

Data Structures and Algorithms

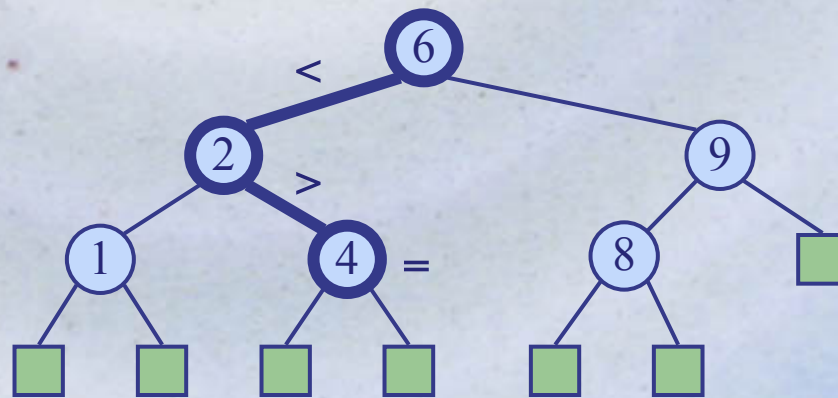
Search Trees



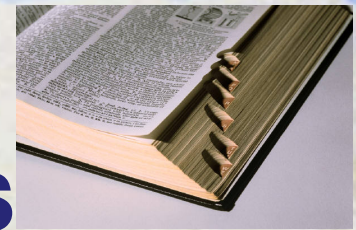
Outline

- Binary Search Trees
- AVL Trees
- (2,4) Trees

Binary Search Trees



Ordered Dictionaries

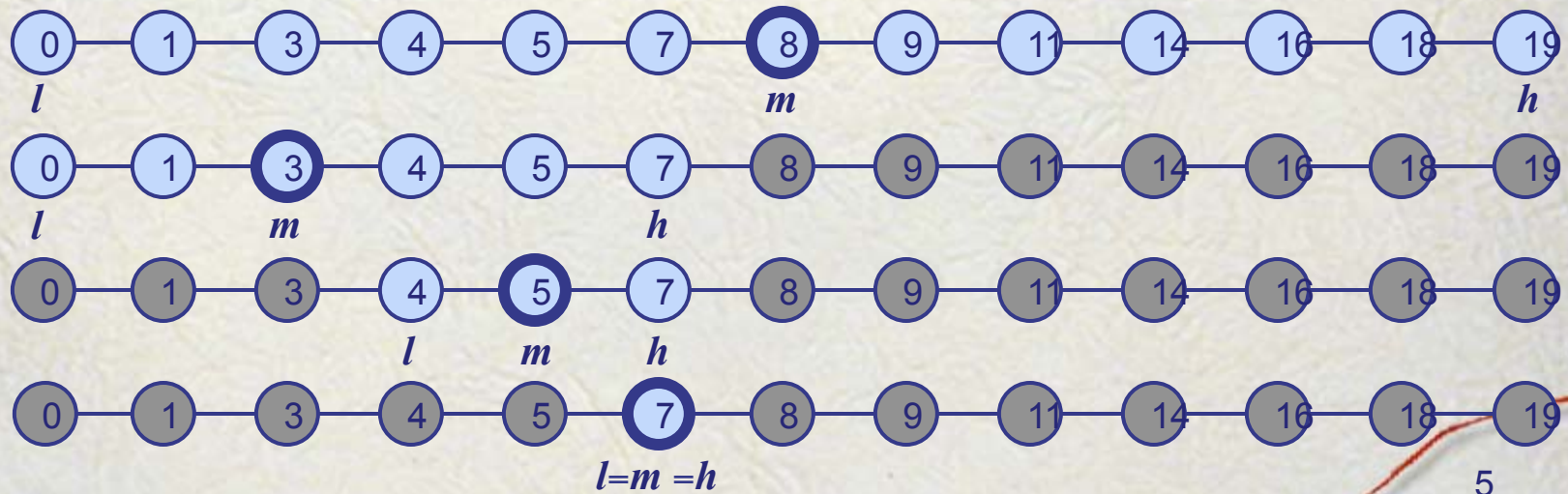


- Keys are assumed to come from a total order.
- New operations:
 - `first()`: first entry in the dictionary ordering
 - `last()`: last entry in the dictionary ordering
 - `successors(k)`: iterator of entries with keys greater than or equal to `k`; increasing order
 - `predecessors(k)`: iterator of entries with keys less than or equal to `k`; decreasing order

Binary Search



- Binary search can perform operation $\text{find}(k)$ on a dictionary implemented by means of an array-based sequence, sorted by key
 - similar to the high-low game
 - at each step, the number of candidate items is halved
 - terminates after $O(\log n)$ steps
- Example: $\text{find}(7)$

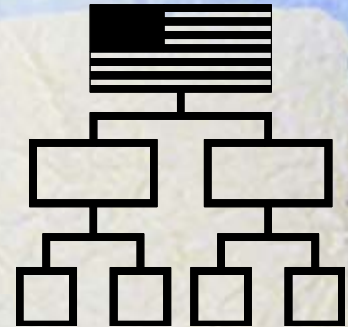


Search Tables



- A search table is a dictionary implemented by means of a sorted sequence
 - We store the items of the dictionary in an array-based sequence, sorted by key
 - We use an external comparator for the keys
- Performance:
 - find takes $O(\log n)$ time, using binary search
 - insert takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
 - remove take $O(n)$ time since in the worst case we have to shift $n-1$ items to compact the items after the removal
- The lookup table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search Trees

