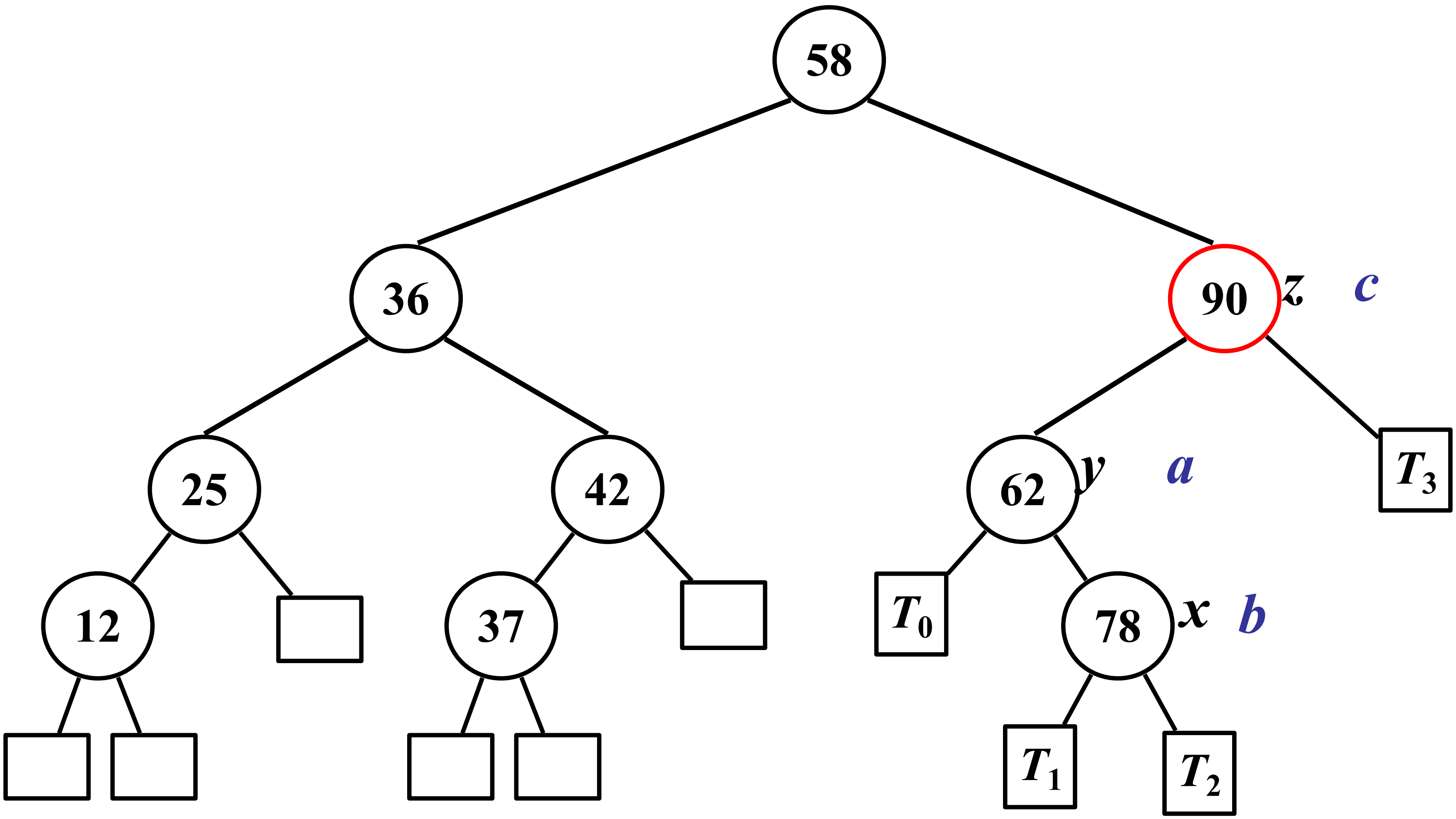
2. Restructure such that *b* becomes the new root



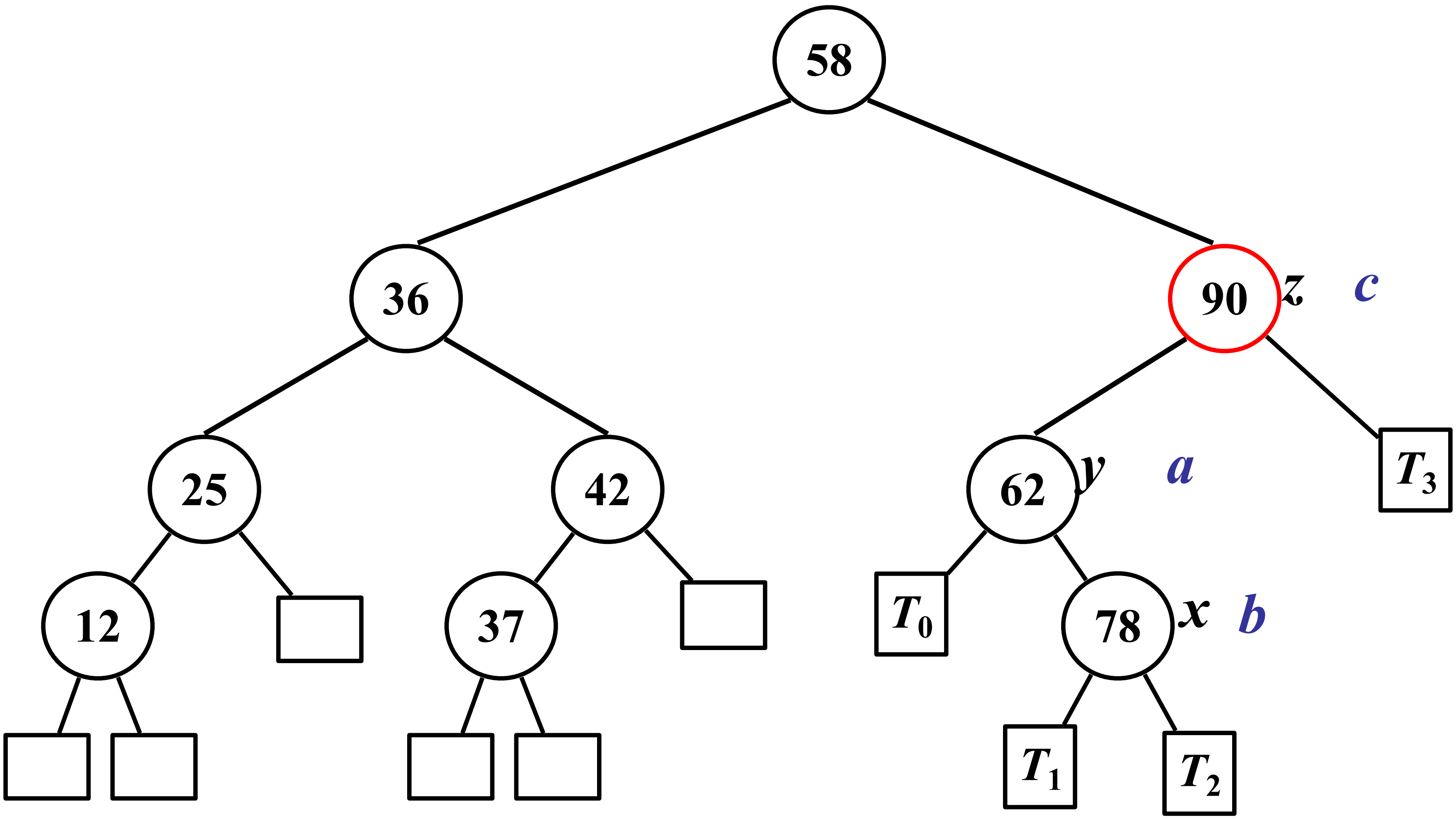
