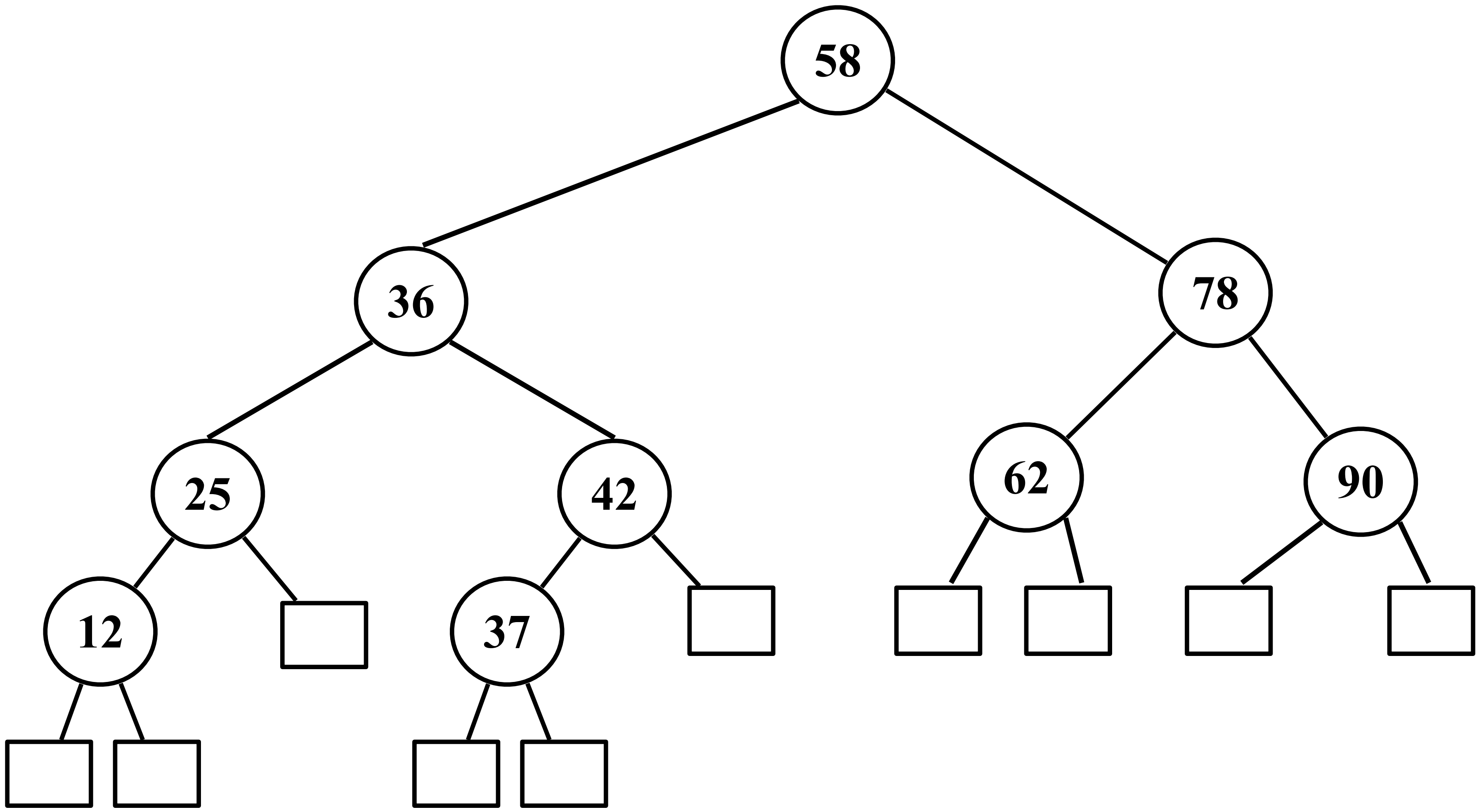
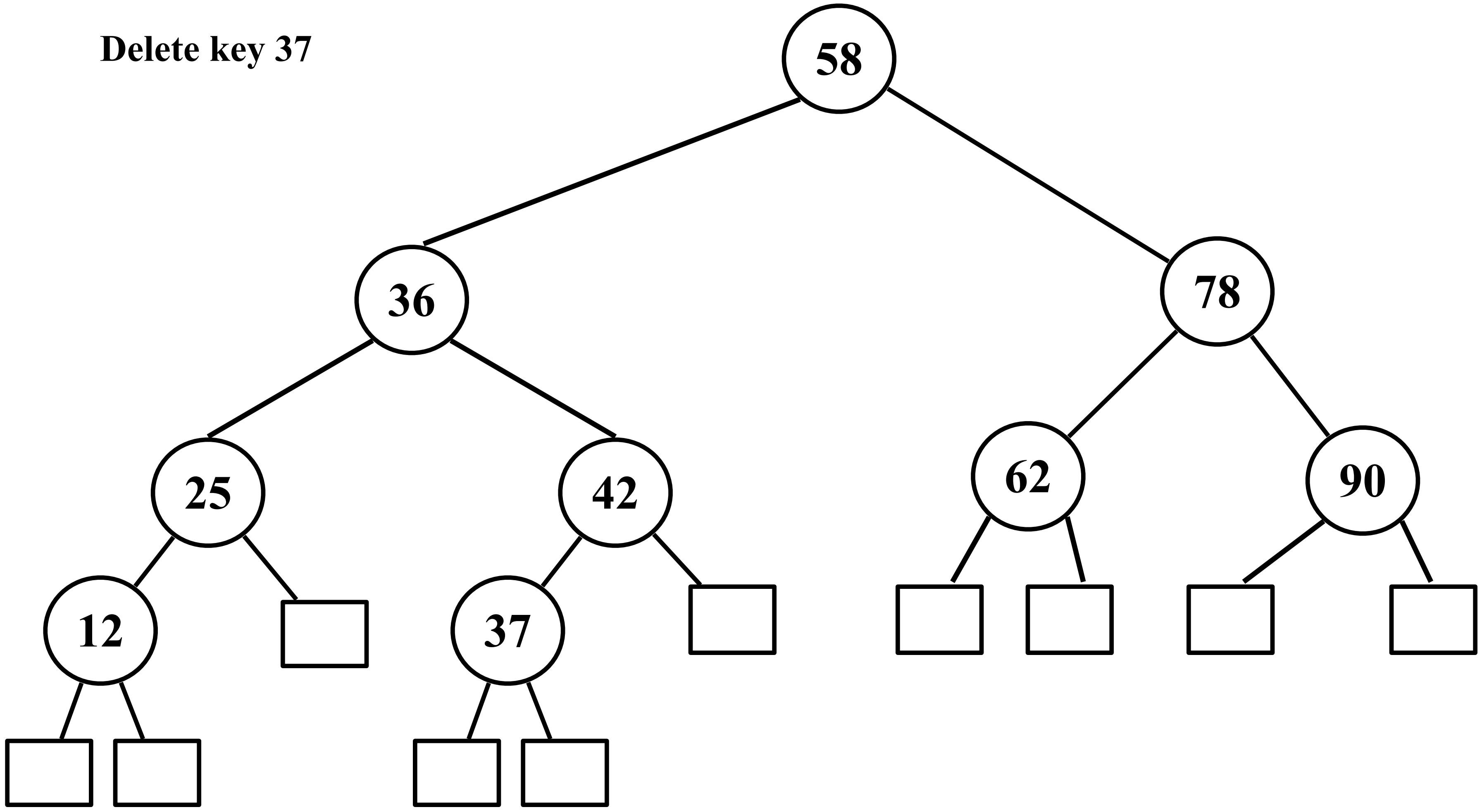
Deletion

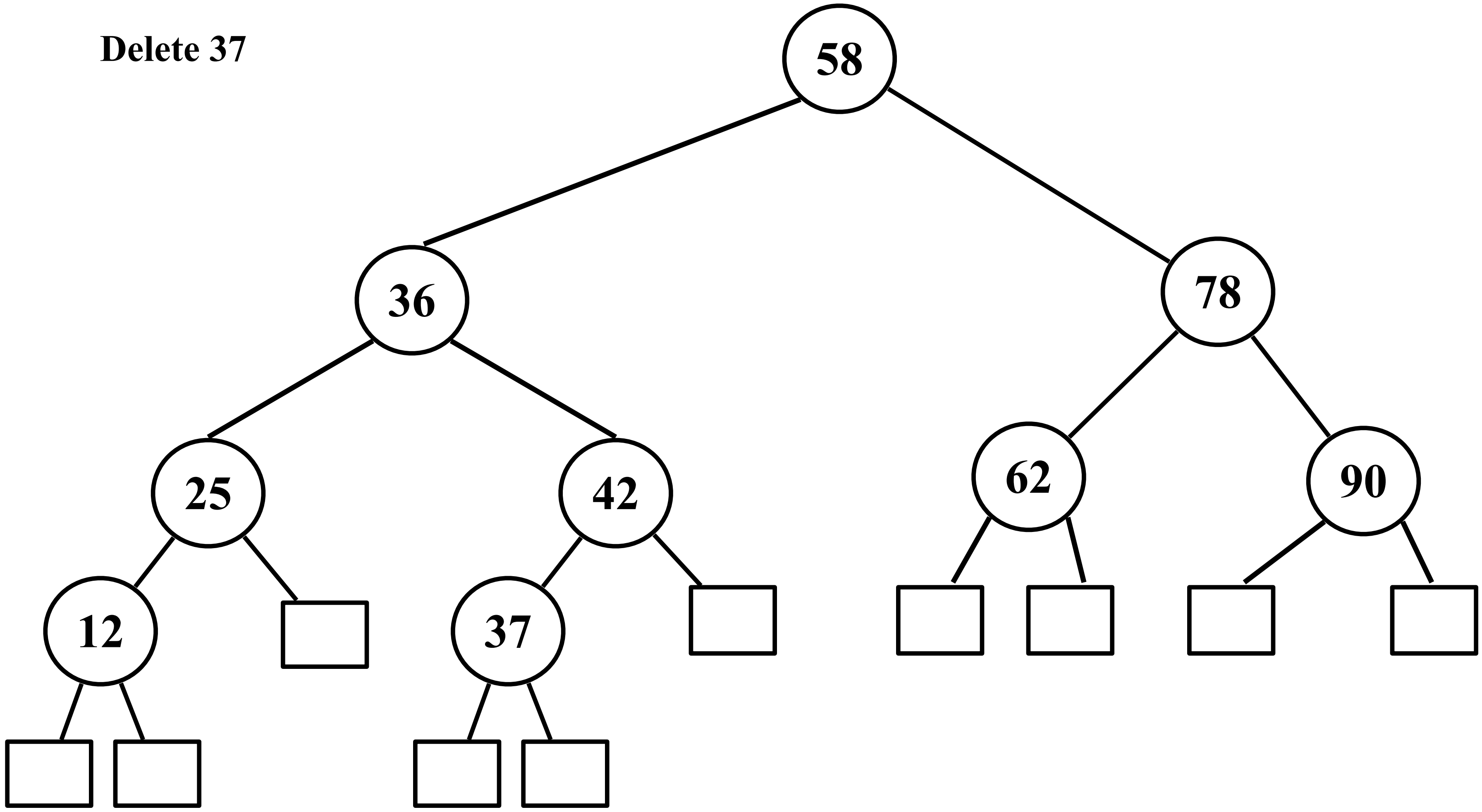


Deletion

Delete key 37

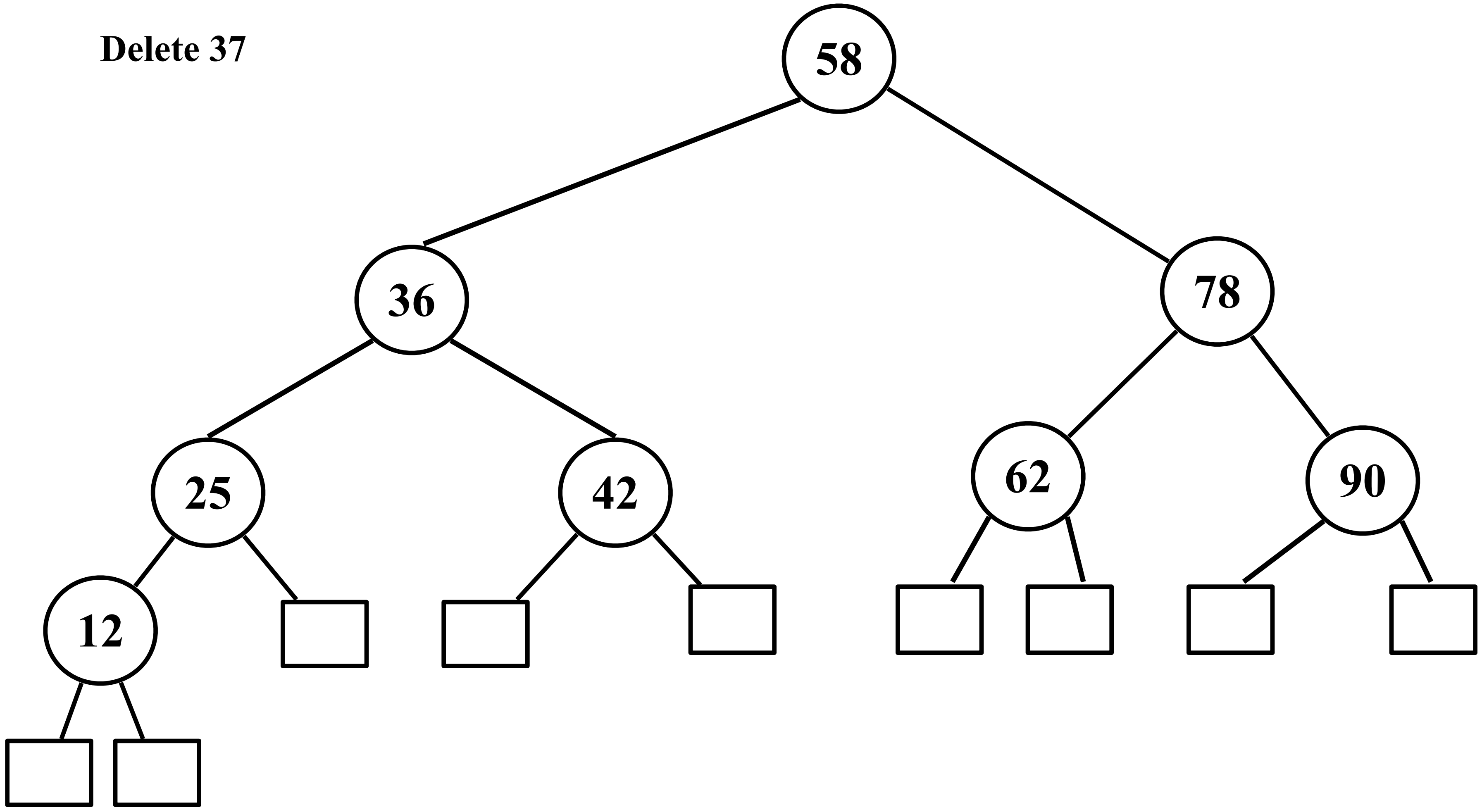


Step 1. Proceed as in binary search trees



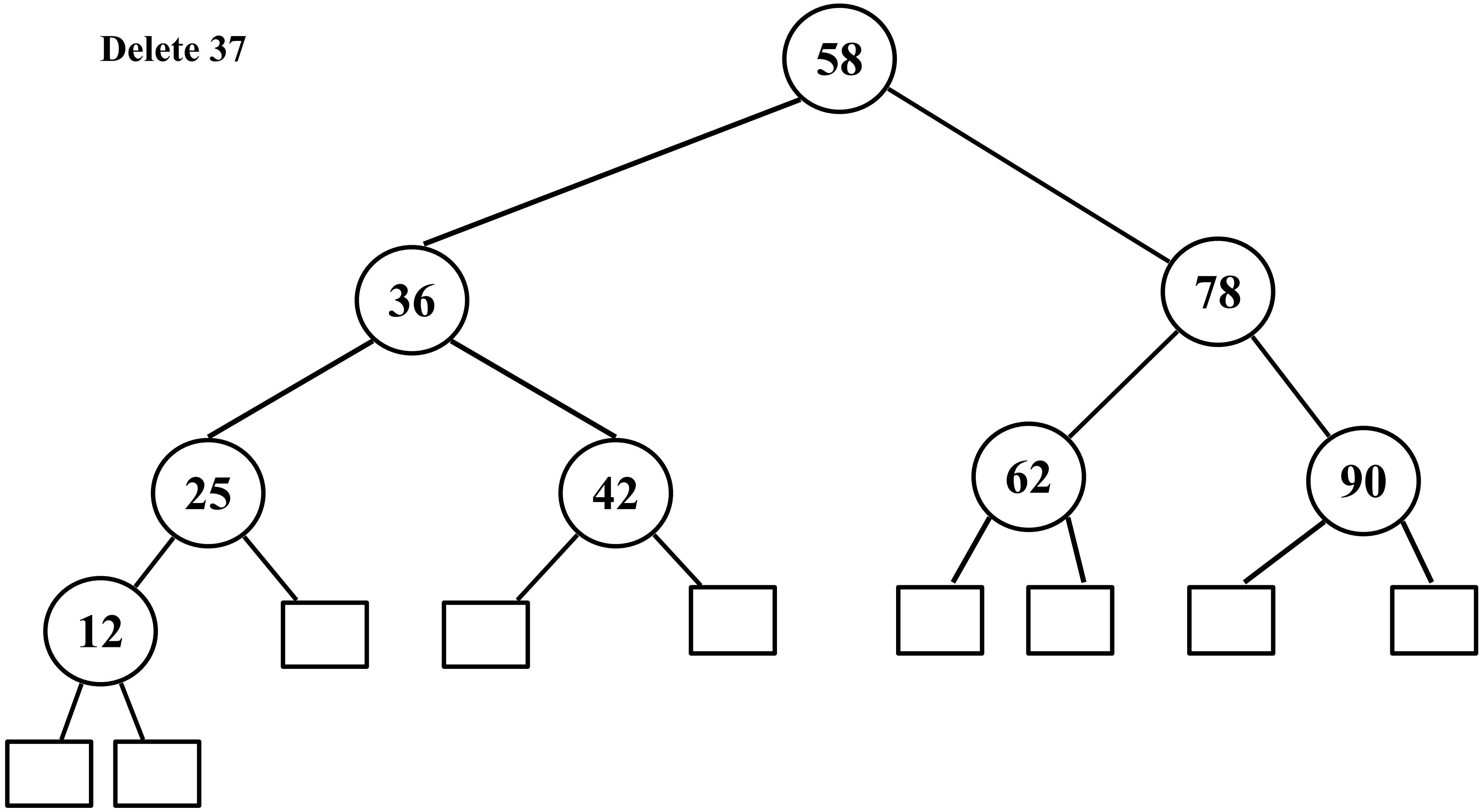
Step 1. Proceed as in binary search trees