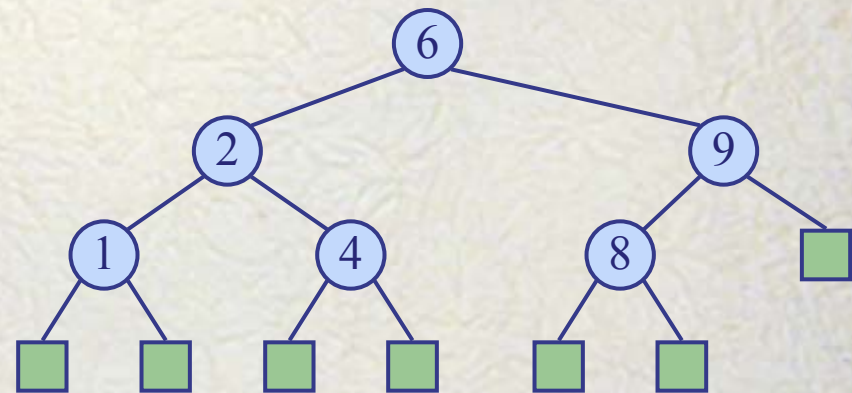


- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:

- Let u , v , and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v . We have
 $key(u) \leq key(v) \leq key(w)$

- External nodes do not store items

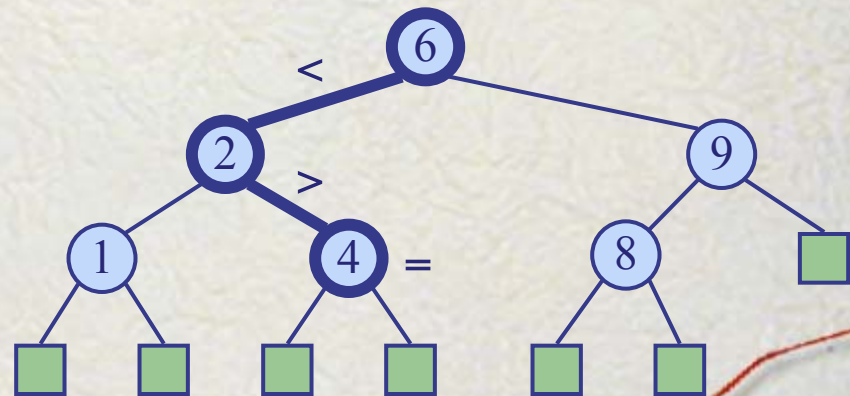
- An inorder traversal of a binary search tree visits the keys in increasing order



Search

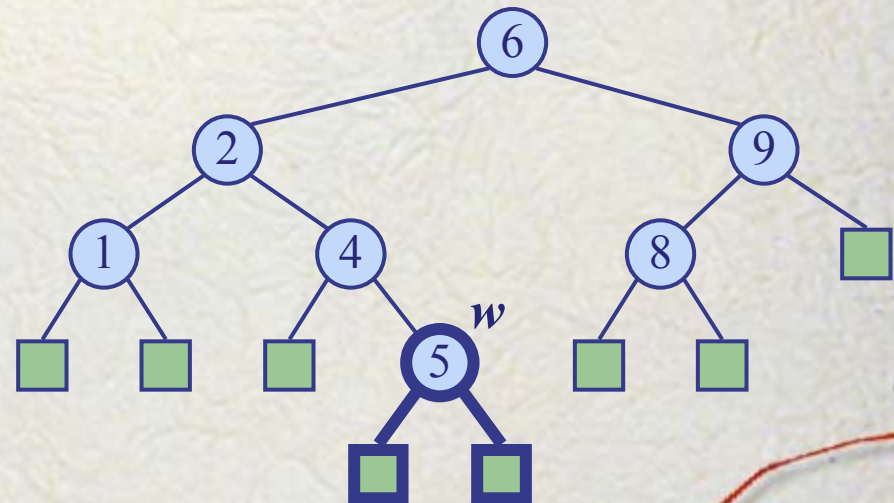
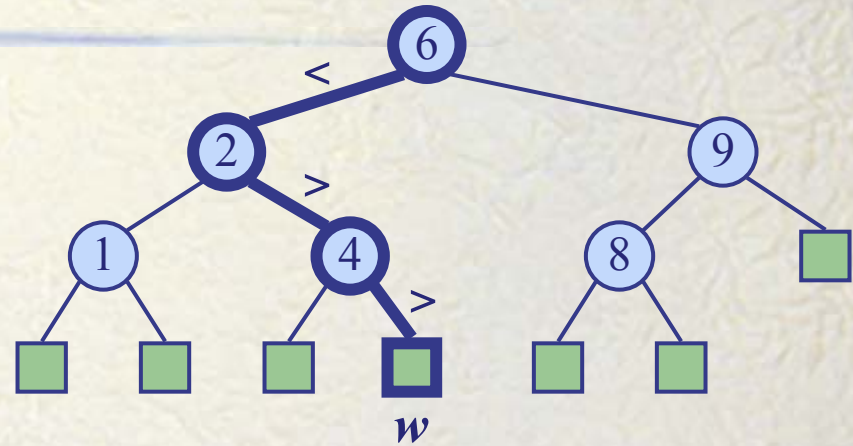
- To search for a key k , we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return null
- Example: find(4):
 - Call `TreeSearch(4,root)`

```
Algorithm TreeSearch( $k, v$ )  
  if T.isExternal( $v$ )  
    return  $v$  //  $v$  has null value  
  if  $k < \text{key}(v)$   
    return TreeSearch( $k, T.\text{left}(v)$ )  
  else if  $k = \text{key}(v)$   
    return  $v$   
  else {  $k > \text{key}(v)$  }  
    return TreeSearch( $k, T.\text{right}(v)$ )
```



