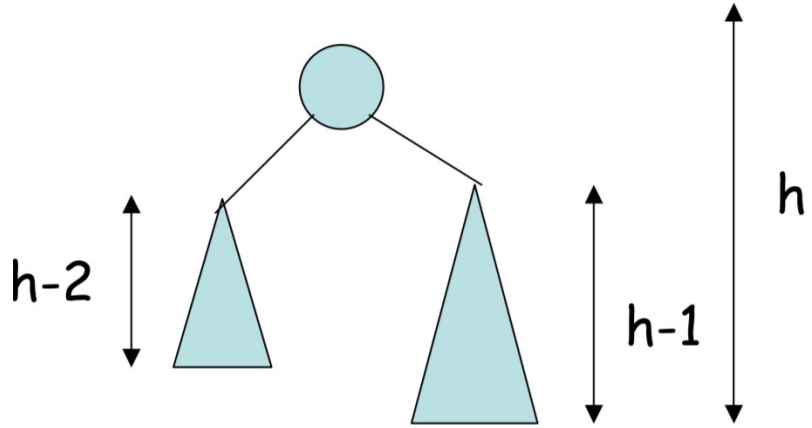
# INT202
# Complexity of Algorithms
# Search Algorithms

XJTLU/SAT/INT
SEM2 AY2020-2021

# AVL trees (by Adel'son-Vel'skii and Landis)

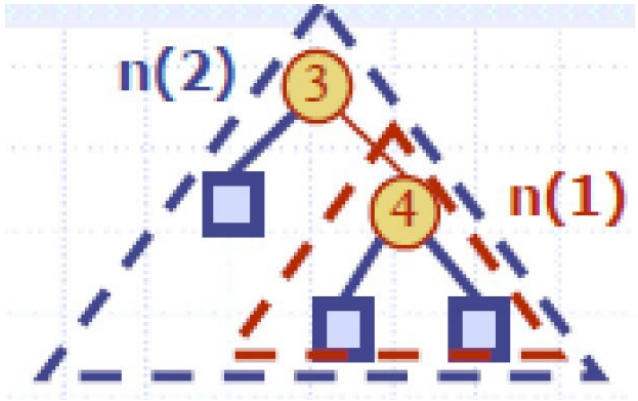
▷ *Height-Balance Property*: for any node n, the heights of n's left and right subtrees can differ by at most 1.




**Height of an AVL Tree**

- **Theorem**: The height of an AVL tree storing n keys is O(log n).

- **Theorem**: The height of an AVL tree storing n keys is O(log n).
- Proof: n(h): the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2.




- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
  n(h) > 2n(h-2), n(h-2) > 2n(h-4), n(h-4) > 2n(n-6) => n(h) > $2^i$n(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
  Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

h-2i = 1 or 2, depending on if h is even or odd: i = $\lceil h/2 \rceil - 1$.

log n(h) > log $2^{h/2-1}$, log n(h)>h/2-1

# AVL trees (by Adel'son-Vel'skii and Landis)

An *AVL tree* is a tree that has the Height-Balance Property.

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

*Consequence 1*: A *search* in an AVL tree can be performed in time $O(\log n)$.

*Consequence 2*: Insertions and removals in AVL need more careful implementations (using *rotations* to maintain the height-balance property).

# Insertion in AVL trees

An *insertion* in an *AVL tree* begins as an insertion in a *general BST*, i.e., attaching a new *external* node (leaf) to the tree.

- This action may result in a tree that *violates* the *height-balance property* because the heights of some nodes increase by 1.

# Insertion in AVL trees

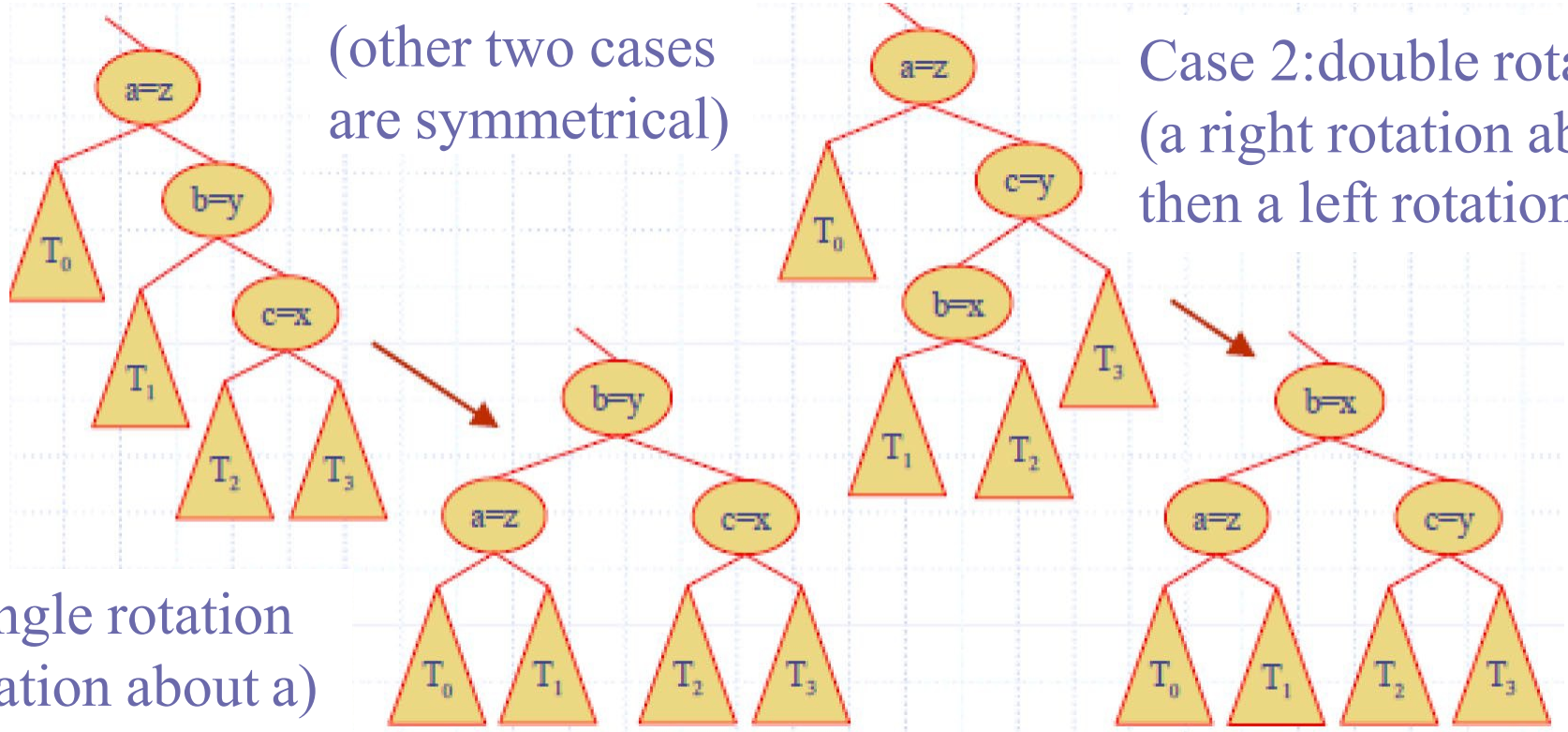
If an insertion causes T to become unbalanced, we travel up the tree from the newly created node until we find the first node x such that its grandparent z is unbalanced node.

• Since z became unbalanced by an insertion in the subtree rooted at its child y, to rebalance the subtree rooted at z, we must perform a restructuring

- we rename x, y, and z to a, b and c based on the order of the nodes in an in-order traversal.
- z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other subtrees formerly children of x, y, and z.

# Restructuring

- let (a,b,c) be an inorder listing of x, y, z
- z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other subtrees formerly children of x, y, and z.