Delete 37

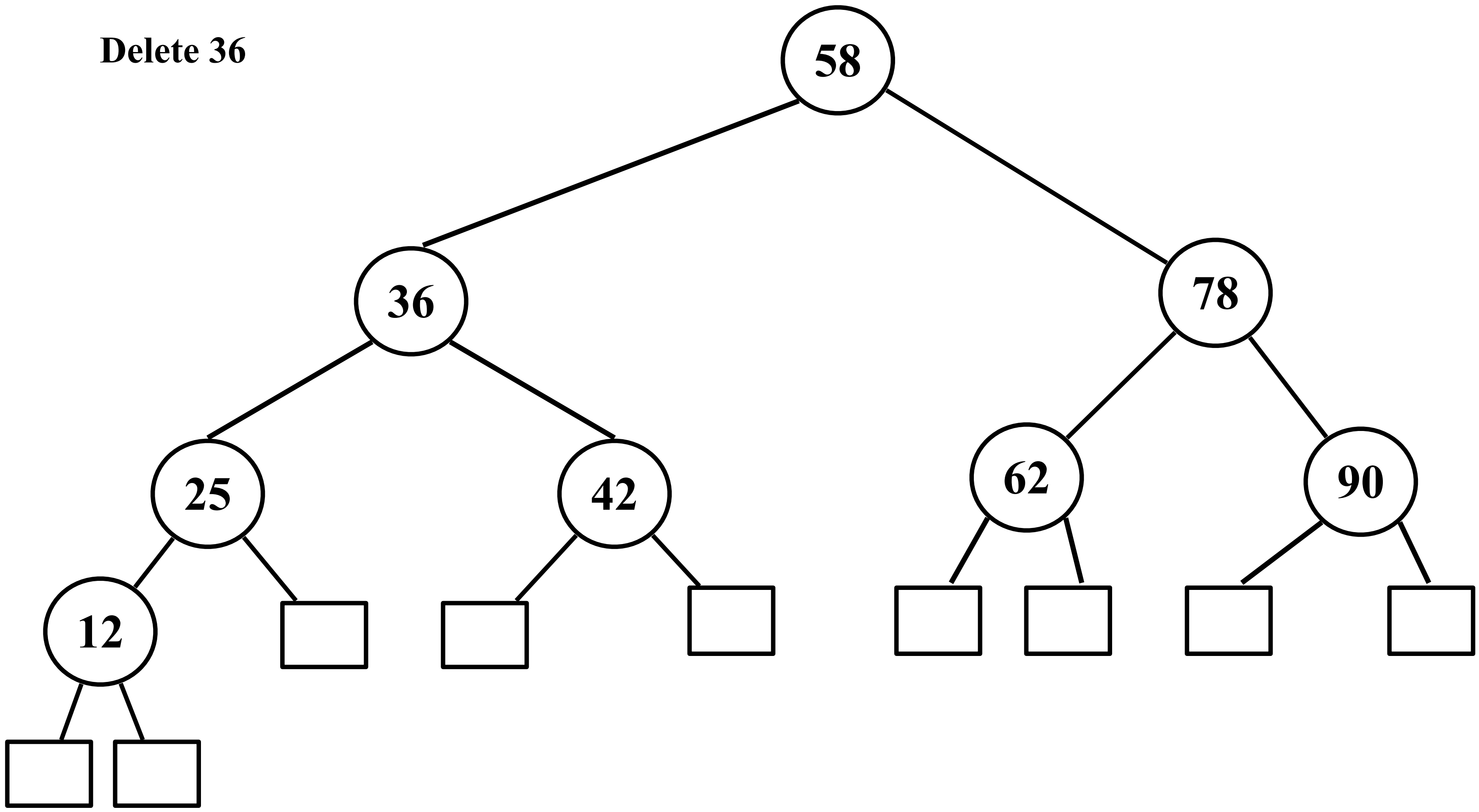


Step 2. Check height balance property

Delete 37

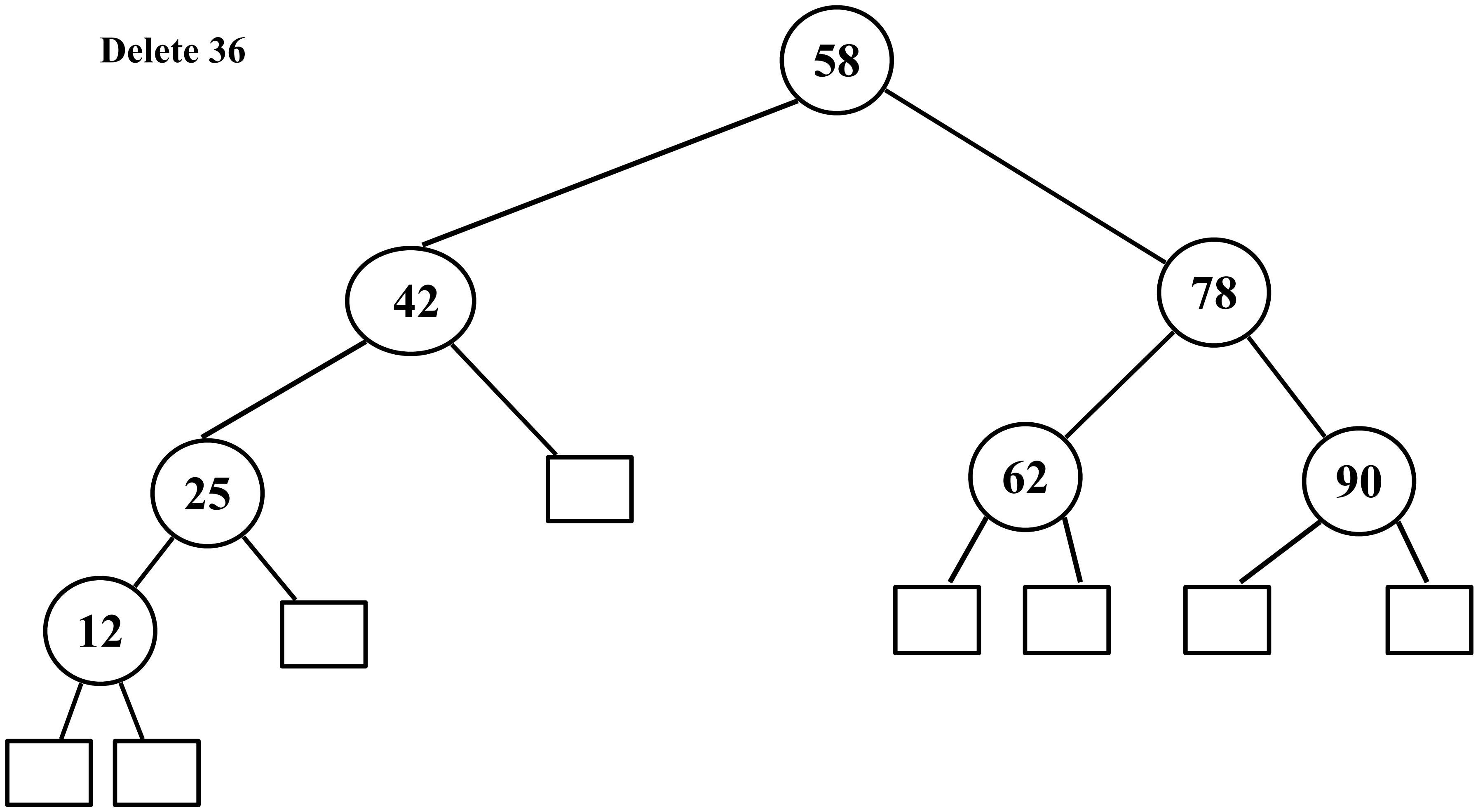


Another Deletion: Step 1



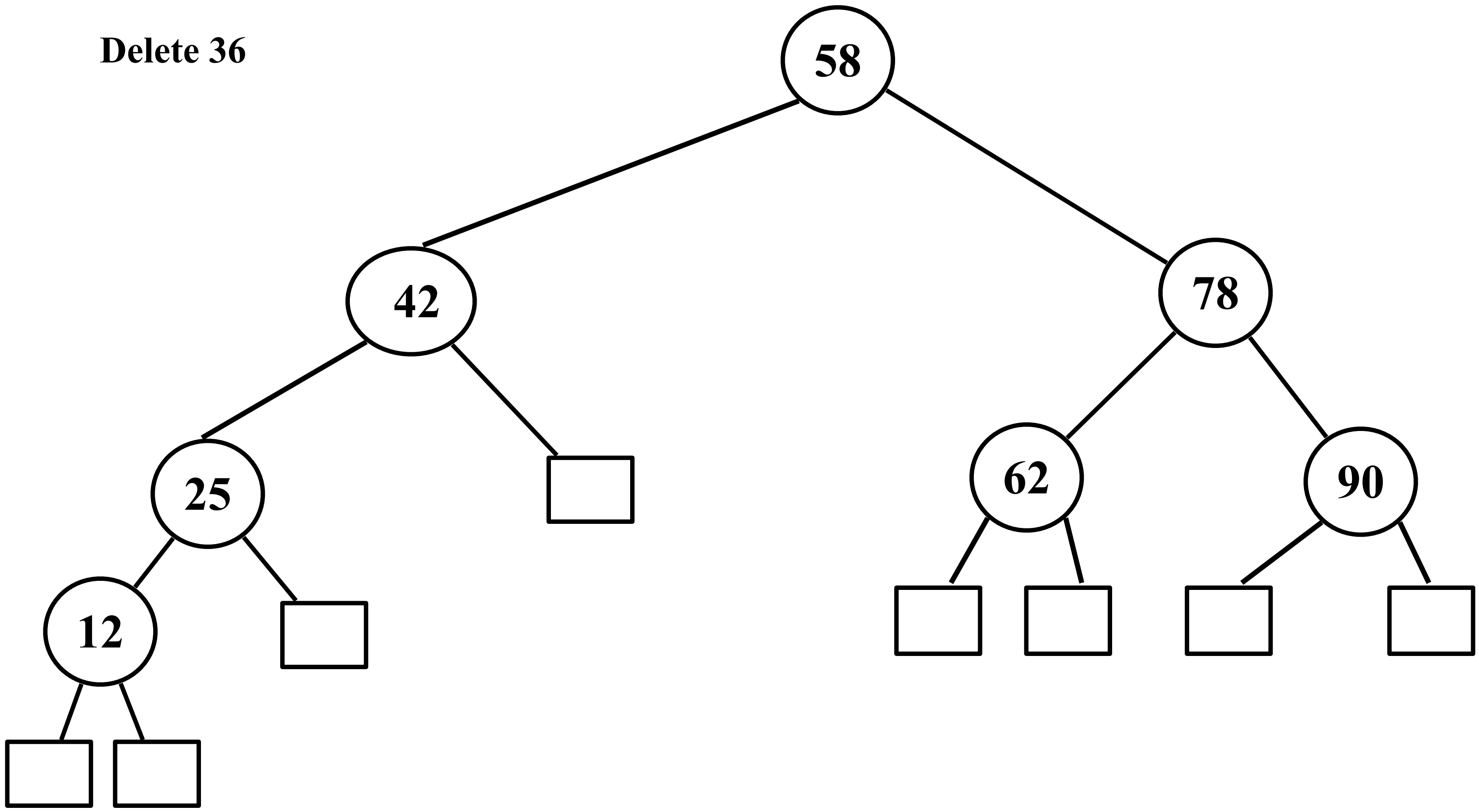
Step 1

Delete 36



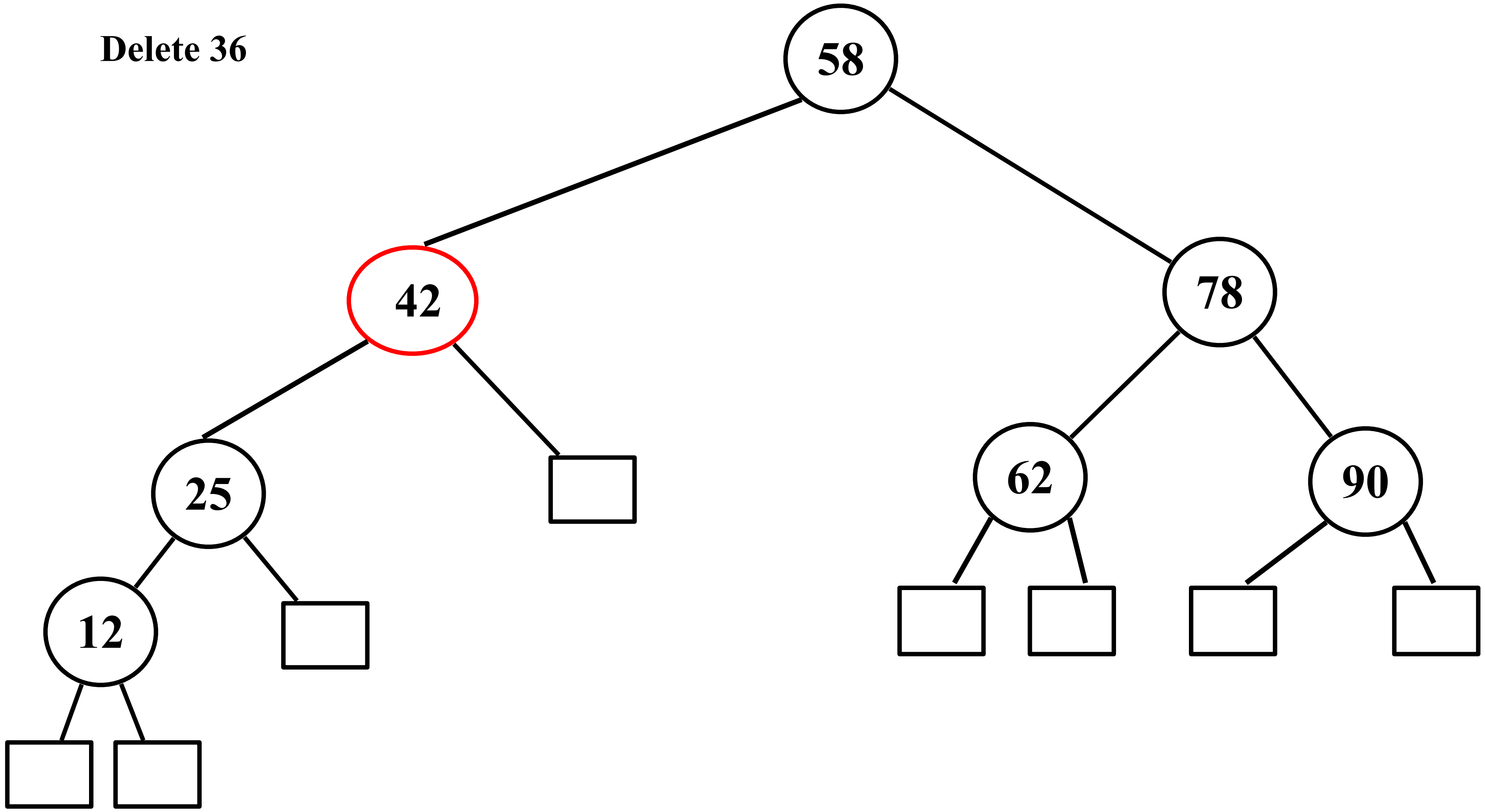
Step 2

Delete 36



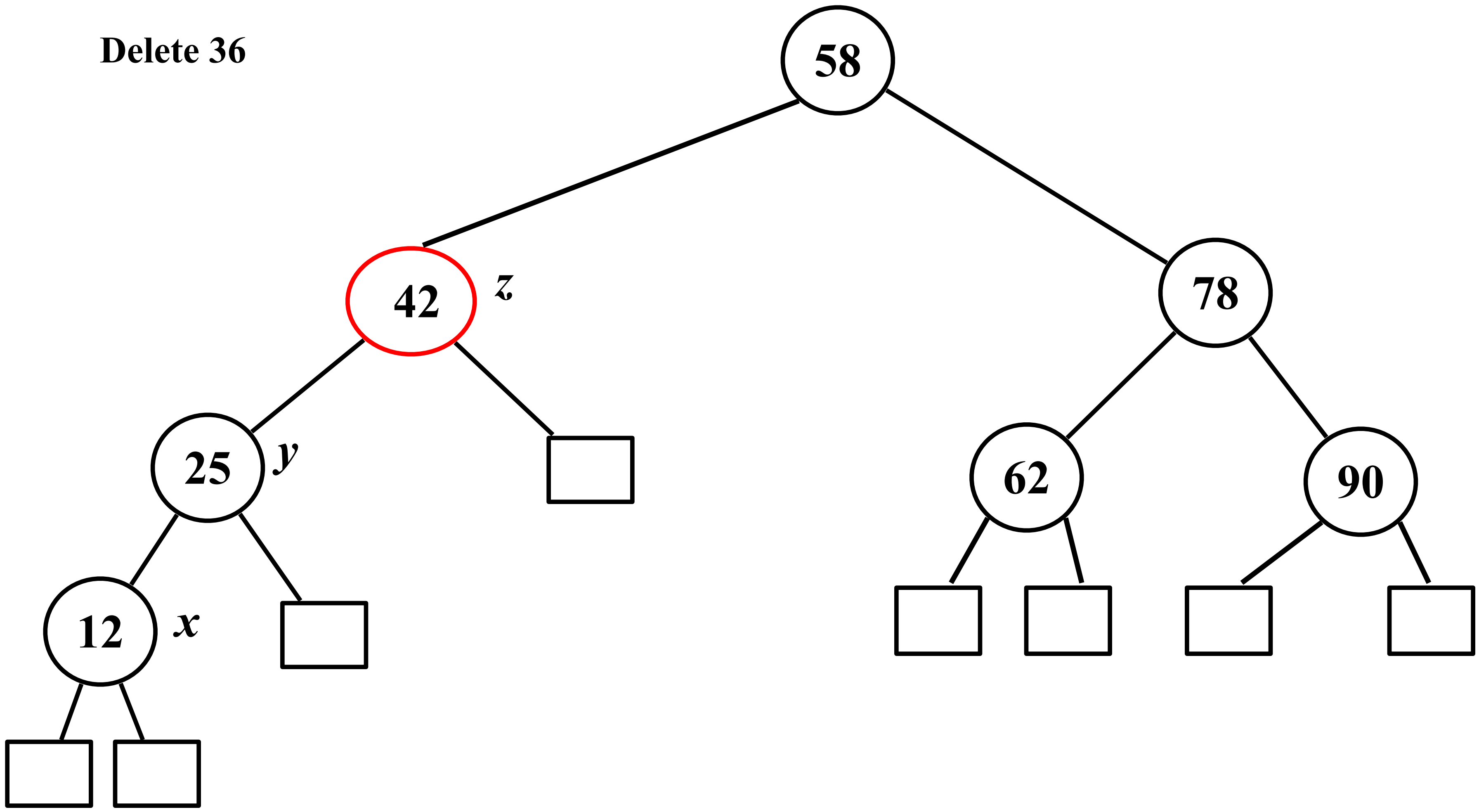
Restructure!

Delete 36



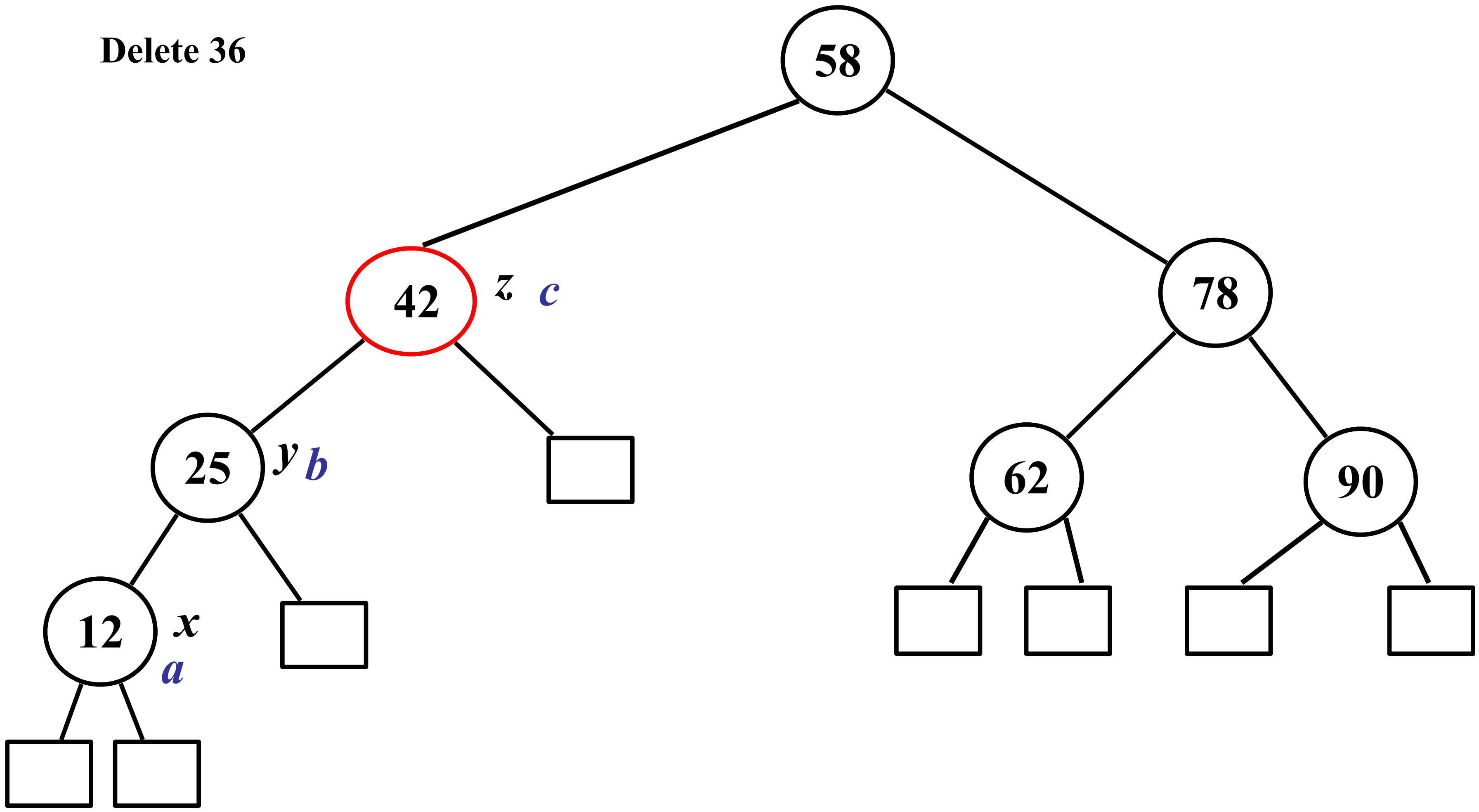
Restructure!

Delete 36



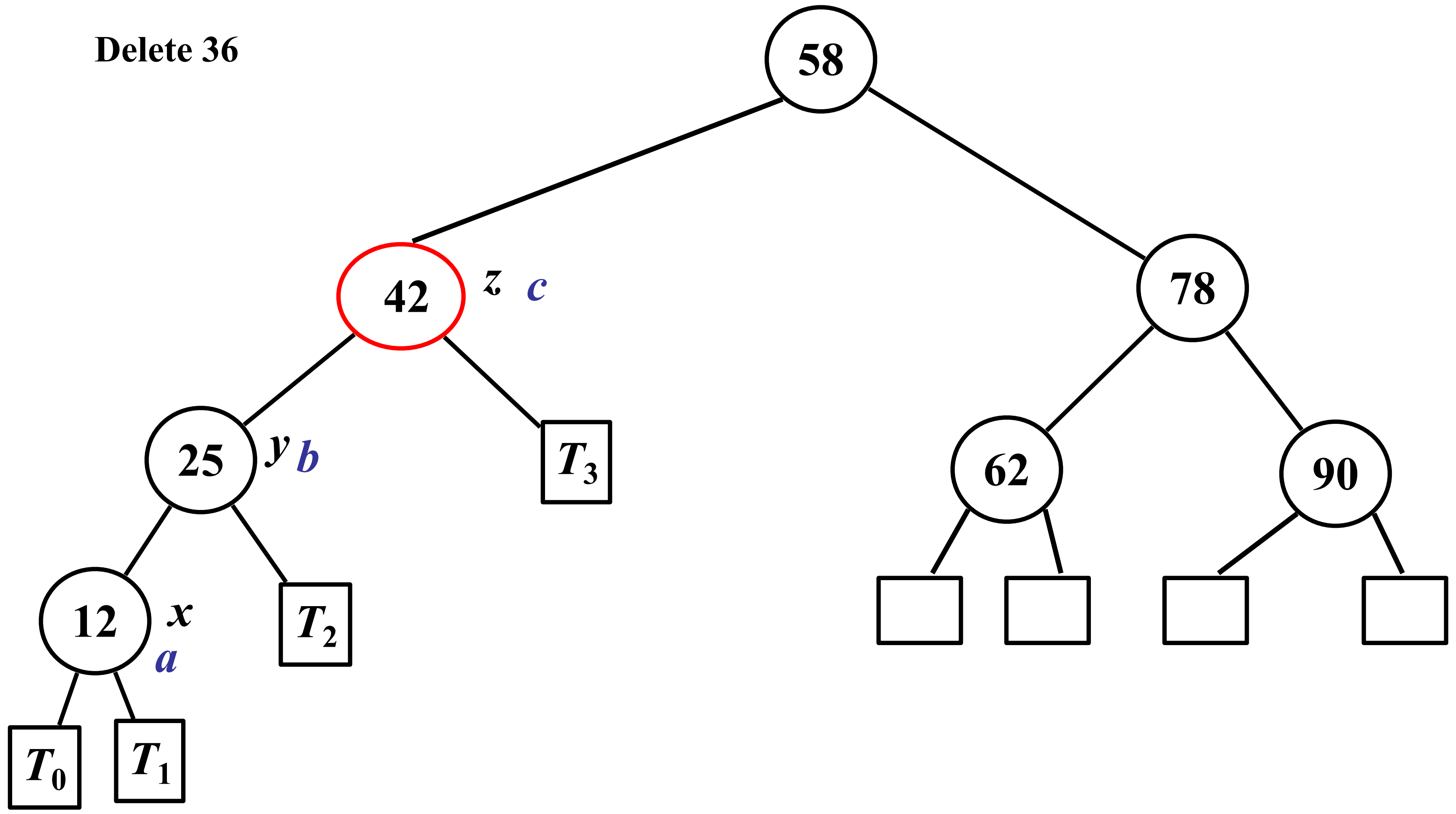
Restructure!

