מבני נתונים

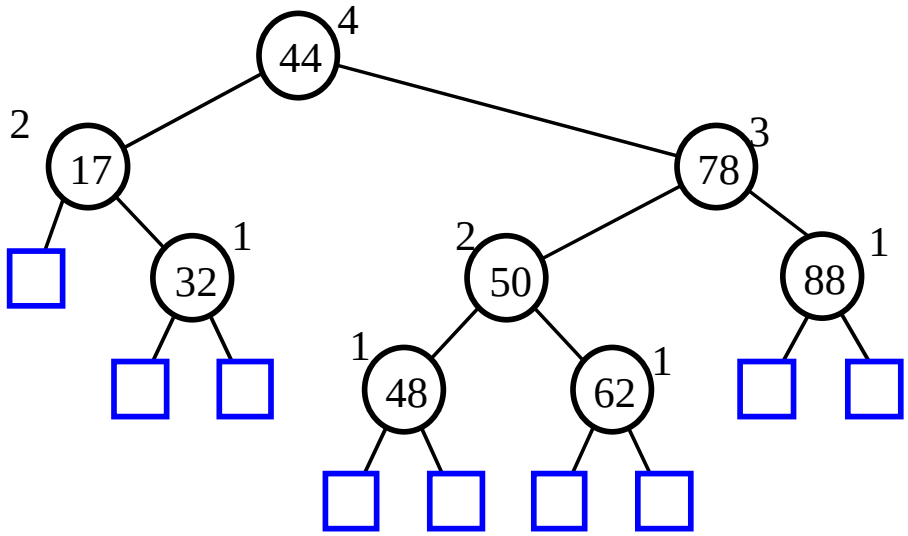
תרגול 3

ליאור שפירא

עצי AVL

G.M. Adelson-Velskii / E.M.Landis 1962

- עץ חיפוש בינארי מאוזן (מבחינת גובה)
- עבור כל צומת פנימית v של עץ T
  - גובה תתי העצים מתחת ל-v יכול להיות שונה בעד 1



גובה של עץ AVL

- למה: גובה של עץ T המחזיק n מפתחות הוא $O(\log n)$
- הוכחה:
  - נמצא את $n(h)$, מס' מינימלי של צמתים פנימיים בעץ AVL בעל גובה h
  - בבירור $n(1)=1$, $n(2)=2$
  - עבור $n\geq 3$, עץ AVL בעל גובה h מכיל את השורש, תת עץ בגובה $h-1$ ותת עץ בגובה של לפחות $h-2$
  - ז"א: $n(h)\geq 1+n(h-1)+n(h-2)$

גובה של עץ AVL

$$n(h-1) > n(h-2) \Rightarrow n(h) \geq 2n(h-2)$$

$$n(h) \geq 2n(h-2)$$

$$n(h) \geq 4n(h-4)$$

$$\ldots$$

$$n(h) \geq 2^i n(h-2i)$$

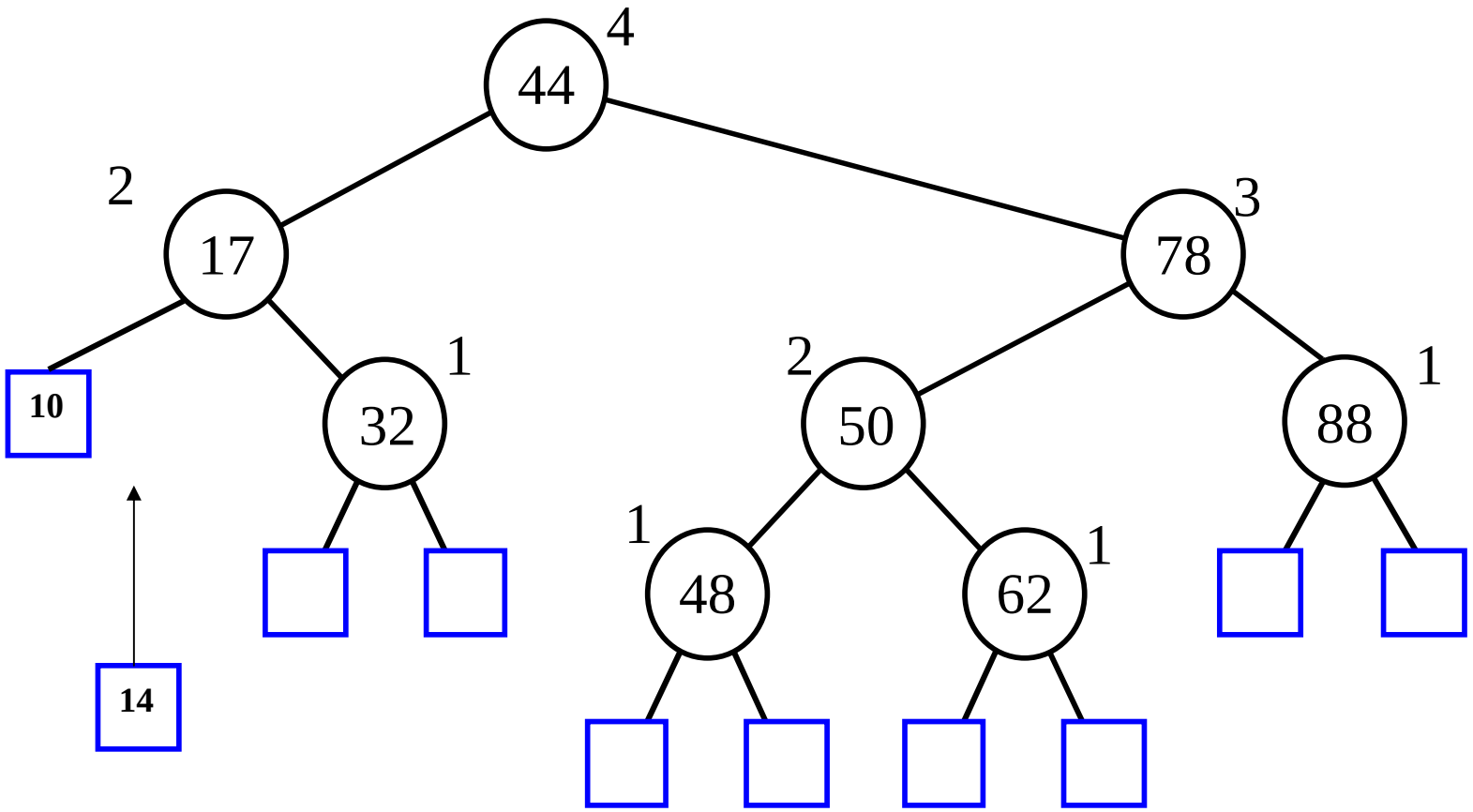
סדרה

$$\Rightarrow n(h) \geq 2^{h/2-1}$$ פתרון הסדרה

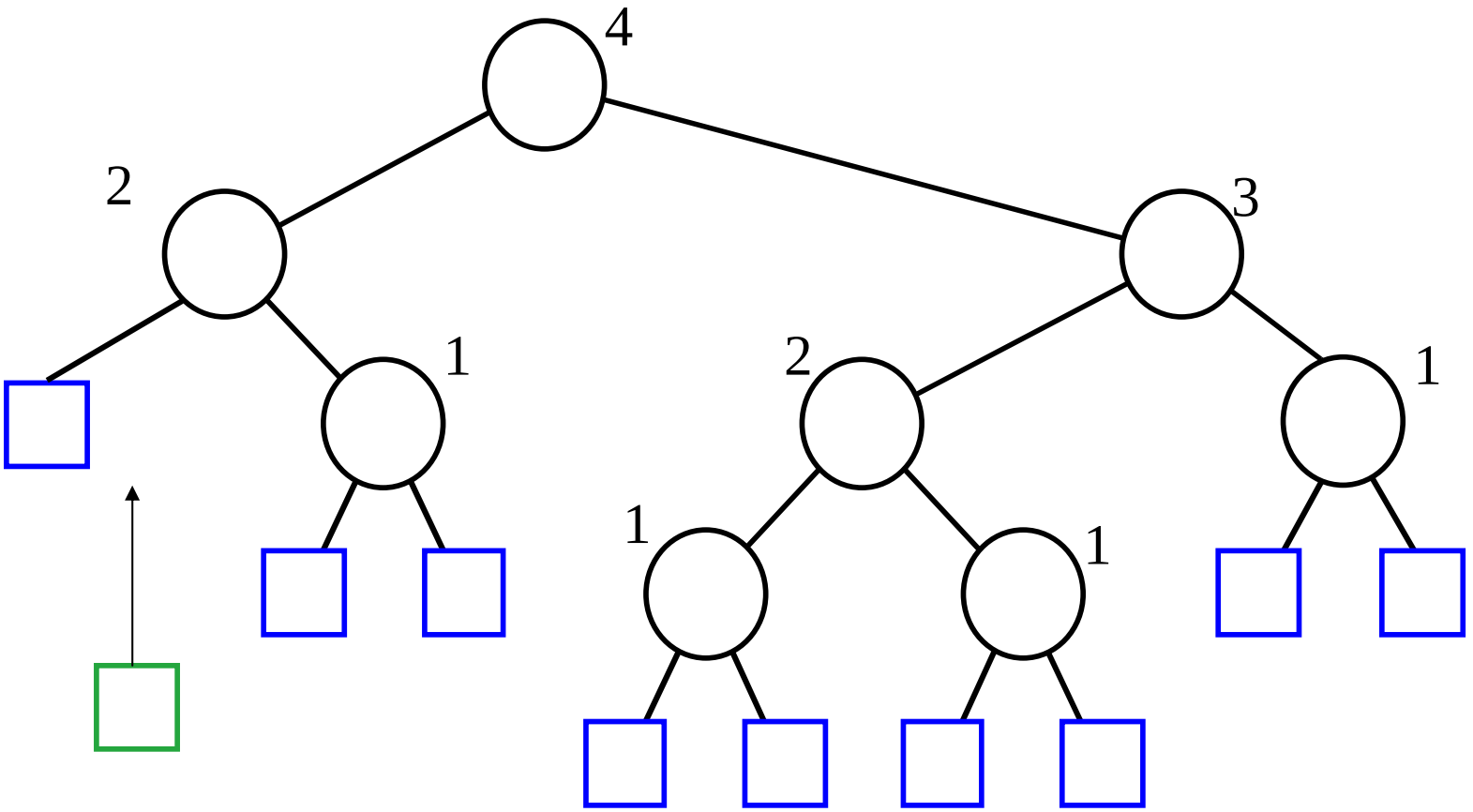
$$\Rightarrow h \leq 2\log n(h) + 2$$ לוג לשני הצדדים