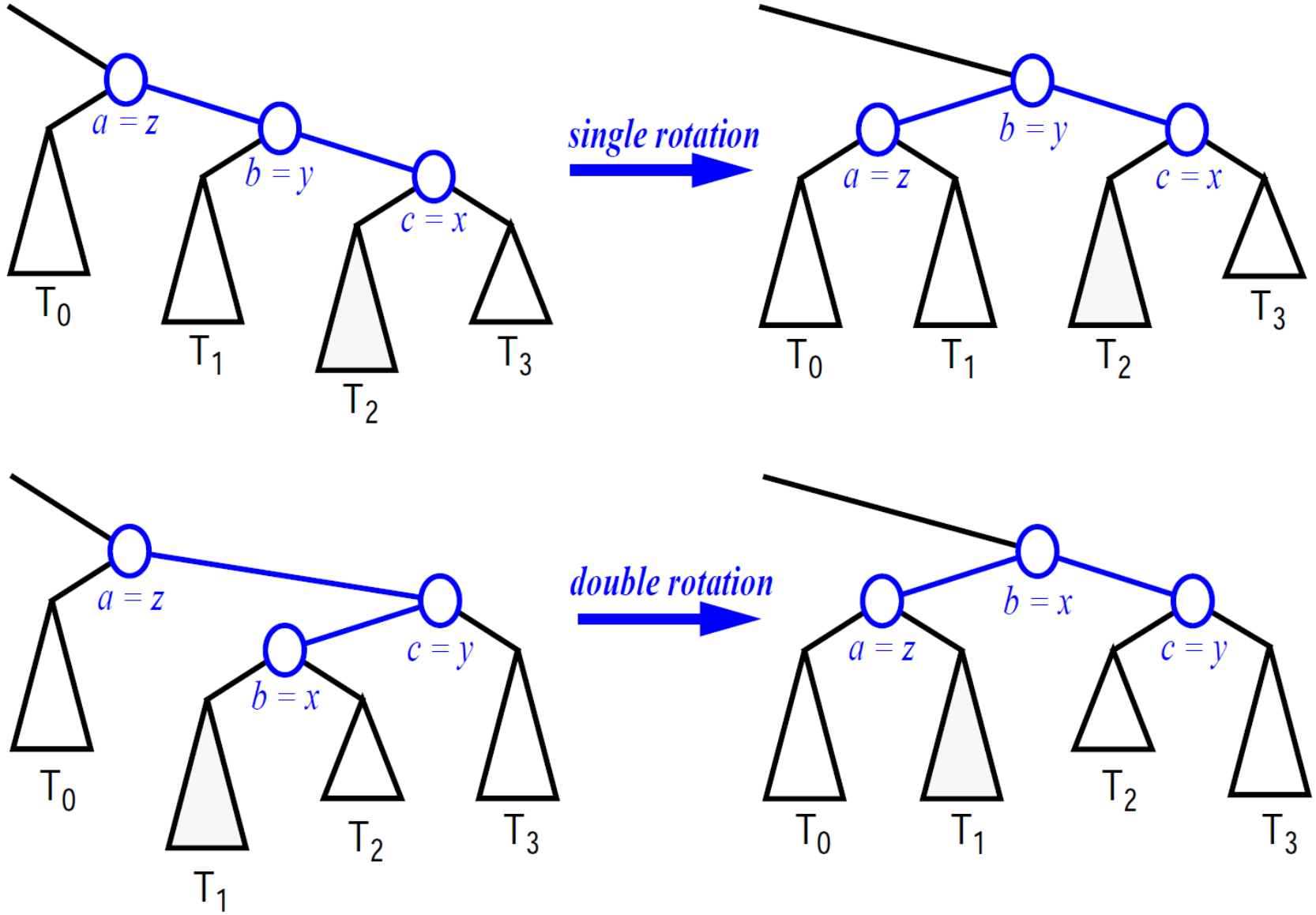


# Restructuring

- 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 a trinode restructuring (which corresponds to a single or double rotation) involving nodes x, y, and z
- 1: Let (a,b,c) be a left-to-right (inorder) listing of the nodes x, y, and z, and let (T0, T1, T2, T3) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.
- 2: Replace the subtree rooted at z with a new subtree rooted at b.
- 3: Let a be the left child of b and let T0 and T1 be the left and right subtrees of a, respectively.
- 4: Let c be the right child of b and let T2 and T3 be the left and right subtrees of c, respectively.

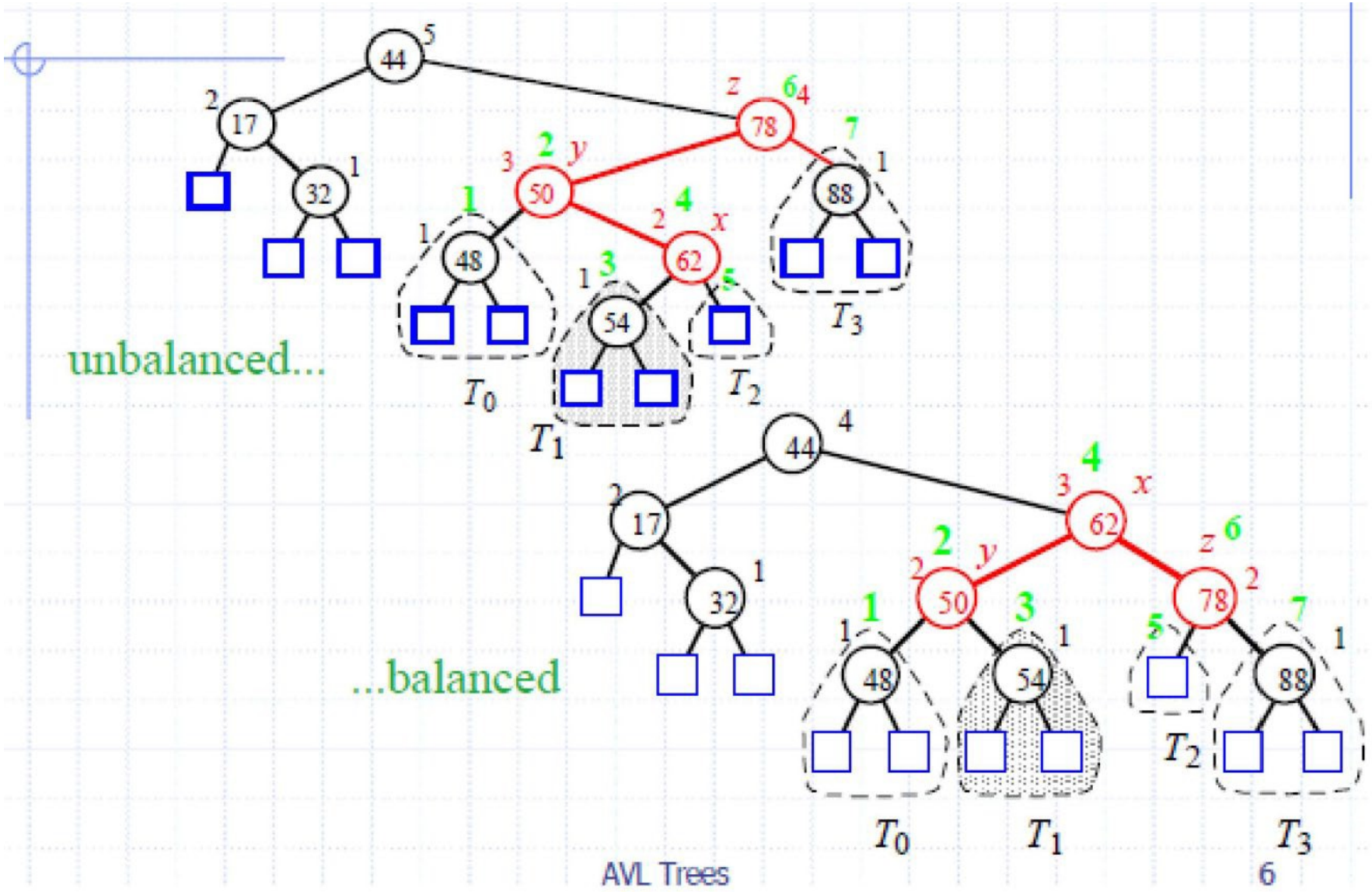
# Insertion Example

- 2: Replace the subtree rooted at z with a new subtree rooted at b.
- 3: Let a be the left child of b and let T0 and T1 be the left and right subtrees of a, respectively.
- 4: Let c be the right child of b and let T2 and T3 be the left and right subtrees of c, respectively.