Delete 36



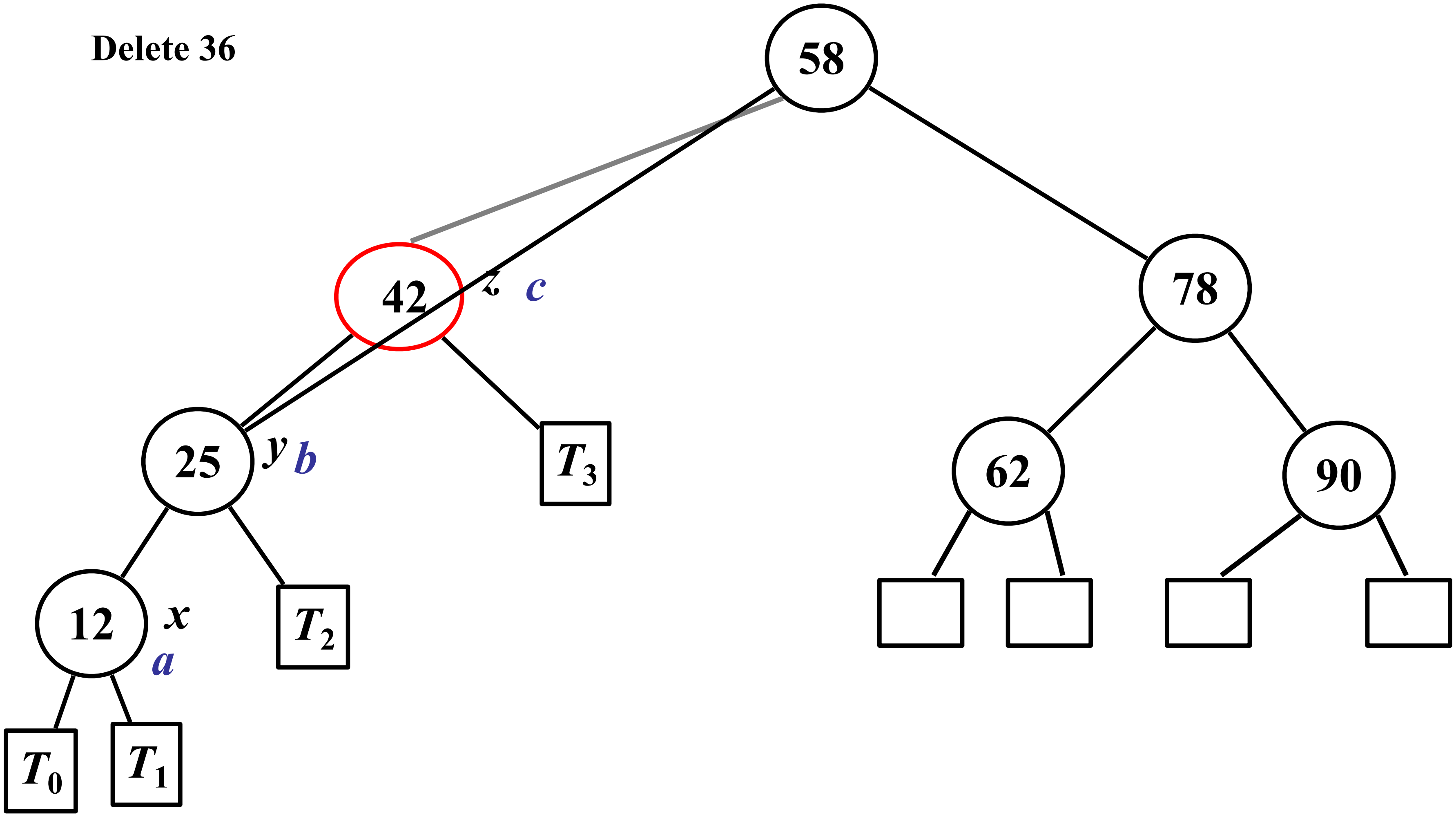
Restructure!

Delete 36



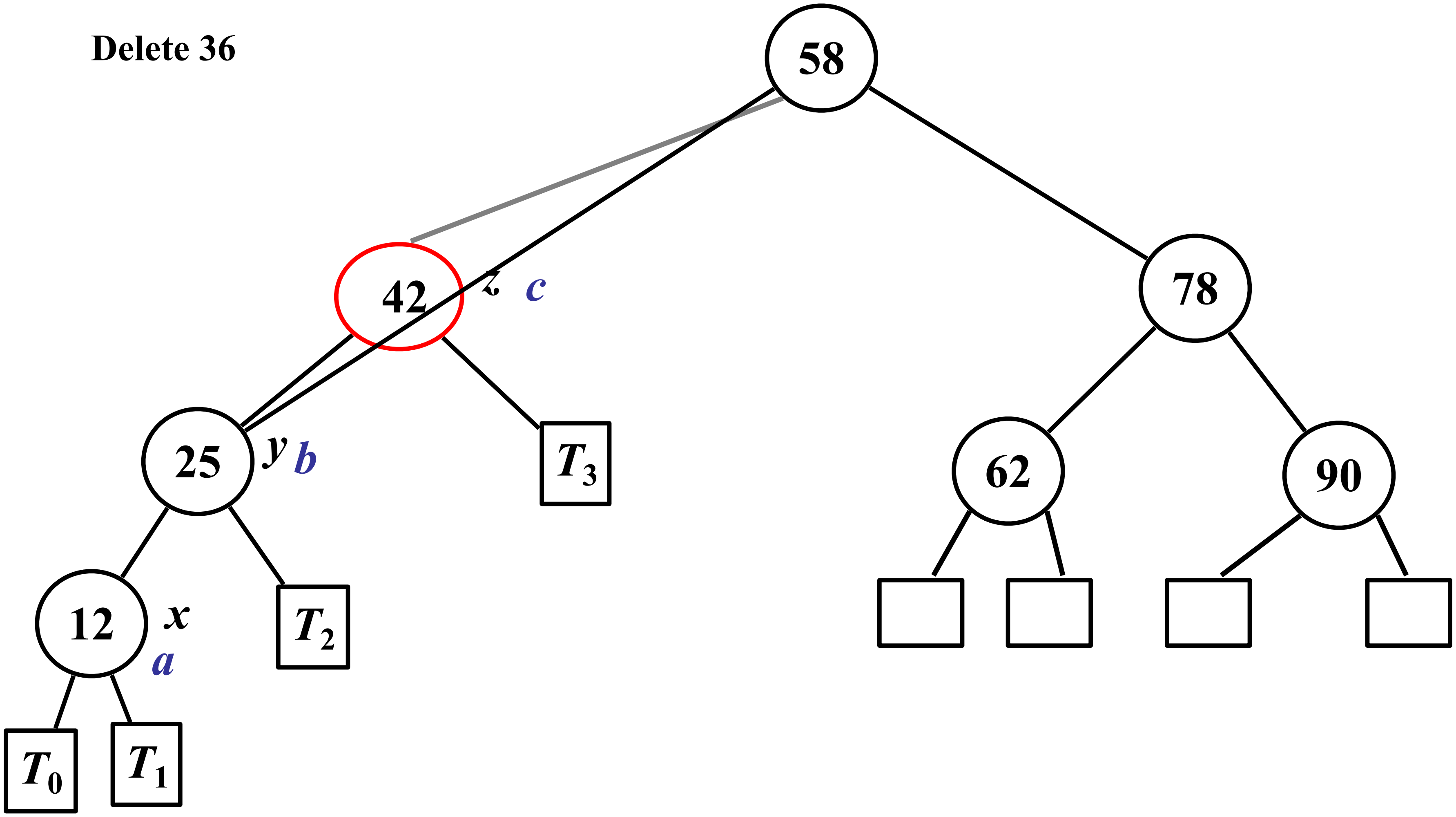
Restructure!

Delete 36



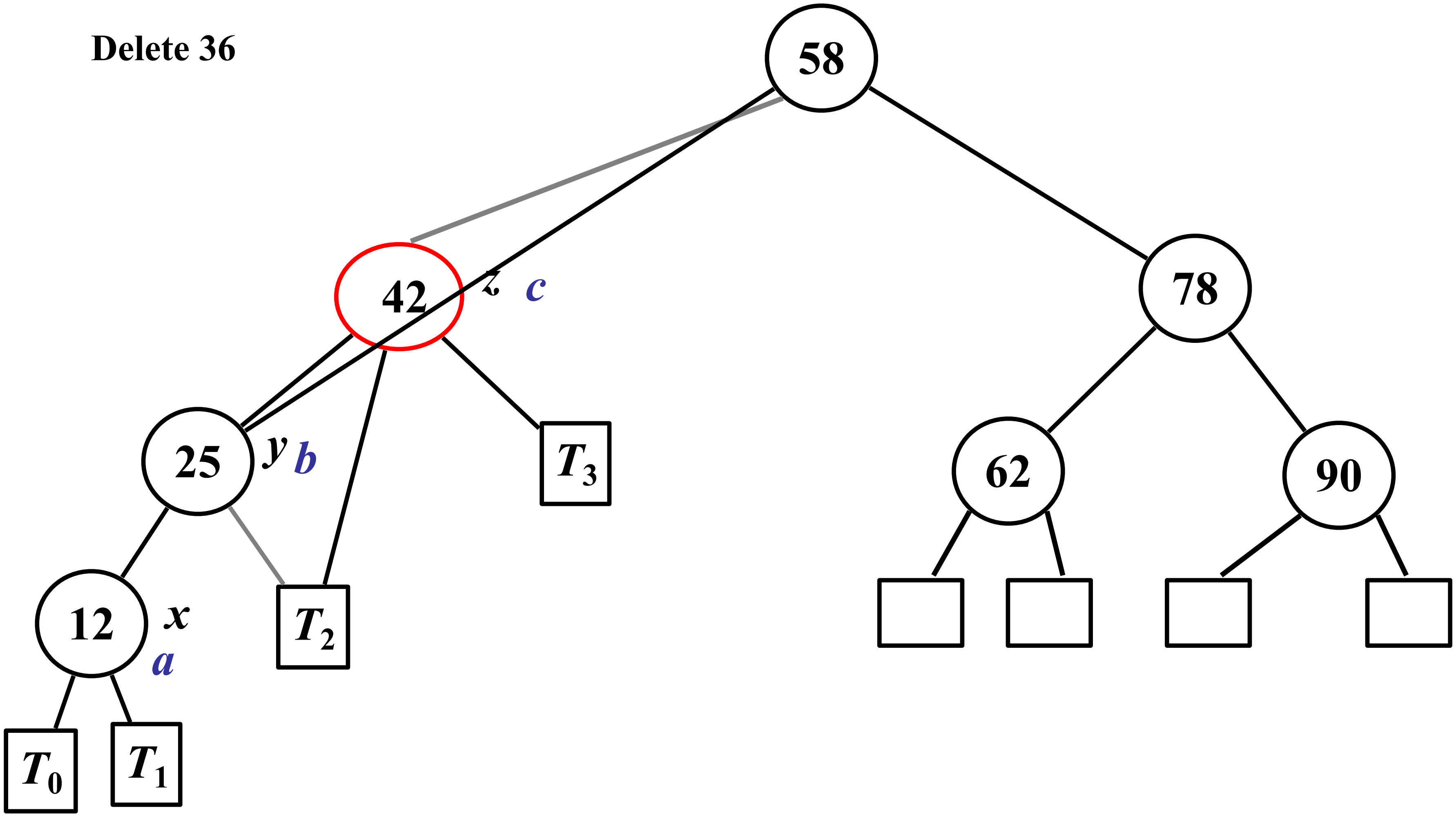
Restructure!

Delete 36



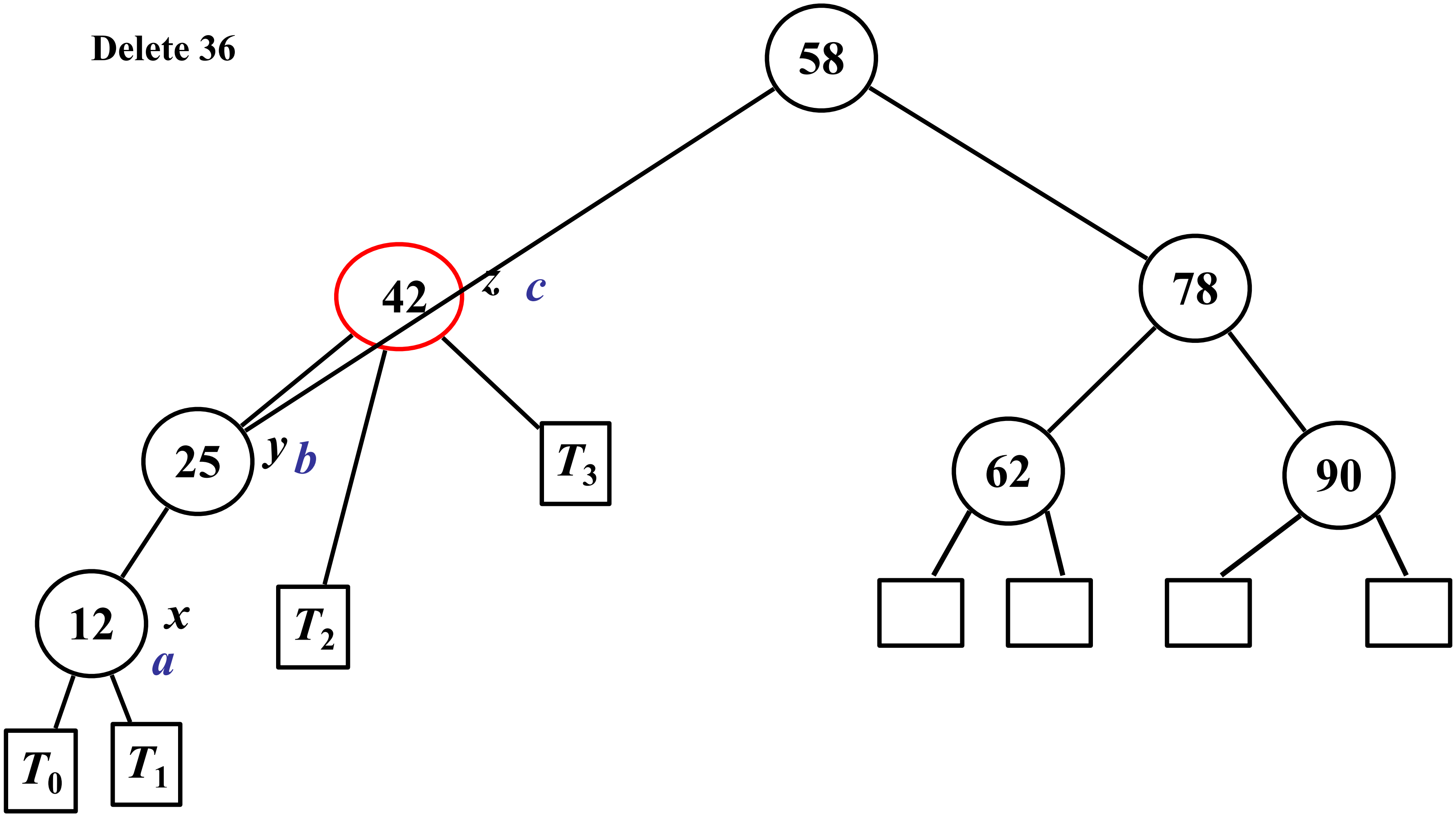
Restructure!

Delete 36



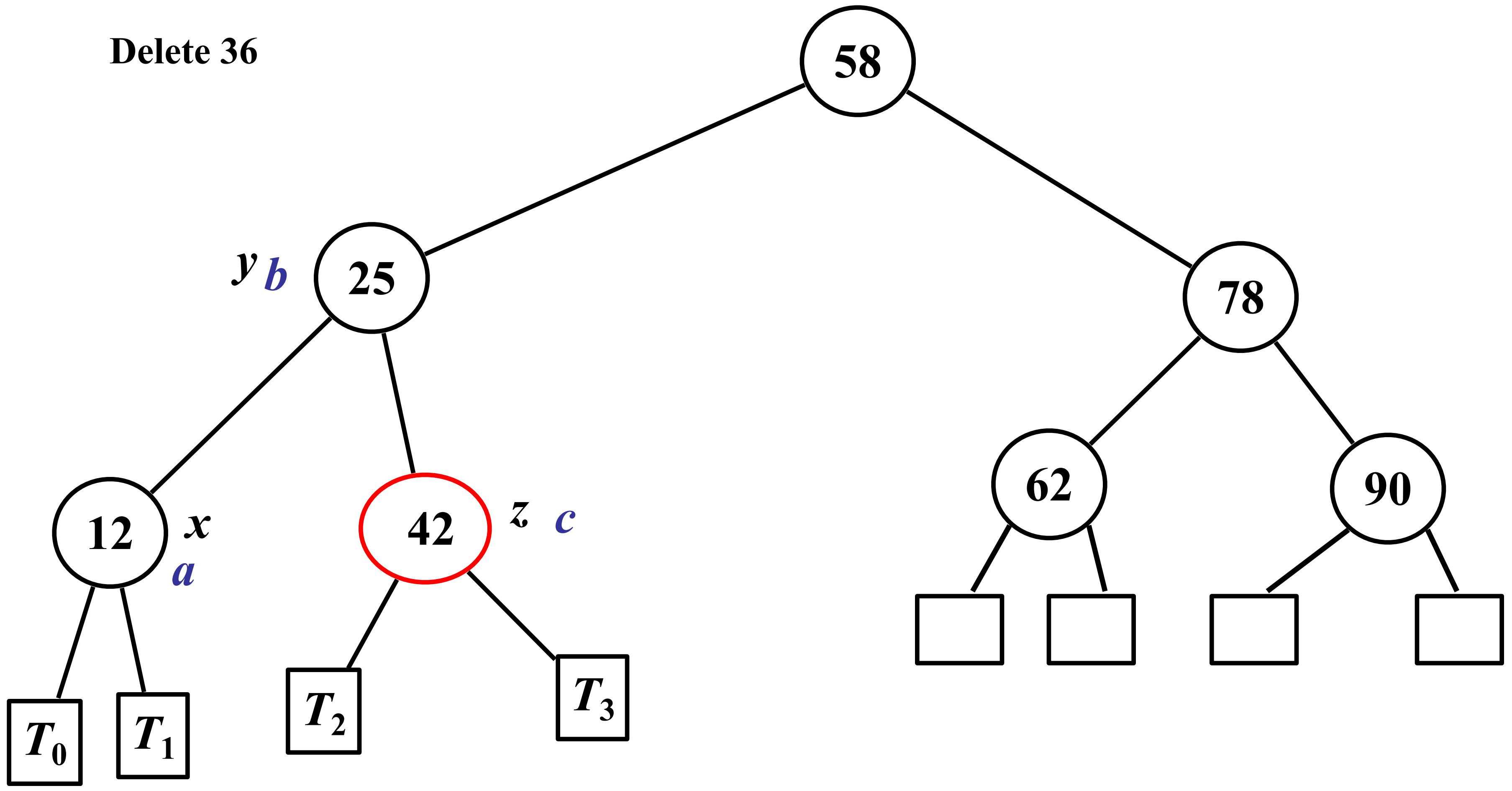
Restructure!