$$\Rightarrow h = O(\log n)$$ מש"ל

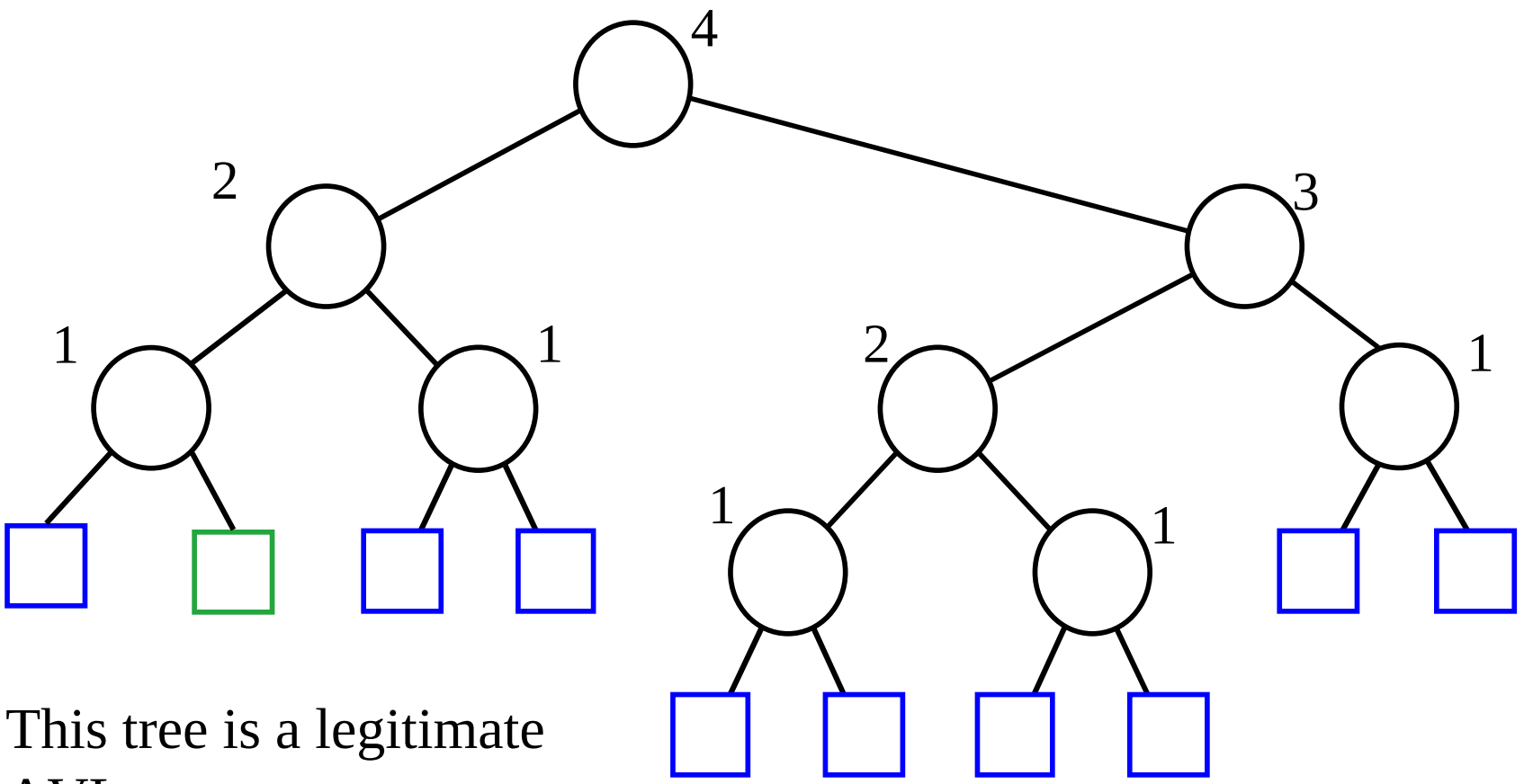
Insertion



נתעלם מהערכים, נתמקד באיזון העץ

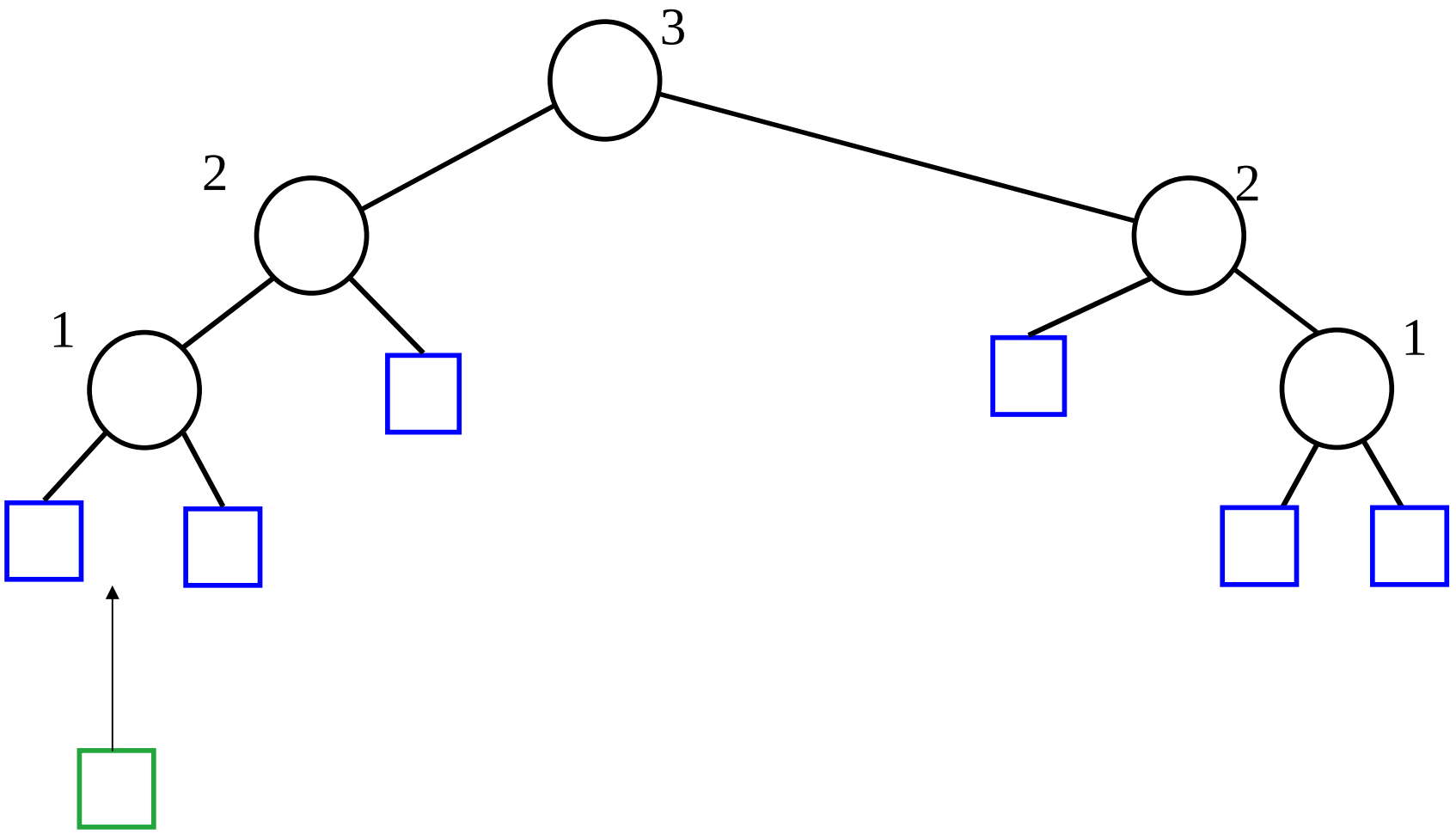


צומת פנימית חדשה

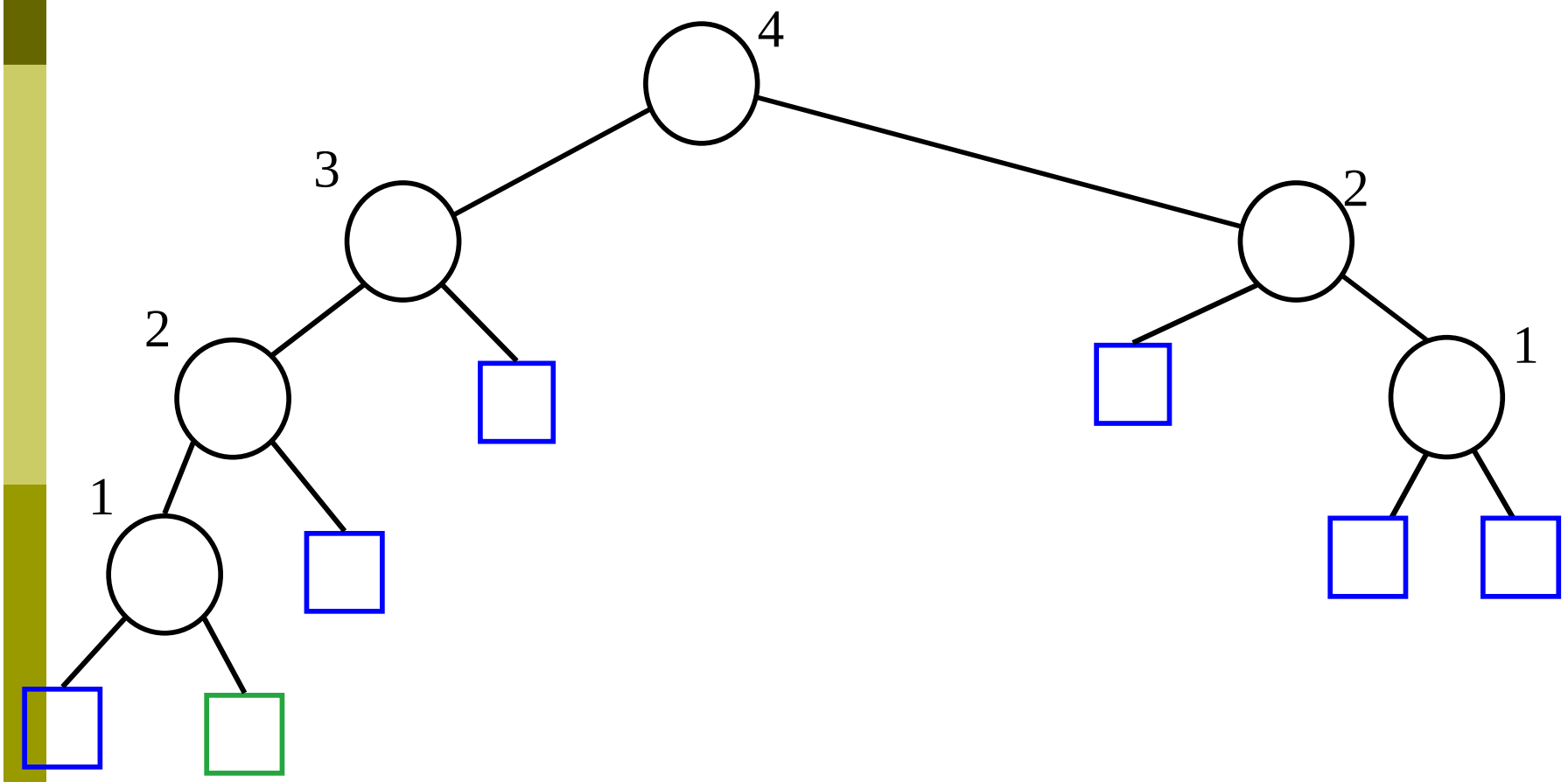