2. Restructure such that *b* becomes the new root



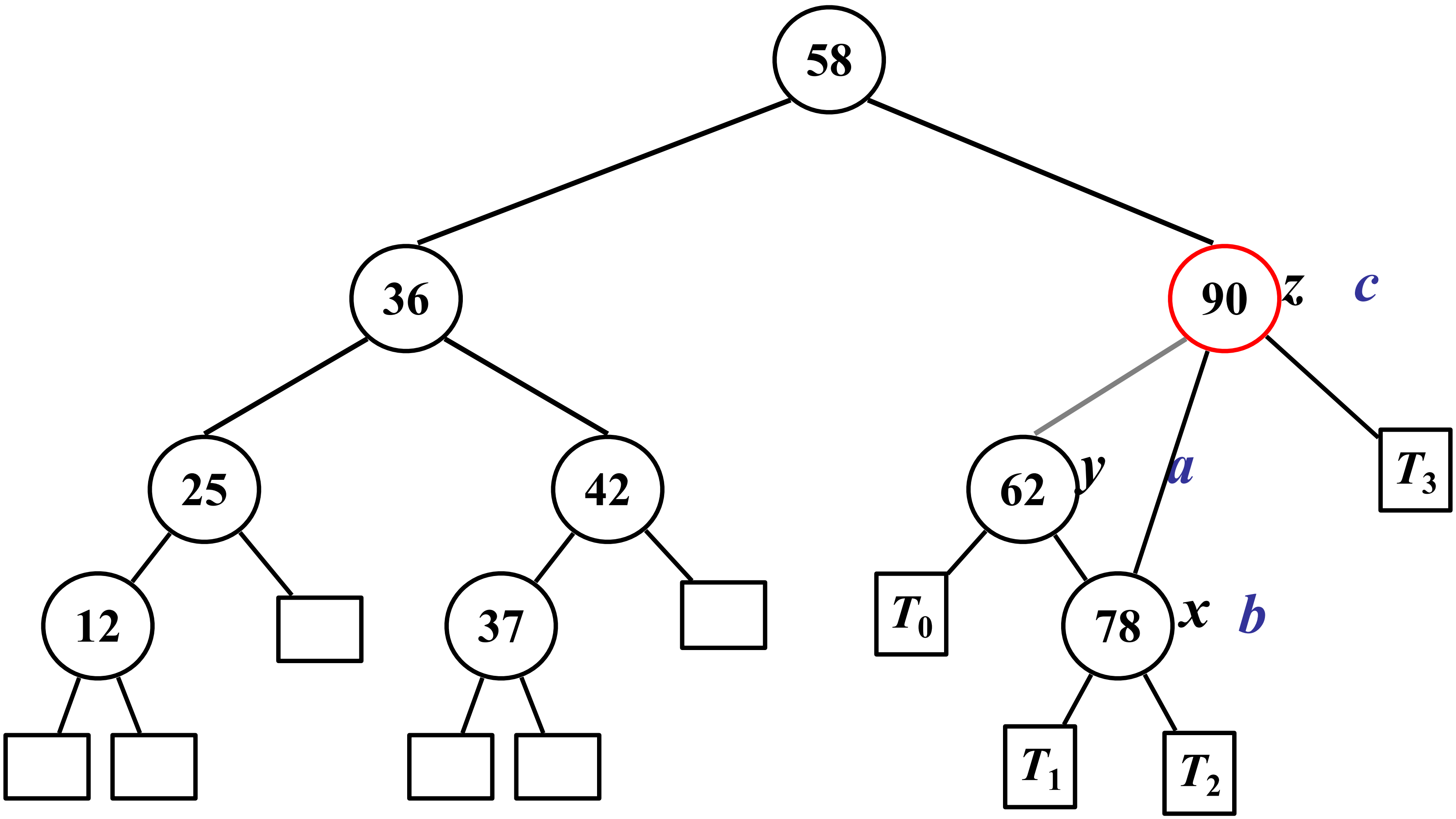
Another view: *Rotate until balanced*.