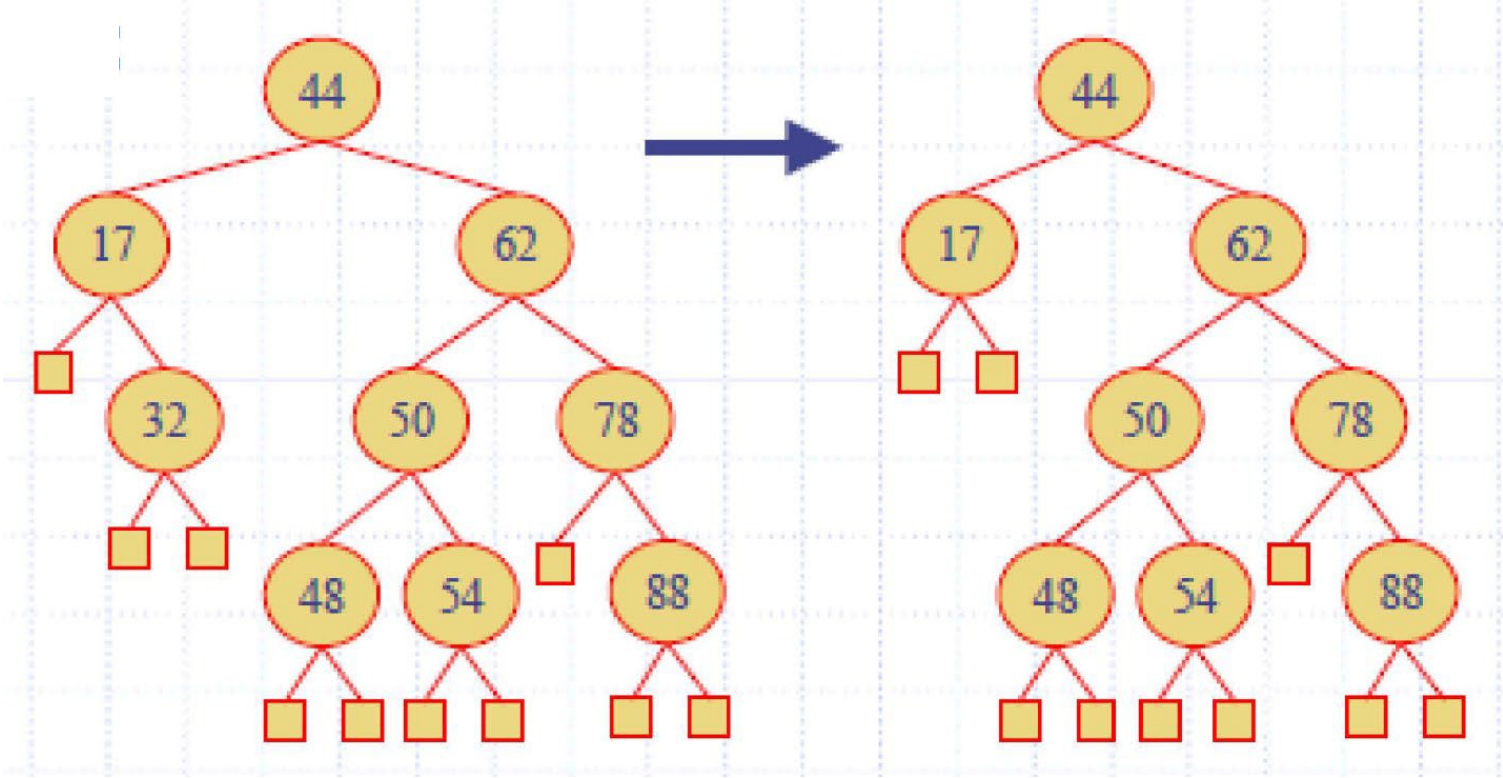
# Insertion Example

- 2: Replace the subtree rooted at z with a new subtree rooted at b.
- 3: Let a be the left child of b and let T0 and T1 be the left and right subtrees of a, respectively.
- 4: Let c be the right child of b and let T2 and T3 be the left and right subtrees of c, respectively.



# Removal in an AVL Trees

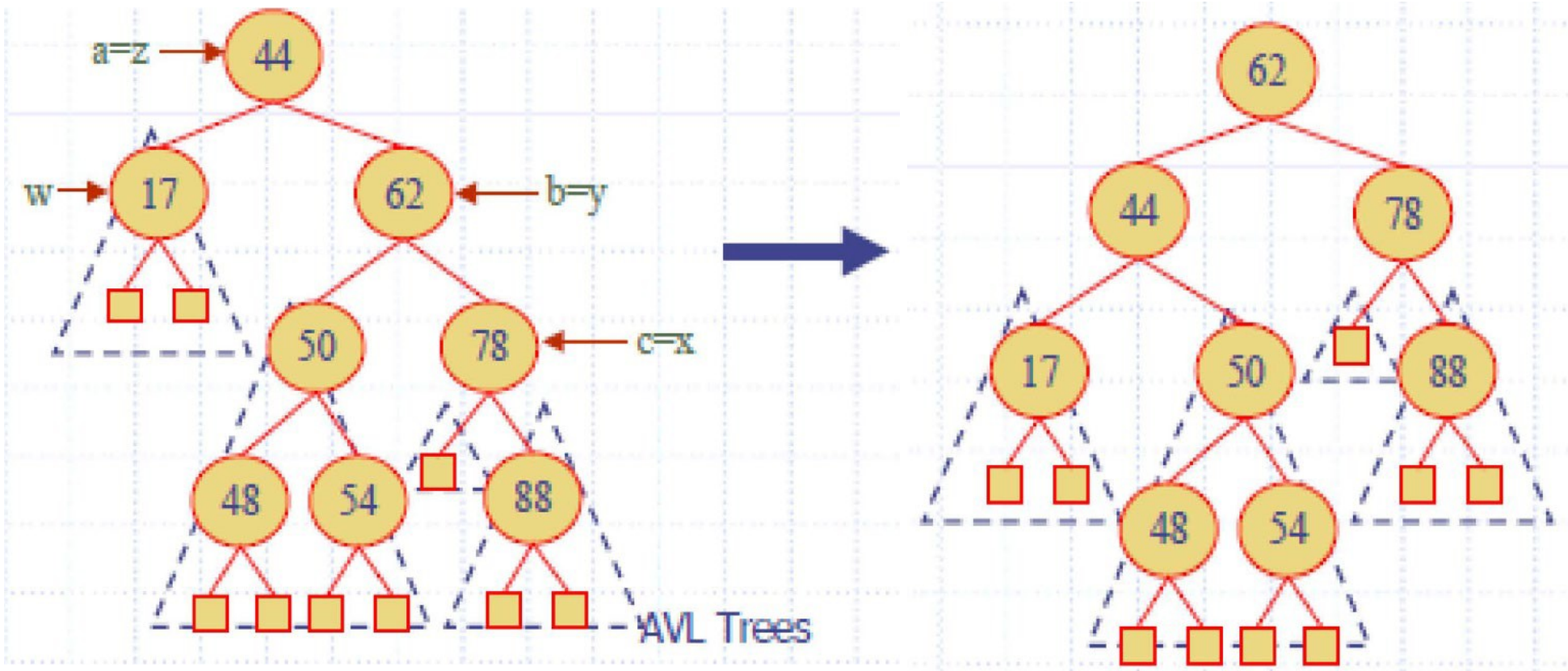
- A removal in an AVL tree begins as a removal in a general BST.
- Action may violate the height-balance property.