This tree is a legitimate AVL tree…

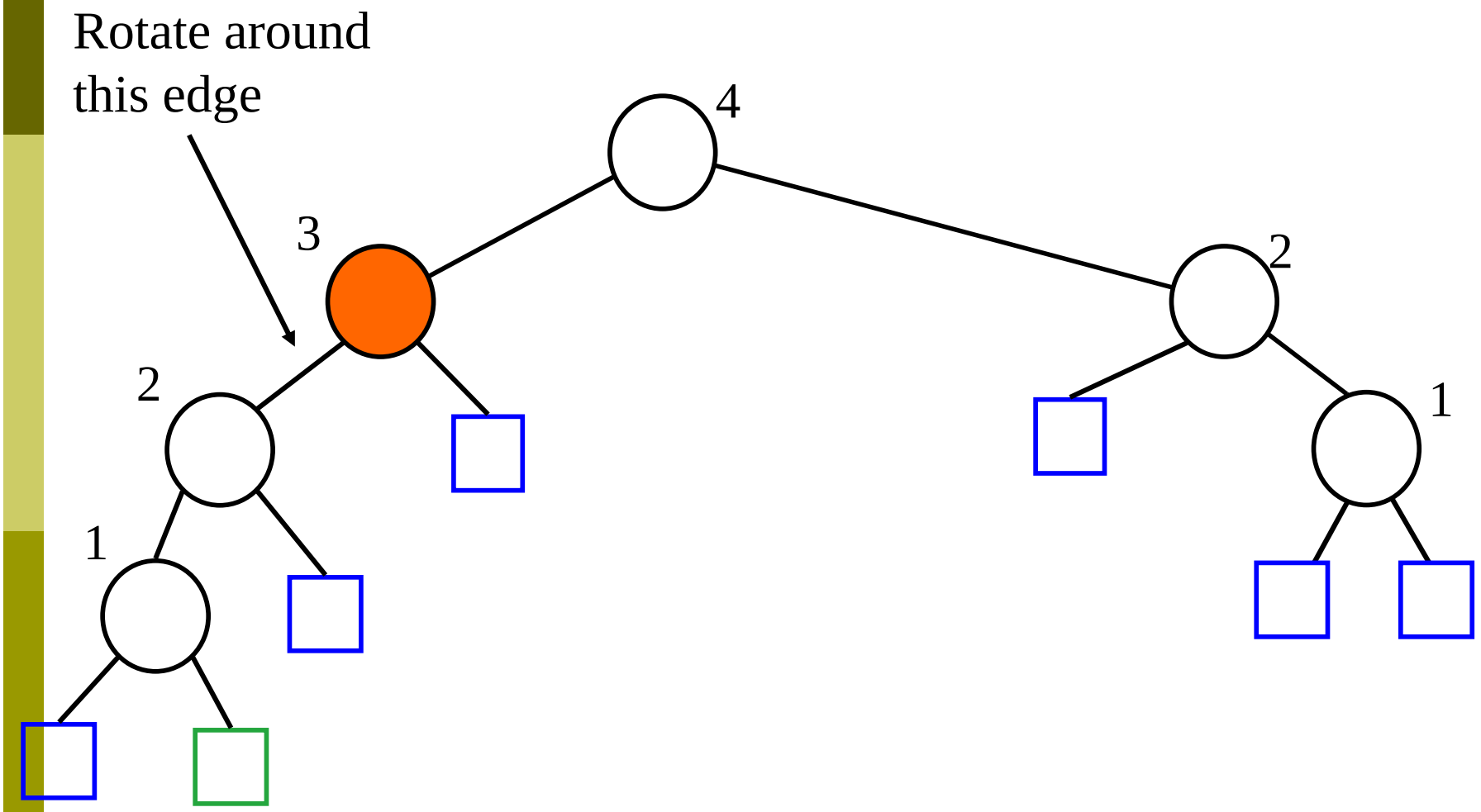
עוד דוגמה



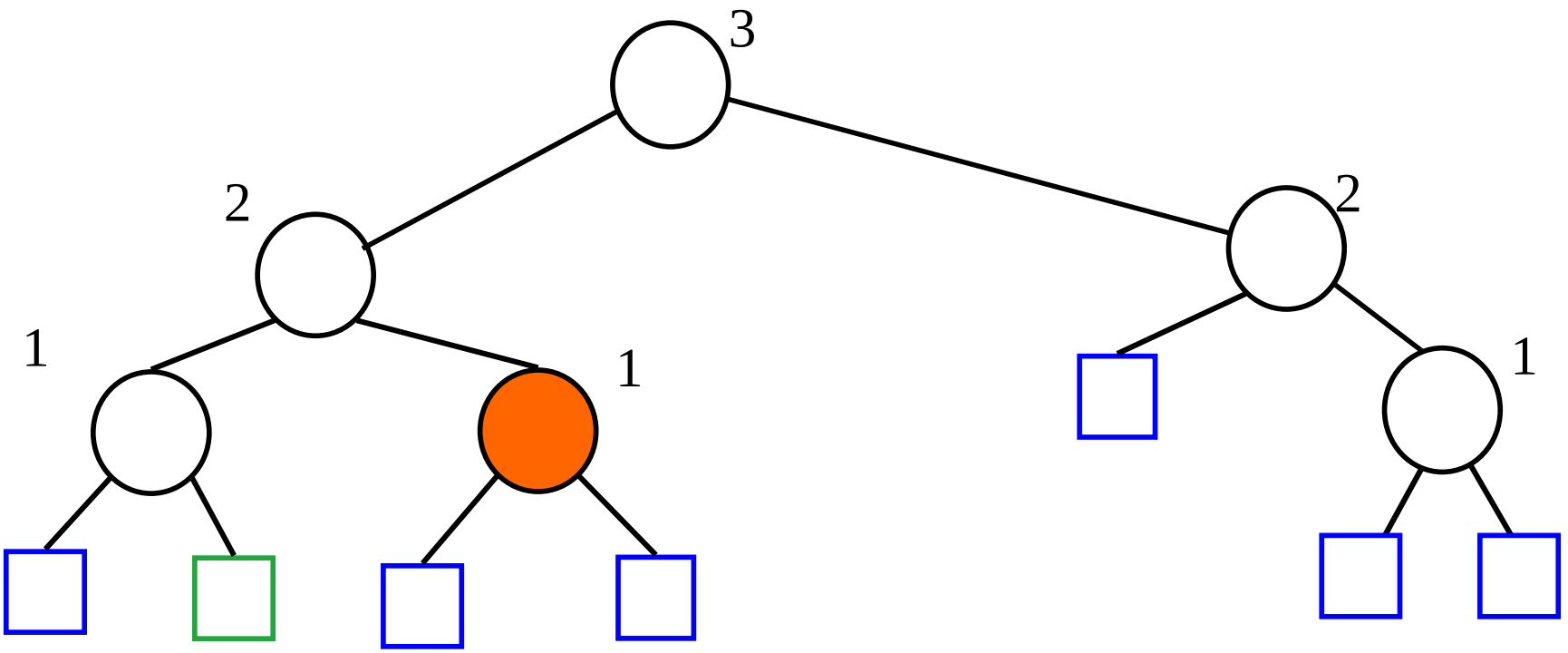
נוסיף את הצומת החדש



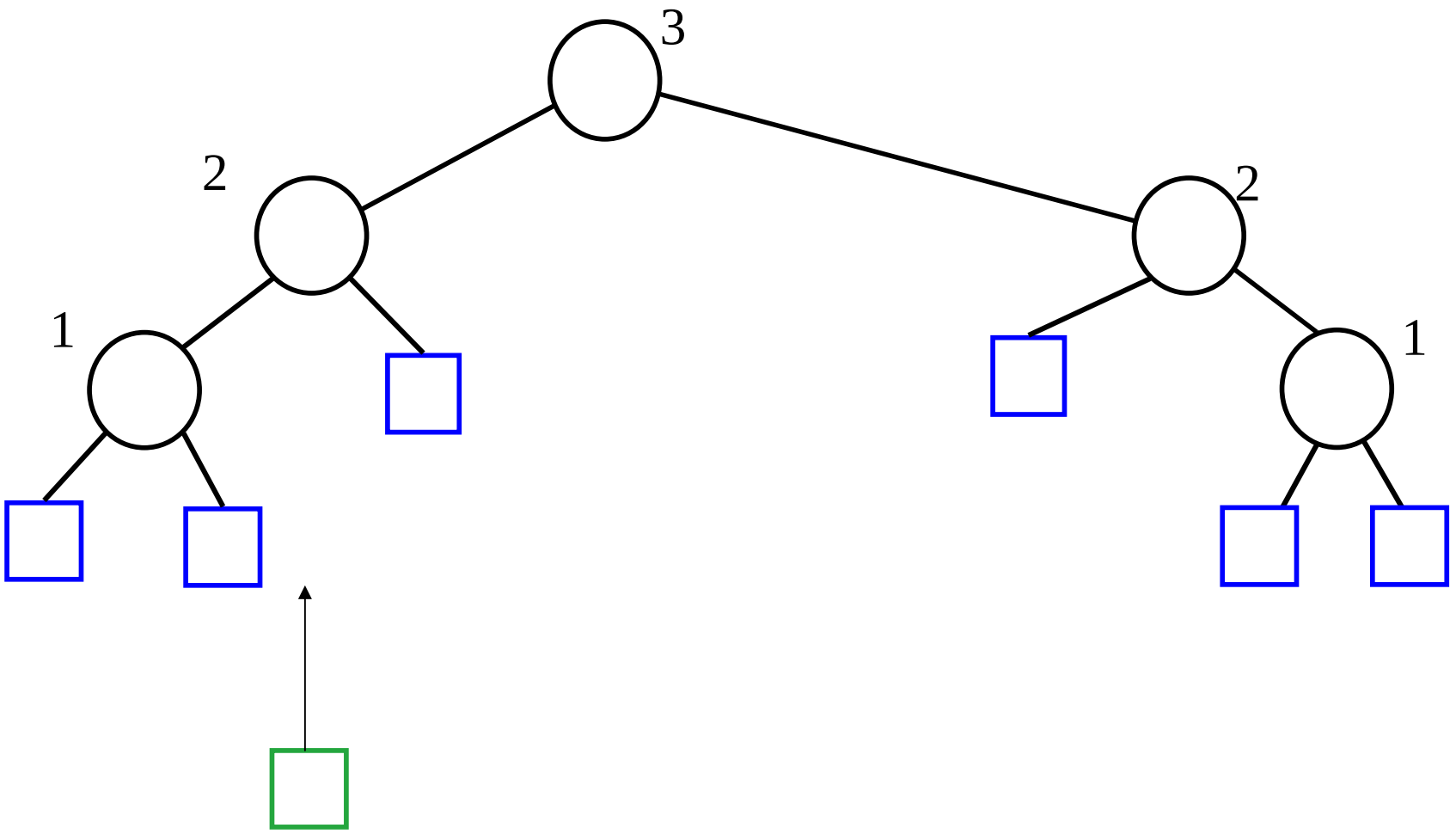
הפרנו את חוקיות העץ



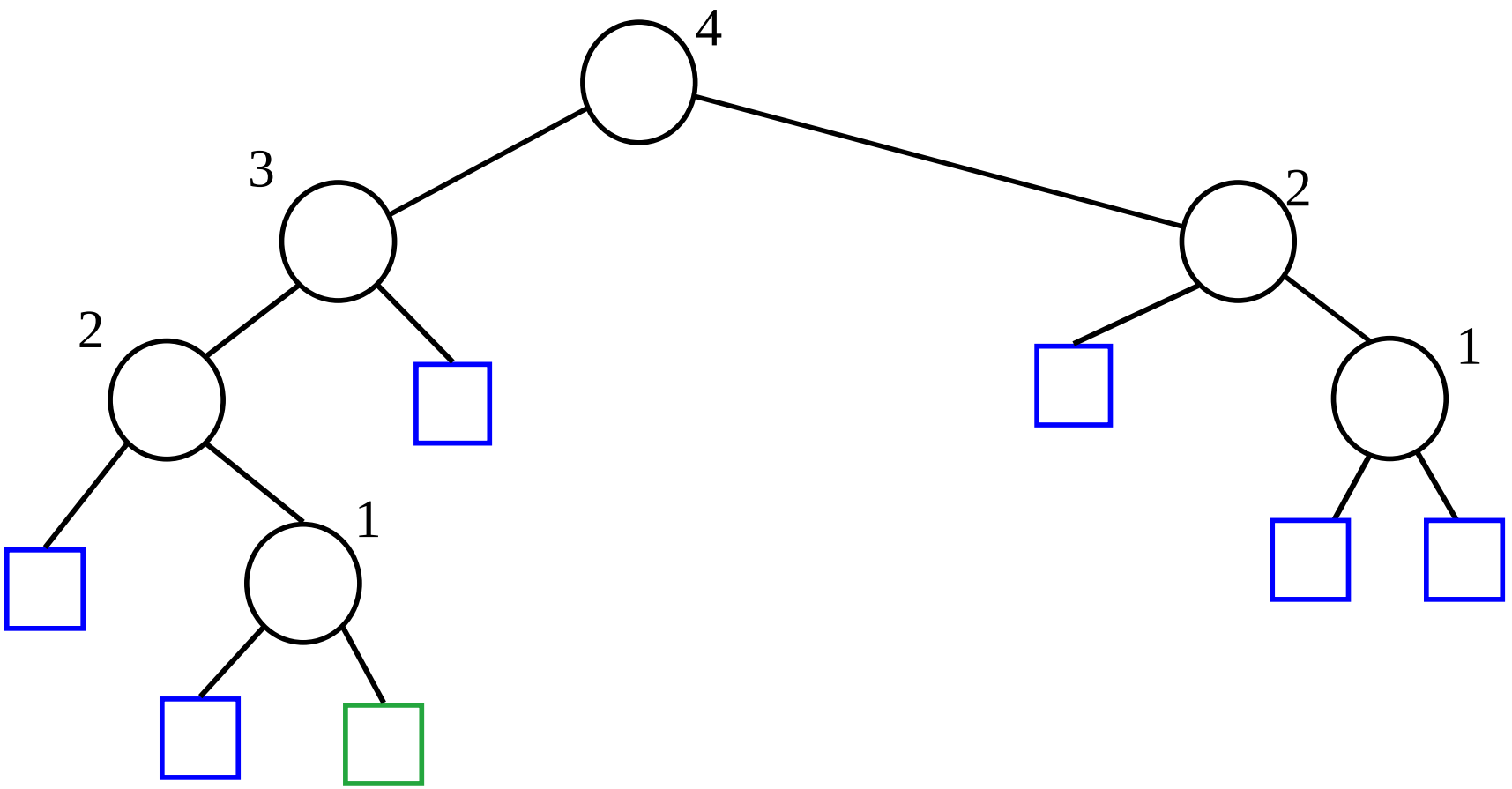