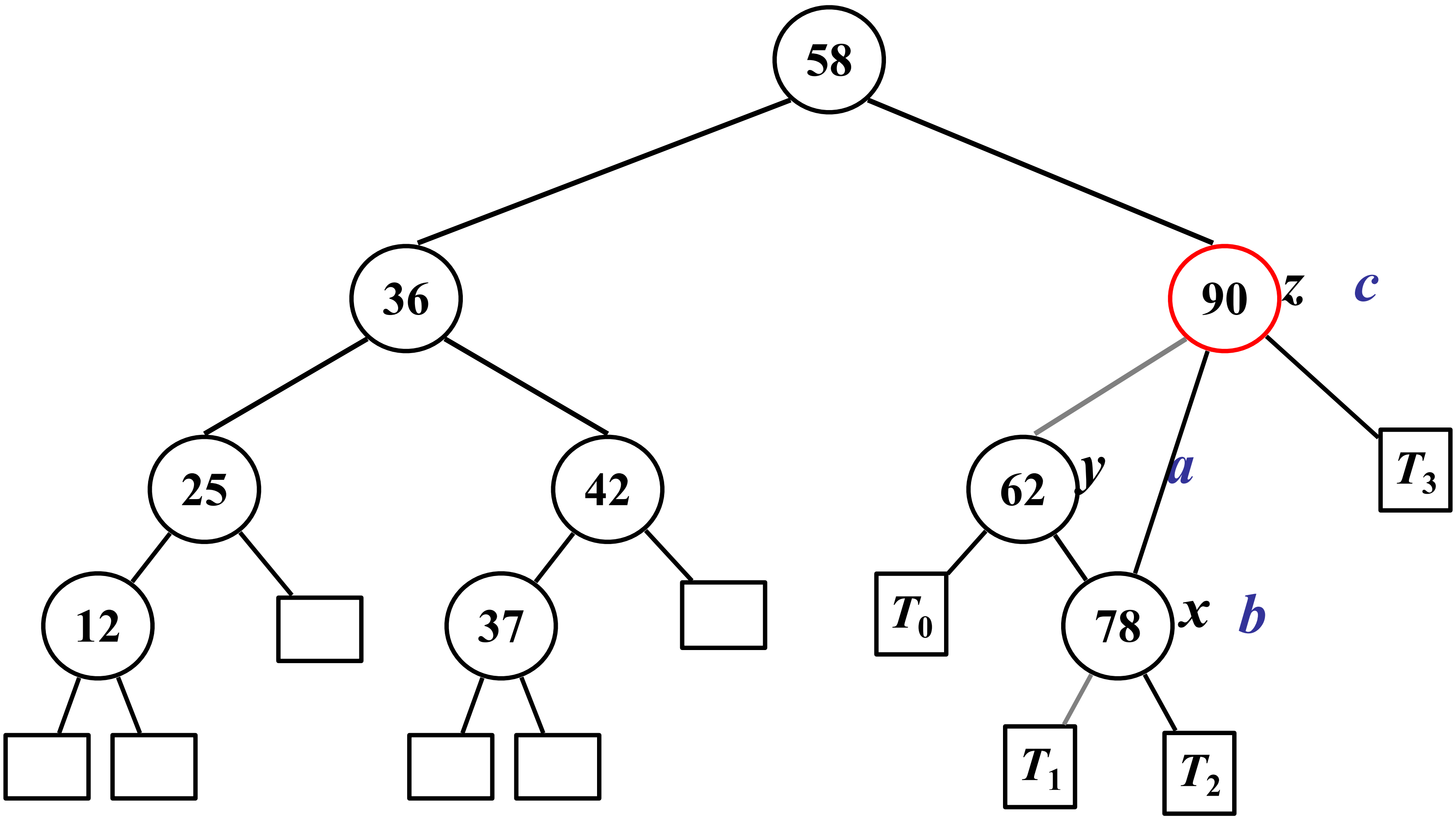
Rotation 1: Rotate such that *b* moves up by a level



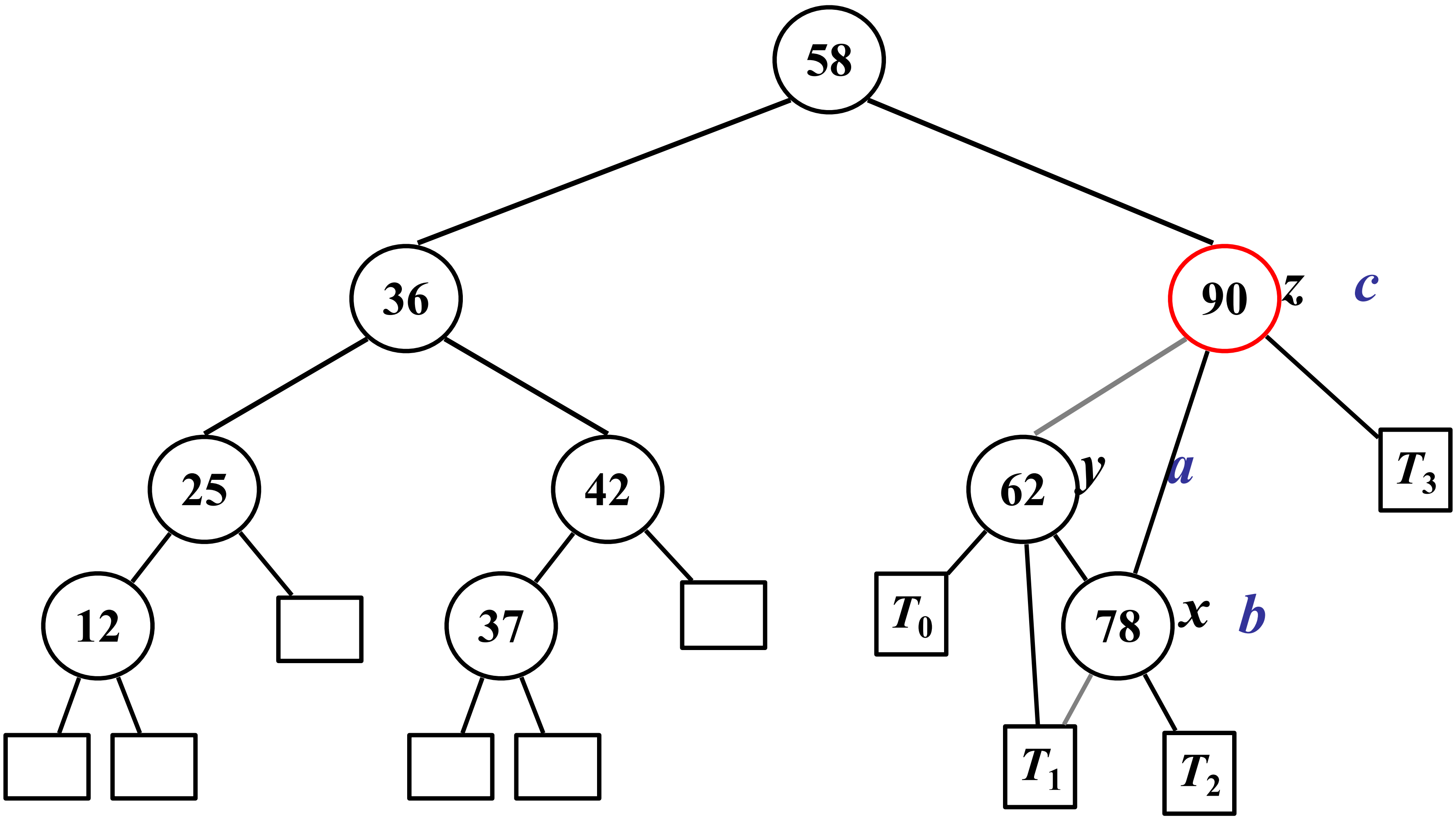
Rotation 1: Rotate such that *b* moves up by a level