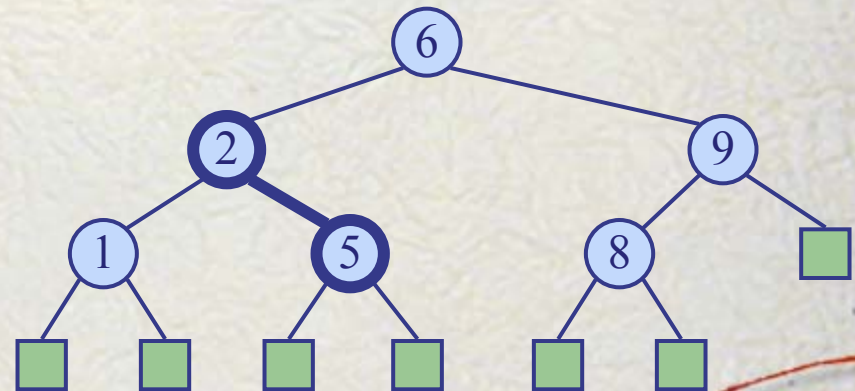
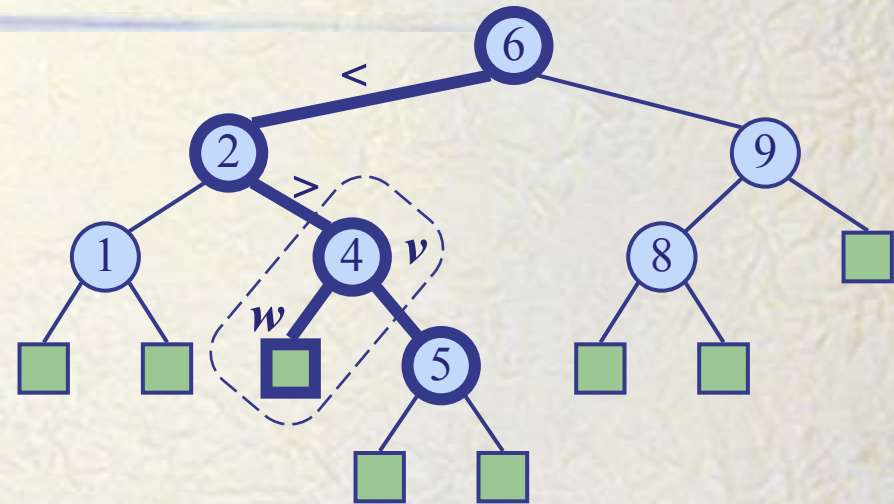
Insertion

- To perform operation $\text{insert}(k, o)$, we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



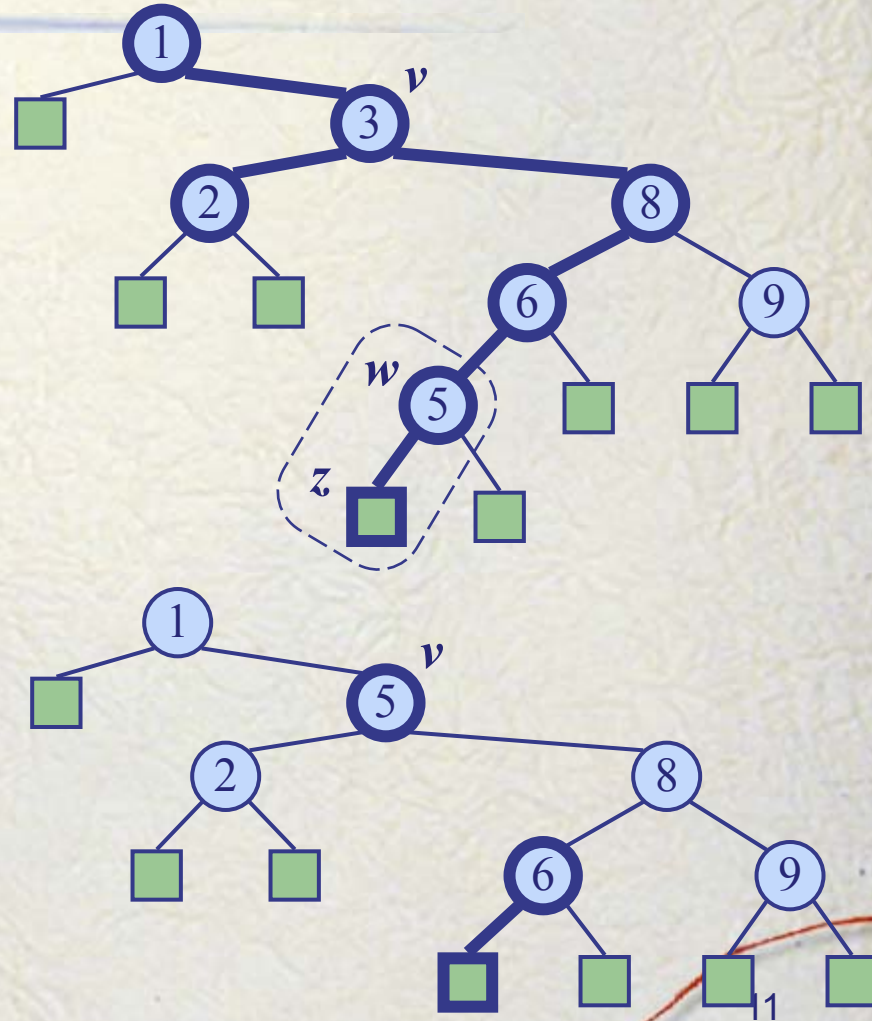
Deletion

- To perform operation $\text{remove}(k)$, we search for key k
- Assume key k is in the tree, and let v be the node storing k
- If node v has a **leaf child** w , we remove v and w from the tree with operation $\text{removeExternal}(w)$, which removes w and its parent
- Example: remove 4