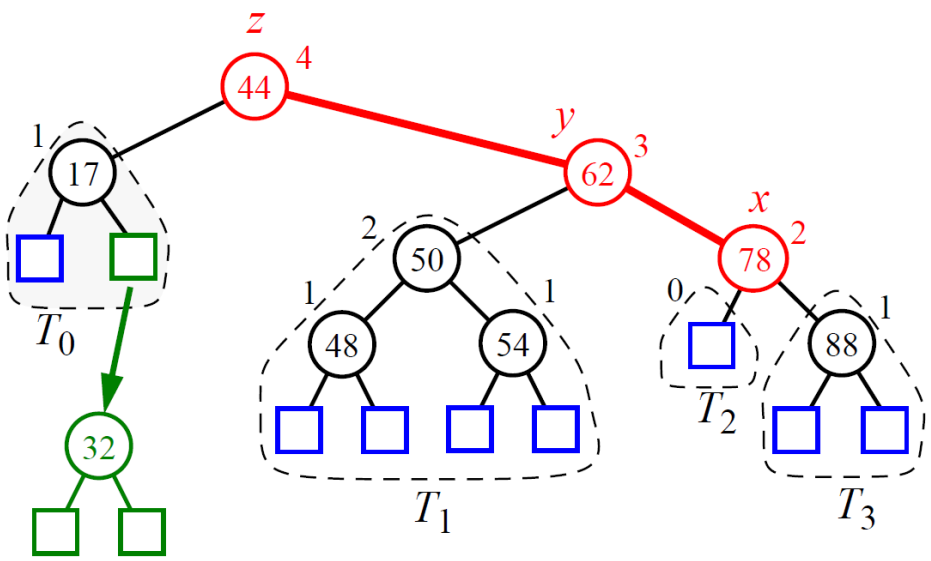

Before deletion of 32 After deletion

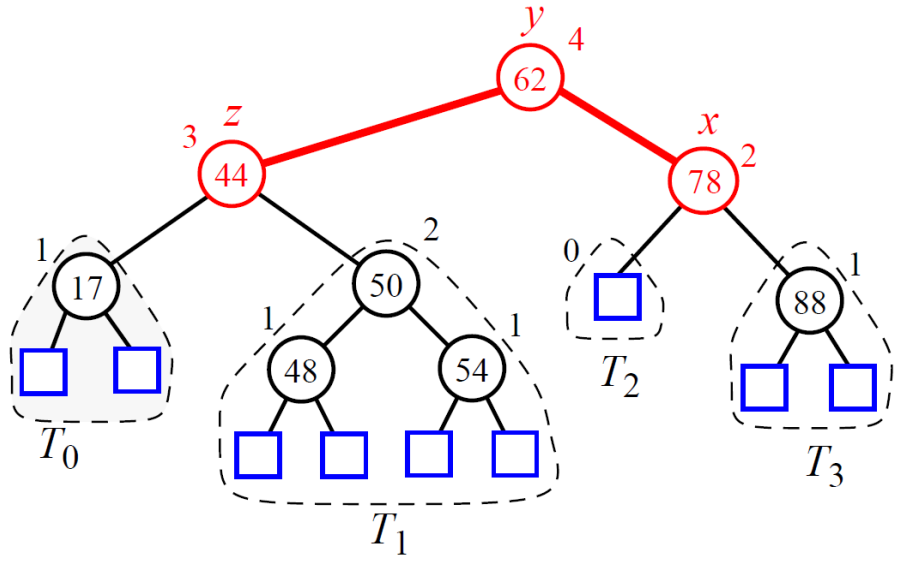
# Removal in an AVL Trees

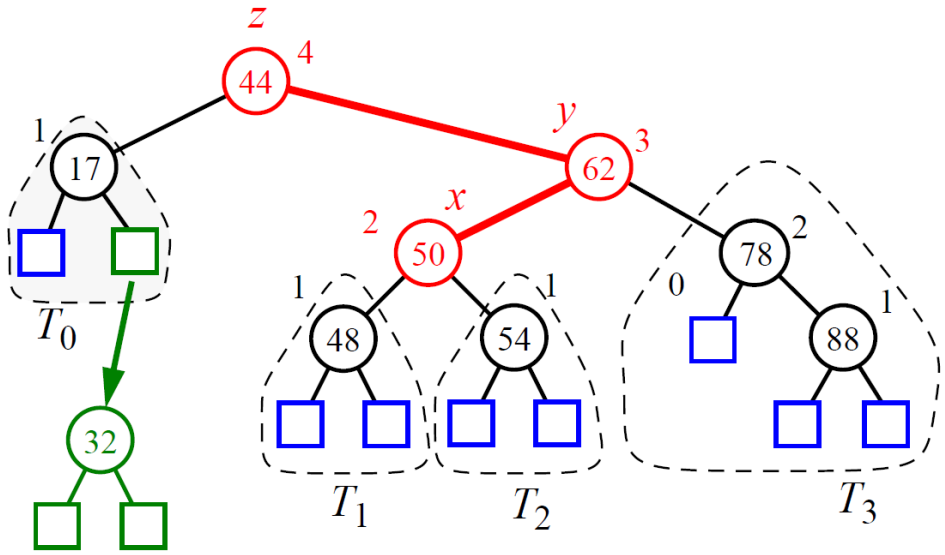
- We perform restructure(x) to restore balance.