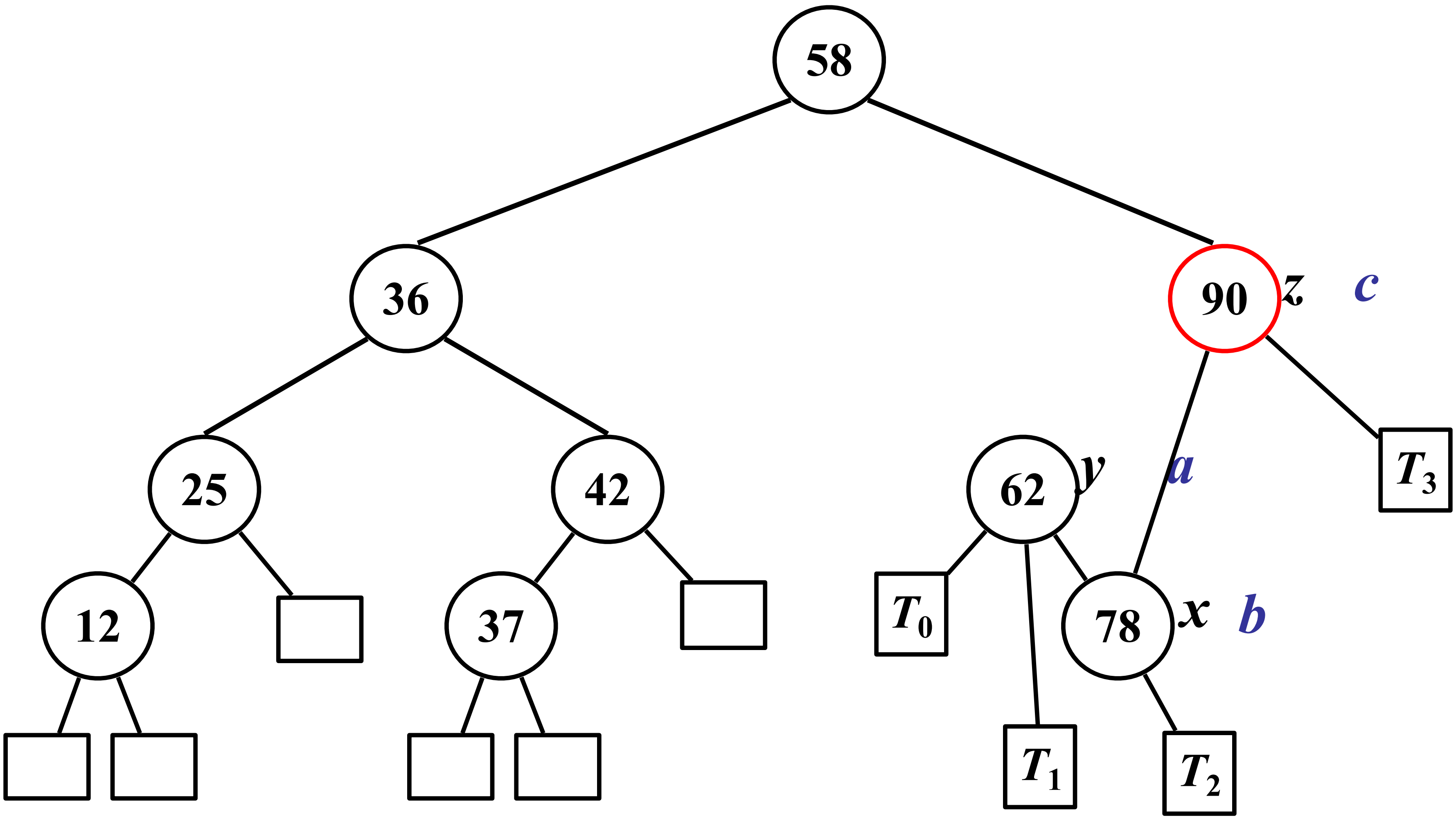
Rotation 1: Rotate such that *b* moves up by a level



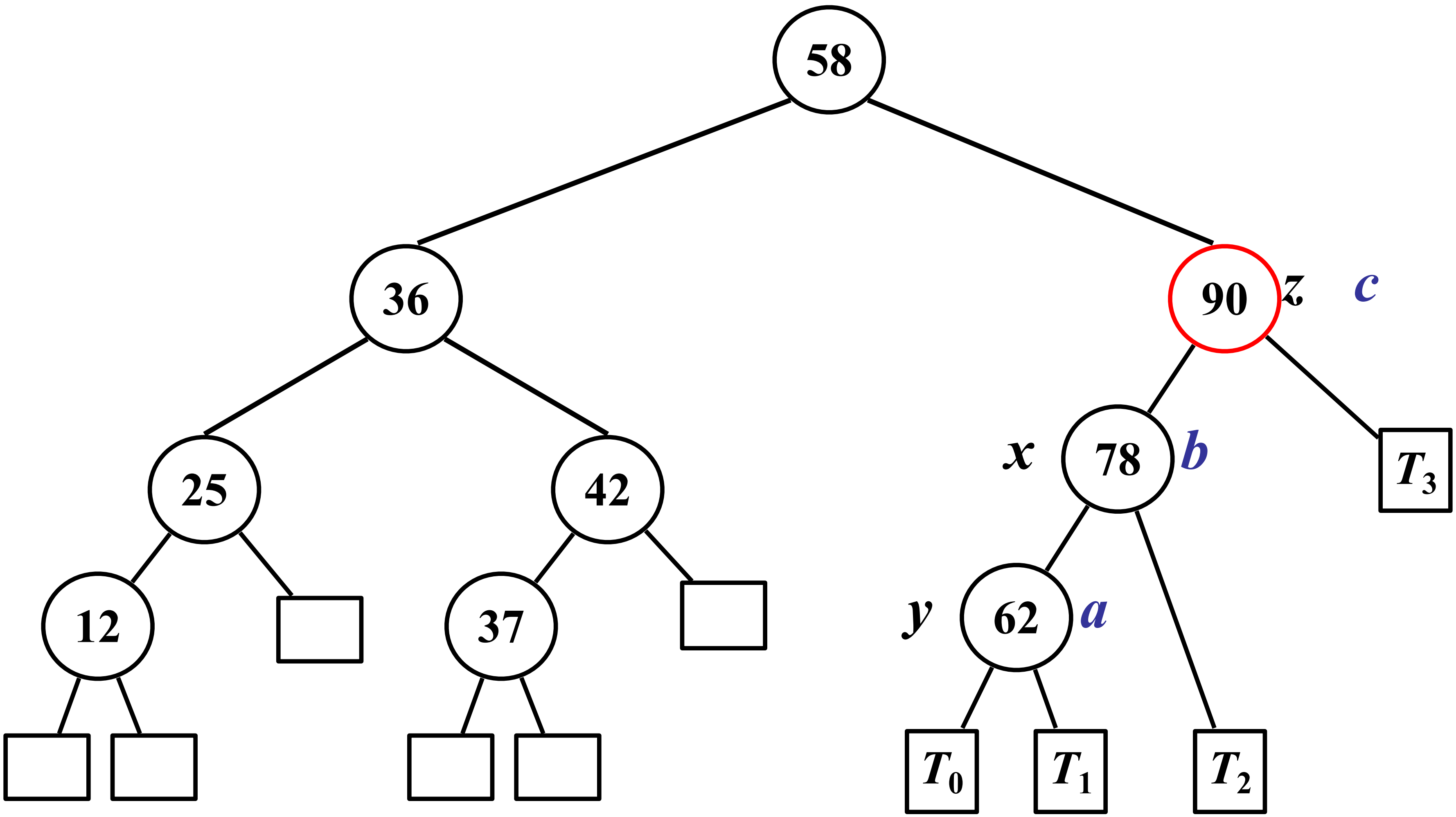
Rotation 1: Rotate such that *b* moves up by a level