Delete 36



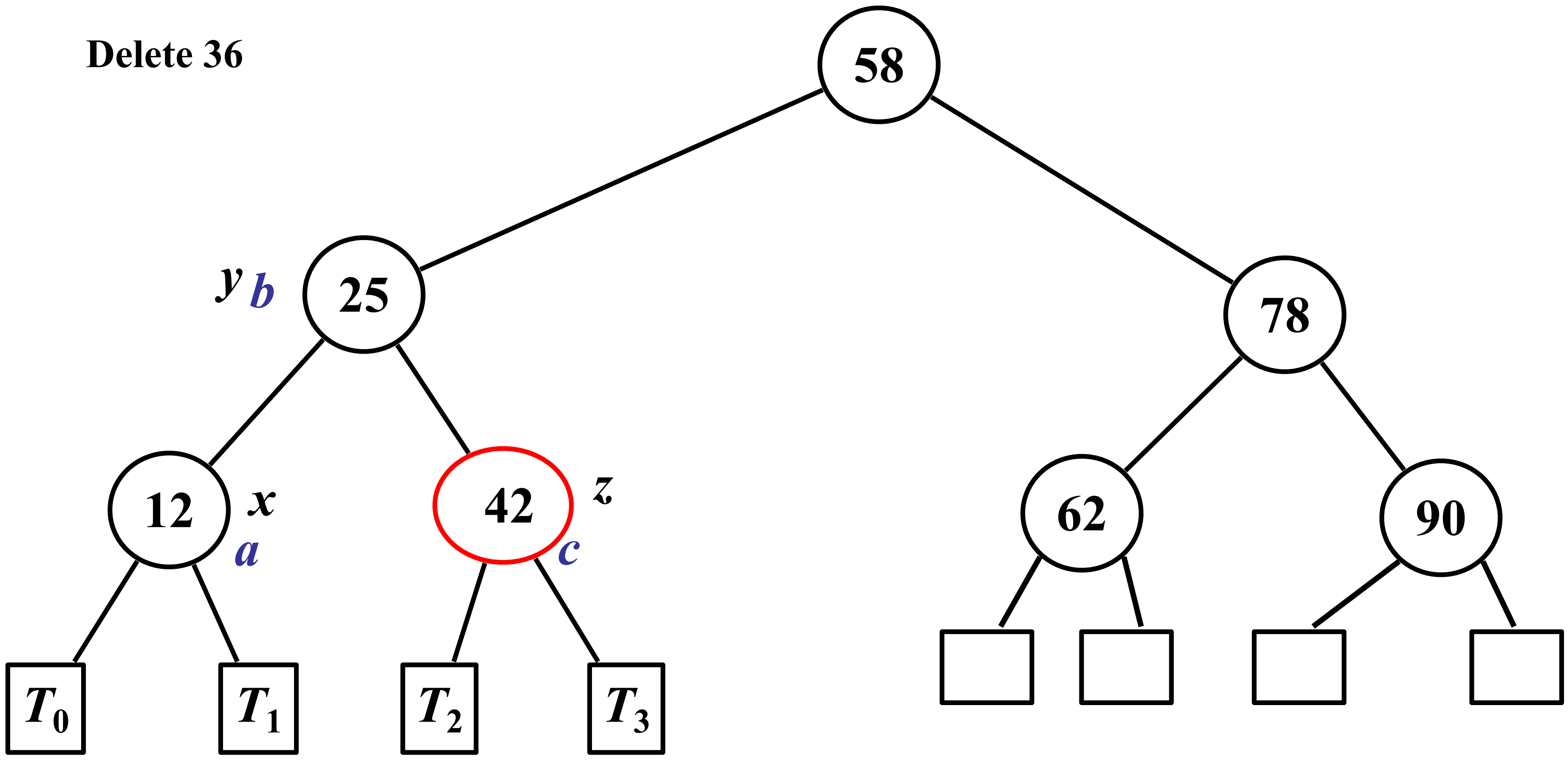
Restructure!

Delete 36



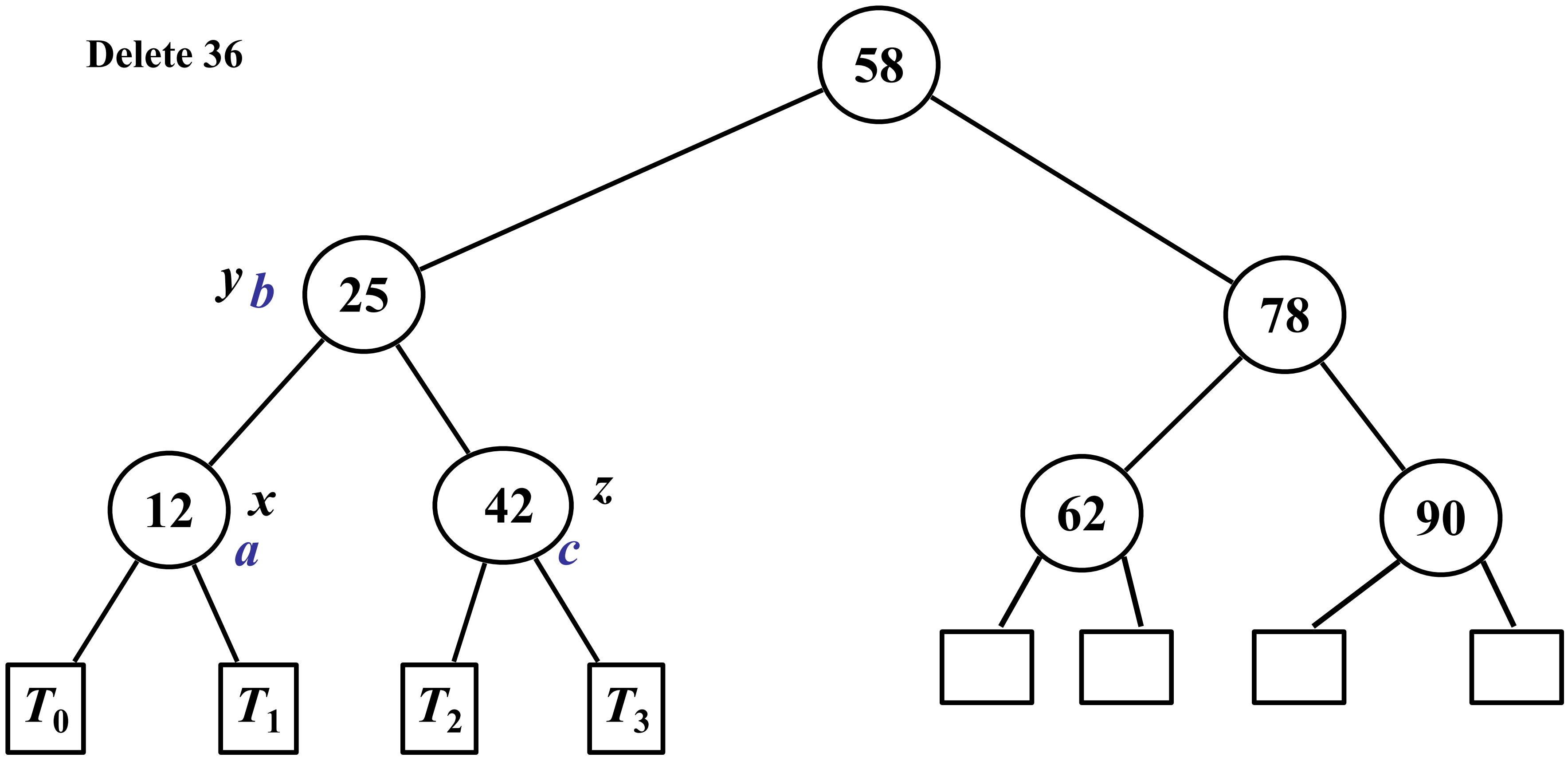
Restructure!

Delete 36



Restructure!

Delete 36



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[1])

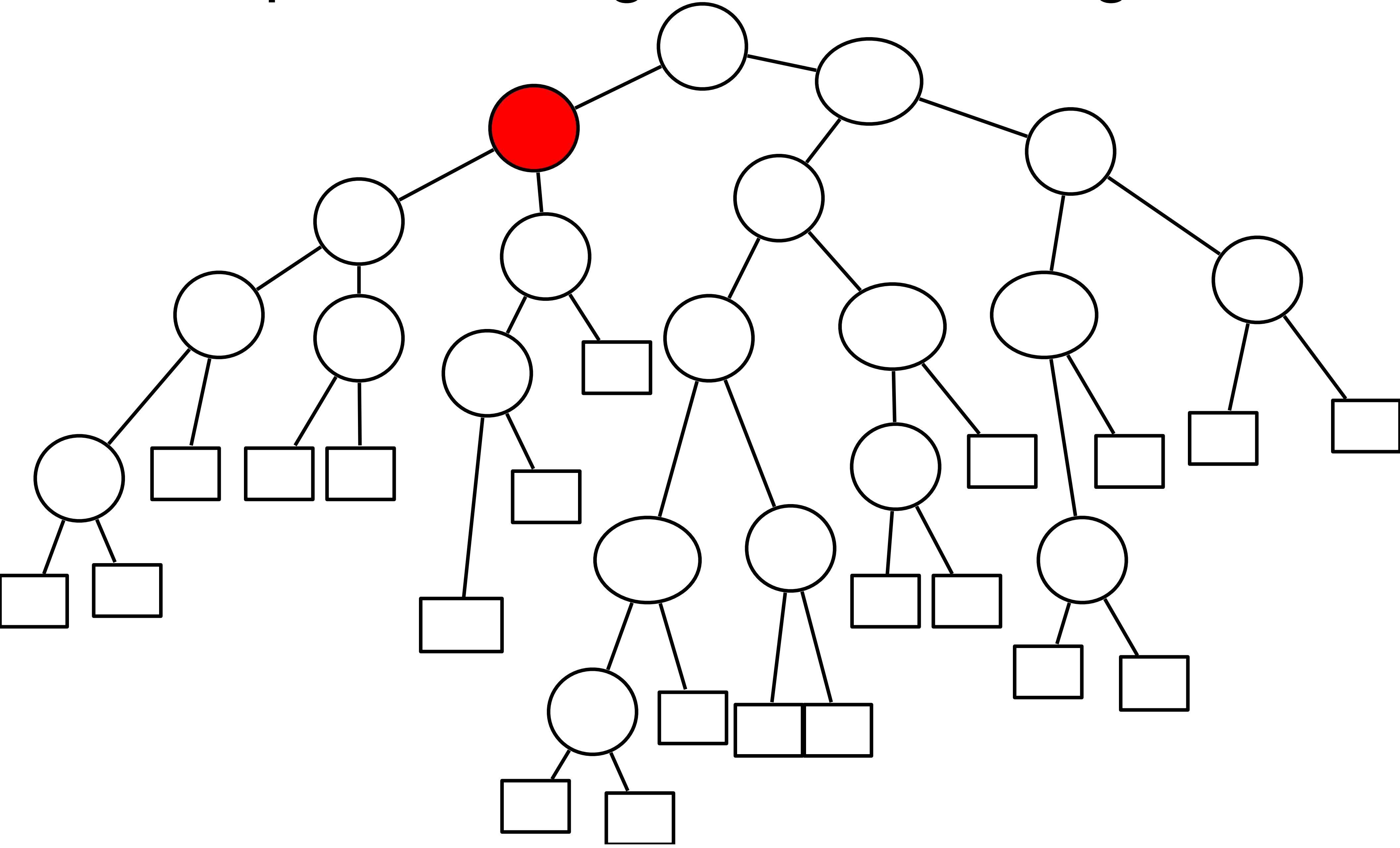
Output: Tree T after trinode-restructuring (corresponding to a single or double rotation) involving x, y, and z

[1]If y's subtrees are both of the same hight, then pick x as right child if y is a right child of z, and as left child otherwise. This is important for rebalancing after deletion.

Searching for unbalanced nodes after deletion

- Start at parent of removed node, walk up the tree. The first unbalanced node encountered is node z
- z's higher sibling is y
- y's higher sibling (if there is one) is x
- If both children of y are of the same height then:
    - if y is z's left child then x is y's left child
    - if y is z's right child then x is y's right child

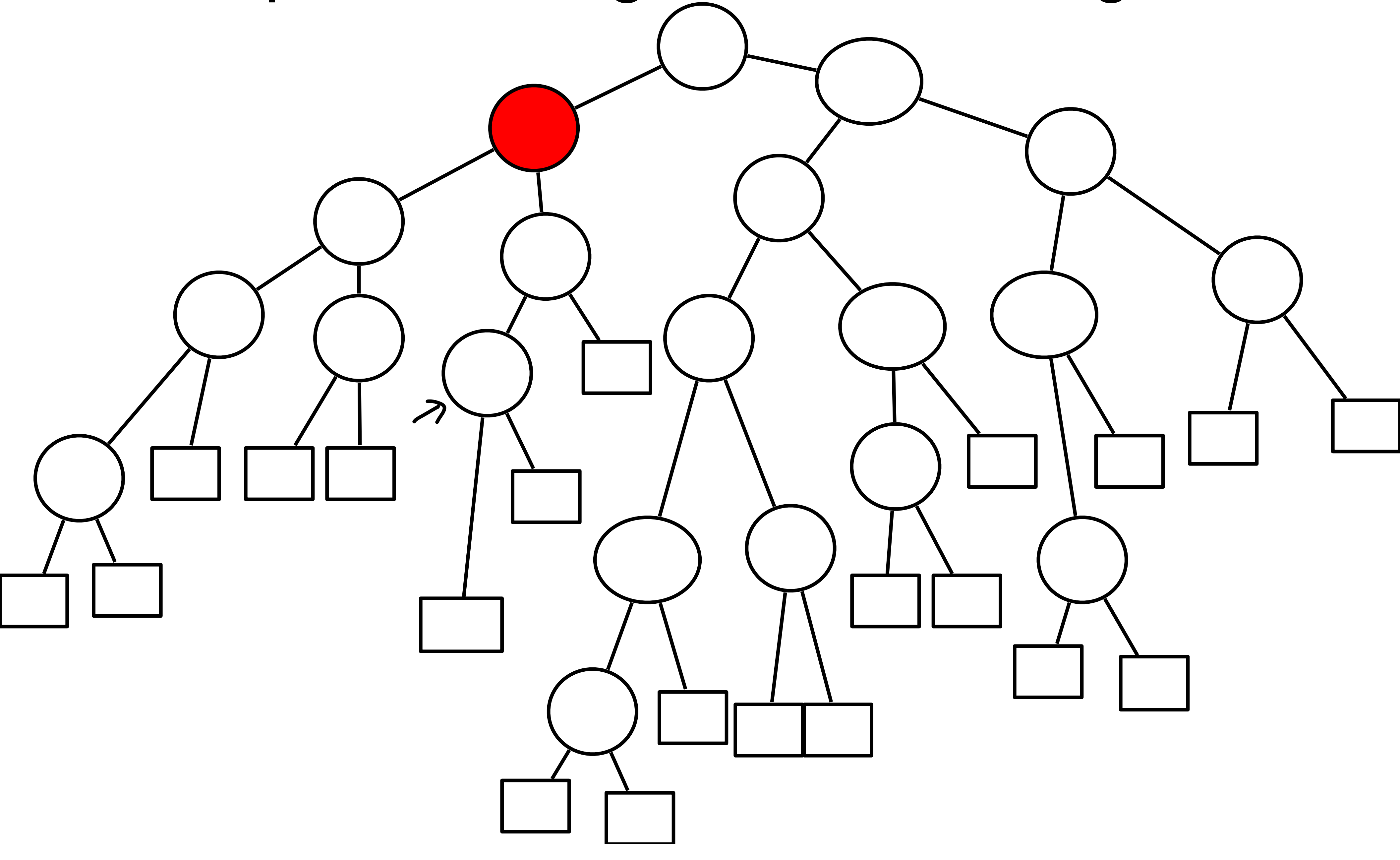
After Deletion: One Restructure-Operation might not be enough!



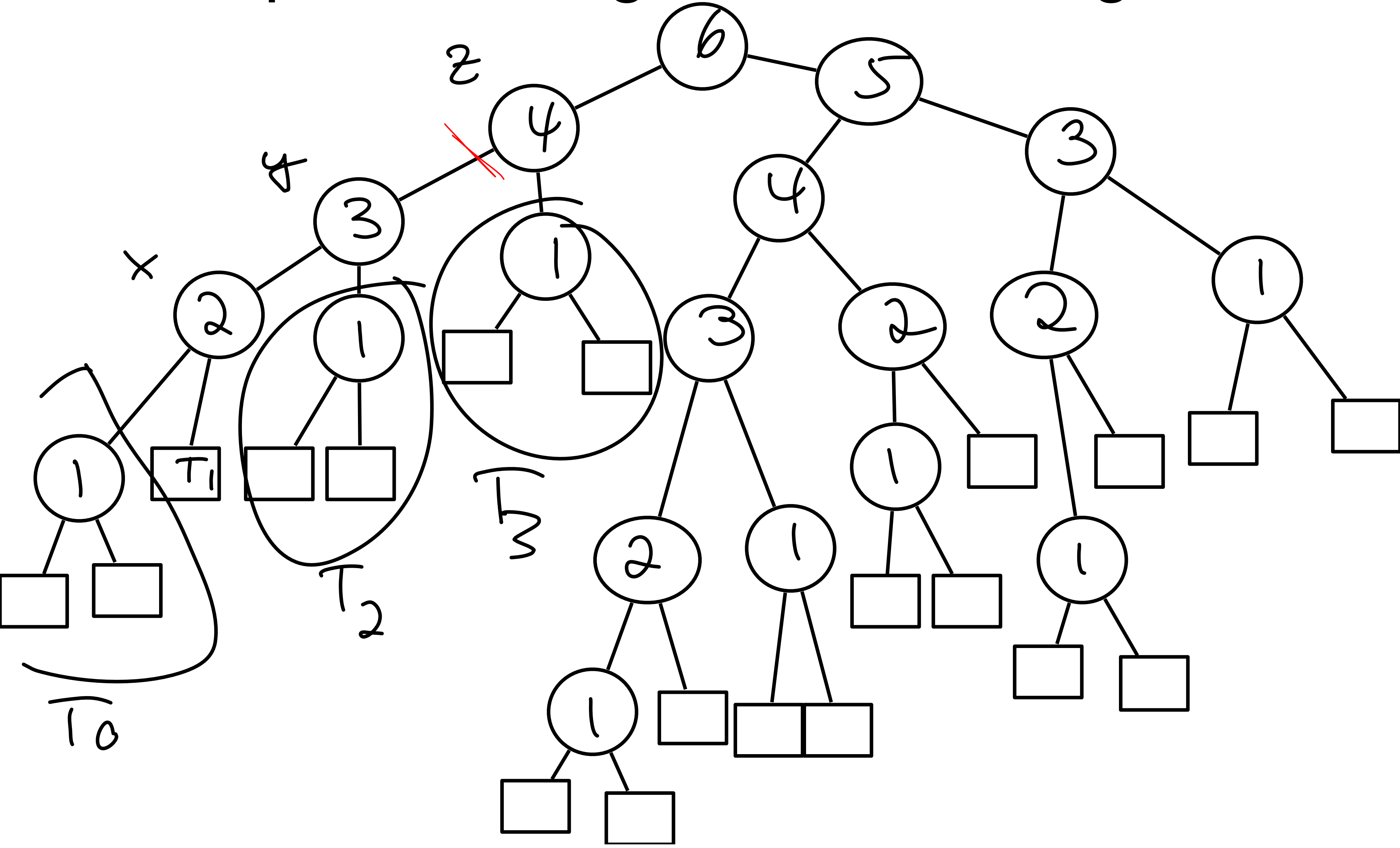
Theorem

- No more than O(log n) many restructuring operations are necessary to rebalance an AVL tree after deleting a key using binary search tree deletion.

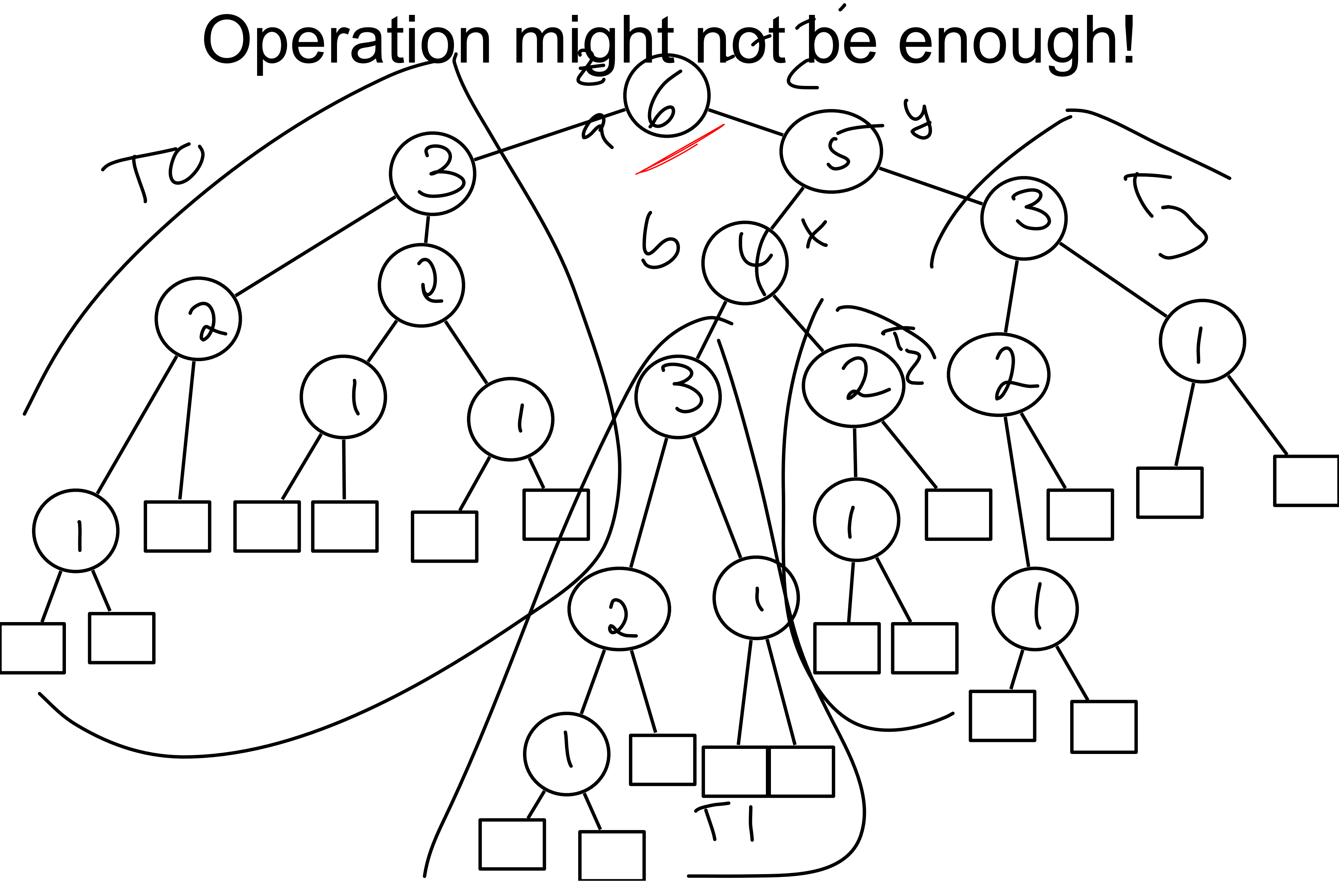
After Deletion: One Restructure-Operation might not be enough!