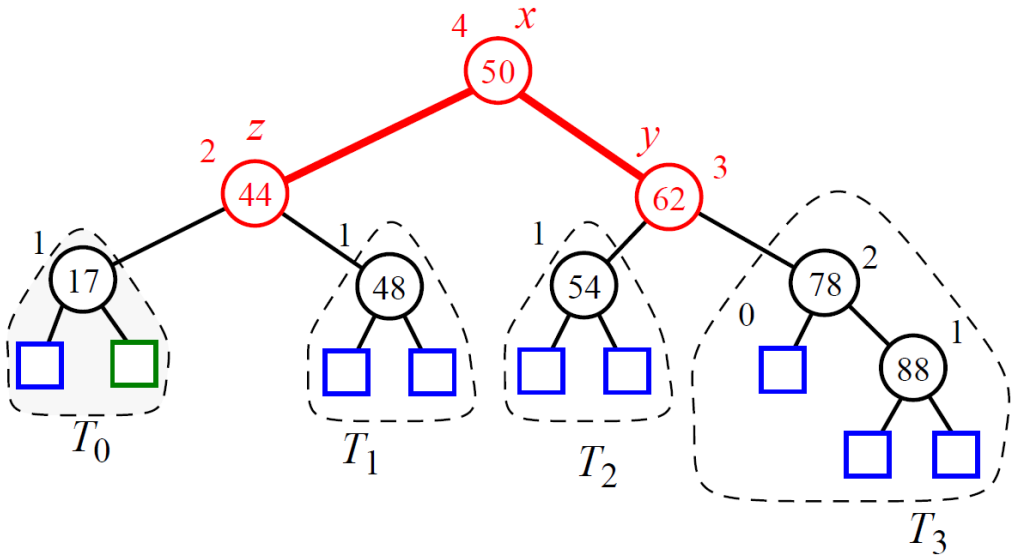


# Removal in an AVL Trees



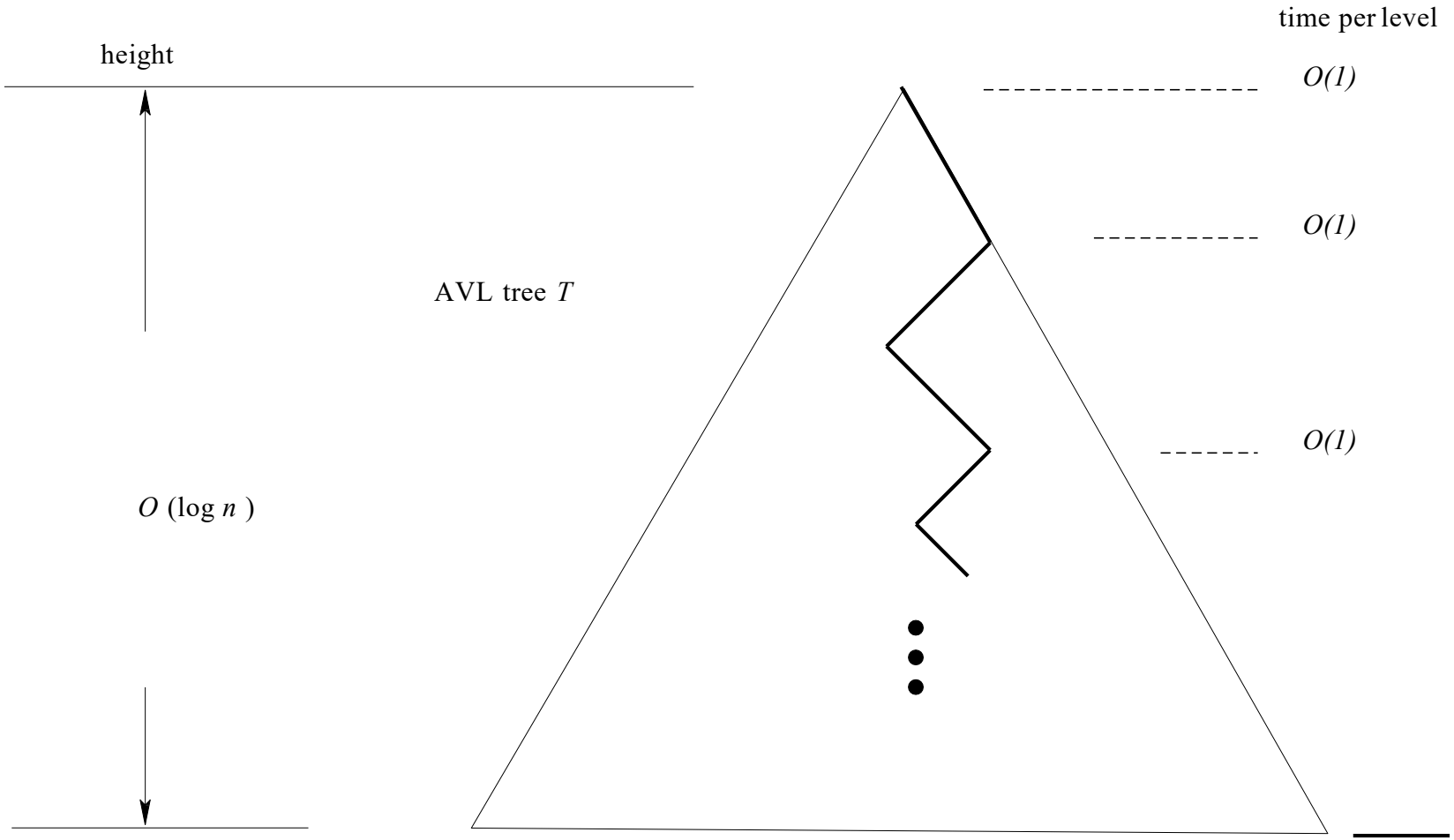



# AVL performance

- All operations (*search, insertion*, and *removal*) on an AVL tree with $n$ elements can be performed in $O(\log n)$ time.



# AVL performance

- A single restructure is O(1)

- Search is O(log n)
  - height of tree is O(log n), no restructures needed

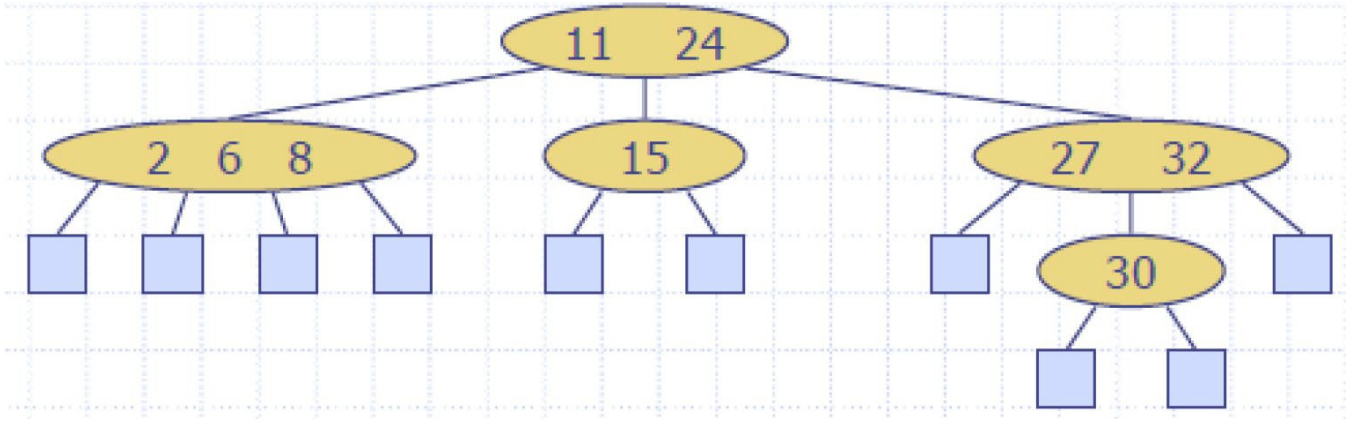
- Insertion is O(log n)
  - Initial search is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

- Removal is O(log n)
  - Initial search is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

# Multi-Way Search Tree

❖ A multi-way search tree is an ordered tree such that

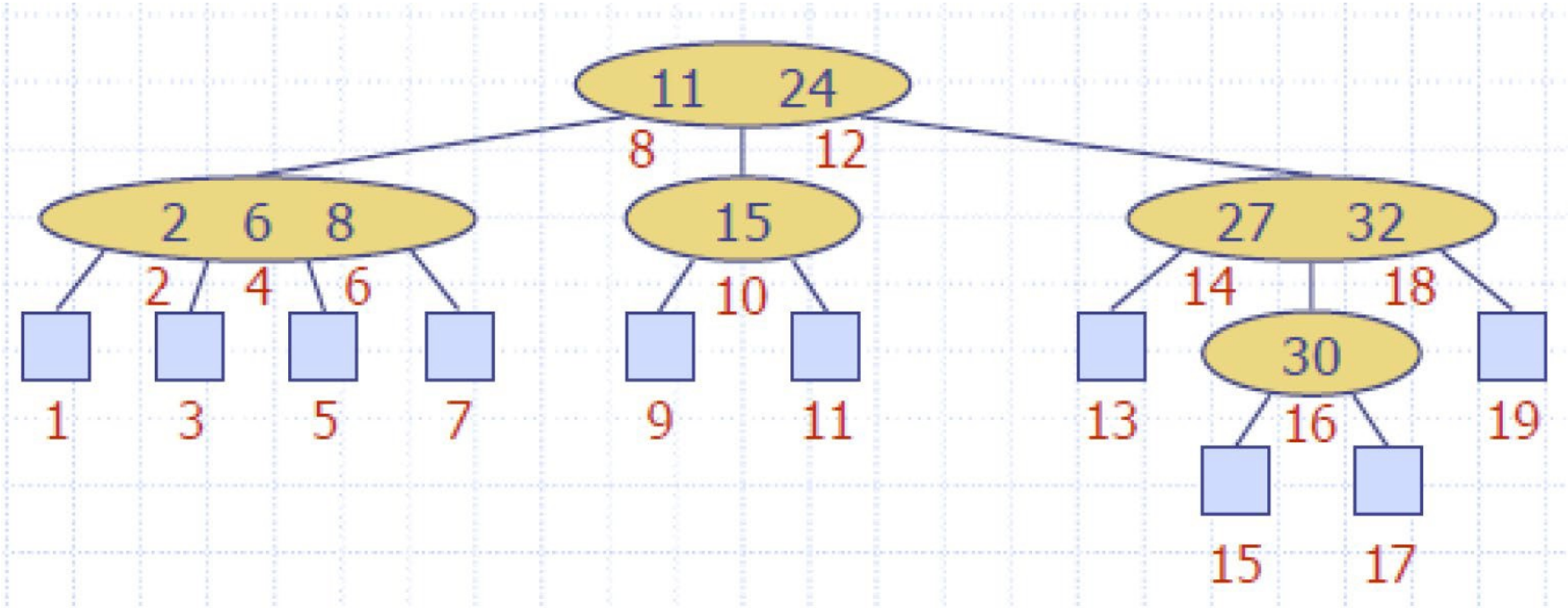
- Each internal node has at least two children and stores d-1 key-element items $(k_i, o_i)$ where d is the number of children
  - For a node with children $v_1, v_2, \ldots v_d$,storing keys $k_1, k_2, \ldots k_{d-1}$
    - keys in the subtree of $v_1$ are less than $k_1$
    - keys in the subtree of $v_i$ are between $k_{i-1}$and $k_i$ (i = 2, . .. , d - 1)
    - keys in the subtree of $v_d$ are greater than $k_{d-1}$
  - The leaves store no items and serve as placeholders