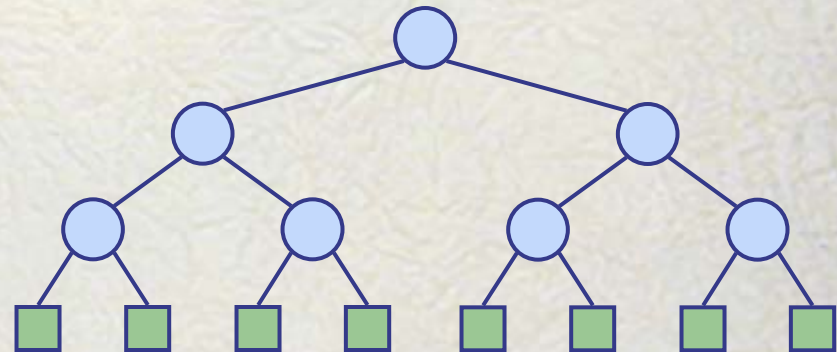
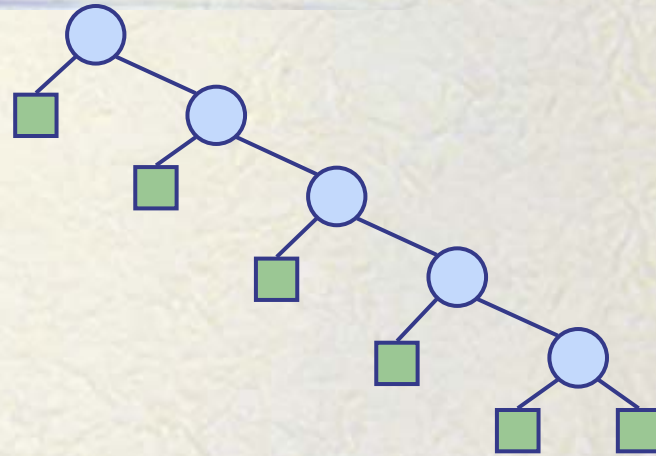
Deletion (cont.)

- We consider the case where the key k to be removed is stored at a node v whose children are both internal
 - we find the internal node w that follows v in an inorder traversal
 - we copy $key(w)$ into node v
 - we remove node w and its left child z (which must be a leaf) by means of operation `removeExternal(z)`
- Example: remove 3

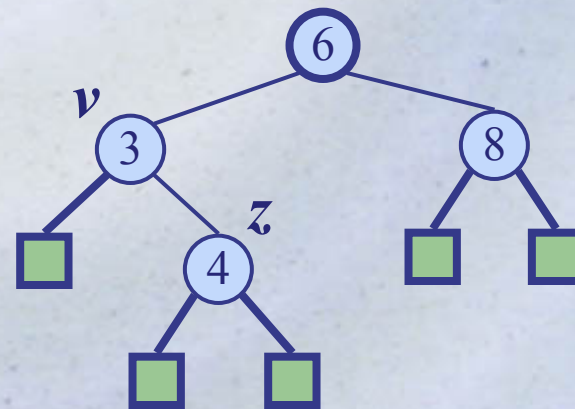


Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
 - the space used is $O(n)$
 - methods find, insert and remove take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case

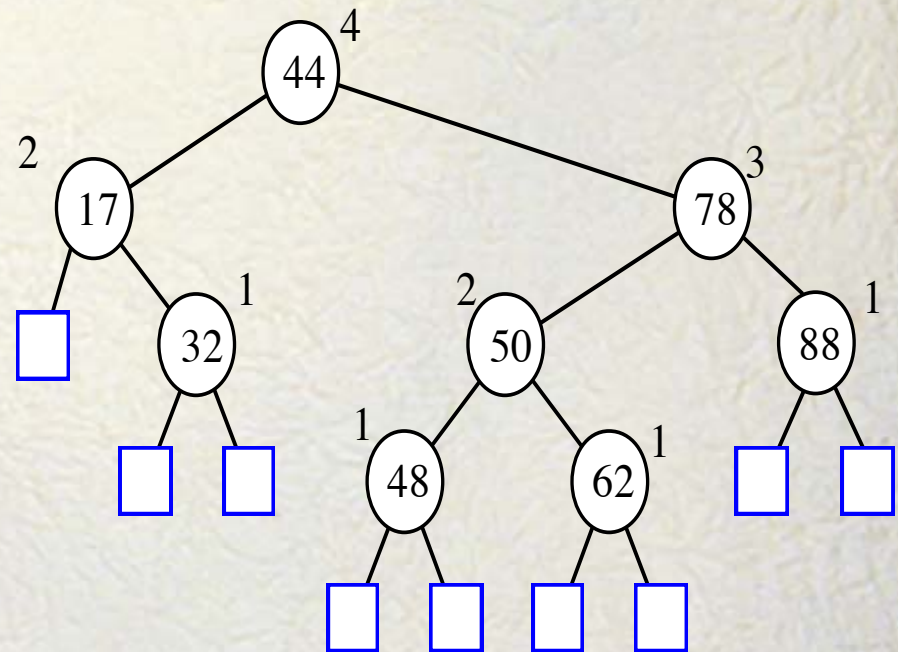


AVL Trees



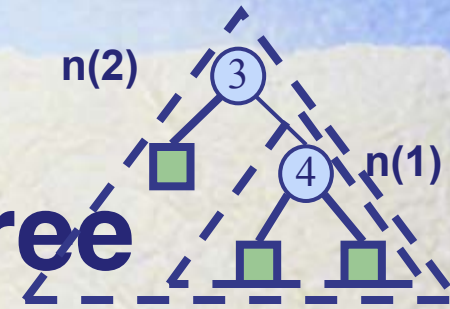
AVL Tree Definition

- **AVL trees are balanced.**
- An AVL Tree is a *binary search tree* such that for every internal node v of T , the *heights of the children of v can differ by at most 1.*