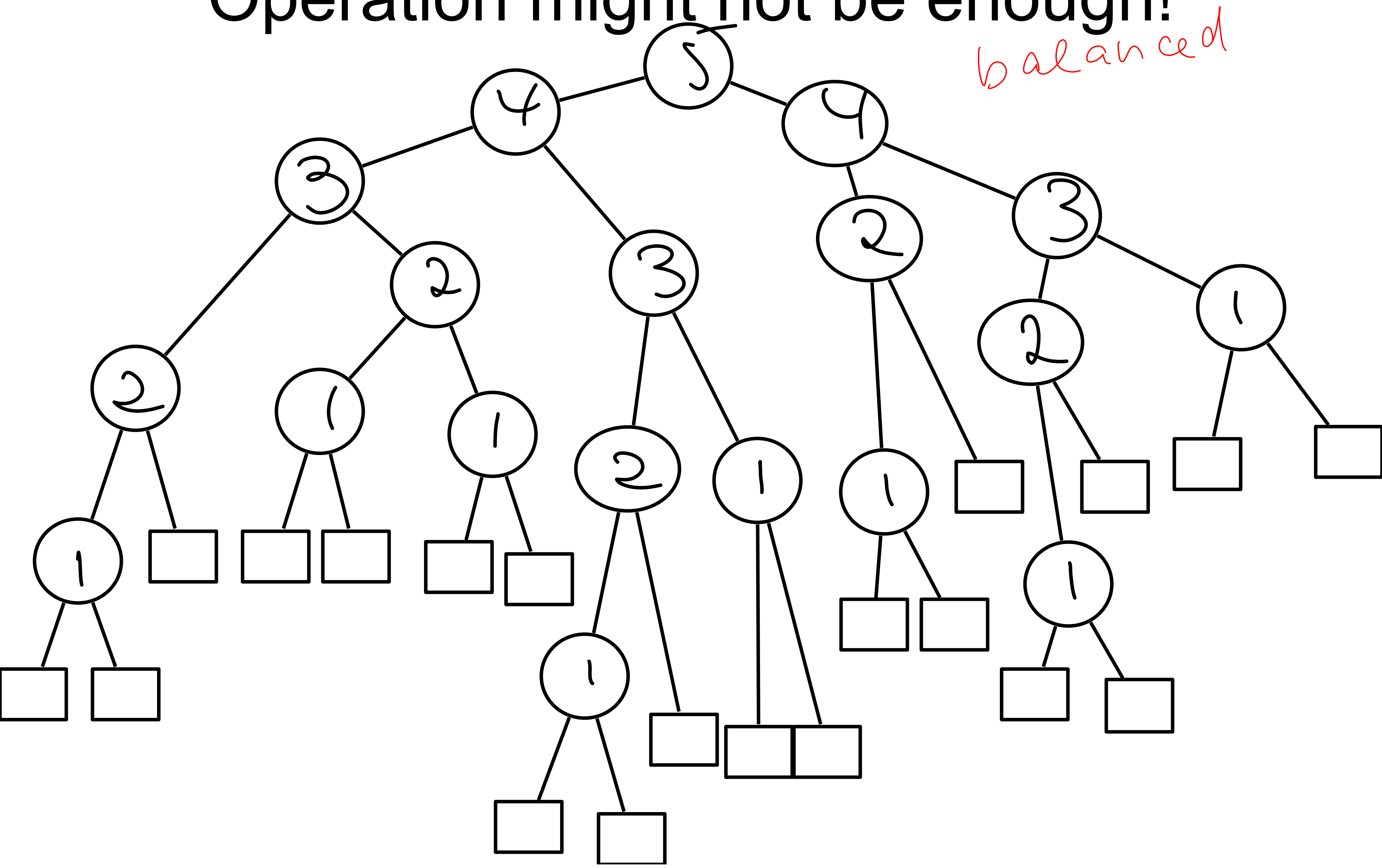
After Deletion: One Restructure-Operation might not be enough!



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



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

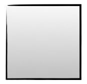


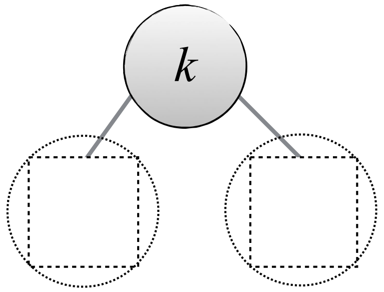
There can be as many as $O(\log n)$ many restructuring operations necessary to delete an element in an AVL tree.

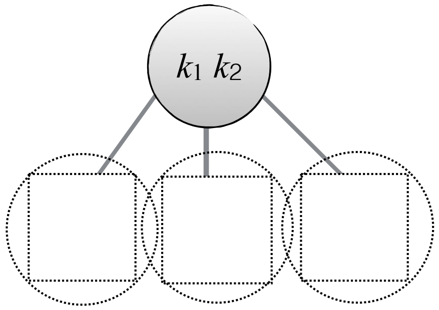
2-3 trees

- Not binary
    - In contrast to AVL trees and red black trees
- How to guarantee balance?
    - Nodes can hold more than one key
- 2-nodes: holds one key, has two children
- 3-nodes: holds two keys, has three children
- Assumption: all keys different
- Later: what if keys can be repeated?

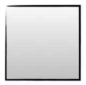
Definition (2-3 tree)

- A *2-3 search tree* is a tree that is
  - either empty 
  - or a *2-node*, with one key k (and associated value) and two links: a left link to a 2-3 search tree with keys smaller than k, and a right link to a 2-3 search tree with keys larger than k
  - or a *3-node*, with two keys $k_1 \leq k_2$ (and associated values) and three links: a left link to a 2-3 search tree with keys smaller than k_1, a middle link to a 2-3 search tree with keys larger than k_1 and smaller than k_2, and a right link to a 2-3 search tree with keys larger than k_2

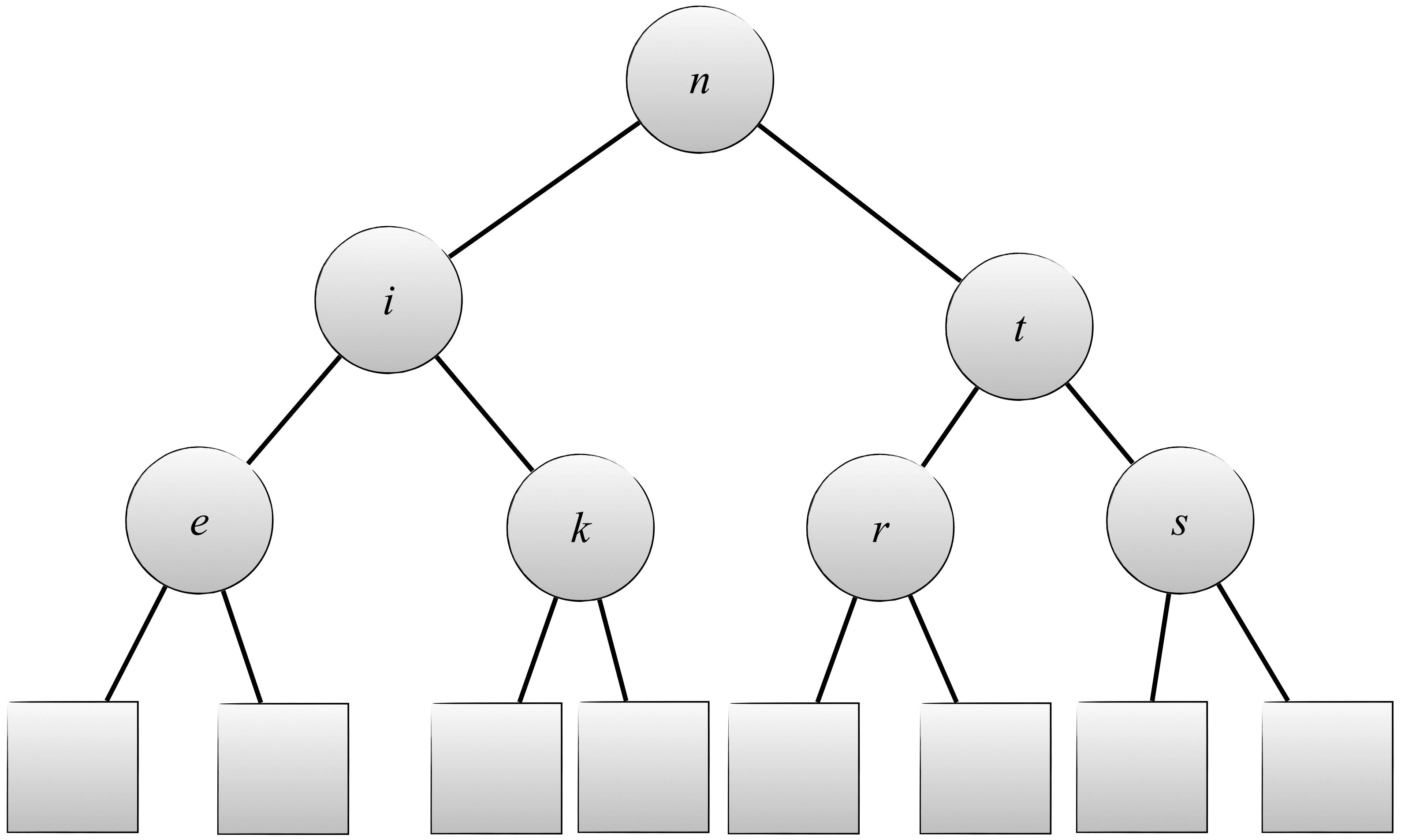




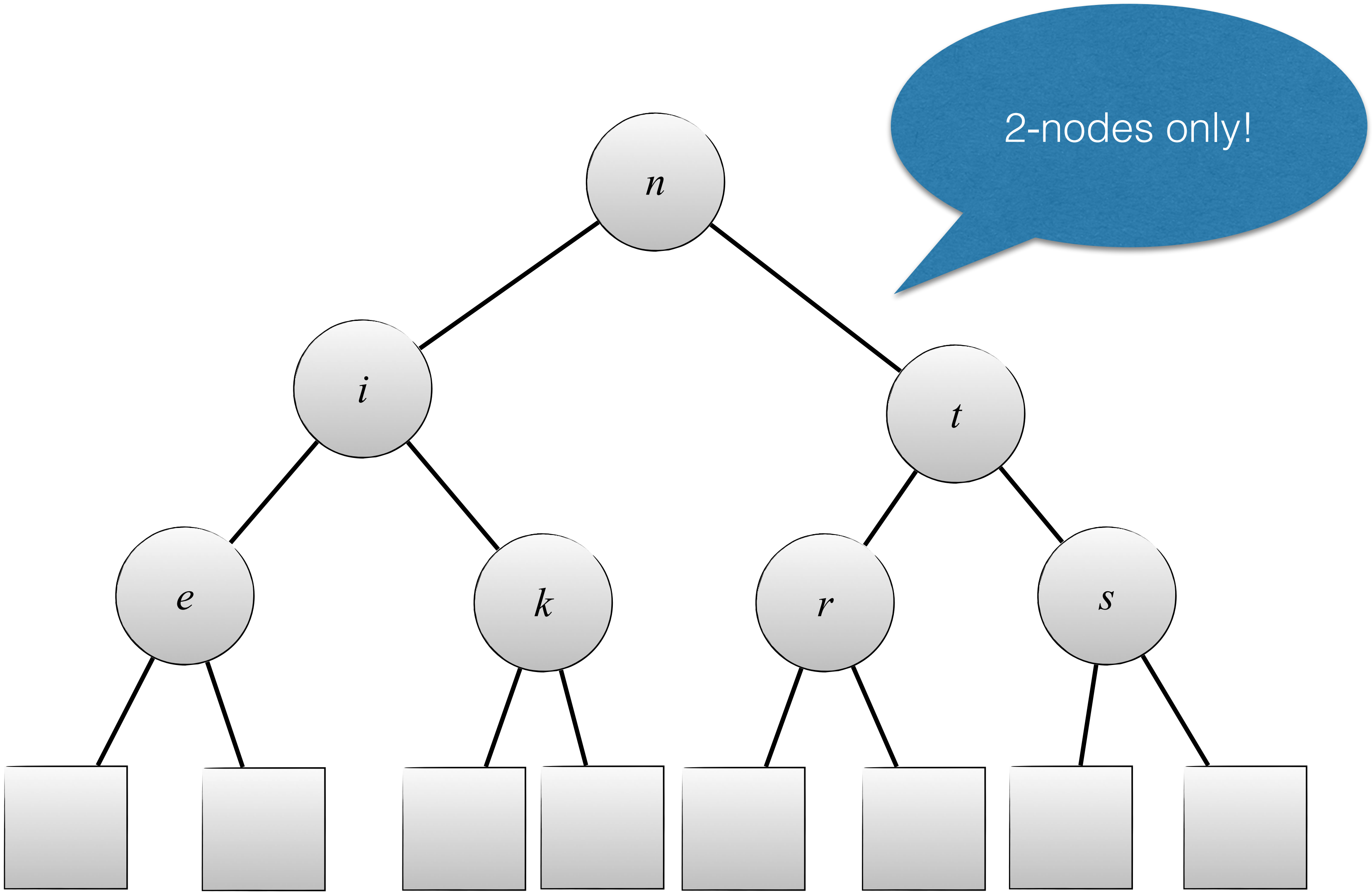
Definition (2-3 tree, continued)

- A link to an empty tree is called a *null link* or *leaf.* 
- A *2-3 tree* is a *perfectly balanced* 2-3 search tree, which is one where all null links have the same distance from the root.

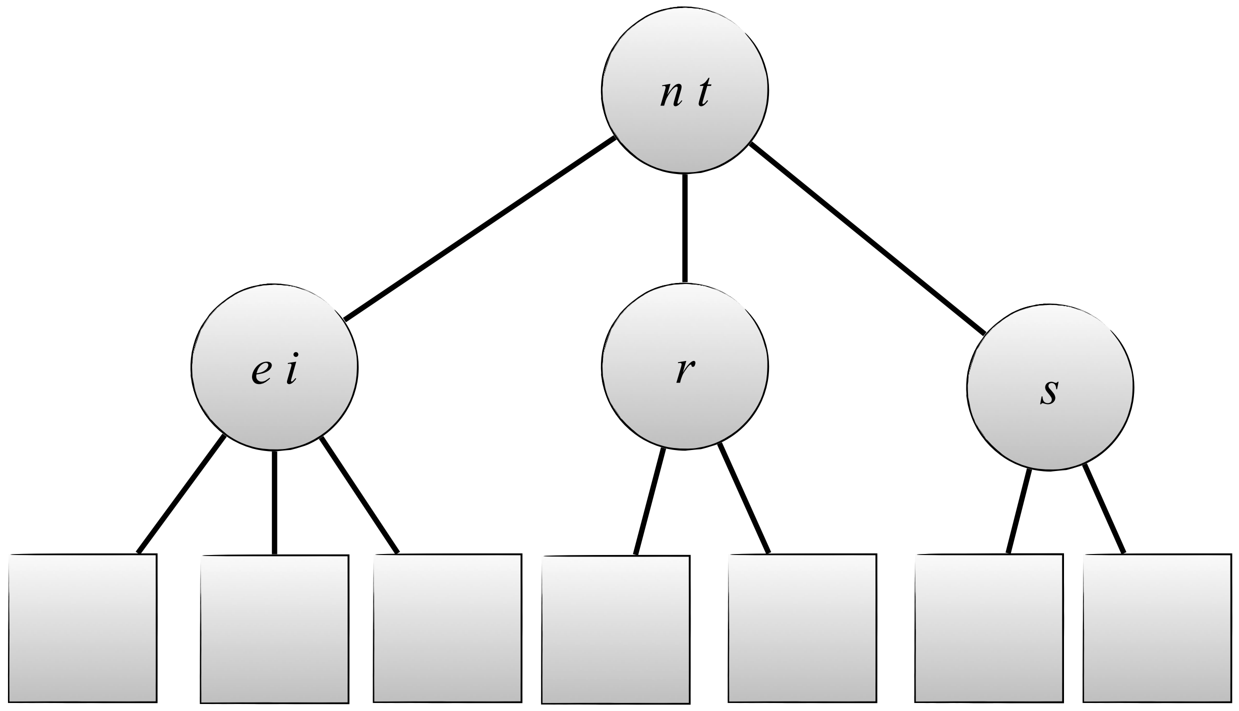
Example of a 2-3 tree



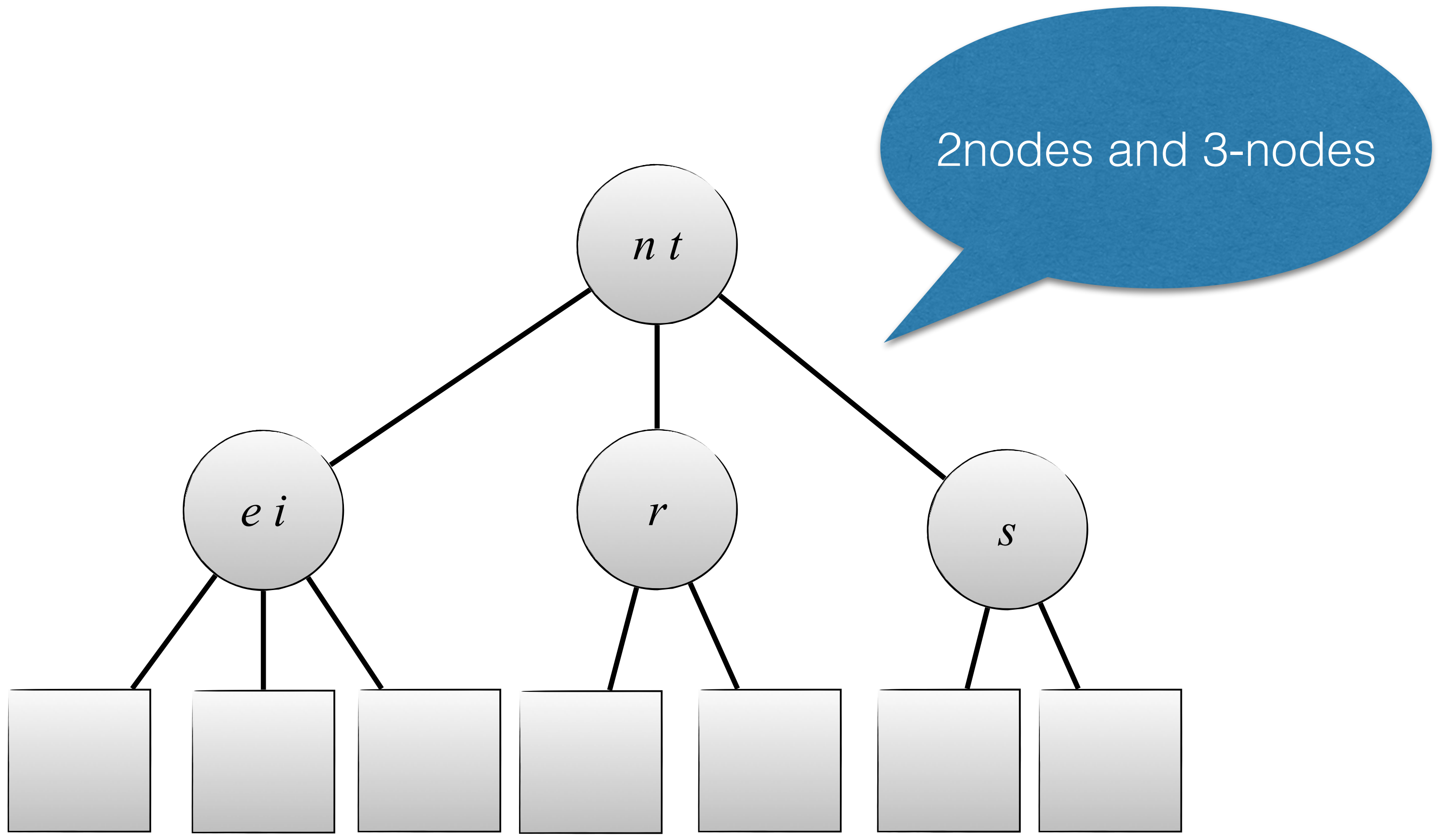
Example of a 2-3 tree



Example of a 2-3 tree



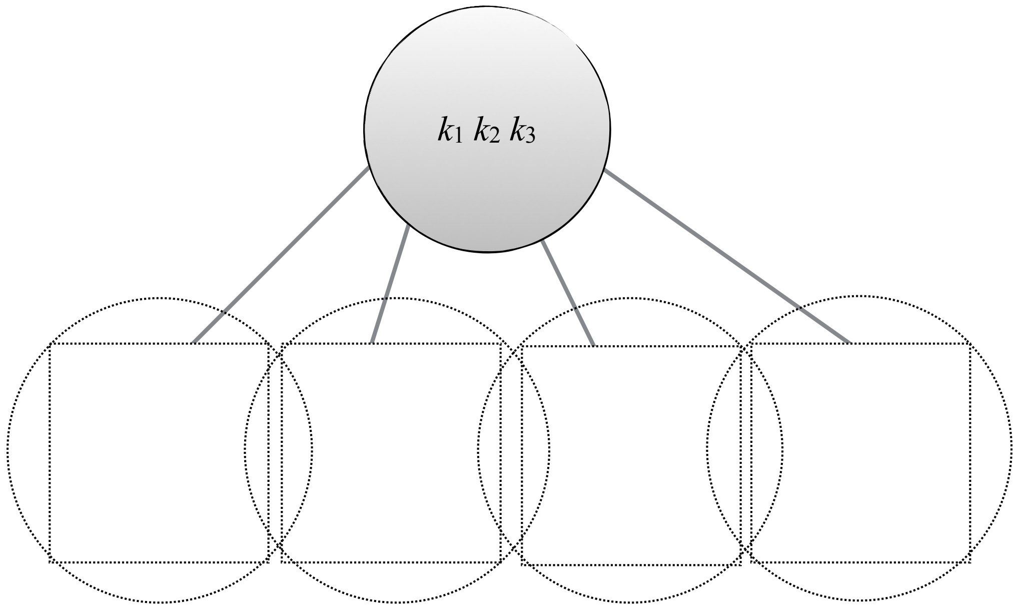
Example of a 2-3 tree



auxiliary nodes: 4-nodes

- On a temporary bases when working with 2-3 trees we will make use of 4-nodes:

- a 4-*node*, with three keys $k_1 \leq k_2 \leq k_3$ (and associated values) and four links: a left link to a 2-3 search tree with keys smaller than k_1, a left middle link to a 2-3 search tree with keys larger than k_1 and smaller than k_2, a right middle link with keys larger than k_2 and smaller than k_3, and a right link to a 2-3 search tree with keys larger than k_3



Supported methods

- Search a key
- Insert an element/key and associated value
- Delete an element/key and associated value

2-3 trees: search

- Generalization of binary search
- **If root node is a 2-node then** compare search key s against root key k
  - If $s = k$ then return element with key k
  - Else if $s < k$ then recurse on left subtree
  - Else if $s > k$ then recurse on right subtree

2-3 tree: search (continued)

- **If root node is 3-node then** compare search key s with 3-node keys k_1 and k_2
  - If $s = k_1$ then return element with key k_1
  - If $s = k_2$ then return element with key k_2
  - If $s < k_1$ then recurse on left subtree
  - If $s > k_2$ then recurse on right subtree
  - Else recurse on middle subtree

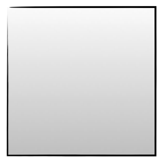
2-3 tree: search (continued)

- **If root is empty/leaf then** search key is not contained in the 2-3 tree

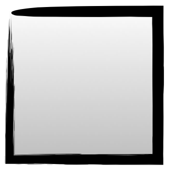
2-3 trees: insertion of an element with key k

- We only insert if key k is not yet in the tree
- **Case 1.** We search for key k. If the tree is empty, the root node is replaced by a 2-node with key k
- **Otherwise**, the search terminates in a parent of two leaves
- We distinguish two cases
  - **Case 2.** The search terminates in a 2-node
  - **Case 3.** The search terminates in a 3-node

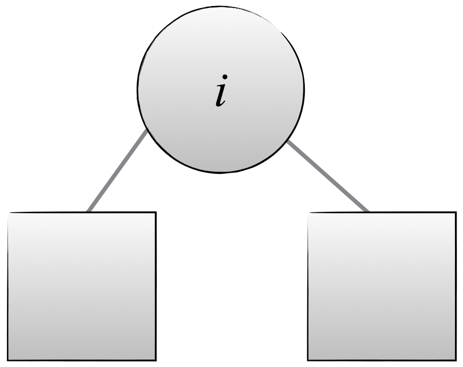
Case 1. Inserting key i into an empty tree



Case 1. Inserting key i into an empty tree



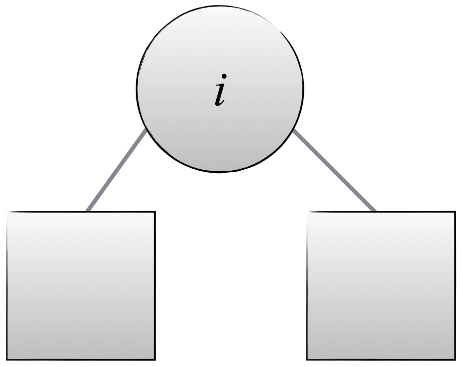
Case 1. Inserting key i into an empty tree



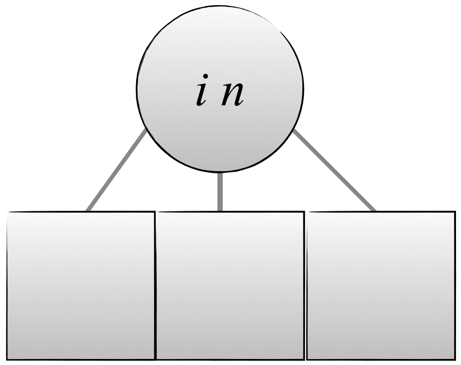
Case 2. Search terminates in a 2-node

- Replace the 2-node with a 3-node containing both its original key and the new key to be inserted
- Note: The tree remains perfectly balanced and satisfies the search tree properties

Case 2. Inserting key n



Case 2. Inserting key n



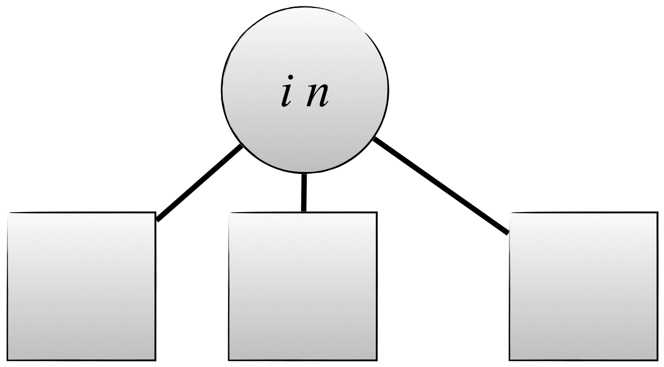
Case 3. Search terminates in a 3-node

- We distinguish the following cases
  - **Case 3.1** The node is root (and is parent of leaves only)
  - **Case 3.2** The node's parent is a 2-node
  - **Case 3.3** The node's parent is a 3-node
  - These are all cases since the search tree is perfectly balanced.

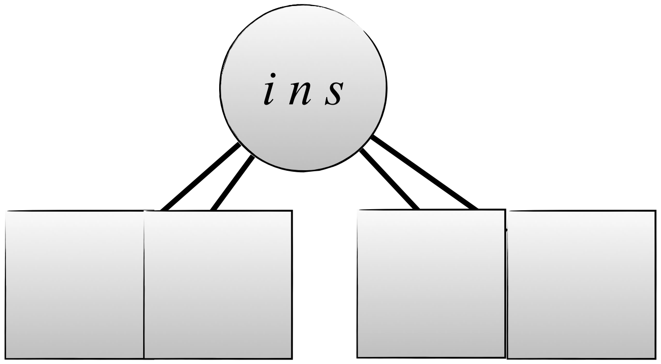
Case 3.1 Termination 3-node is root

- Temporarily replace 3-node by 4-node with original keys and inserted new key
- Convert this tree rooted by the 4-node into a 2-3 tree consisting of three 2-nodes as follows:
    - The new root contains key k_2.
    - The left child of the root contains key k_1
    - The right child of the root contains key k_3
    - The children of the 2-nodes containing k_1 and k_3 are all leaves.

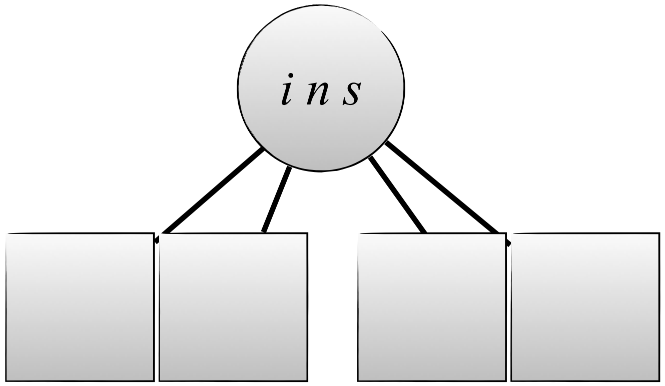
Case 3.1. Insert key s



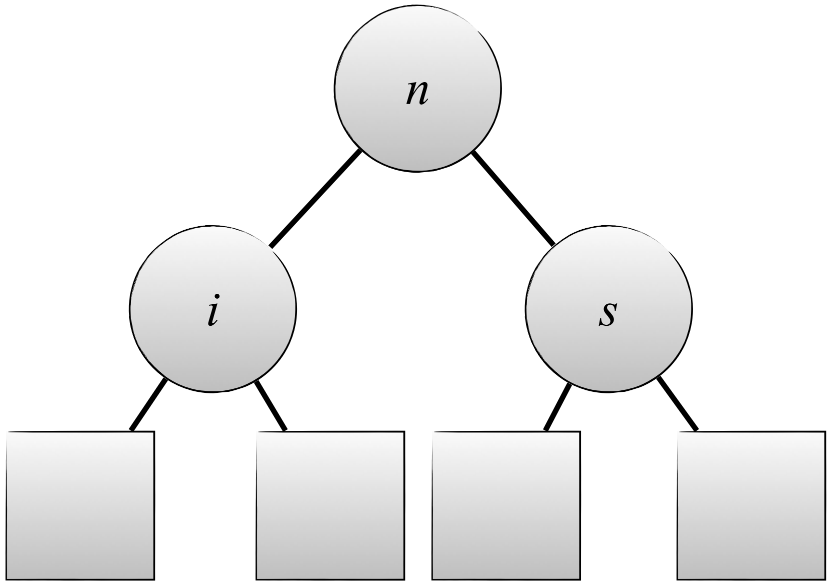
Case 3.1. Insert key *s*



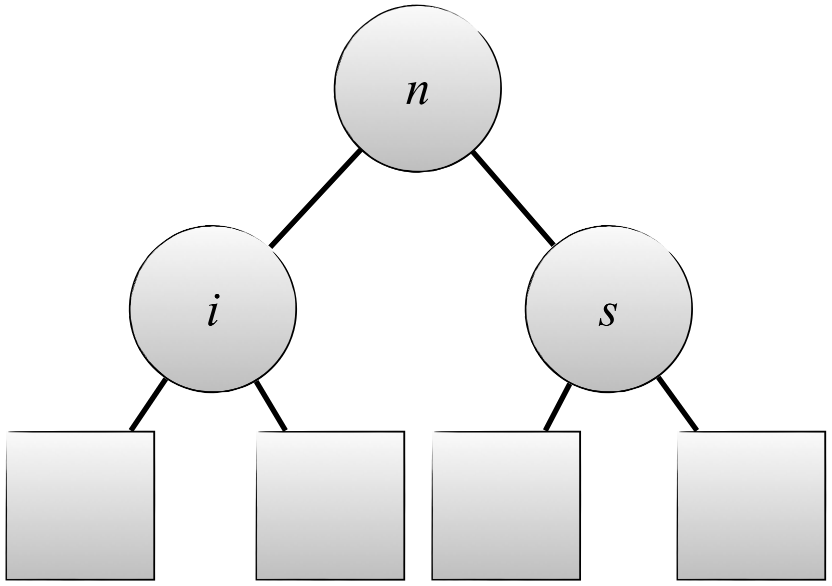
Case 3.1. Insert key s



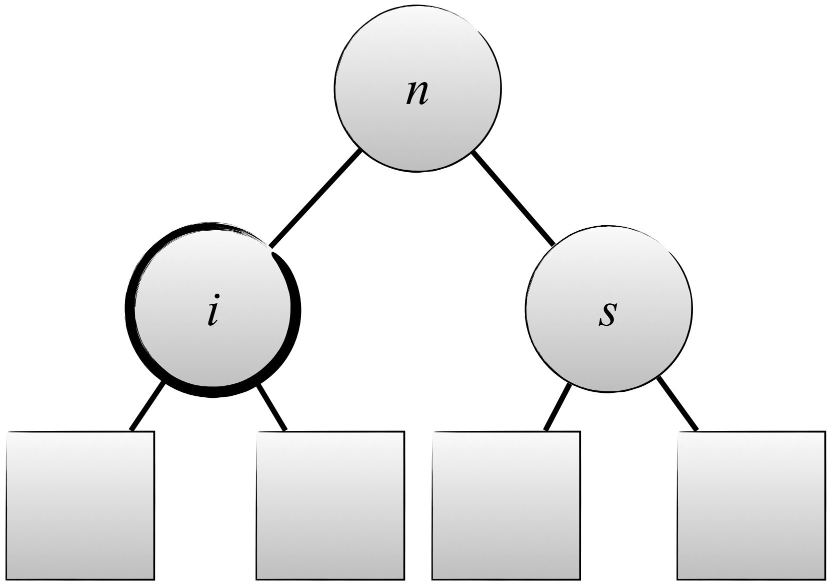
Case 3.1. Insert key *s*



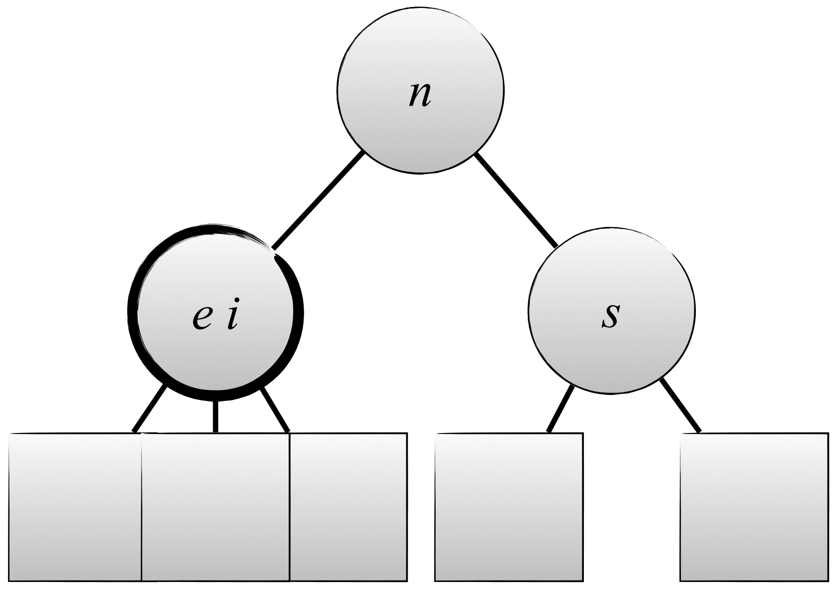
Case 2. Insert key e



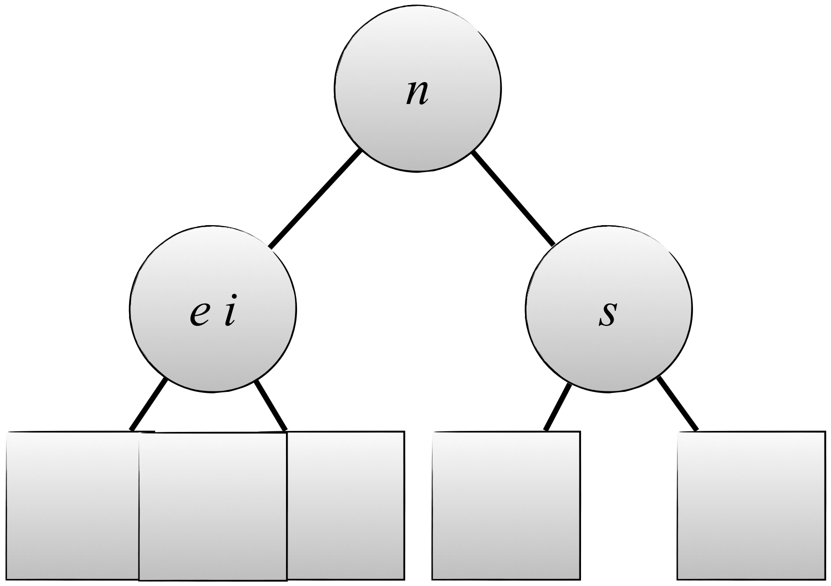
Case 2. Insert key e



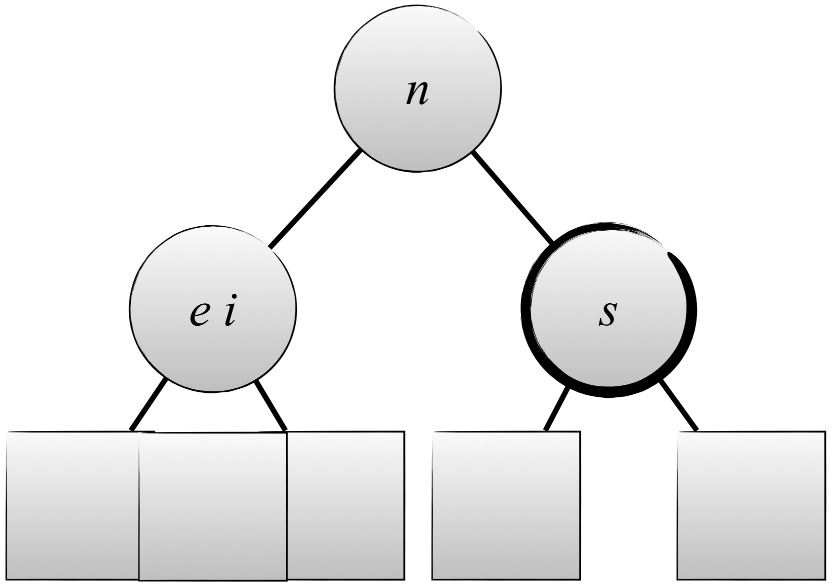
Case 2. Insert key e



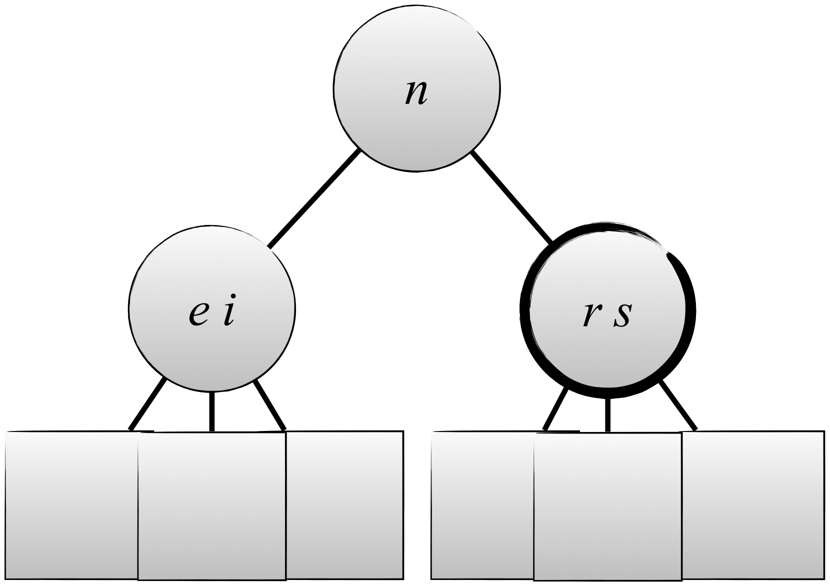
Case 2. Insert key *r*



Case 2. Insert key r



Case 2. Insert key *r*

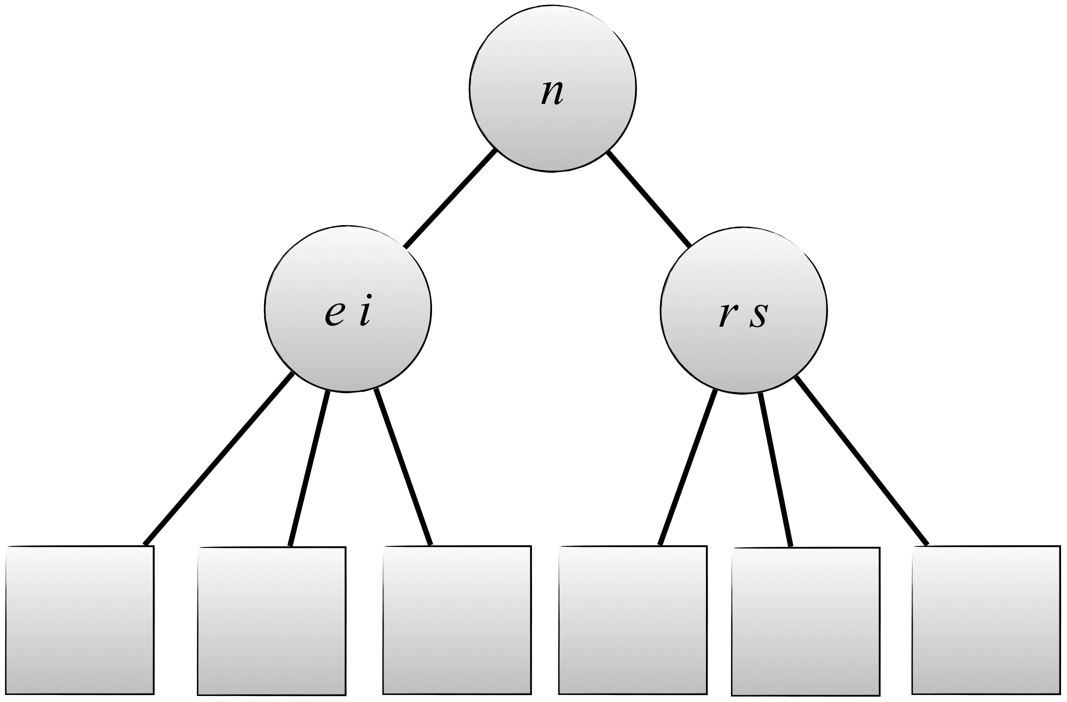


Case 3.2. The parent of the termination 3-node (x) is a 2-node (y)

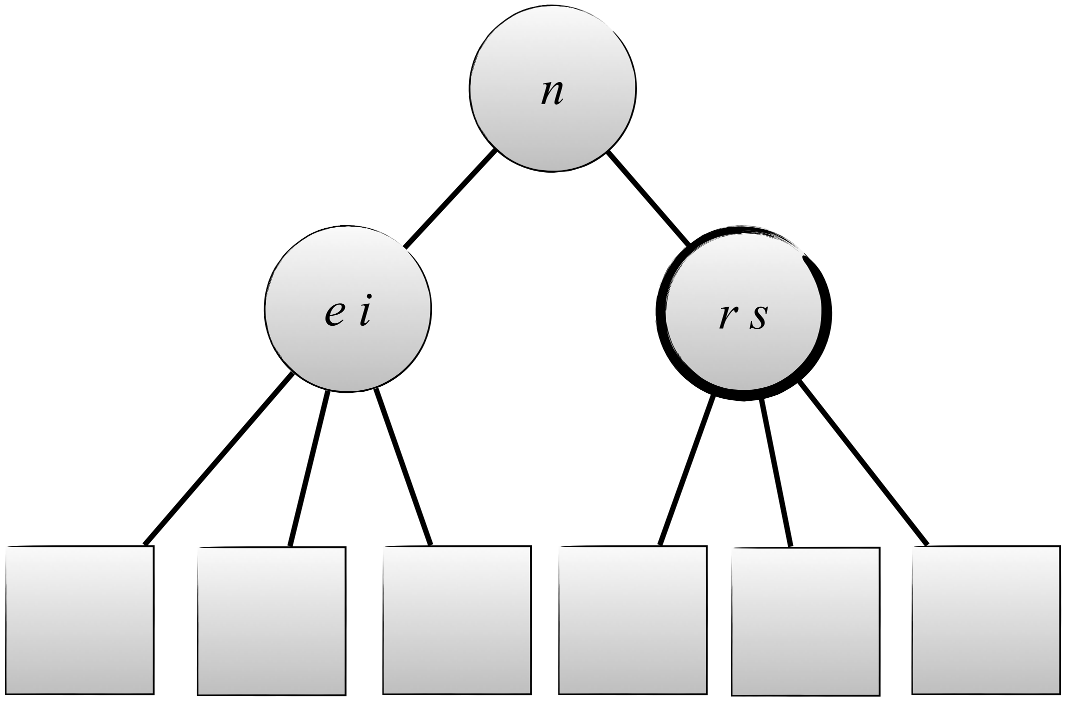
- We replace the 3-node x (temporarily) by a 4-node that contains the original keys of the 3-node and the new key to be inserted.