# Multi-Way Inorder Traversal

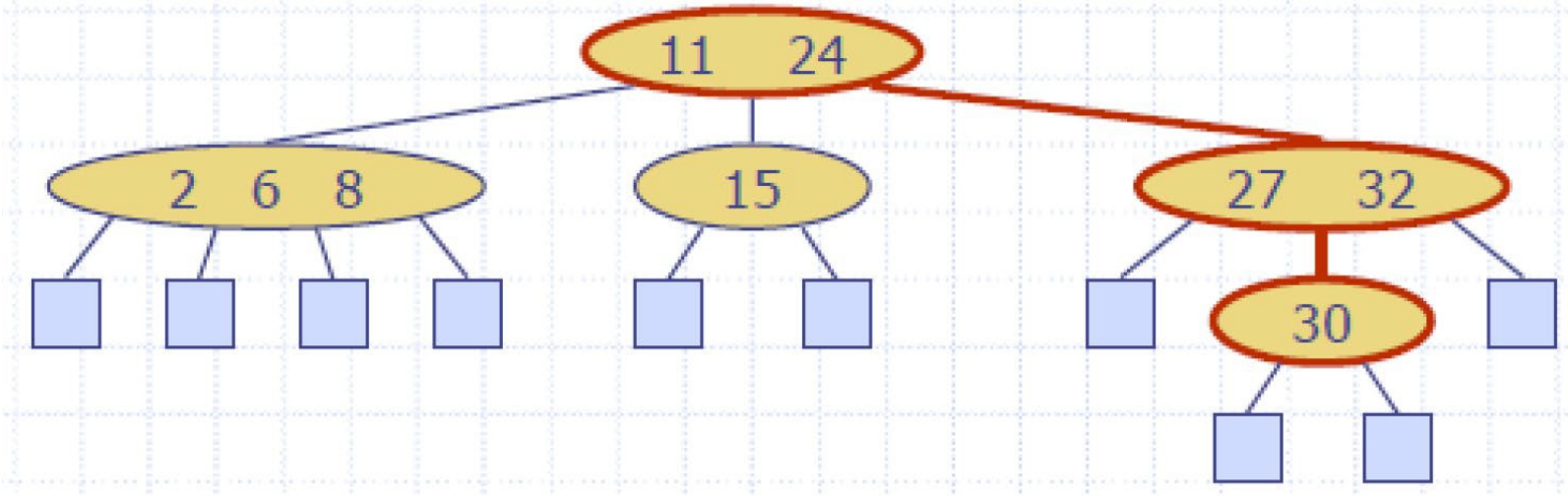
◆We can extend the notion of inorder traversal from binary trees to multi-way search trees

◆Namely, we visit item ($k_i$, $o_i$) of node v between the recursive traversals of the subtrees of $v_i$ rooted at children $v_i$ and $v_{i+1}$



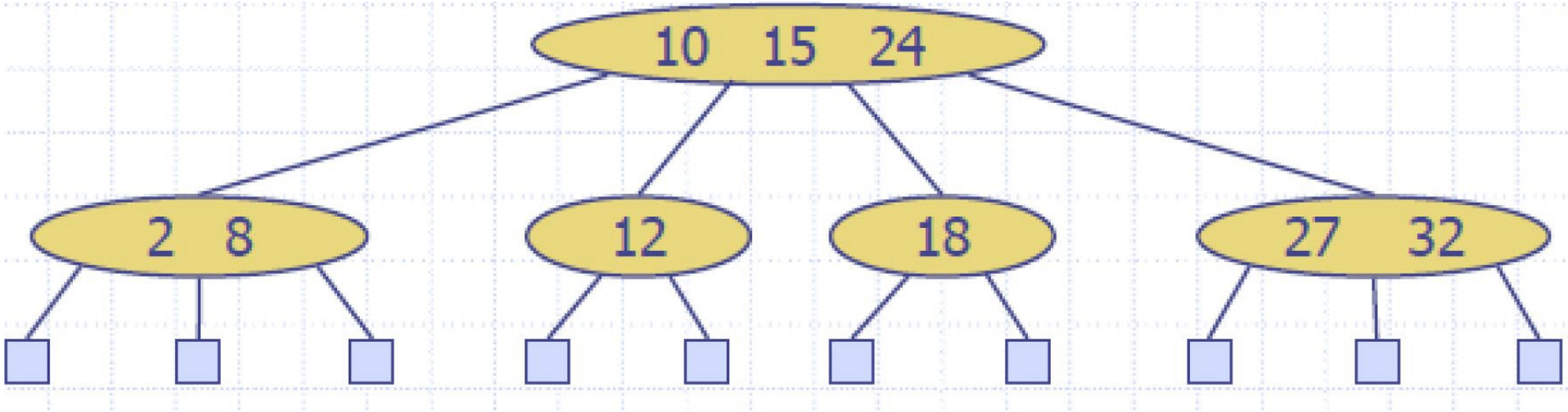
# Multi-Way Searching

- Similar to search in a binary search tree
- A each internal node with children $v_1, v_2, \ldots v_d$ and keys $k_1, k_2, \ldots k_{d-1}$
  - $k = k_i$ (i = 1, …, d-1) : the search terminates successfully
  - $k < k_1$ : we continue the search in child $v_1$
  - $k_{i-1} < k < k_i$ (i = 2, …, d-1): we continue the search in child $v_i$
  - $k > k_{d-1}$: we continue the search in child $v_d$
- Example: search for 30



# (2, 4) trees

- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node



# (2, 4) trees

❖ Theorem: A (2,4) tree storing n items has height *O(log n)*

Proof:

■ Let n be the height of a (2,4) tree with n items

■ Since there are at least $2^i$ items at depth i = 0, …, h-1 and no items at depth, h, we have

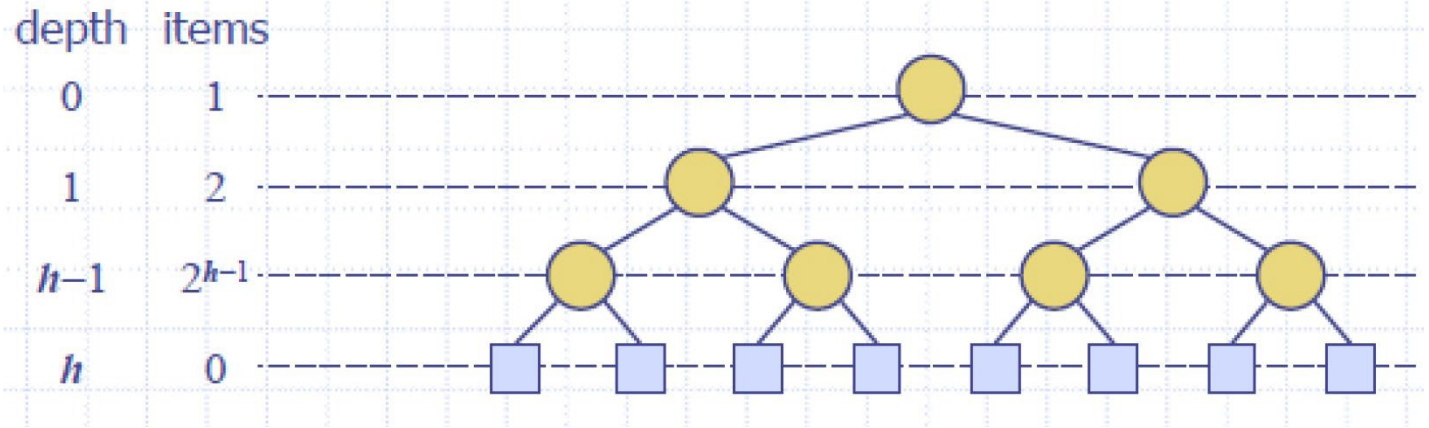
$$n \geq 1 + 2 + 4 + \cdots + 2^{h-1} = 2^h - 1$$