An example of an AVL tree where the heights are shown next to the nodes:

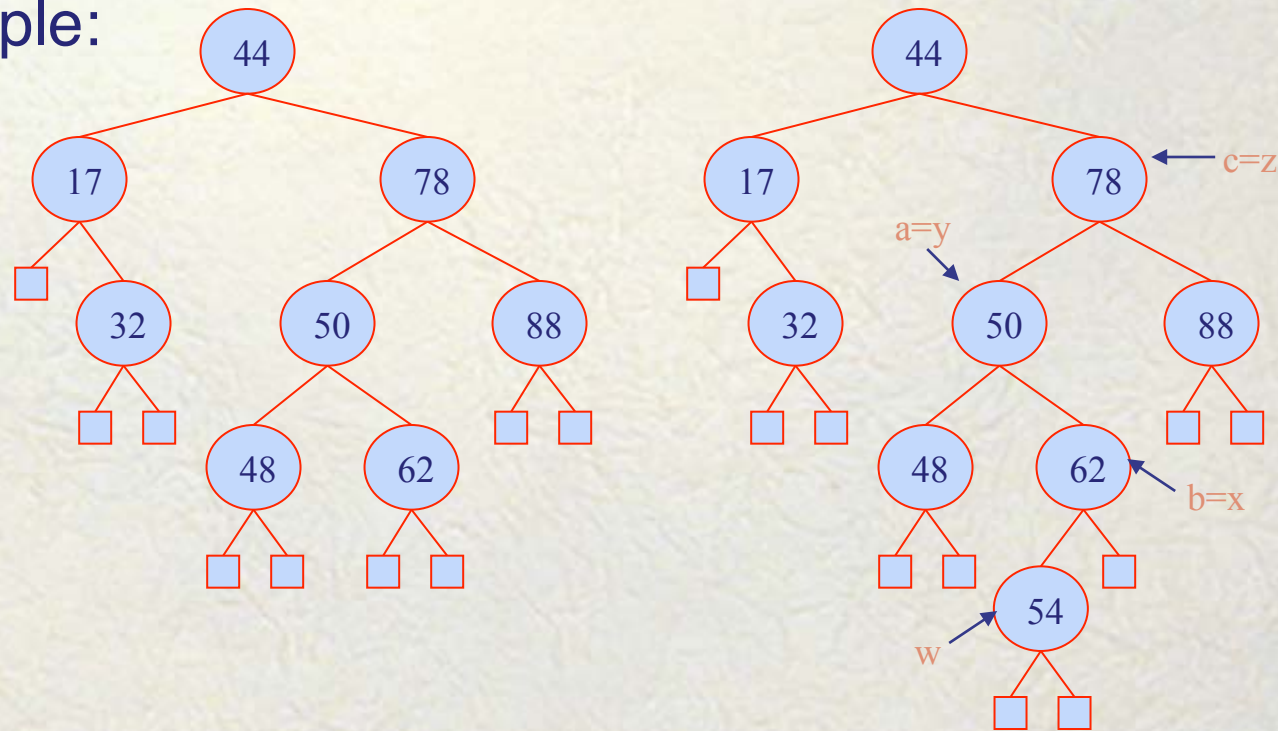
Height of an AVL Tree



- **Fact:** The *height* of an AVL tree storing n keys is $O(\log n)$.
- **Proof:** Let us bound $n(h)$: 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
 $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(h-6)$, ... (by induction),
 $n(h) > 2^i n(h-2i)$
- Solving the base case we get: $n(h) > 2^{h/2-1}$
- Taking logarithms: $h < 2\log n(h) + 2$
- Thus the height of an AVL tree is $O(\log n)$

Insertion in an AVL Tree

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:

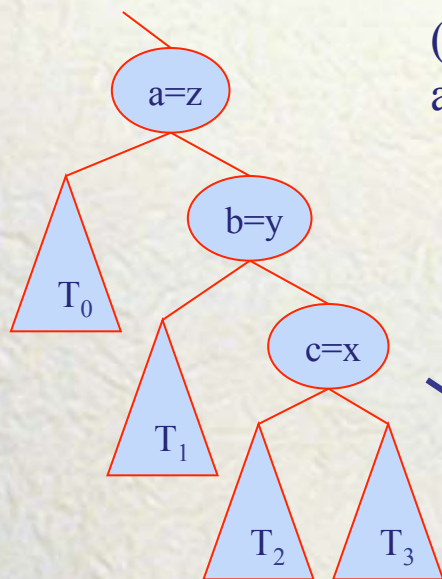


before insertion

after insertion

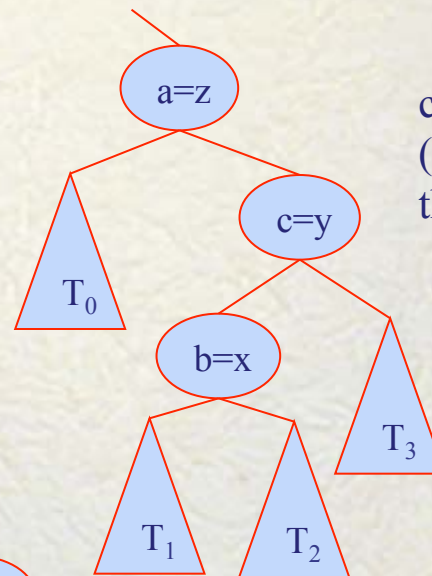
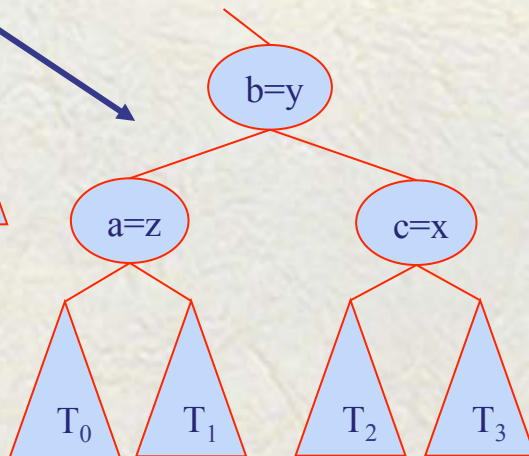
Trinode Restructuring

- let (a,b,c) be an inorder listing of x, y, z
- perform the rotations needed to make b the topmost node of the three

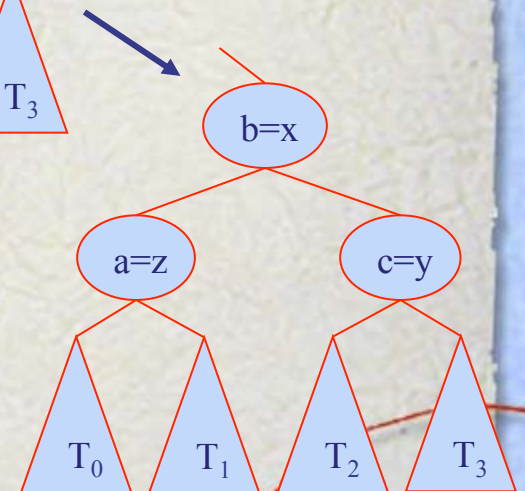


case 1: single rotation
(a left rotation about a)

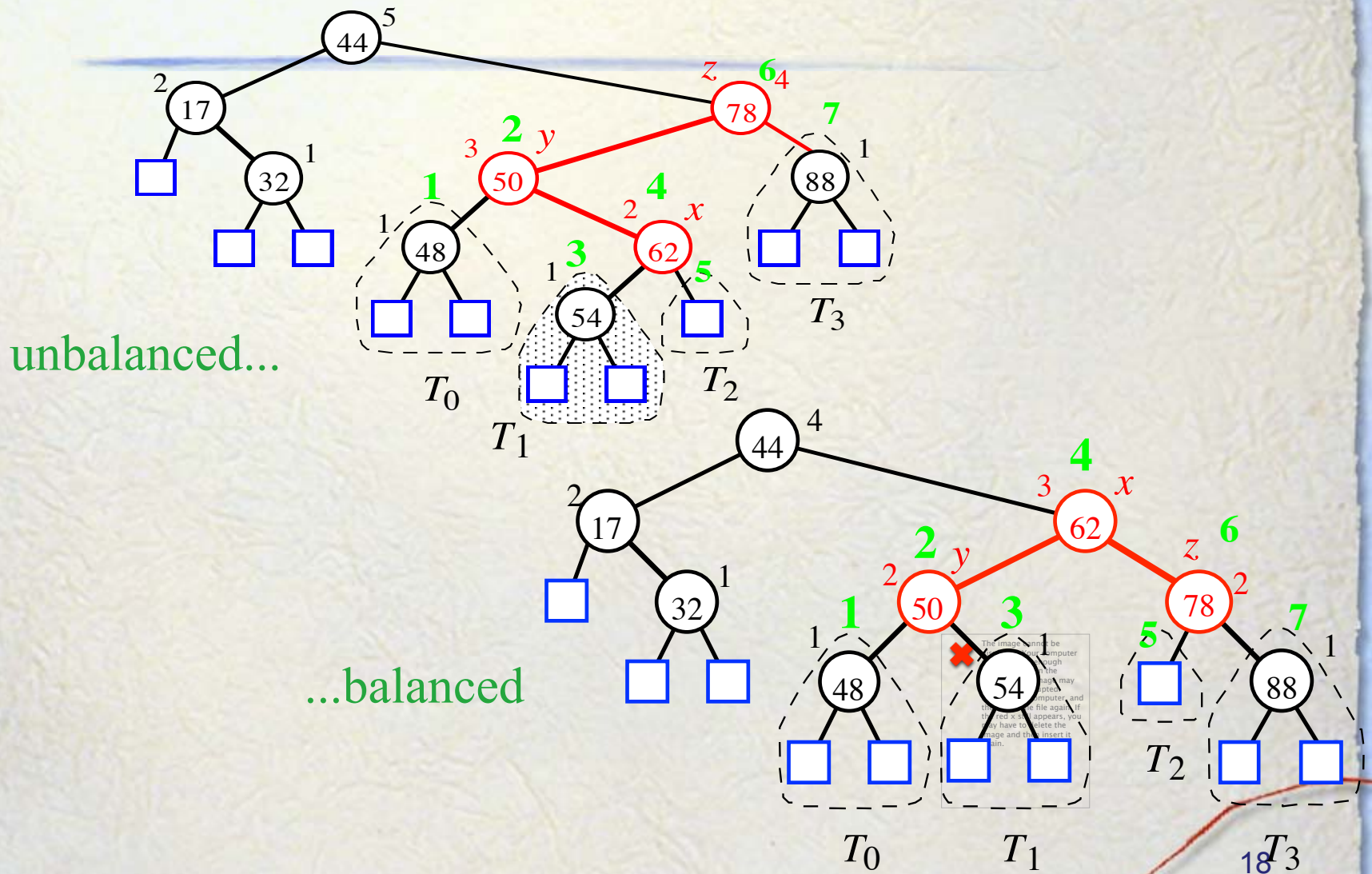
(other two cases
are symmetrical)



case 2: double rotation
(a right rotation about c ,
then a left rotation about a)