נתקן את השגיאות



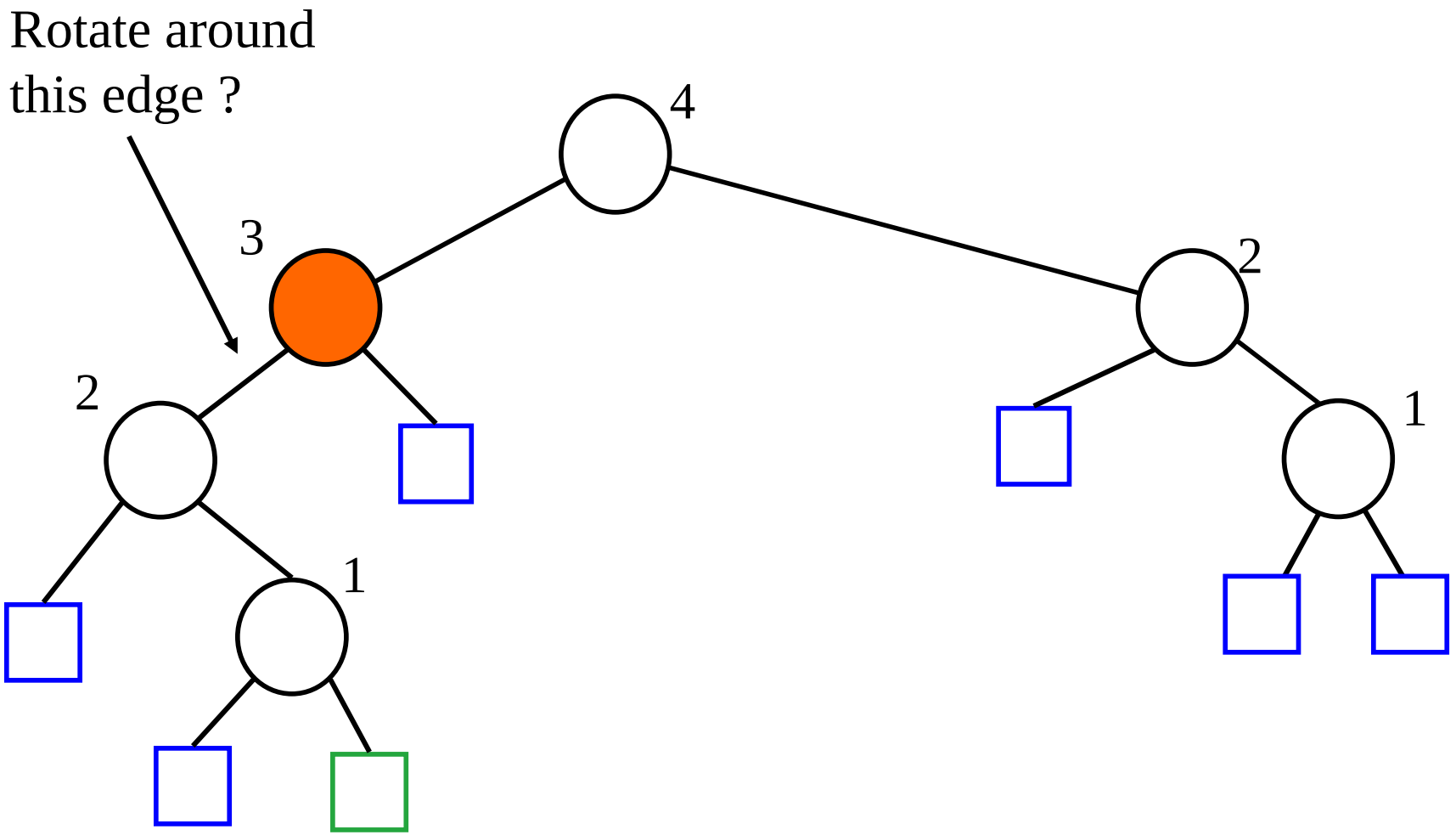
עוד דוגמה



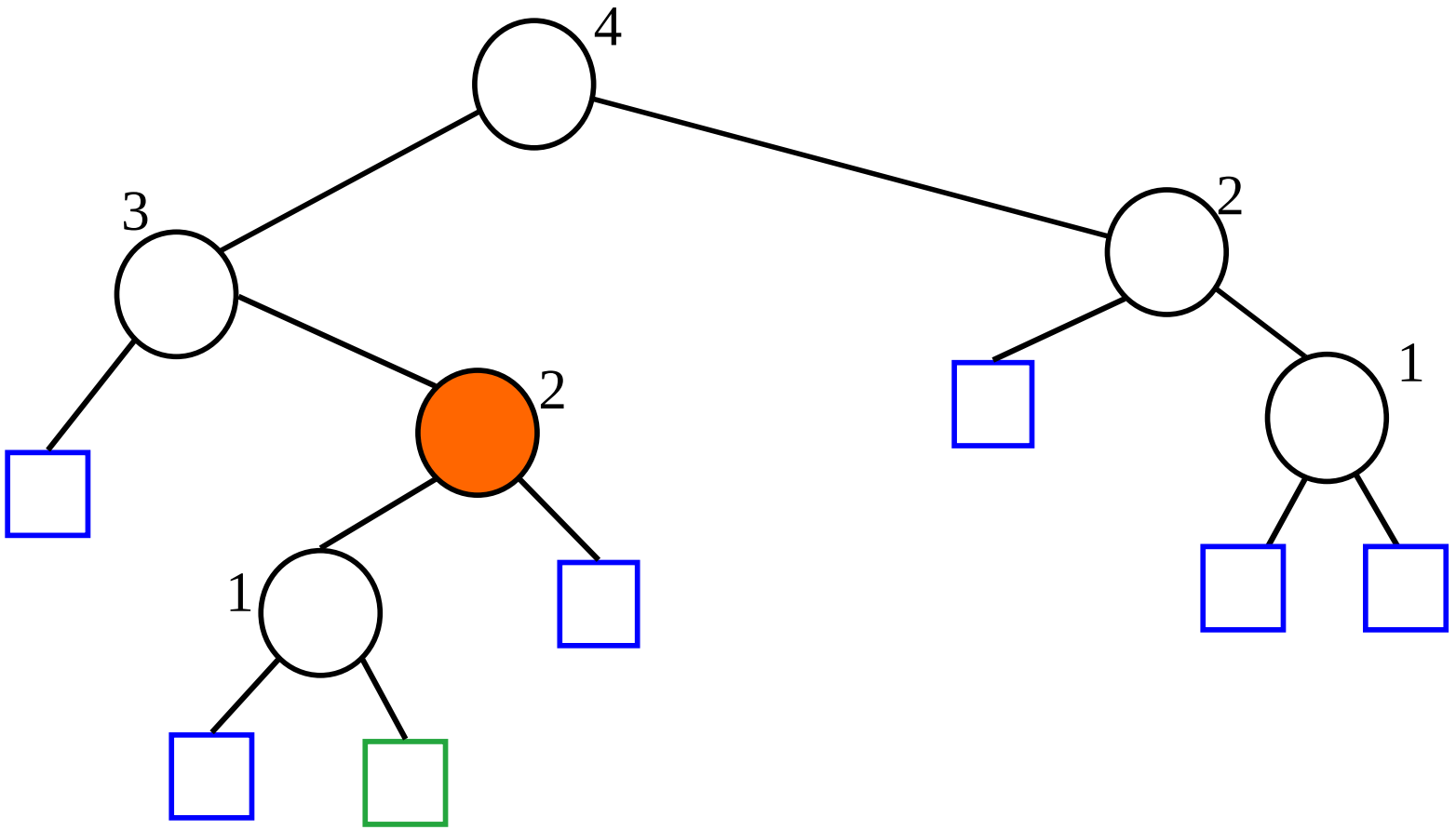
נוסיף את הצומת החדש



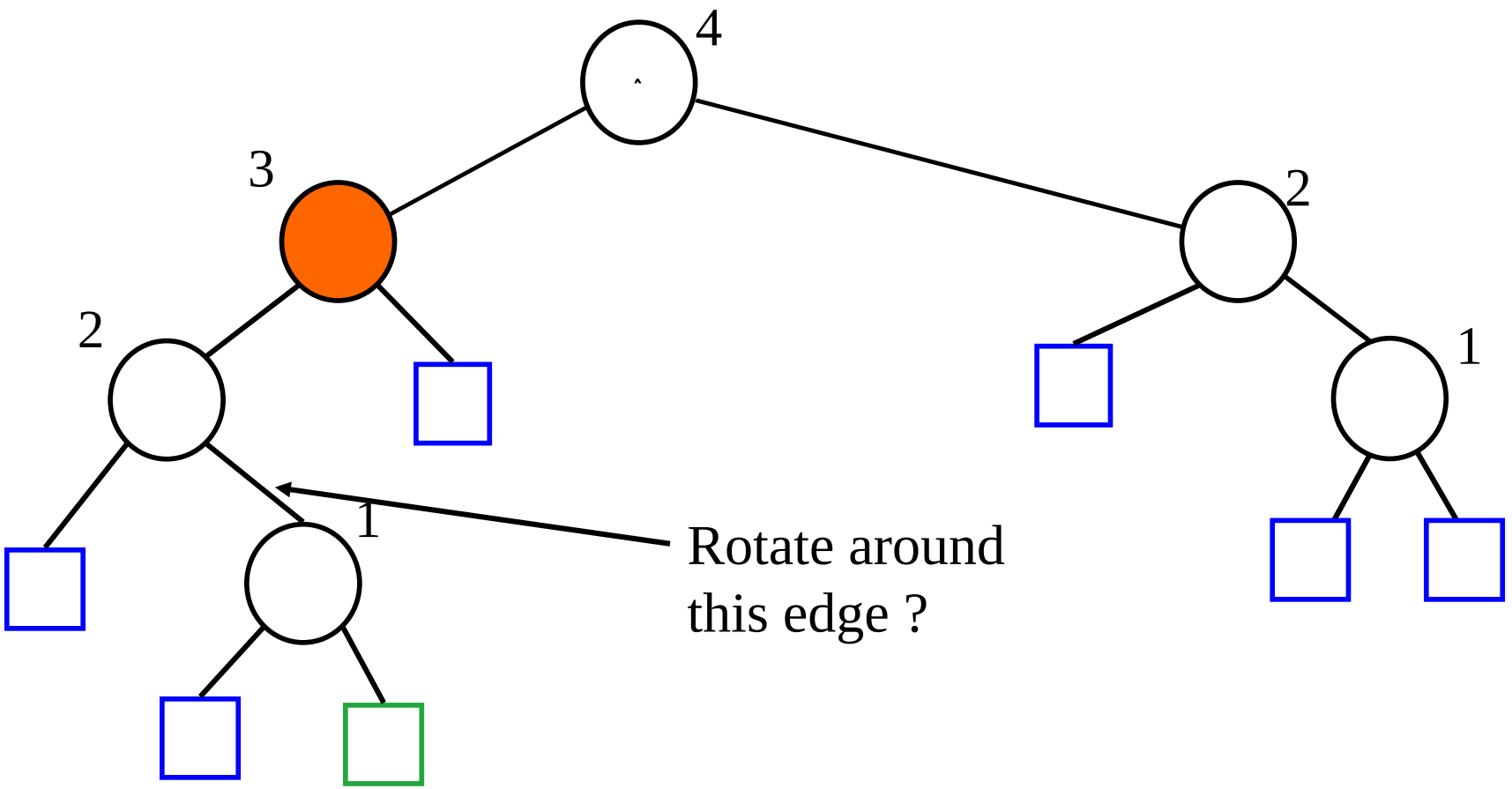
שוב קיבלנו הפרה של החוקים



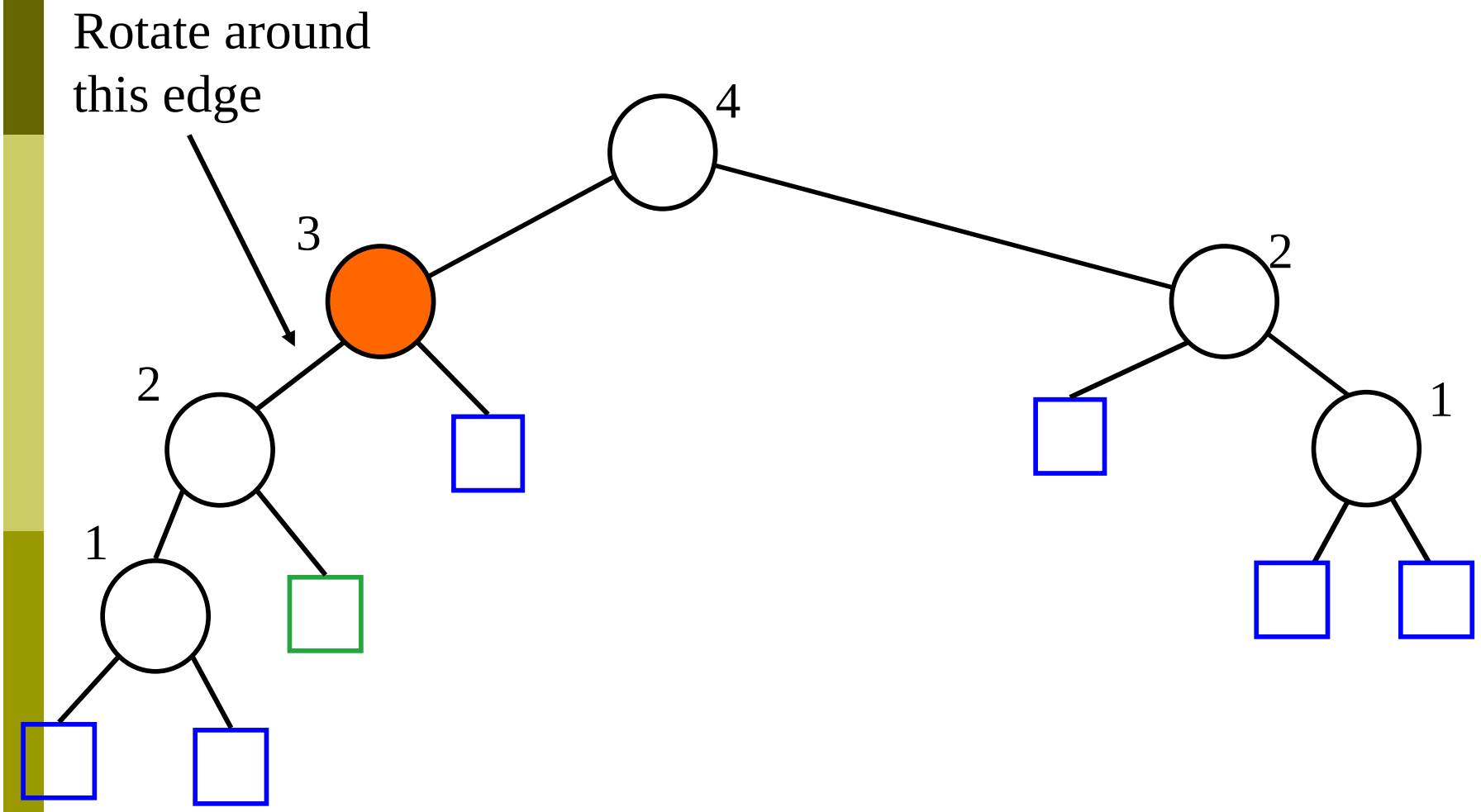