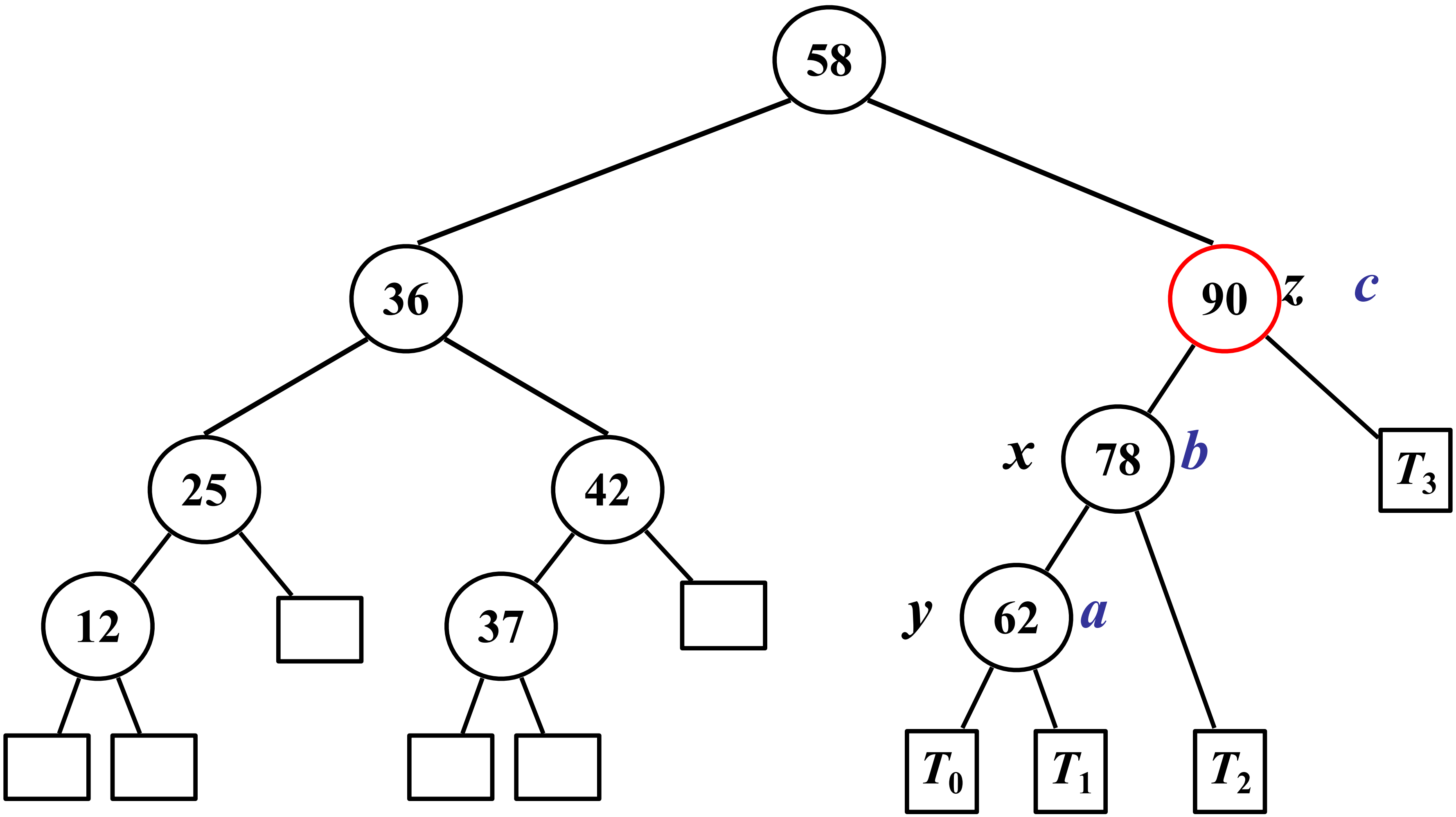
Rotation 1: Rotate such that *b* moves up by a level



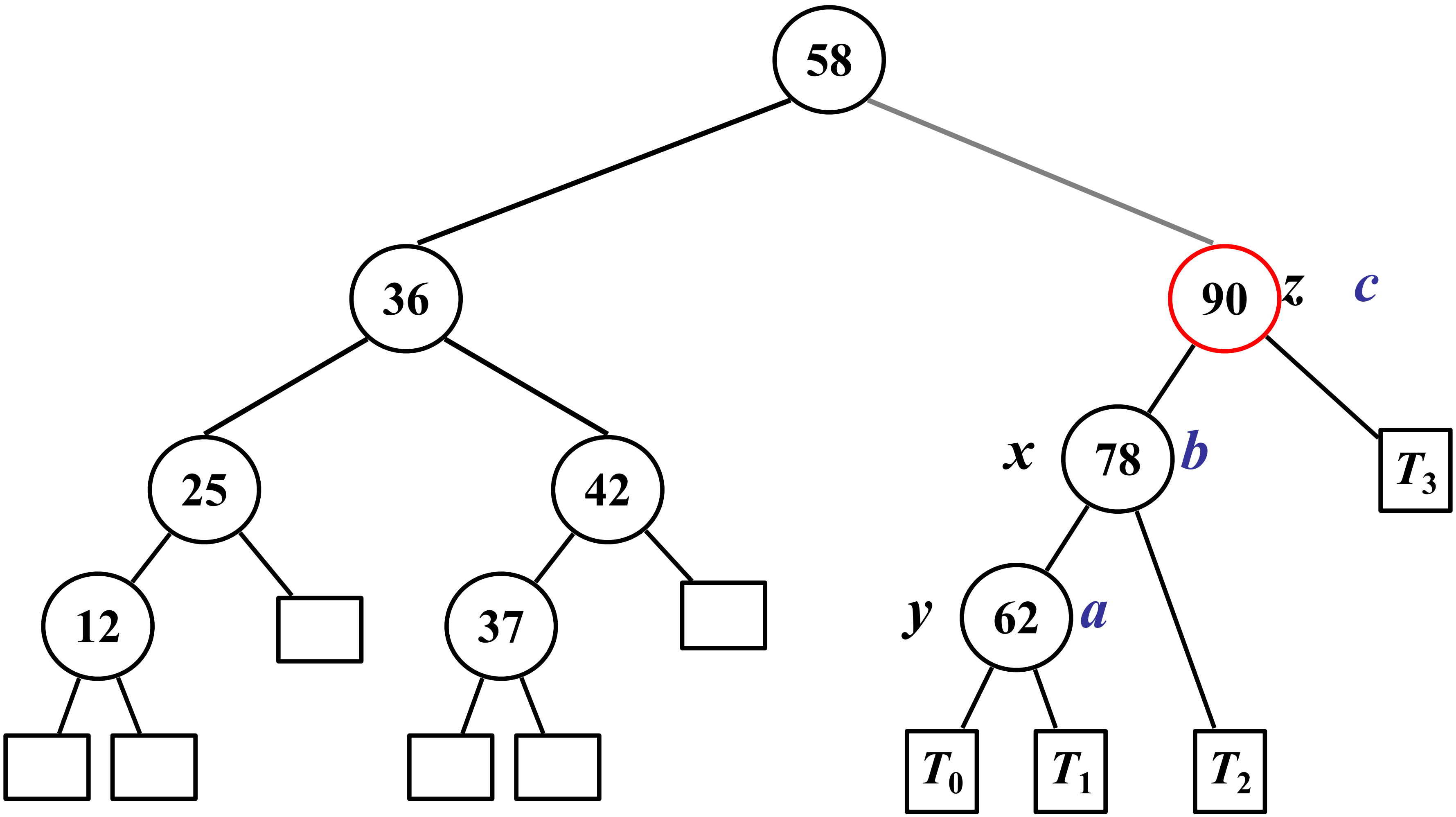
Rotation 1: Rotate such that *b* moves up by a level