■ Thus, $h \leq \log(n+1)$

Sum of a Geometric Series

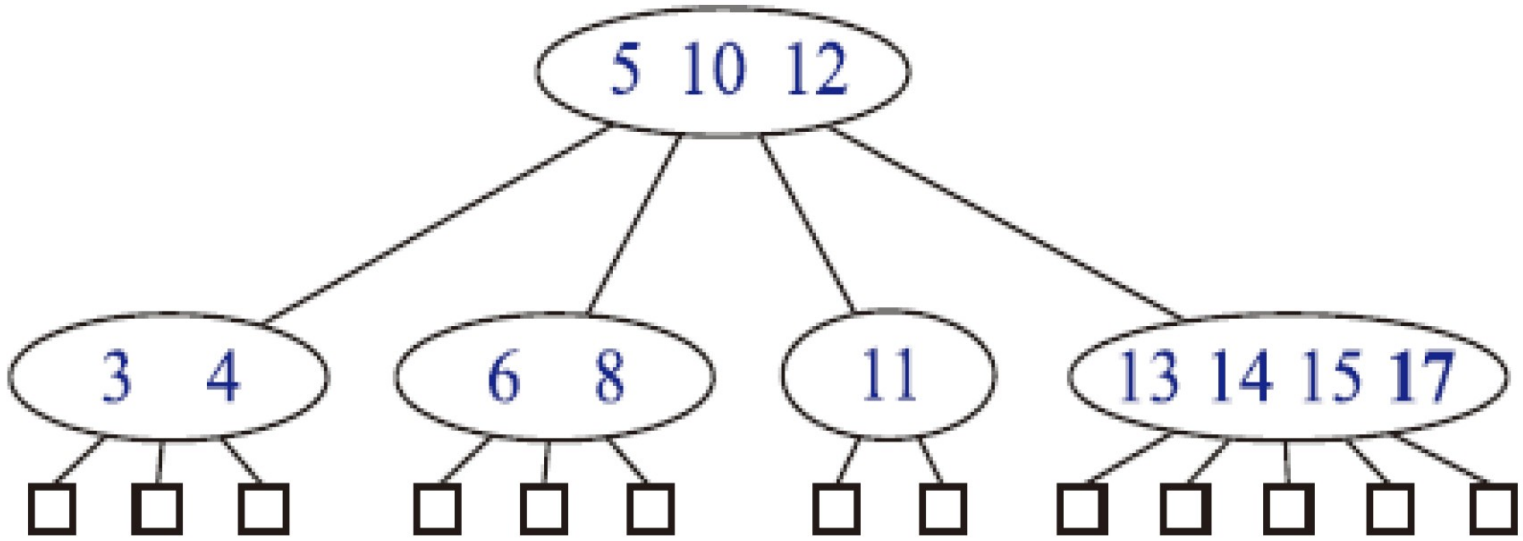
$$\sum_{k=0}^{n-1}(ar^k) = a\left(\frac{1-r^n}{1-r}\right)$$

❖Searching in a (2,4) tree with n items takes *O(log n)* time depth items



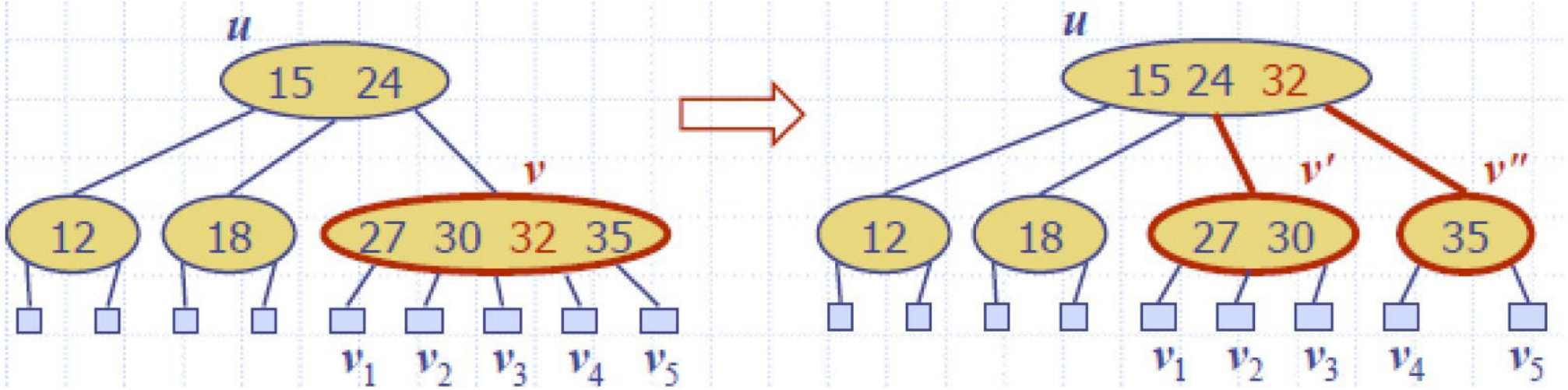
# (2, 4) tree - Insertion

- We insert a new item *(k, o)* at the parent v of the leaf reached by searching for k
  - We preserve the depth property but
  - We may cause an overflow (i.e., node v may become a 5-node)
- Example: inserting key 17 causes an overflow

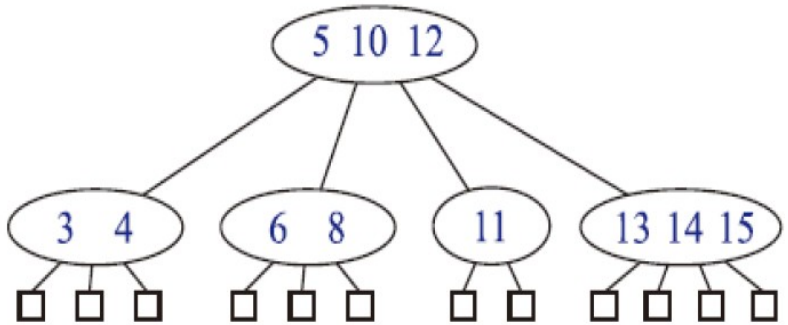


# (2, 4) trees - Split operation

- We handle an overflow at a 5-node v with a split operation:
  - let $v_1, v_2, \ldots v_5$ be the children of $v$ and $k_1 \ldots k_4$ be the keys of $v$
  - node $v$ replaced nodes $v'$ and $v''$
    - $v'$ is a 3-node with keys $k_1 k_2$ and children $v_1 v_2 v_3$
    - $v''$ is a 2-node with key $k_4$ and children $v_4 v_5$
  - key $k_3$ is inserted into the parent $u$ of $v$ (a new root may be created)
- The overflow may propagate to the parent node $u$

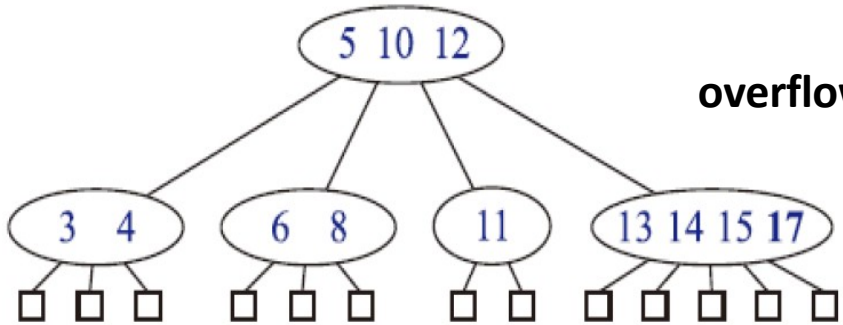


# (2, 4) tree - Insertion



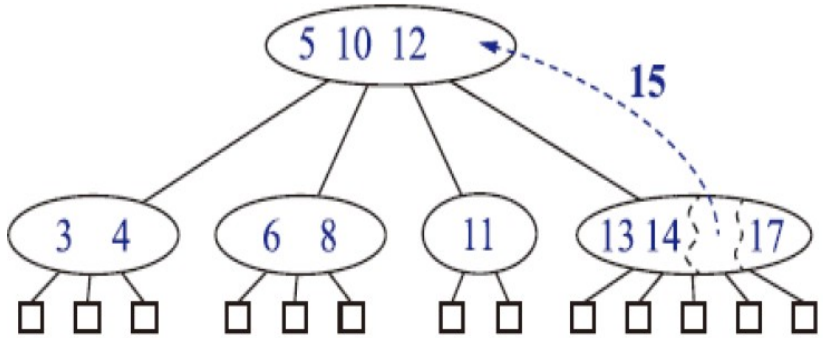

(a)

**overflow**



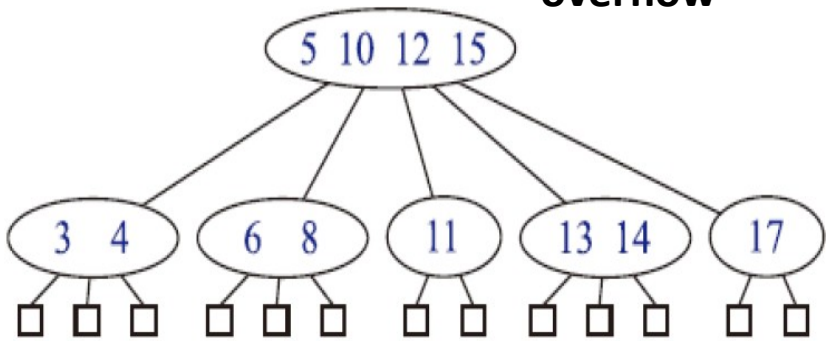

(b)