הפעם הרוטציה לא עוזרת



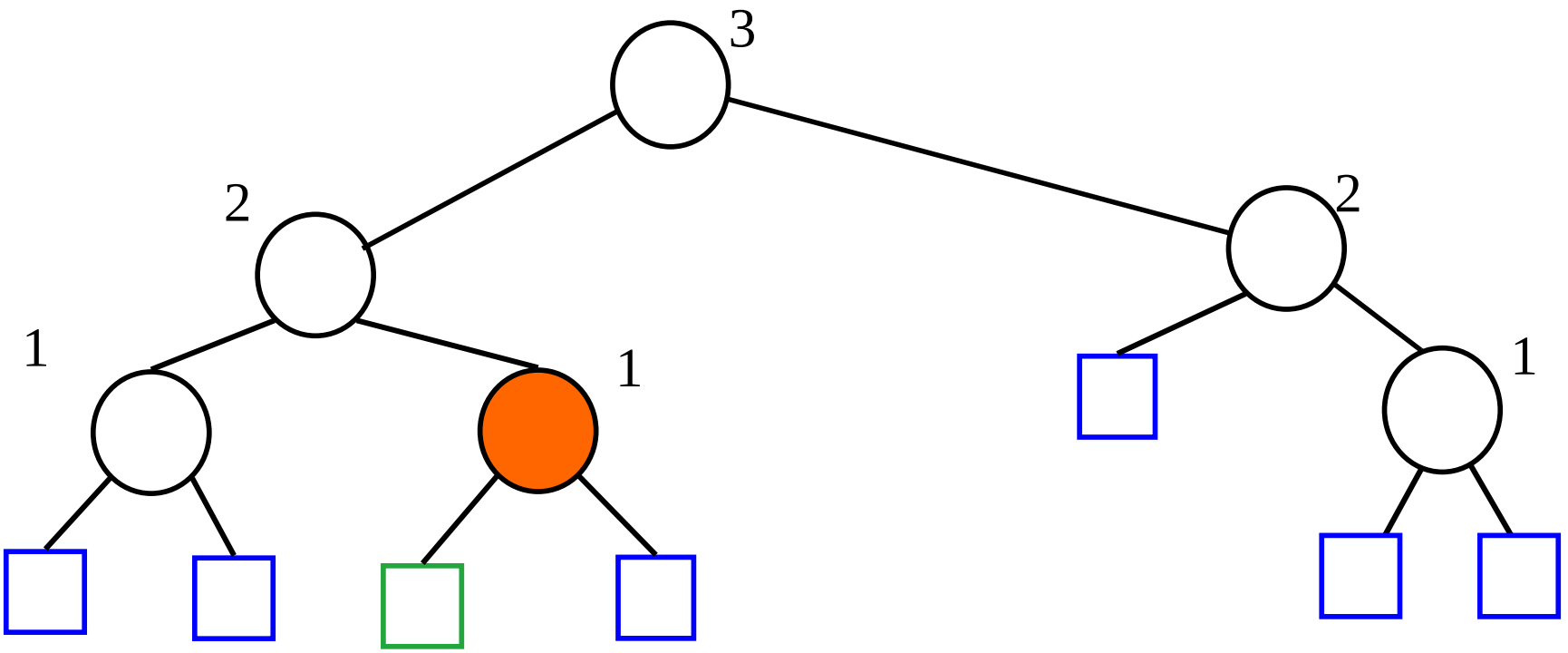
מה ניתן לעשות?



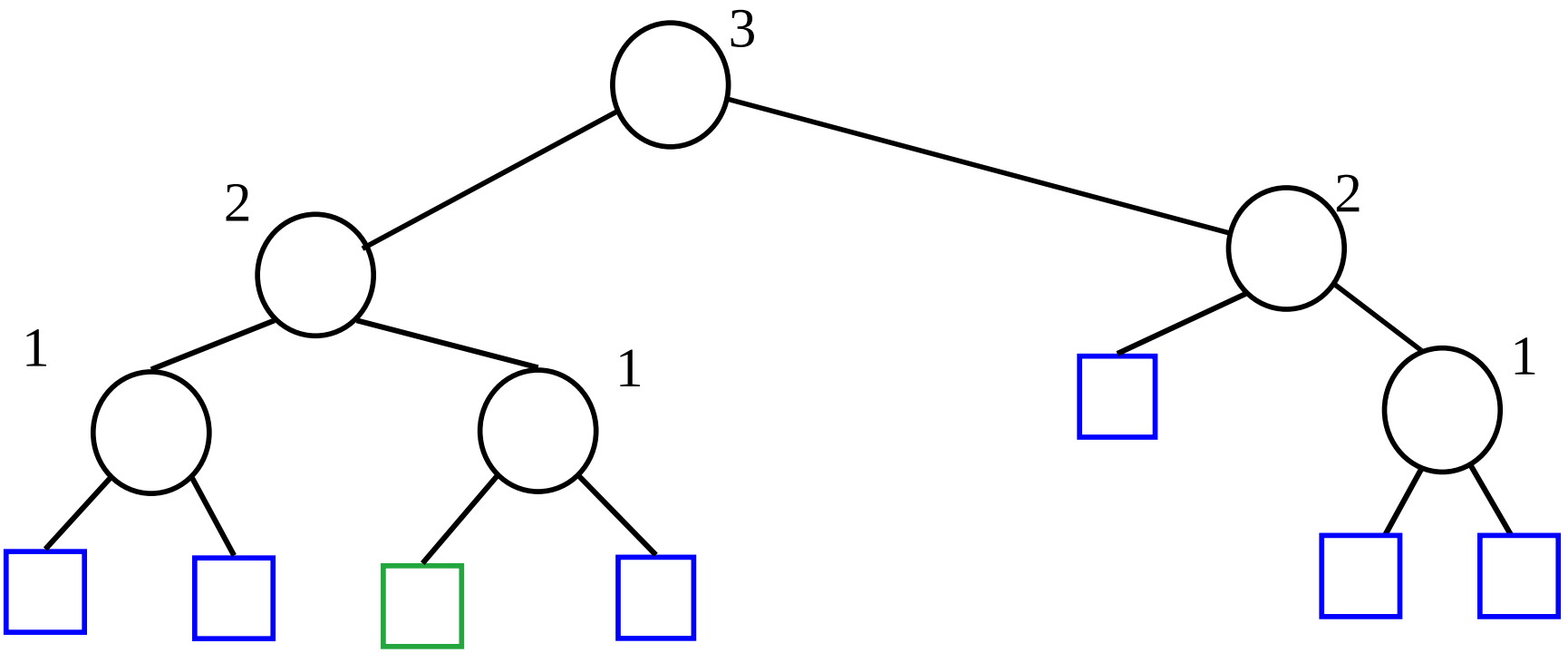
חזרנו למקרה הקודם



תיקנו את הפרת החוקים

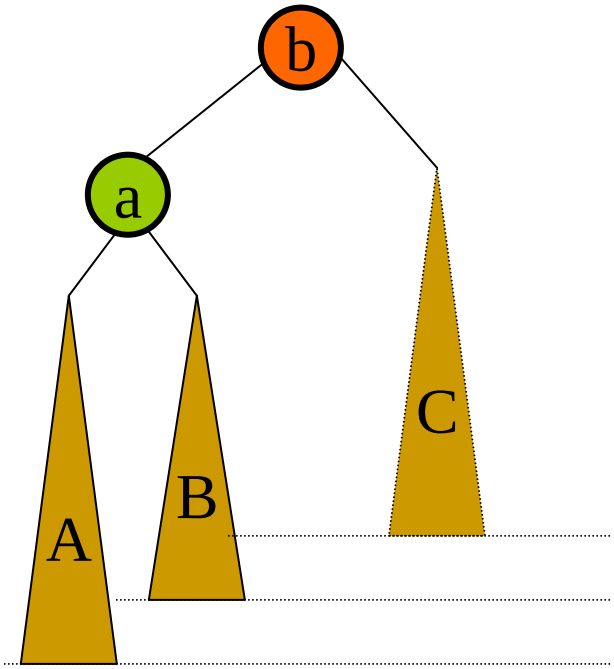


אז מה האלגוריתם הכללי?