- Then, the middle key, k_2, is removed from 4-node x and inserted into the parent 2-node y (making it into a 3 node), and splitting 4-node x with its two remaining keys, k_1 and k_2, into two 2-nodes with parent y.

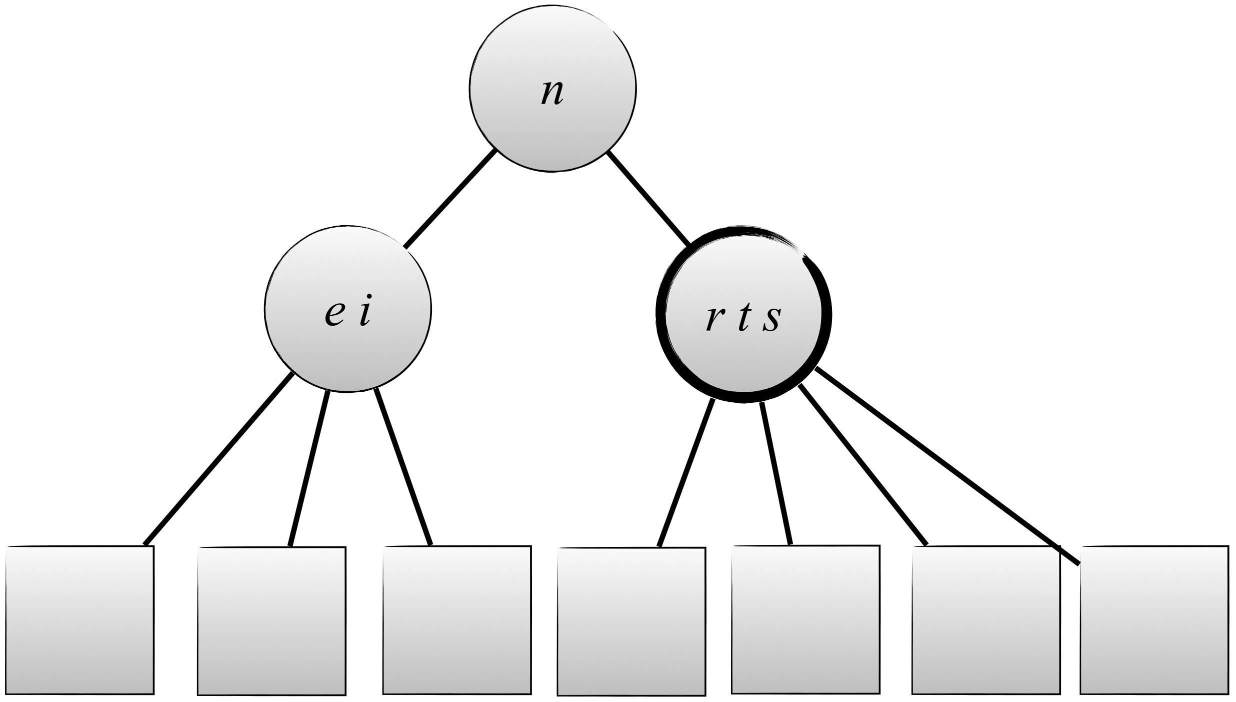
Case 3.2. Insert key t



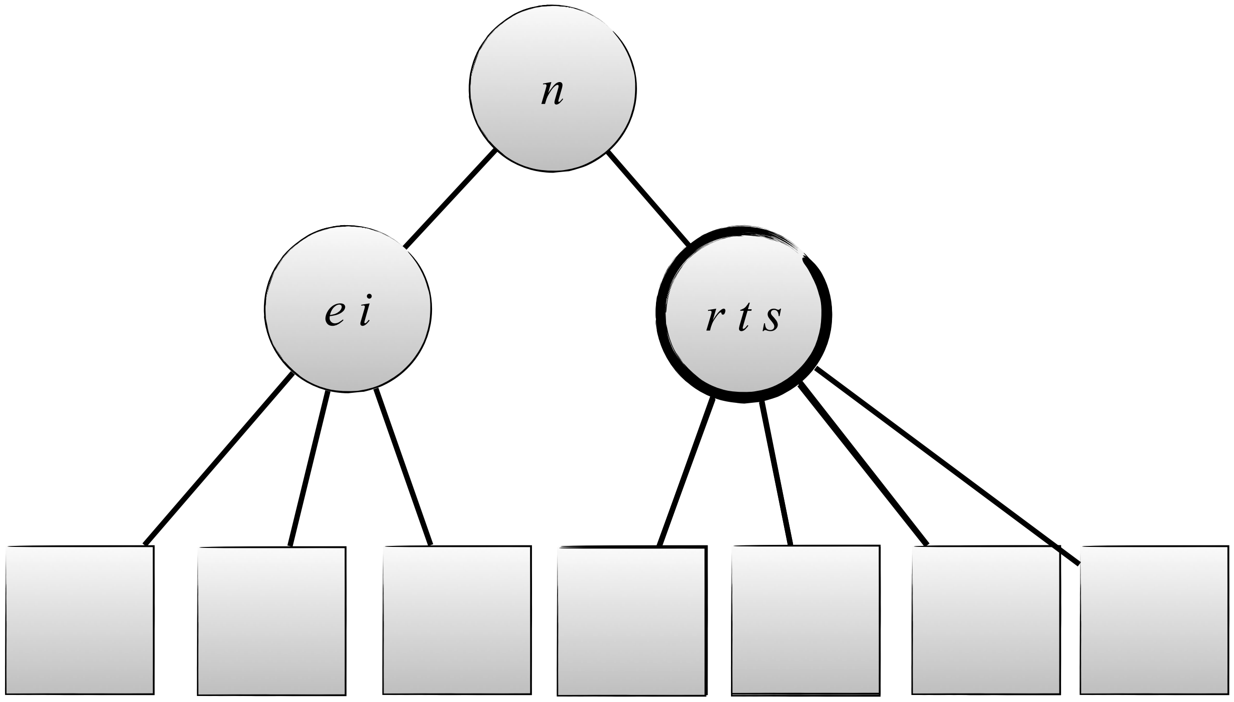
Case 3.2. Insert key *t*



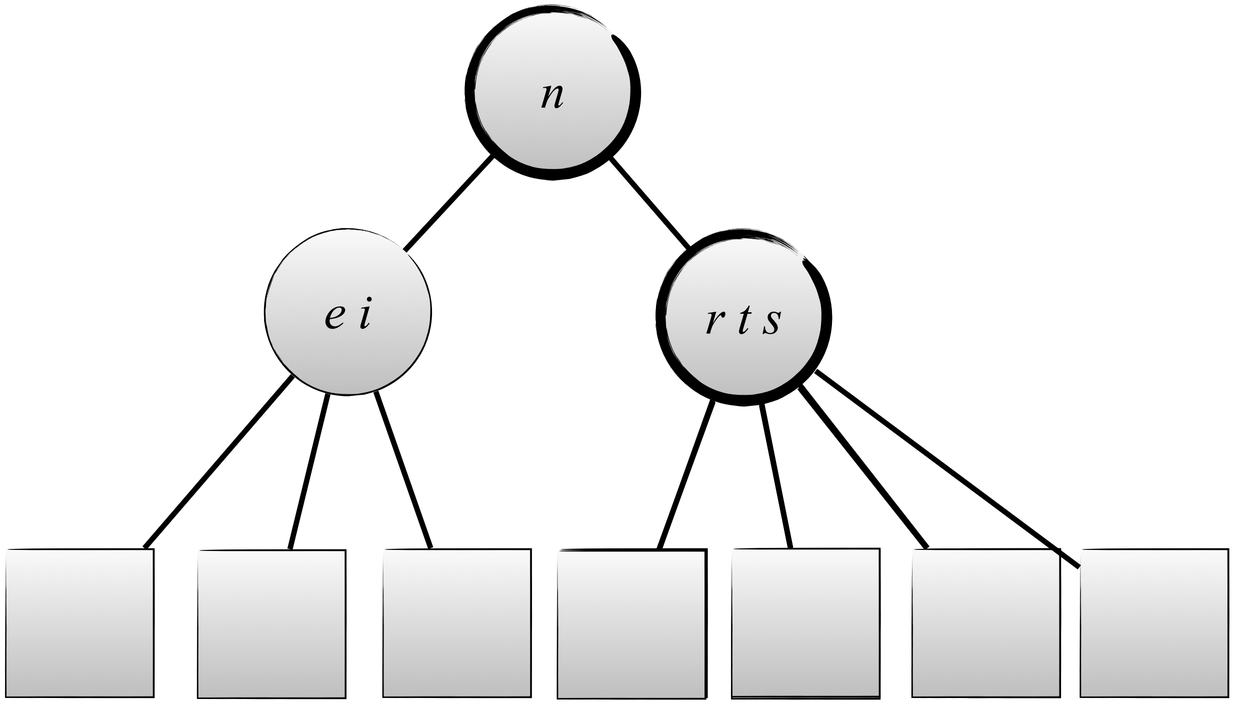
Case 3.2. Insert key t



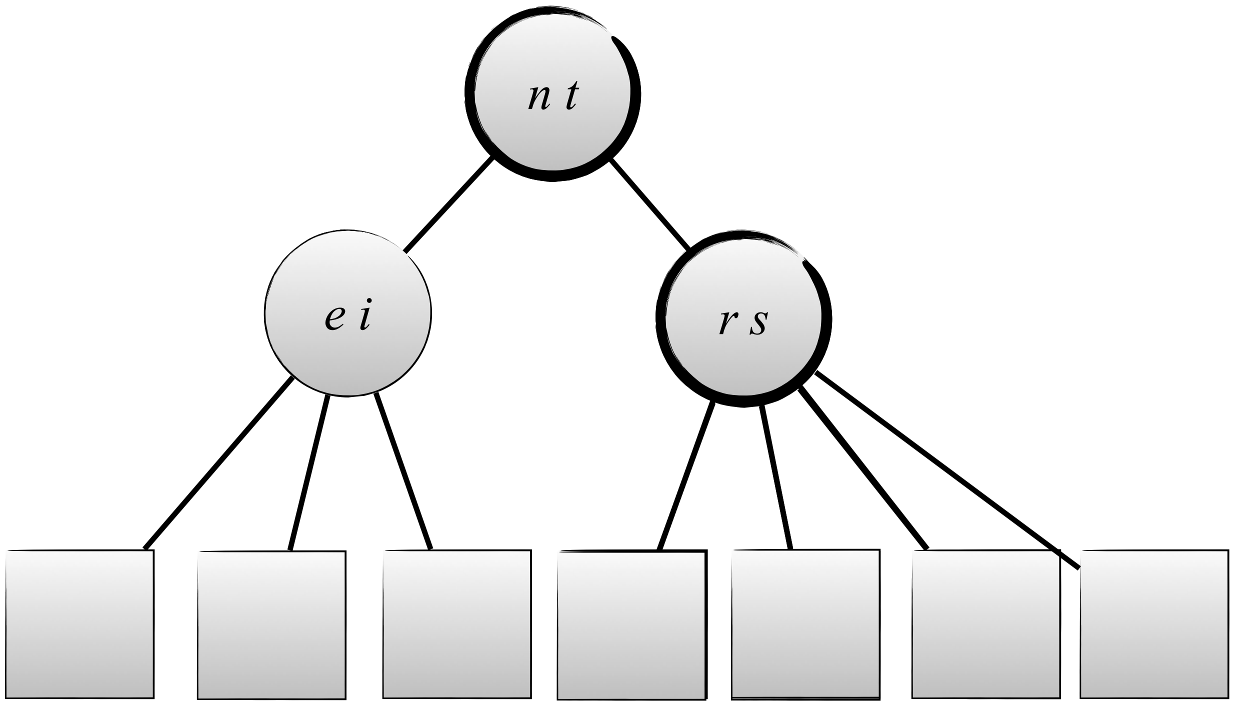
Case 3.2. Insert key t



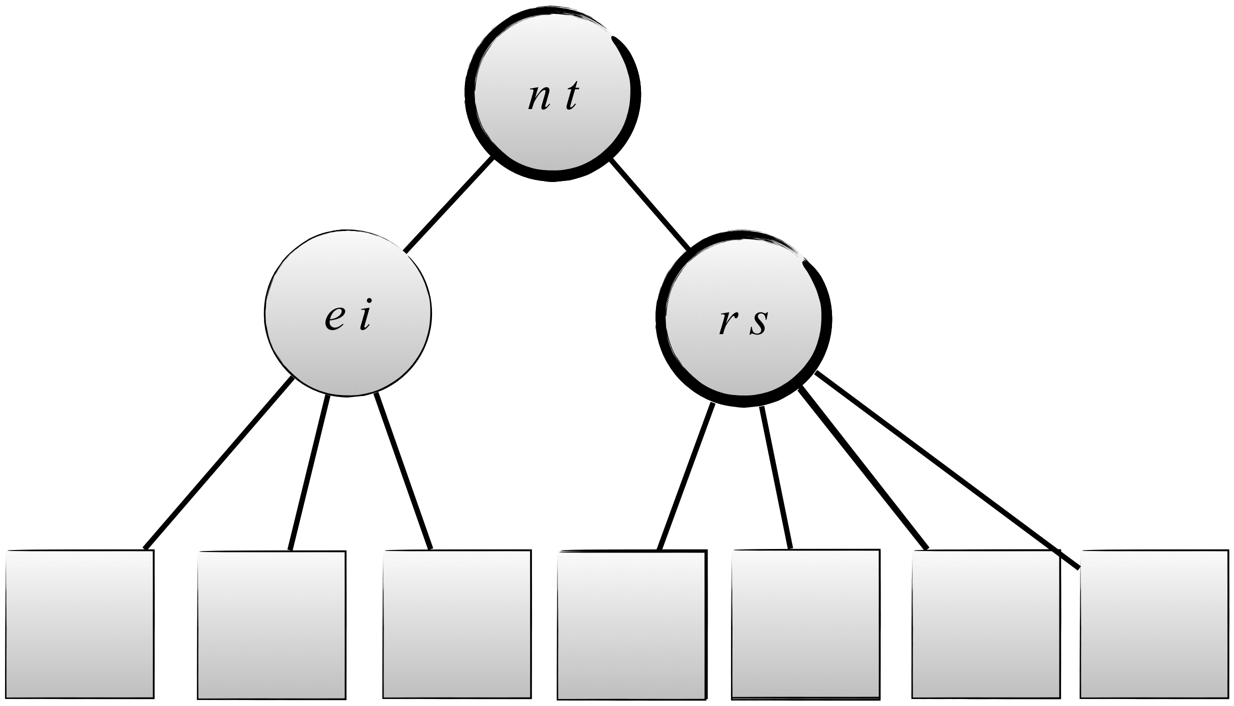
Case 3.2.Insert key t



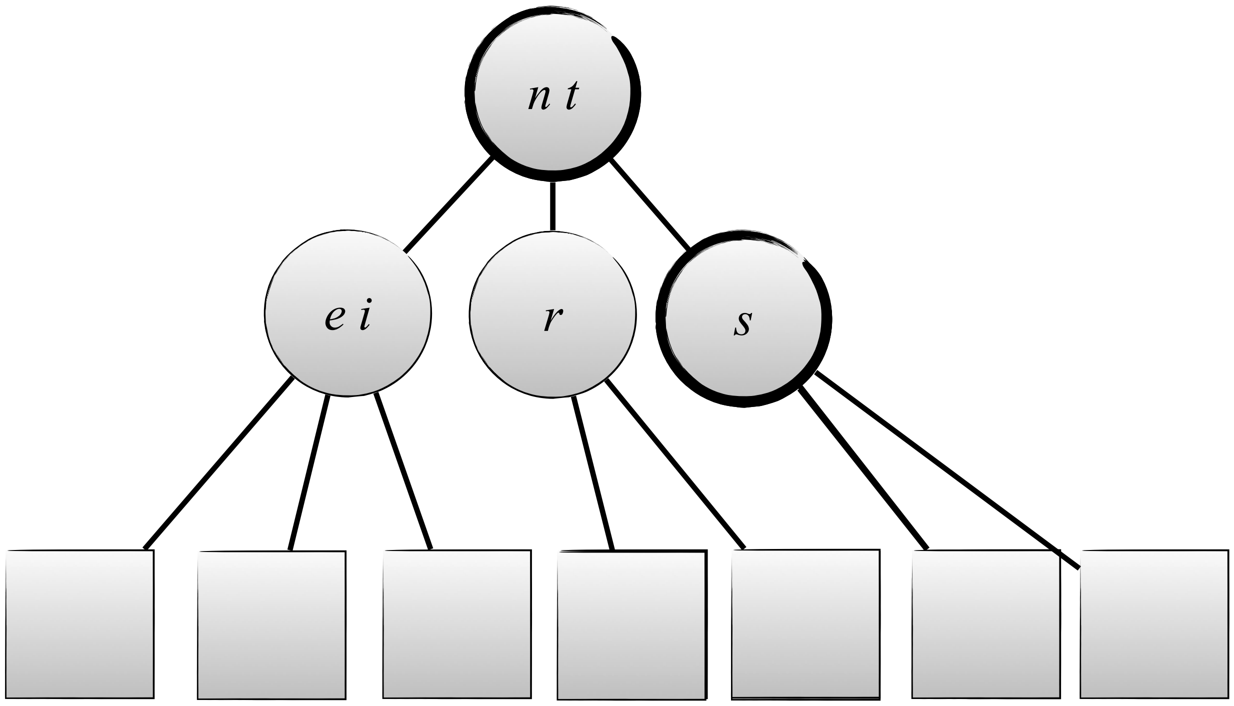
Case 3.2. Insert key t



Case 3.2. Insert key t



Case 3.2. Insert key t



Case 3.3. The parent of the termination 3-node is a 3-node

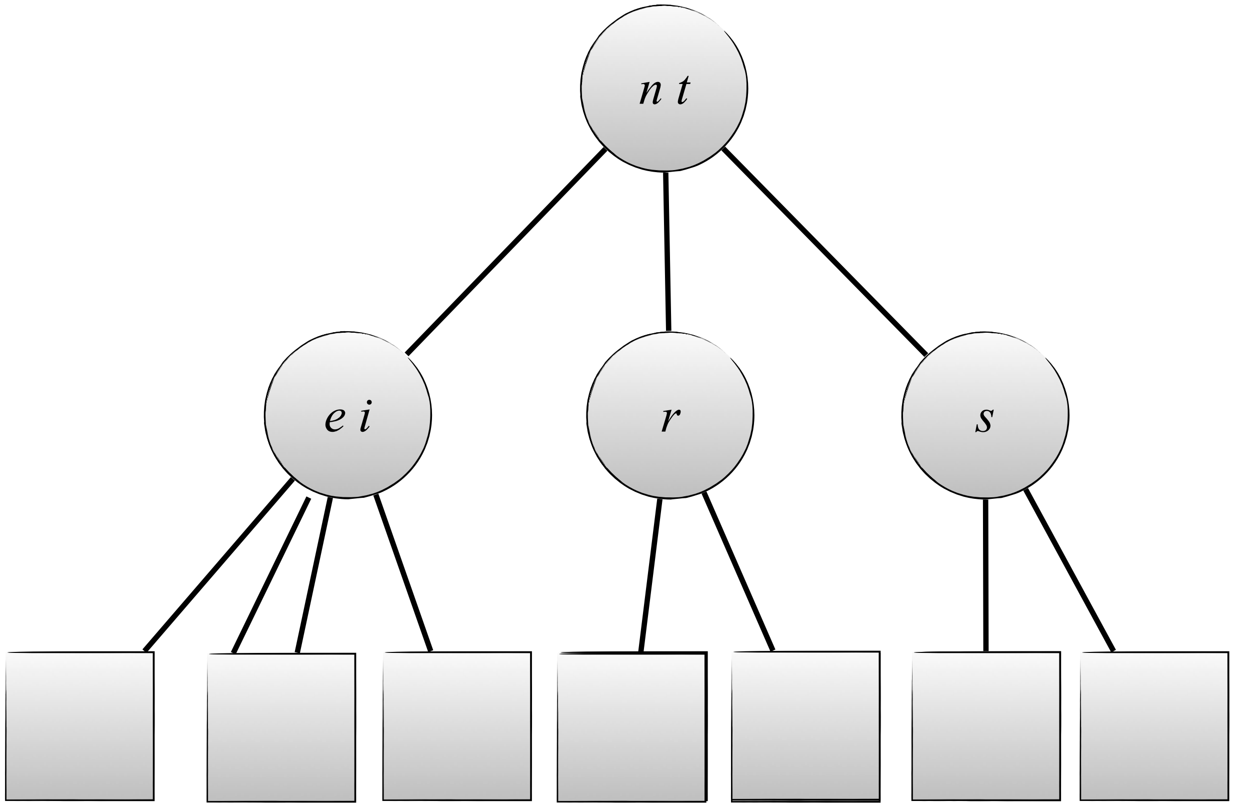
- We replace the termination 3-node x (temporarily) by a 4-node that contains the original keys of the 3-node and the new key to be inserted.