***node split***




(c)

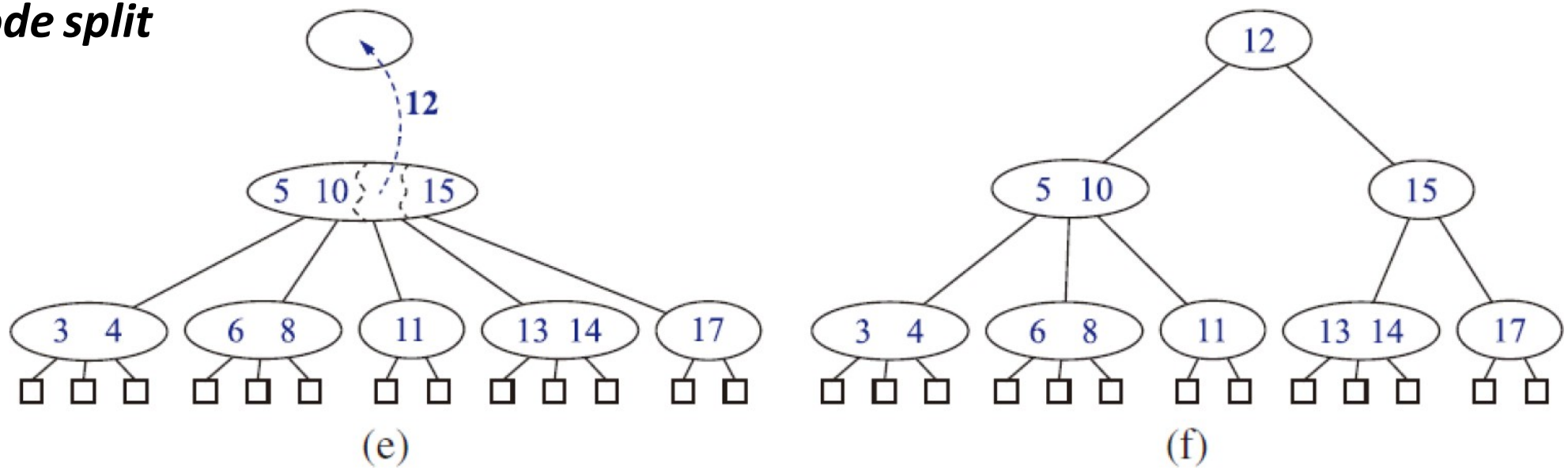
**overflow**




(d)

# (2, 4) tree - Insertion

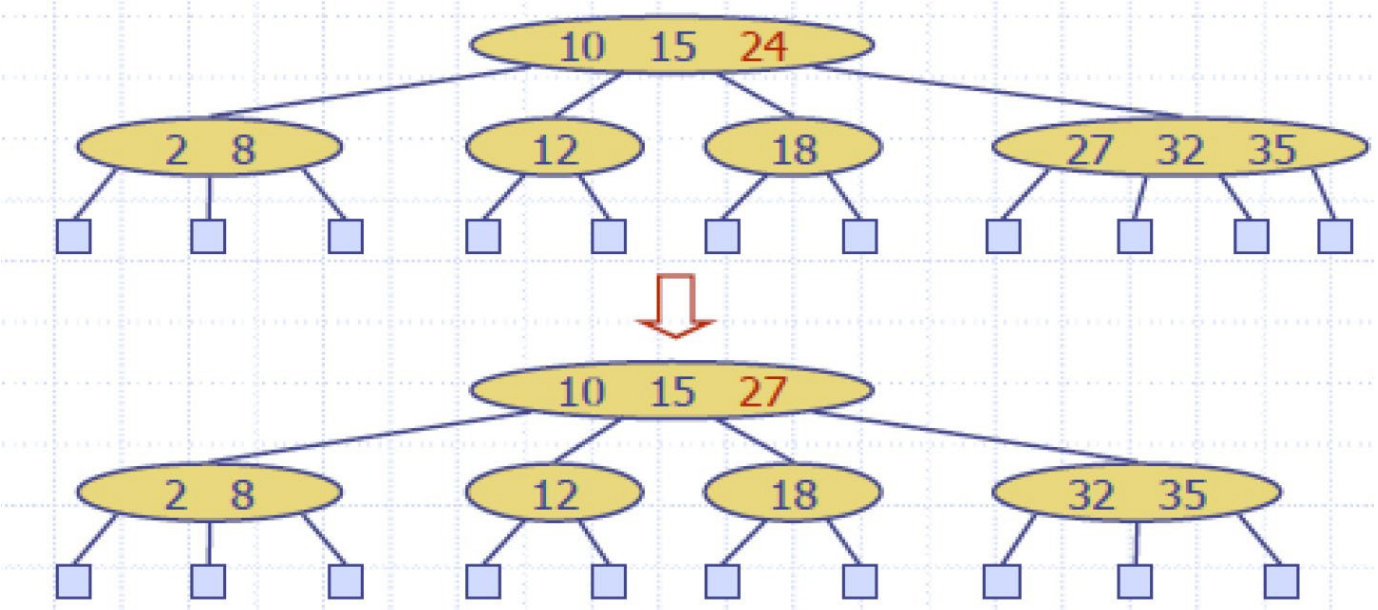
***node split***




A split operation either eliminates the overflow or propagates it into the parent of the current node.
Indeed, this propagation can continue all the way up to the root of the search tree.
But if it does propagate all the way to the root, it will finally be resolved at that point.

# (2, 4) trees - Deletion

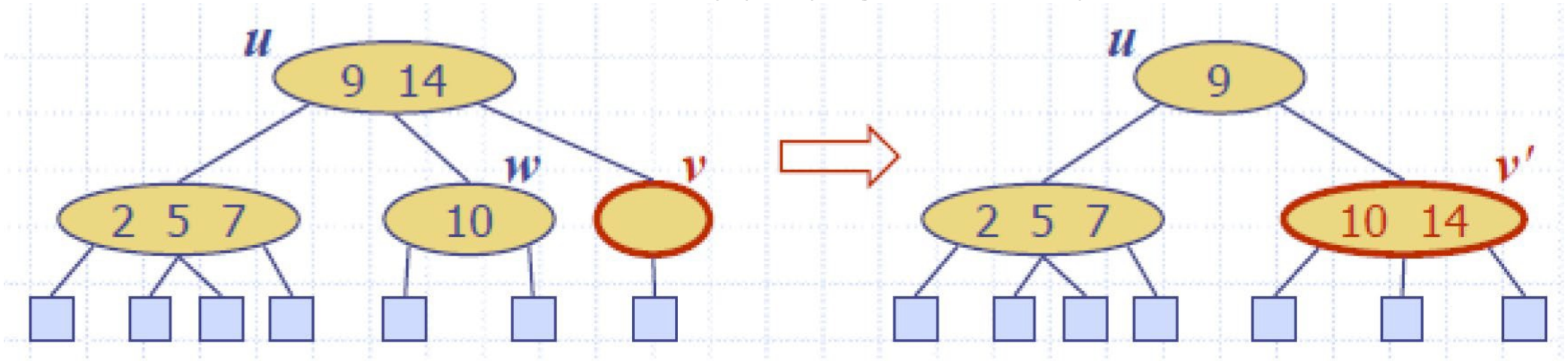
- We reduce deletion of an item to the case where the item is at the node with leaf children
- Otherwise, we replace the item with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter item
- Example: to delete key 24, we replace it with 27 (inorder successor)



# (2, 4) trees - Deletion

## Underflow and Fusion

- Deleting an item from a node $v$, may cause an **underflow**, where node $v$ becomes a 1-node with one child and no keys
- To handle an underflow at node $v$ with parent $u$, we consider two cases
- Case 1: the adjacent siblings of $v$ are 2-nodes
  - Fusion operation: we merge $v$ with an adjacent sibling $w$ and move an item from $u$ to the merged node $v'$
  - After a fusion, the underflow may propagate to the parent $u$



# (2, 4) trees - Deletion

## Underflow and Transfer

❖ Case 2: an adjacent sibling w of v is a 3-node or a 4-node

- Transfer operation
  1. we move a child of $w$ to $v$
  2. we move an item from $u$ to $v$
  3. we move an item from $w$ to $u$
- After a transfer, no underflow occurs