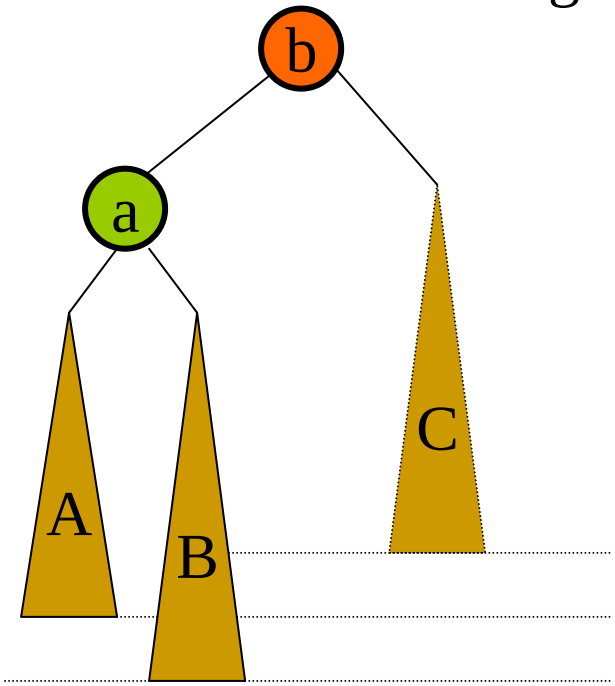


סוגי חוסר שיווי משקל

- Left-left

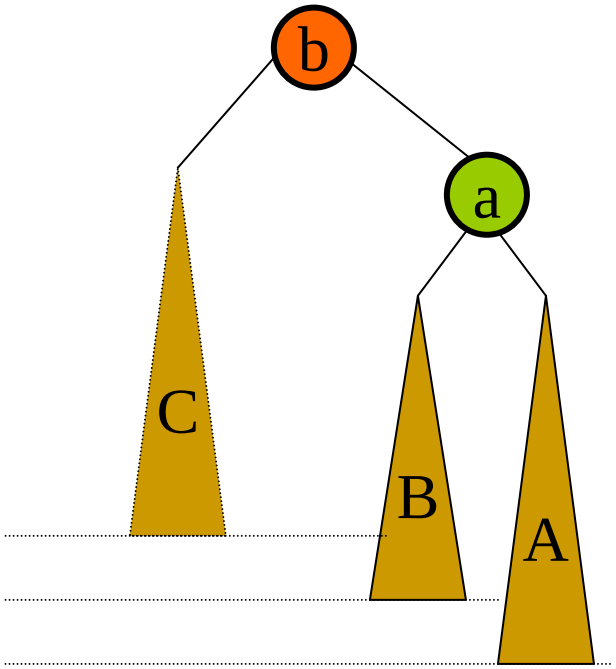


- Left-Right

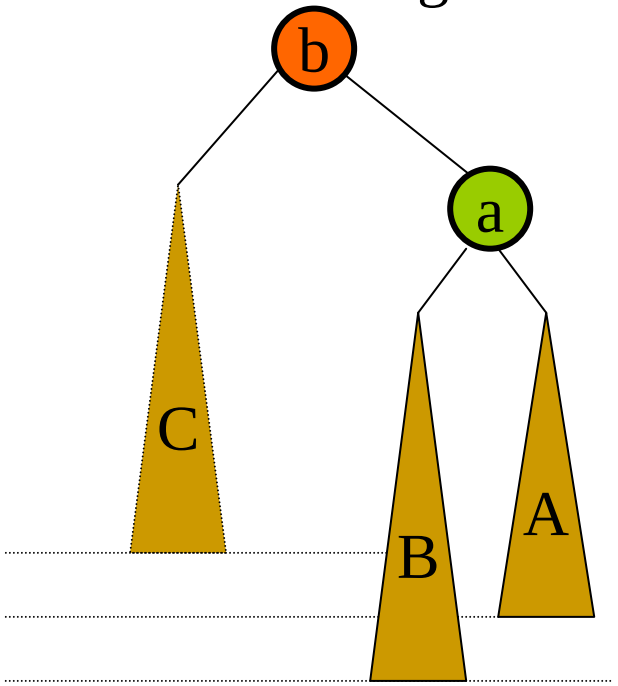


ותמונות המראה שלהם

- Right-Right



- Right-Left



נסתכל על הבעיה לוקאלית

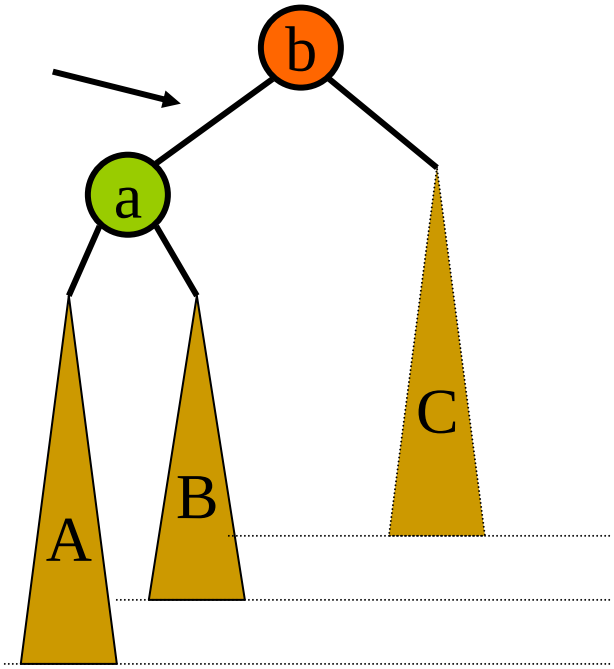
שני עקרונות מנחים

- Imbalance will only occur on the path from the inserted node to the root (only these nodes have had their subtrees altered - local problem)
- Rebalancing should occur at the *deepest unbalanced node* (local solution too)

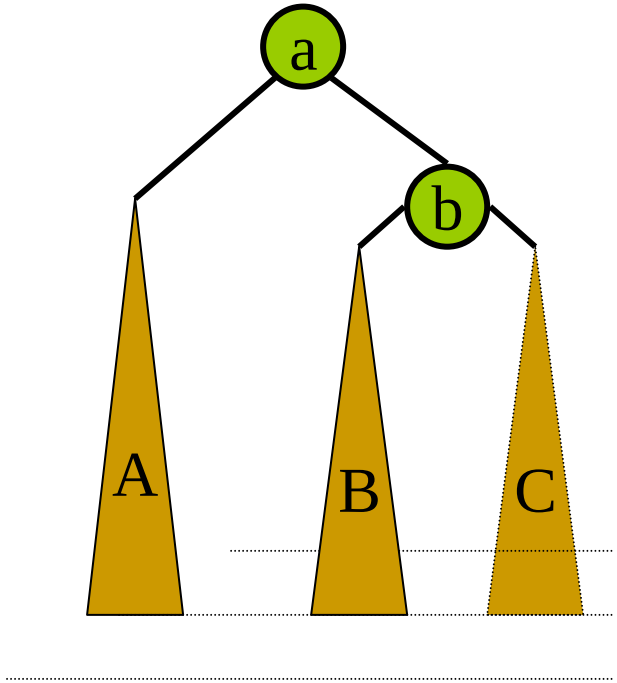
Left-left → single rotation

Rotate around this edge

Before

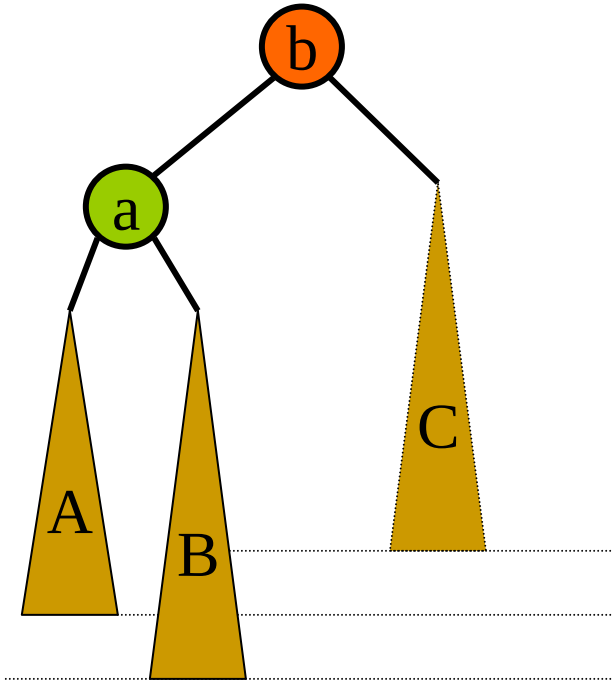


After



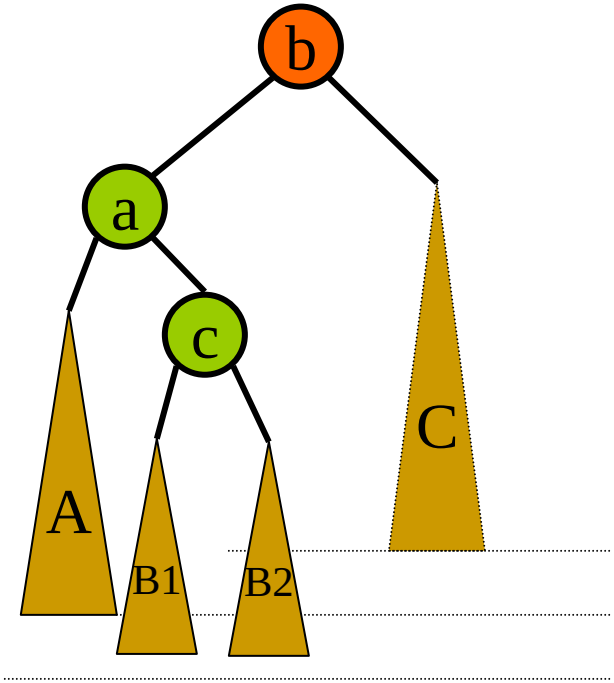
Left-Right fixing

- Before:



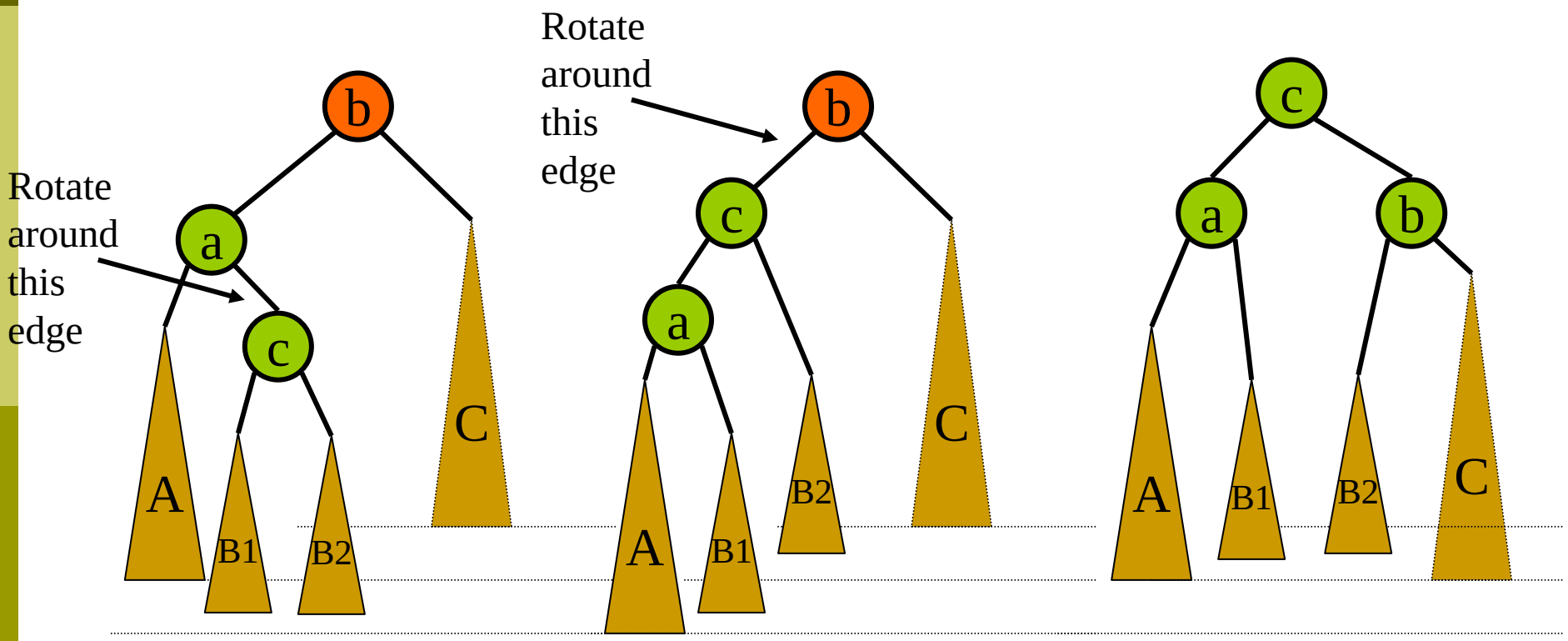
Lets look at this more carefully

- Before:



We need two rotations here…(double rotation)

- Before:

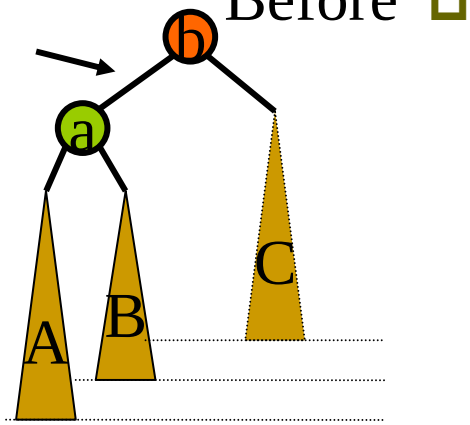


- After:

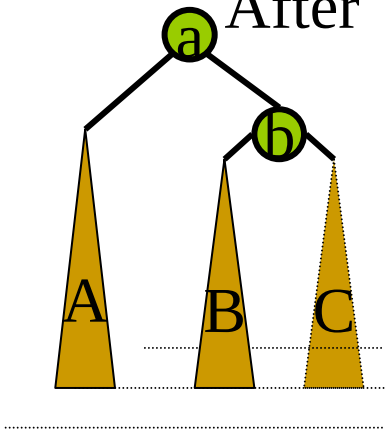
בהוספת צומת חוסר האיזון "מוגבל"

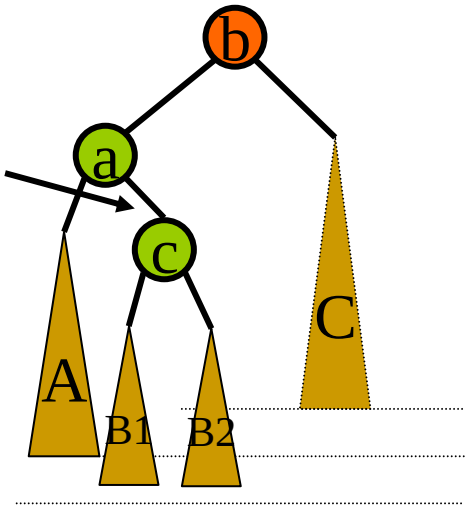
- שימו לב שבשני המקרים של חוסר איזון בהוספת צומת, לאחר הrebalancing, גובה תת העץ שהשורש שלו הוא הצומת שהייתה בחוסר איזון, נשאר אותו דבר! ז"א שלאחר איזון (רוטציה או שתיים) אין צורך להמשיך ולבדוק חוסר איזון יותר גבוה בעץ, כי בוודאות אין כזה!

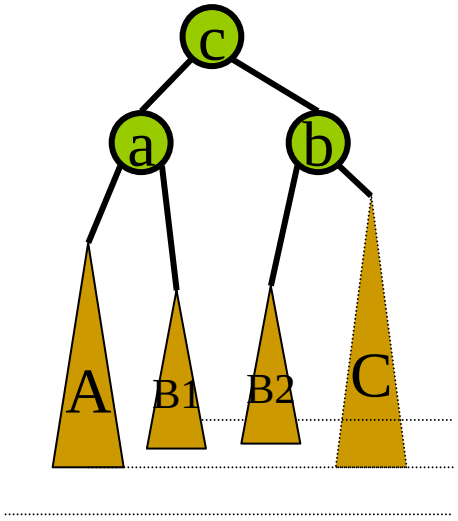
- Before



- After







סיכום

- We fix the first node x on the way up where there is a violation
- After the fix, x is at the same height as it was before, so no nodes further up towards the root will need to be updated.
- Can implement using just 2 bits per node for rebalancing (the difference between the heights of the children

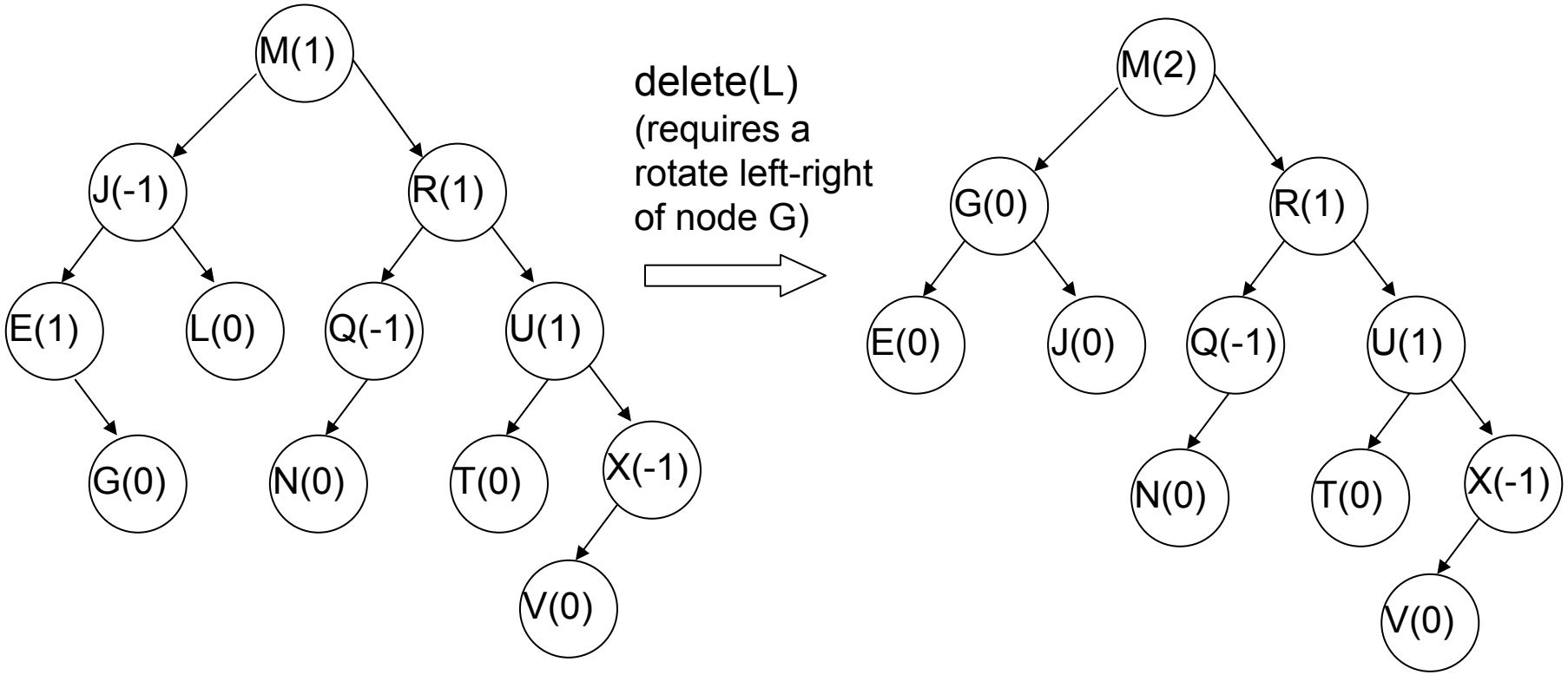
AVL Tree Delete

- An AVL delete is similar to a regular binary tree delete
  - search for the node
  - remove it
    - zero children: replace it with null
    - one child: replace it with the only child
    - two children: replace it with right-most node in the left subtree

AVL Tree Delete

- Complications arise from the fact that deleting a node can unbalance a number of its ancestors
  - insert only required you find the first unbalanced node
  - delete will require you to go all the way back to the root looking for imbalances
    - Must balance any node with a ±2 balance factor (+2 the left sub-tree is 2 levels deeper, -2 the right sub-tree is 2 levels deeper)

AVL Tree Delete



Notice that even after fixing J, M is still out of balance

AVL Tree Delete

- Traversing back to the root
  - now we need to return 2 values
    - one indicating the height of the sub-tree has changed
    - another to return the deleted value
  - one of the values will need to be “returned” through the parameters
    - will create a data TreeNode to place the returned data into

Returning Two Values

- Here is a simple example:

```
void main() {
        TreeNode data = new TreeNode(null);
        if(someMethod(data))
            System.out.println(data.key.toString());   // prints 5
}

boolean someMethod(TreeNode data) {
        data.key = new Integer(5);
        return true;
}
```

Return Values

- The delete method should return true if the height of the subtree changes
  - this allows the parent to update its balance factor
- A TreeNode reference should be passed into the delete method
  - if the key is found in the tree, the data in the node should be copied into the TreeNode element passed into delete

Delete Situations

- Node to delete is a leaf
  - copy nodes data into the TreeNode data field
  - make the nodes parent refer to null
    - remember to consider deleting the root
  - return true
    - the height of the sub-tree went from 1 to zero
- Node to delete has only one child
  - copy nodes data into the TreeNode data field
  - make the nodes parent refer to its only child
    - remember to consider deleting the root
  - return true
    - height of sub-tree has been decreased one level

Delete Situations

- Node to delete has two children
  - copy nodes data into the TreeNode data field
  - find the node to replace this one with
    - descendant farthest right in left sub-tree
  - then make a copy of the replacement node
    - do not want to move the original
  - insert the copied replacement in place of the node to delete
  - delete the original replacement node
    - to do this, call the delete method recursively
      - do not just delete it

Deleting Replacement Node

- So why make a copy of node to replace?
  - remember, we need to keep all nodes between the deleted node and the replacement node balanced
  - well, that's what the delete method does
  - consider what happens when calling delete with the replacement node
    - guaranteed replacement doesn't have two children
      - it gets deleted and returns true
    - replacements parent will get back true and update its balance factor
      - it will then return (true or false) and eventually we will get back to the node we deleted

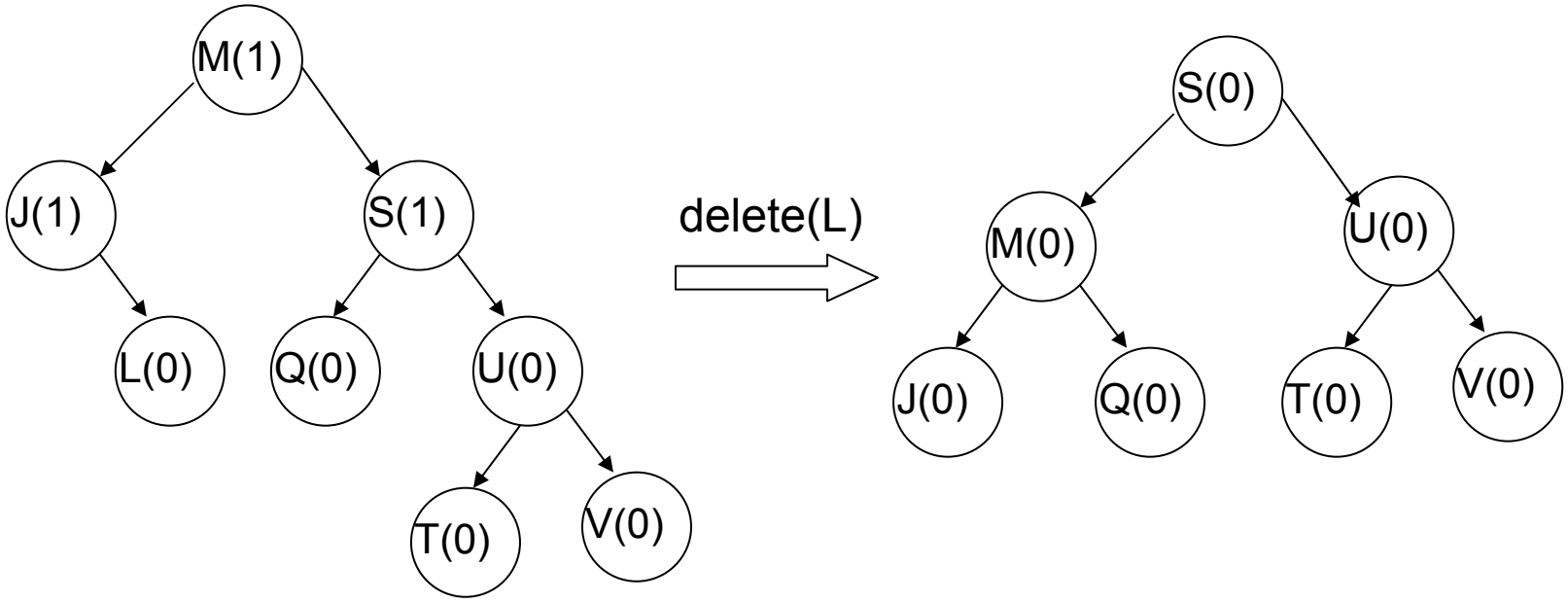
Changing Height

- So how do we know whether or not to return true?
  - if a recursive call returns false, number of levels below unchanged
    - return false
  - if it's a leaf or only has one child, lost a level
    - return true
  - if a recursive call returns true and the balance factor goes to zero, lost a level
    - was unbalanced, now it's not – this only happens if one side or other loses a level to balance things
    - return true