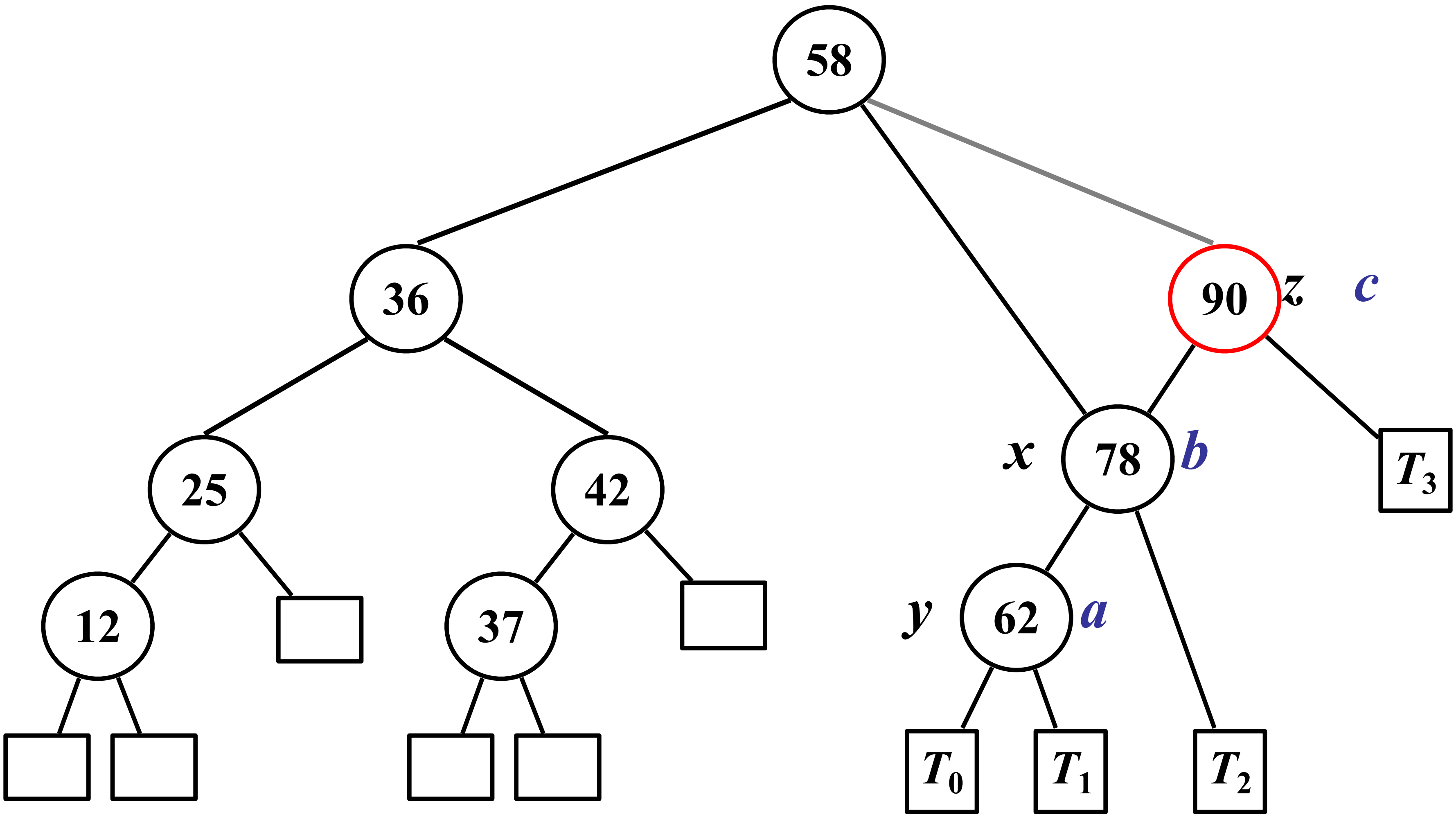
Rotation 2: Rotate such that *b* moves up by another level



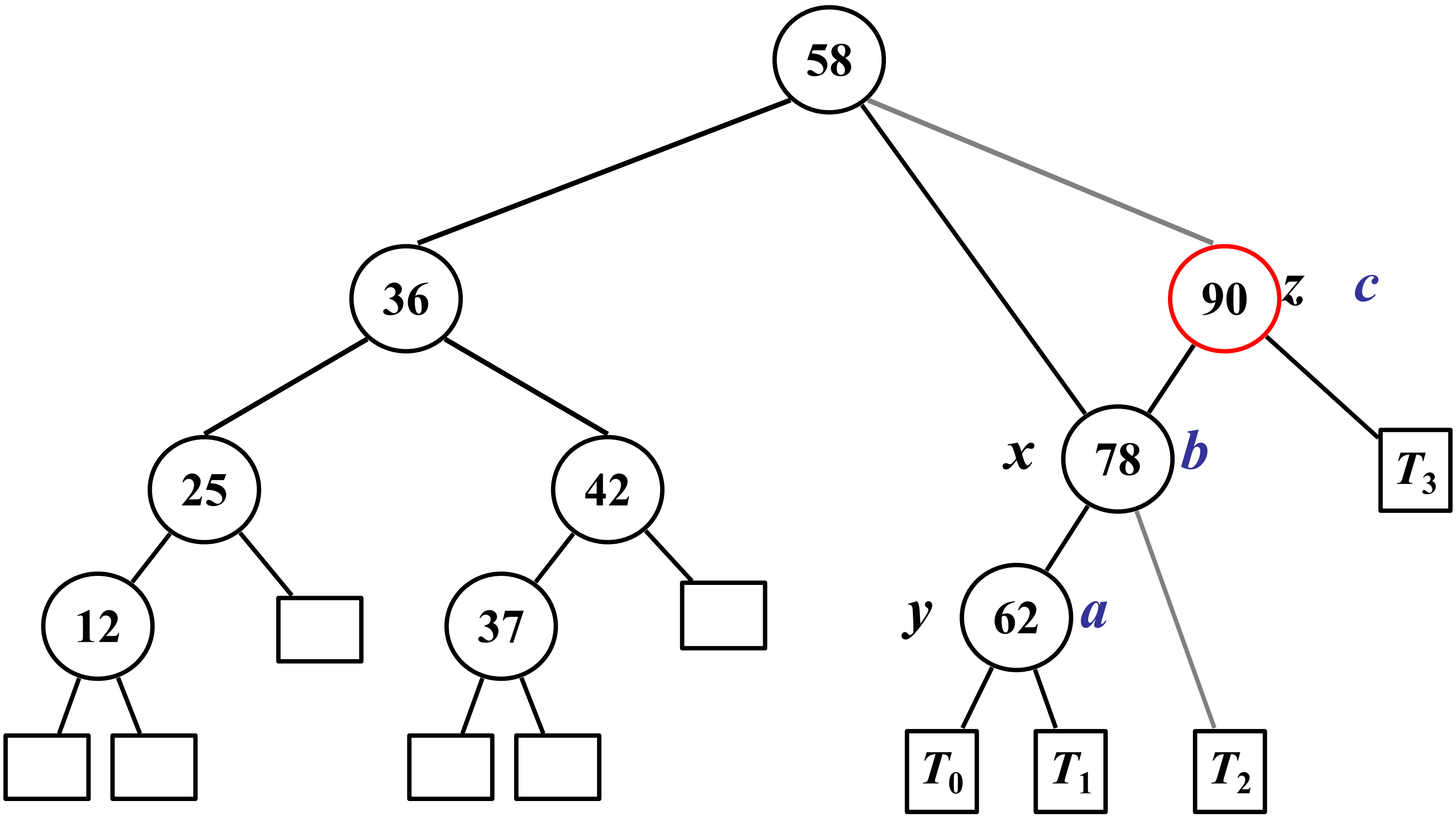
Rotation 2: Rotate such that *b* moves up by another level