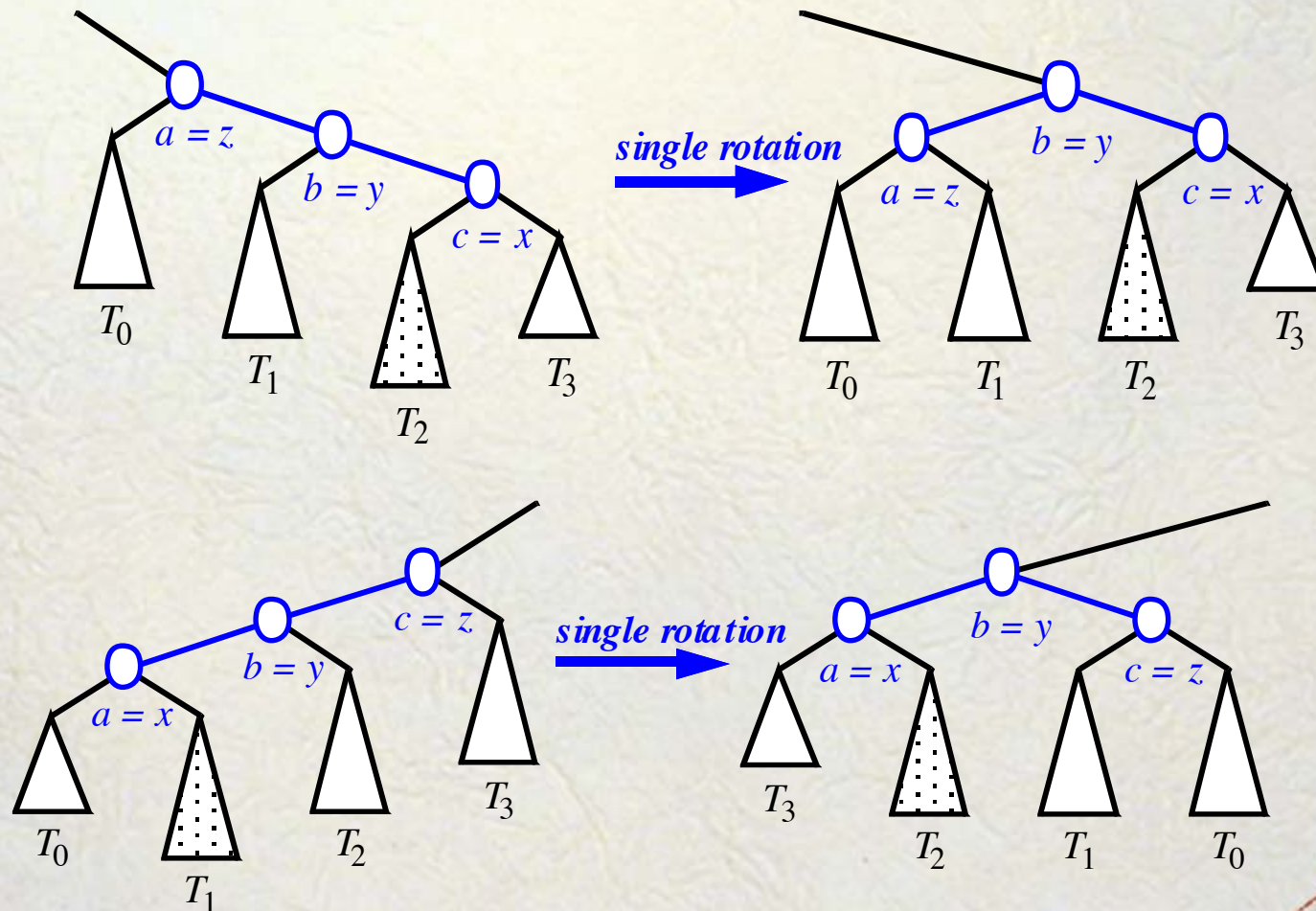


Insertion Example, continued



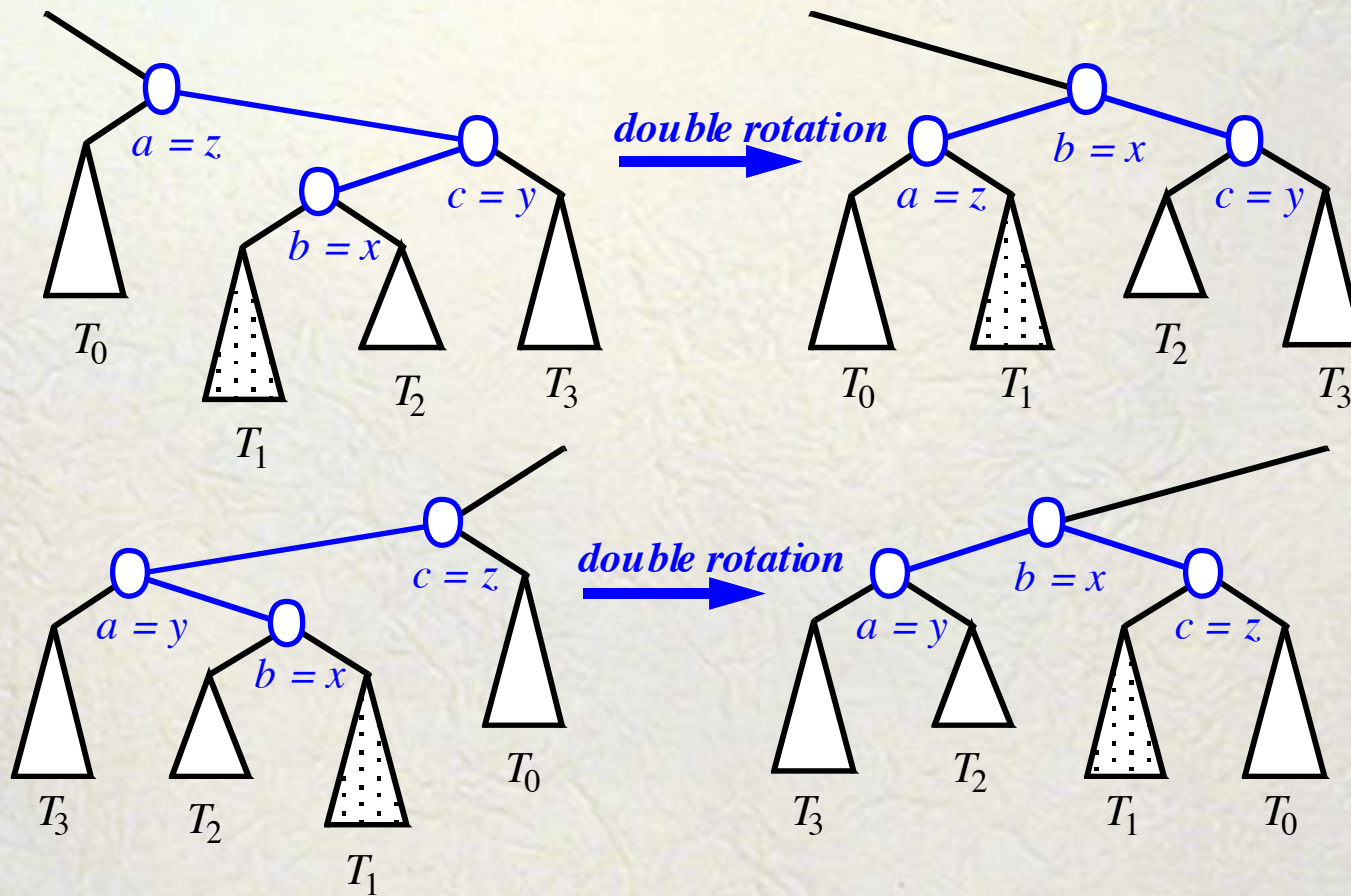
Restructuring (as Single Rotations)