Rotating Nodes

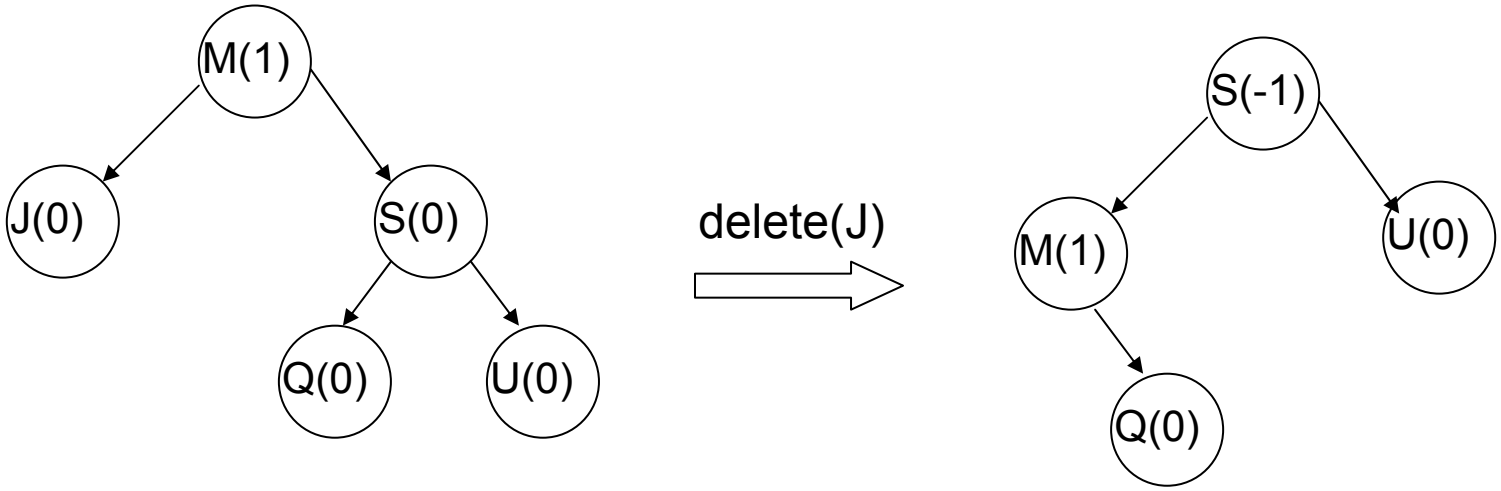
- Very similar theory to insert
  - One major difference
    - if a node was inserted and another node had to be balanced, the child to rotate with had a balance factor of -1 or 1 – never zero
    - when deleting a node, it is possible for the child to rotate with to be zero

Rotating Nodes



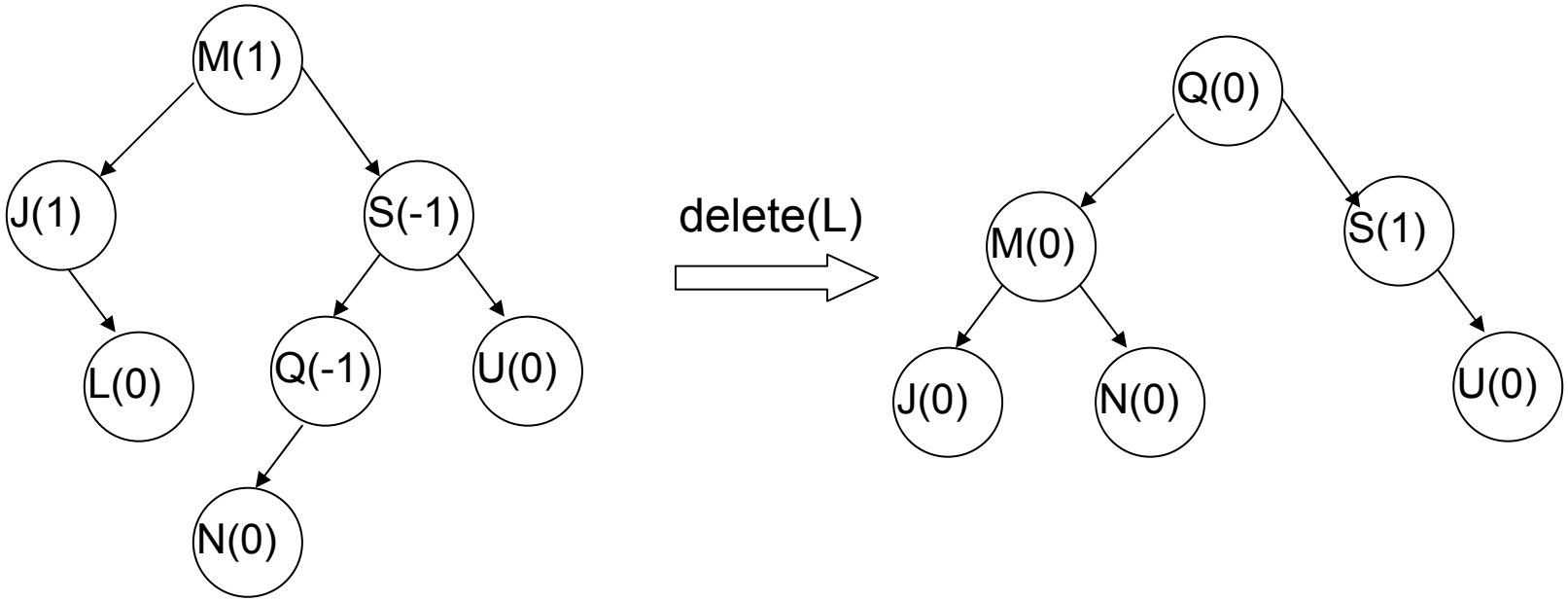
Deleting a node in the left sub-tree (M's balance becomes 2).
Need to rotate M with it's right sub-tree.
S's balance factor is 1 before rotate.
Just do a rotate left of node S.
Notice that the height of the tree *does* change in this case.

Rotating Nodes



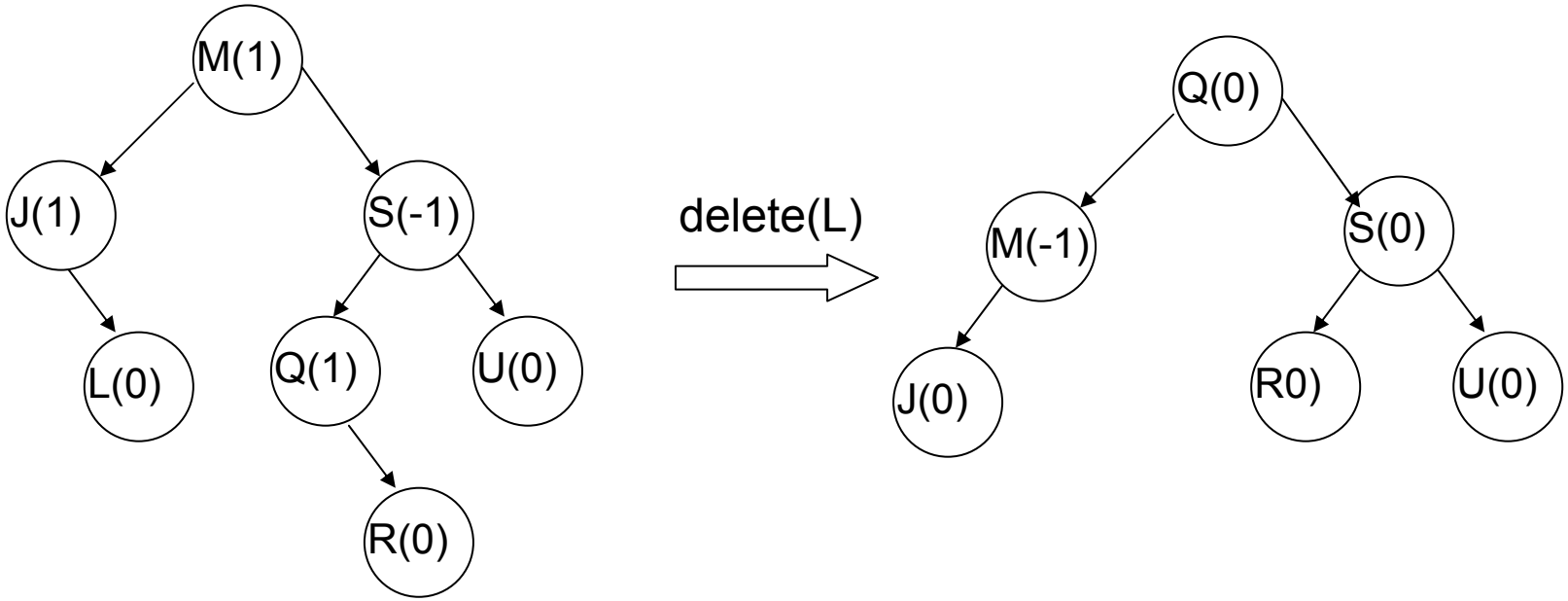
Deleting a node in the left sub-tree (M's balance becomes 2).
Need to rotate M with it's right sub-tree.
S's balance factor is 0 before rotate.
Just do a rotate left of node S.
Notice that the height of the tree *does not* change in this case.

Rotating Nodes



Deleting a node in the left sub-tree (M's balance becomes 2).
Need to rotate M with it's right sub-tree.
S's balance factor is -1 before rotate.
Need to do a right rotate of Q with S and then a left rotate of Q with M.
Notice that the height of the tree changes.

Rotating Nodes



Deleting a node in the left sub-tree (M's balance becomes 2).
Need to rotate M with it's right sub-tree.
S's balance factor is -1 before rotate.
Need to do a right rotate of Q with S and then a left rotate of Q with M.
Notice that the height of the tree changes.

Deleting a Node

- boolean delete(Comparable key, TreeNode subRoot,
  TreeNode prev, TreeNode data) {

  I) if subRoot is null, tree empty or no data
  return false

  II) compare subRoot's key, K_r, to the delete key, K_d
  A) if $K_r < K_d$, need to check the right sub-tree
  -> call delete(key, subRoot.right, subRoot, data)
  -> if it returns true, adjust balance factor (-1)
  -> if it returns false, just return false
  B) if $K_r > K_d$, need to check the left sub-tree
  -> call delete(key, subRoot.left, subRoot, data)
  -> if it returns true, adjust balance factor (+1)
  -> if it returns false, just return false

Delete Continued

- Delete continued

c) if $K_r == K_d$, this is the node to delete

-> if zero or 1 children, make parent "go around" subRoot and return true

-> if two children, find replacement node, copy it, insert copy into subRoots place, and delete the original replacement node

* if delete returns true, increase bal by 1

III) If the code gets this far

A) if subRoot's balance factor equals 2 or -2, balance the tree

B) if, after balancing tree, subRoot's balance factor equals 0

-> return true

C) if, after balancing tree, subRoot's balance factor is not 0

-> return false

זמני ריצה

- a single restructure is O(1)
  - using a linked-structure binary tree
- find is O(log n)
  - height of tree is O(log n), no restructures needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- remove is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

לינקים מעניינים

- AVL Trees on Wikipedia
- AVL Tree applet
- Nice Red-Black tree demo
- Another AVL/RB tree applet

The End