- We then move the middle key up (converting the node back to a 3-node) and insert it into the parent, creating again a temporary 4-node.

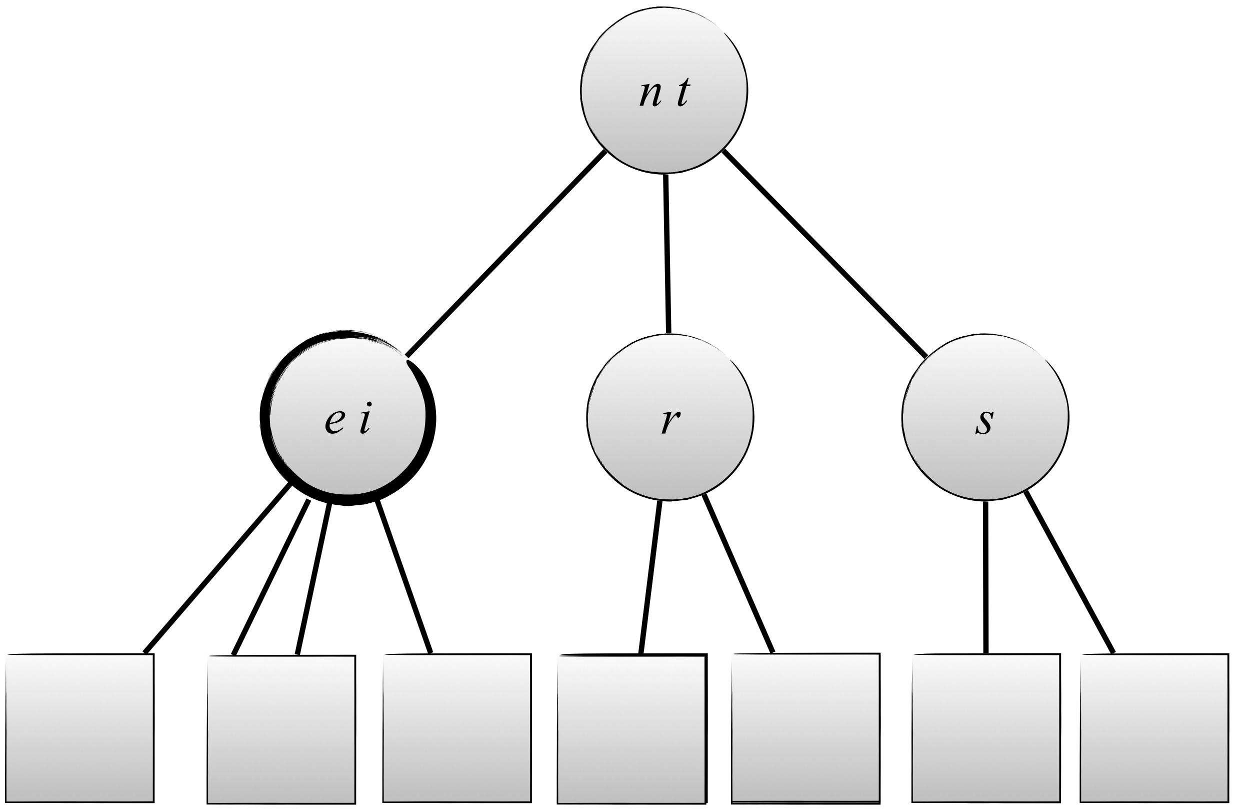
- This 4-node is either the root, has a 2-node as parent or has a 3-node as parent.

- In the first case, we continue as on the previous slide. In the second case, we continue to move the middle key up the tree as above. The last case is discussed on the next slide (splitting the root).

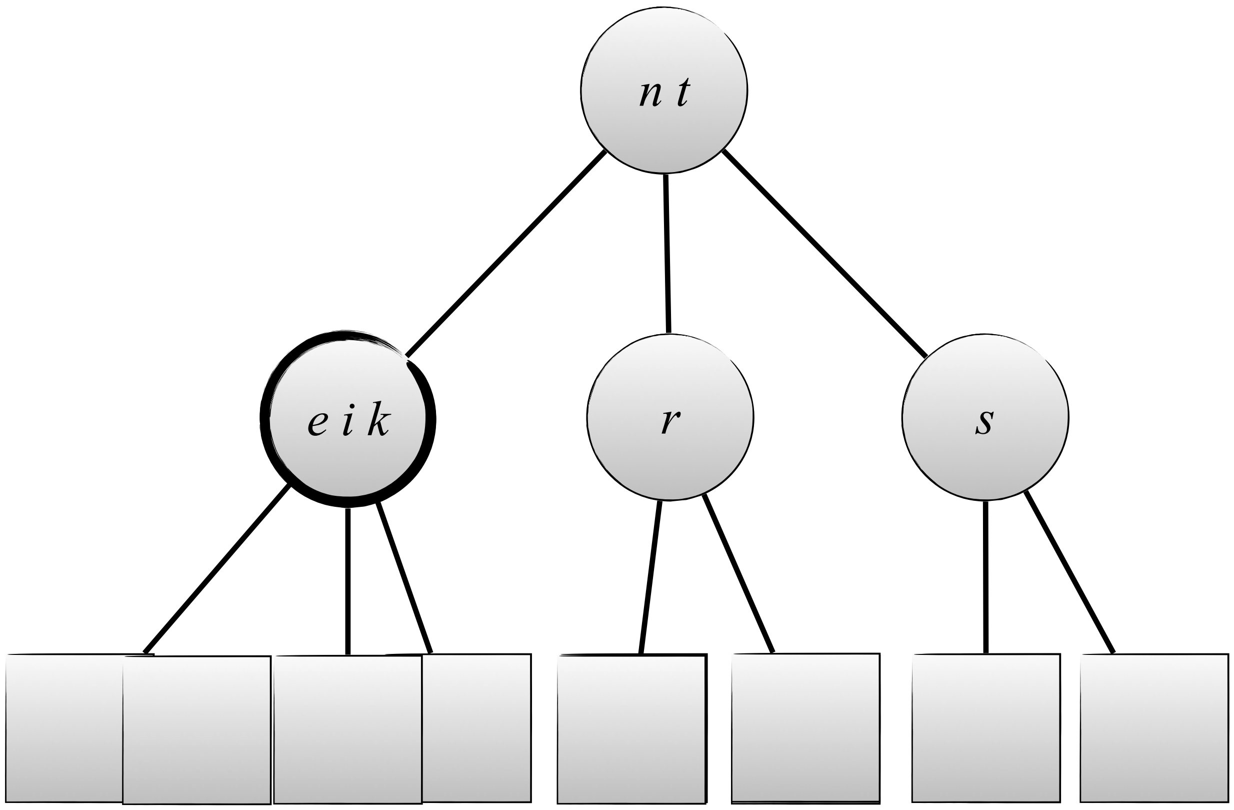
Case 3.3. Insert key k



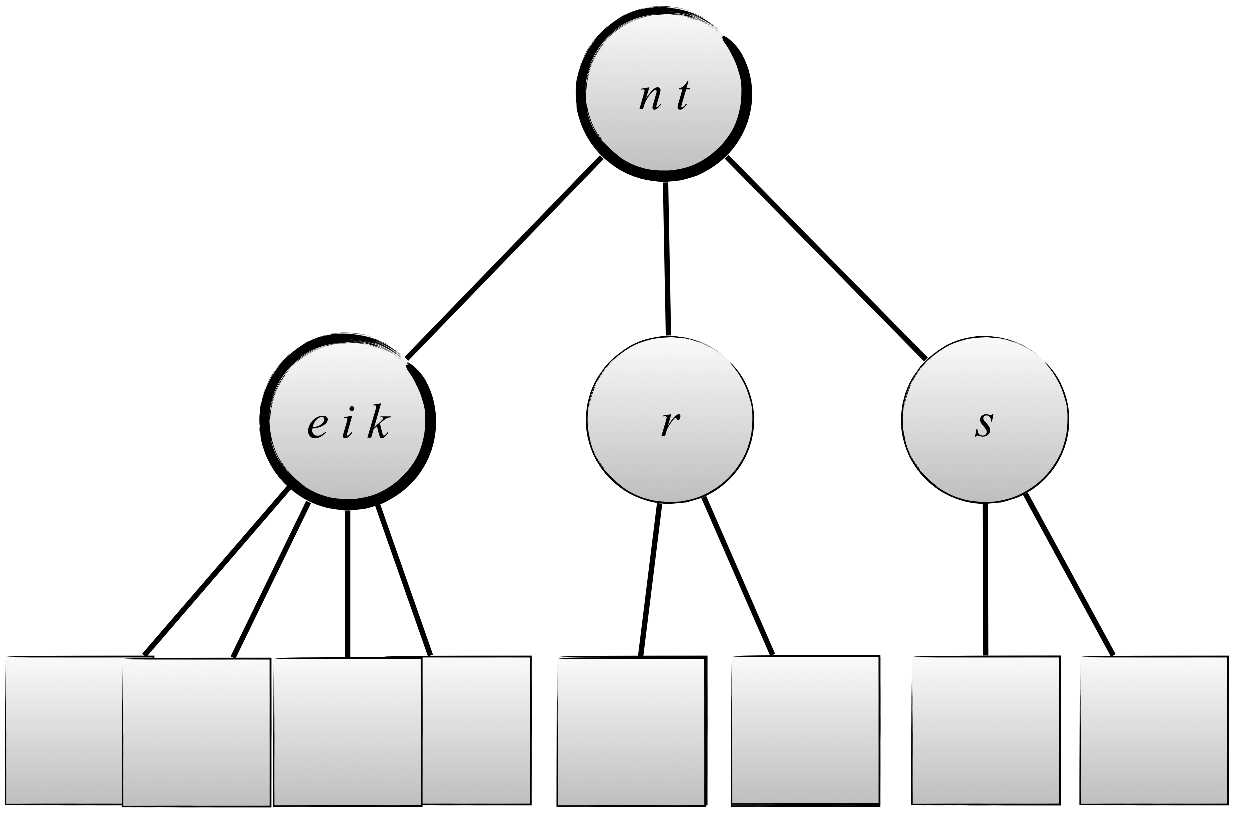
Case 3.3. Insert key k



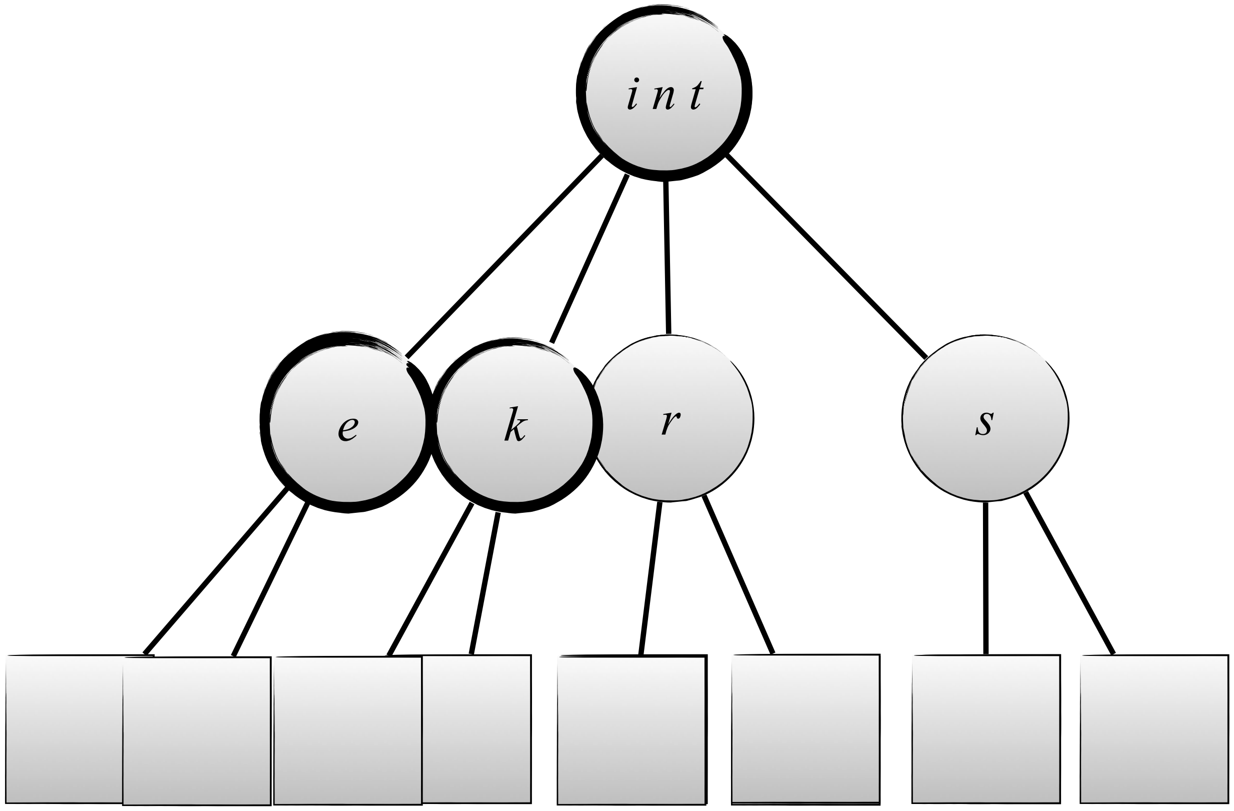
Case 3.3. Insert key k



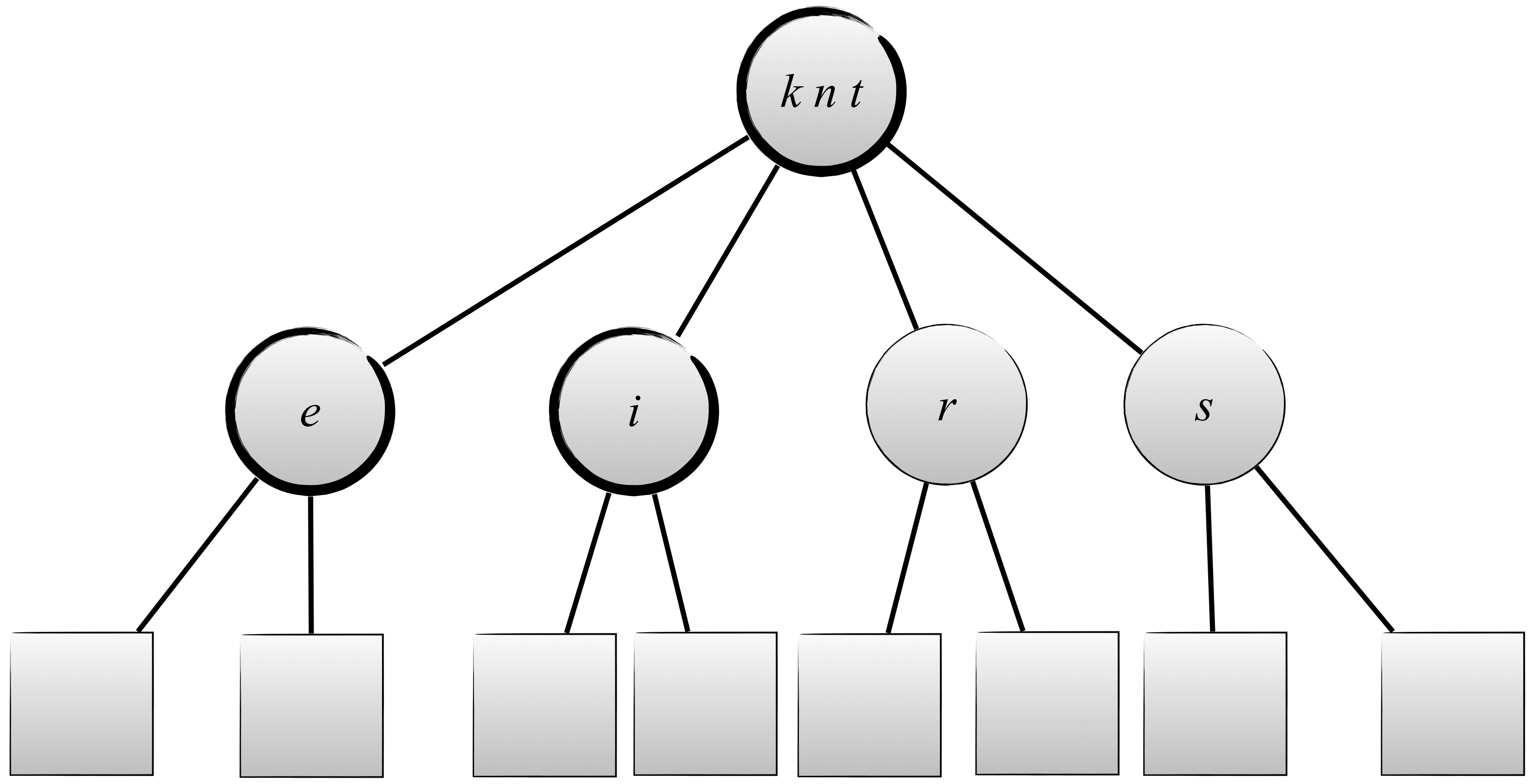
Case 3.3. Insert key k



Case 3.3. Insert key k



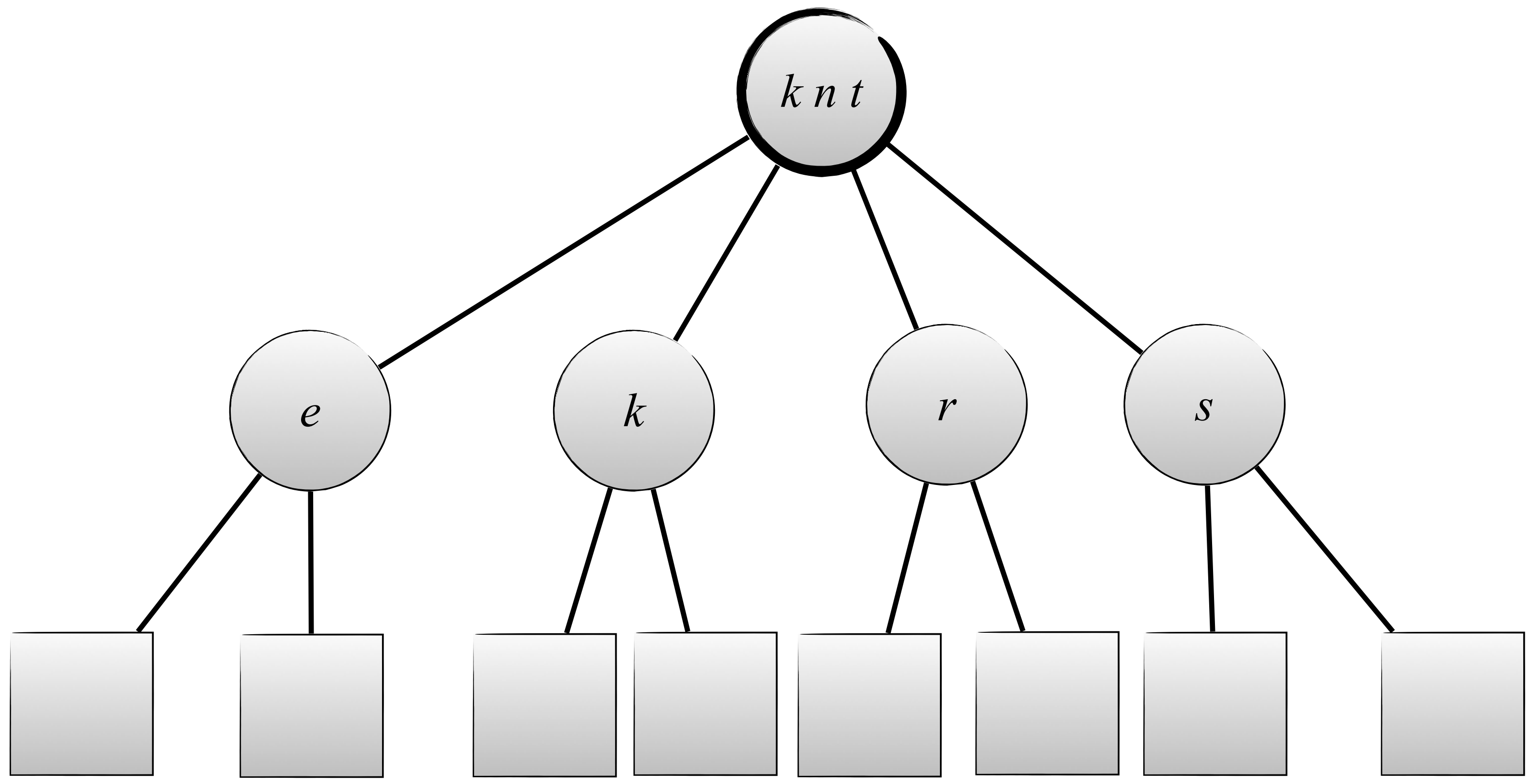
Case 3.3. Insert key k



Splitting the root

- Split the root into three (temporary) 2-nodes (this increases the height of the tree by one), leaving the tree perfectly balanced
  - k_2 is the root key
  - k_1 the key of the root's left child; its two children are the two leftmost children of the 4-node
  - k_3 the key of the root's right child; its two children are the two rightmost children of the 4-node

Insert key k



Insert key k