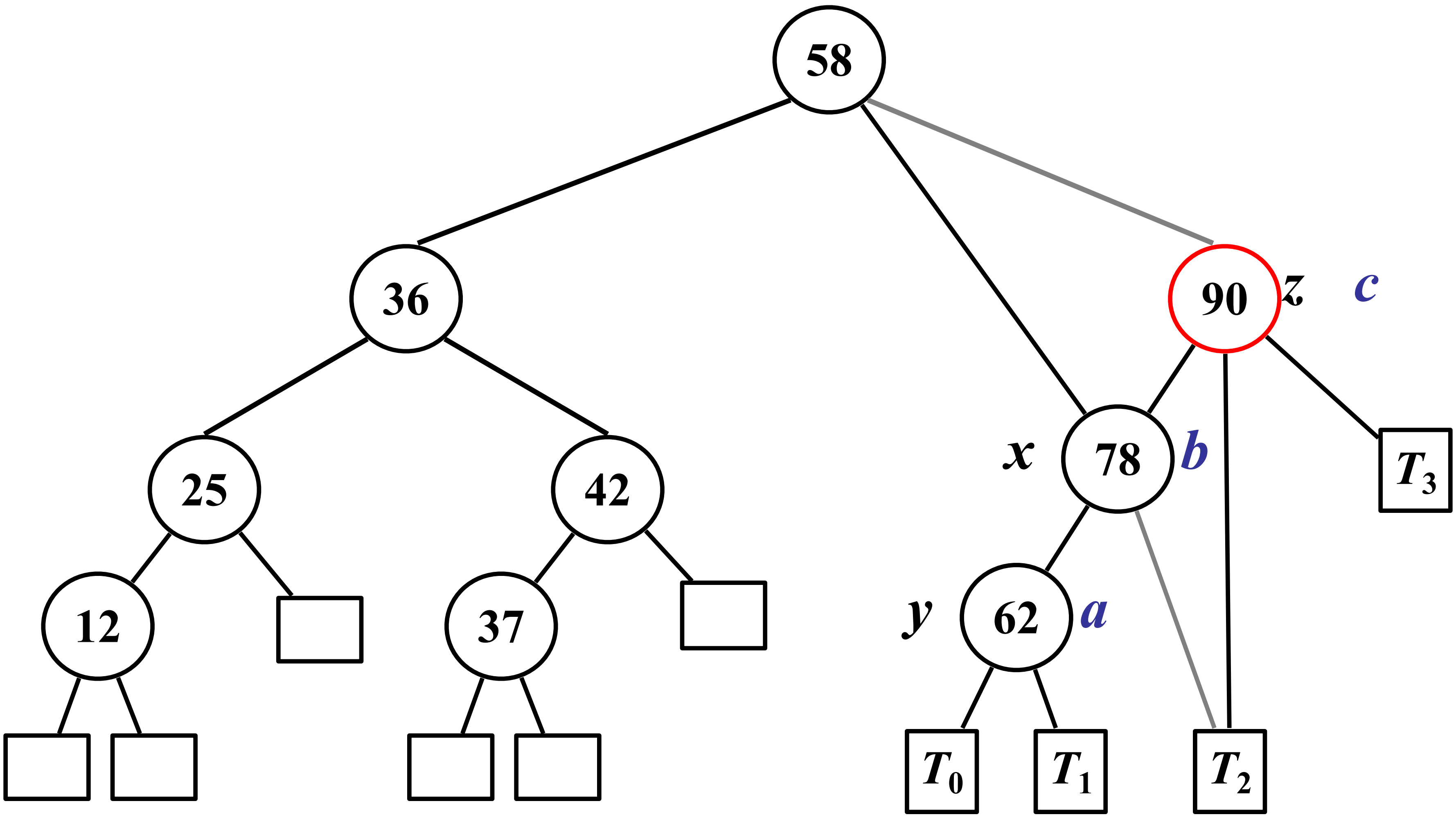
Rotation 2: Rotate such that *b* moves up by another level



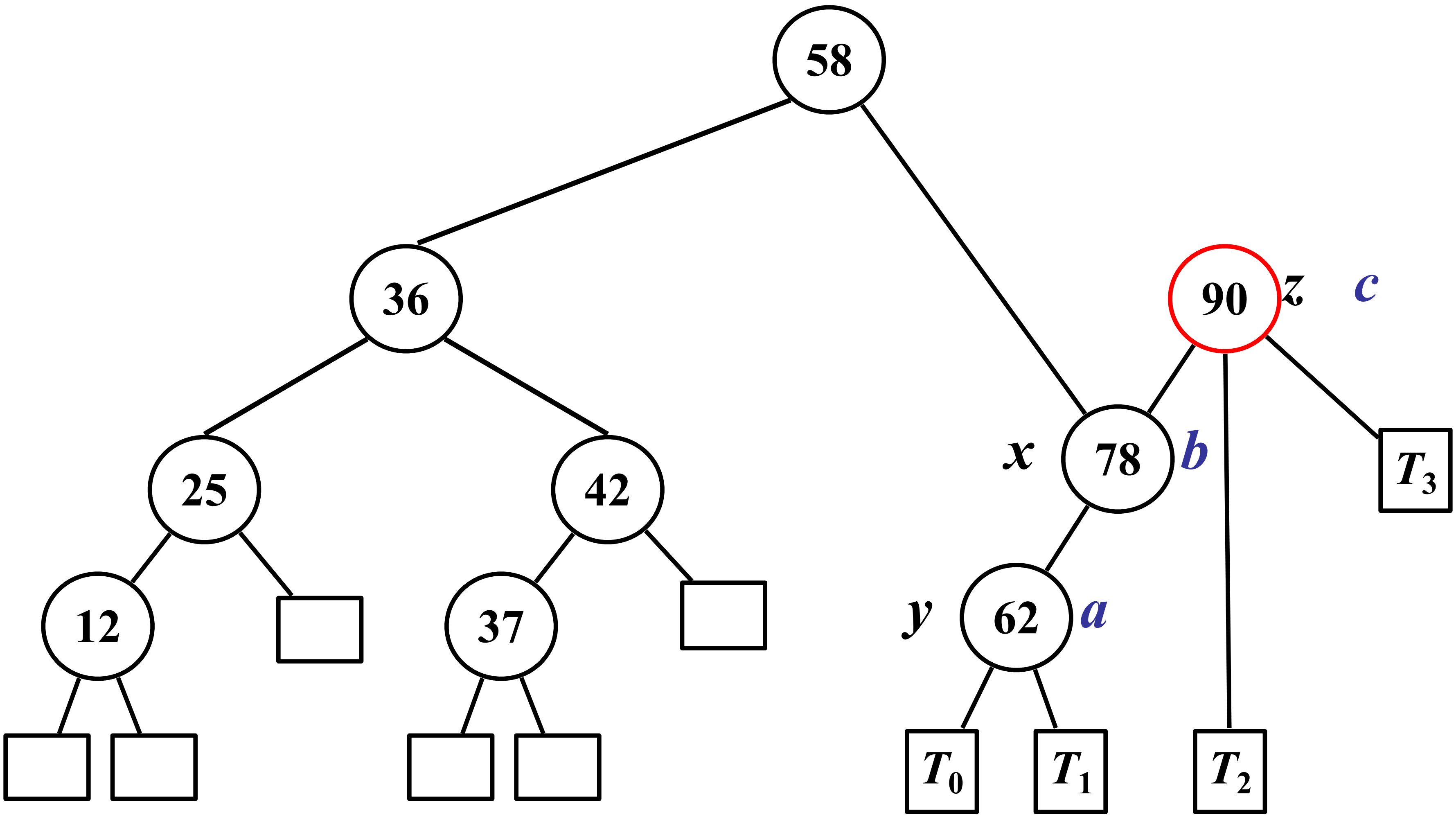
Rotation 2: Rotate such that *b* moves up by another level



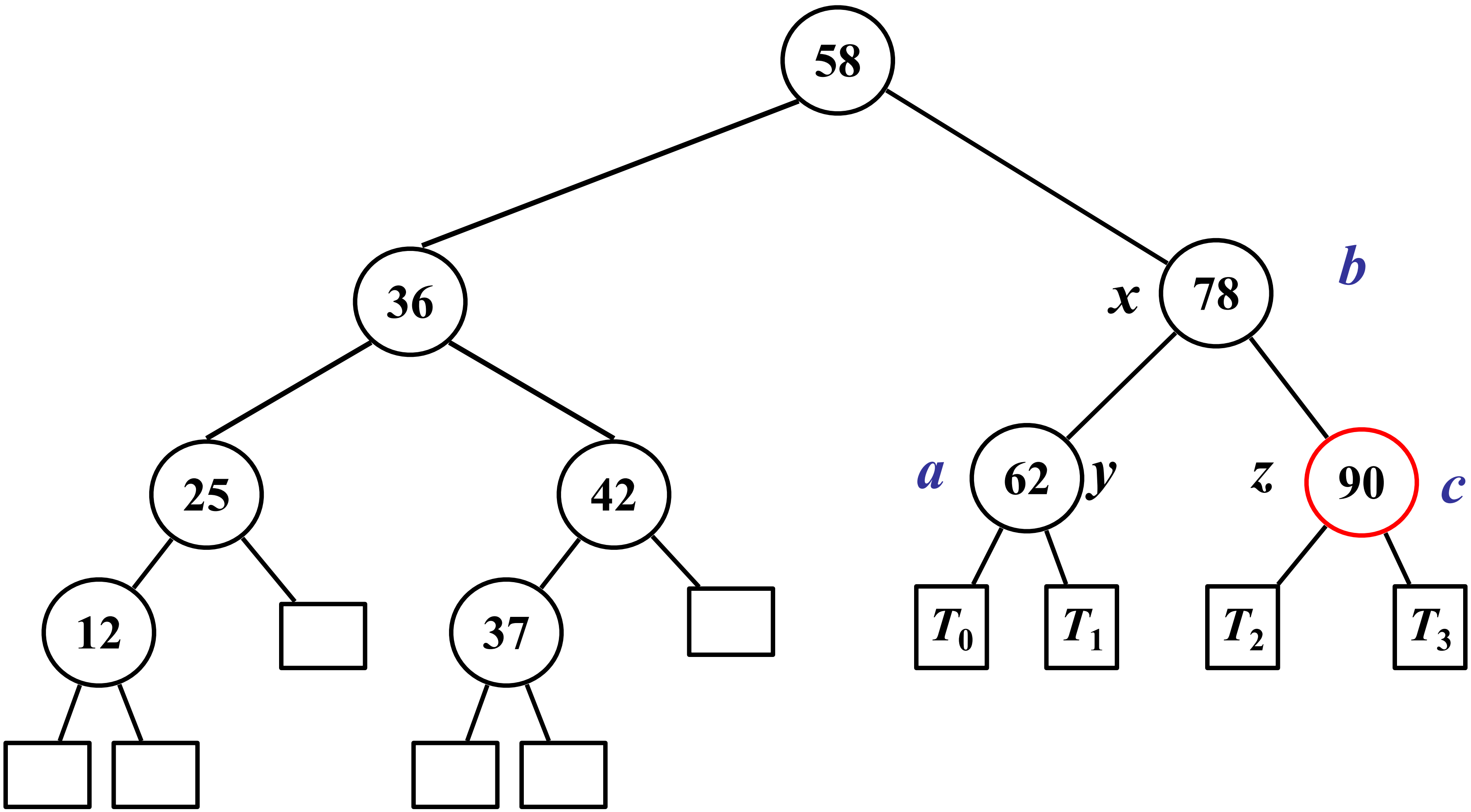
Rotation 2: Rotate such that *b* moves up by another level



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