- Single Rotations:



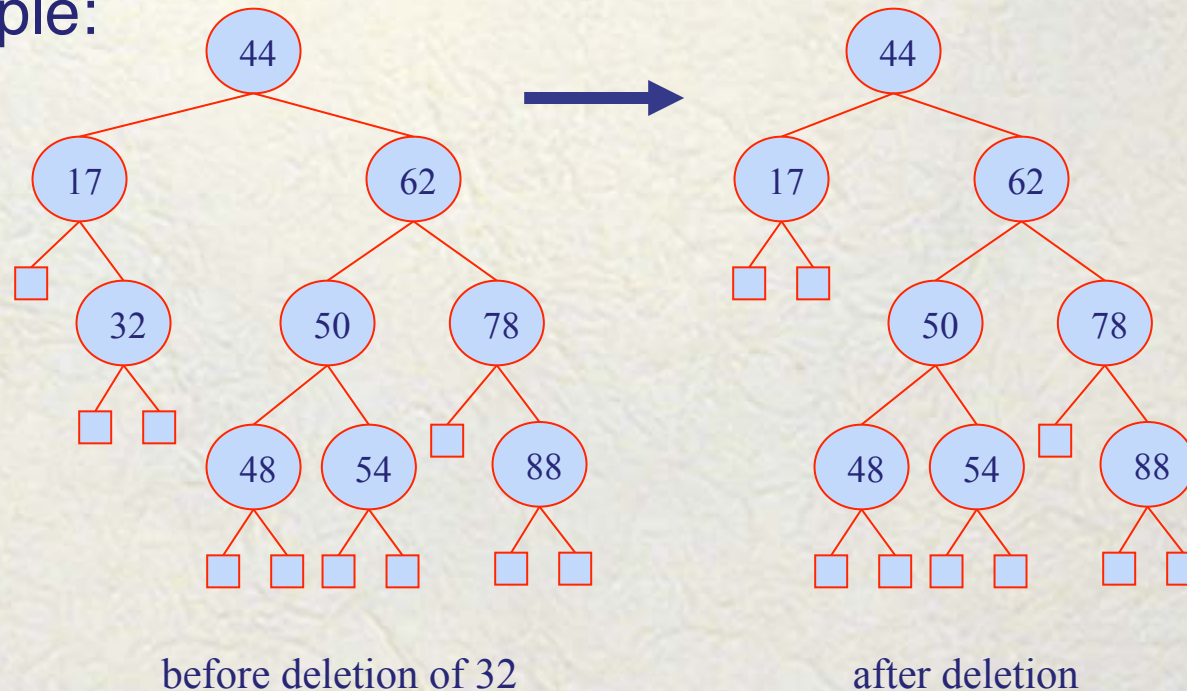
Restructuring (as Double Rotations)

- double rotations:



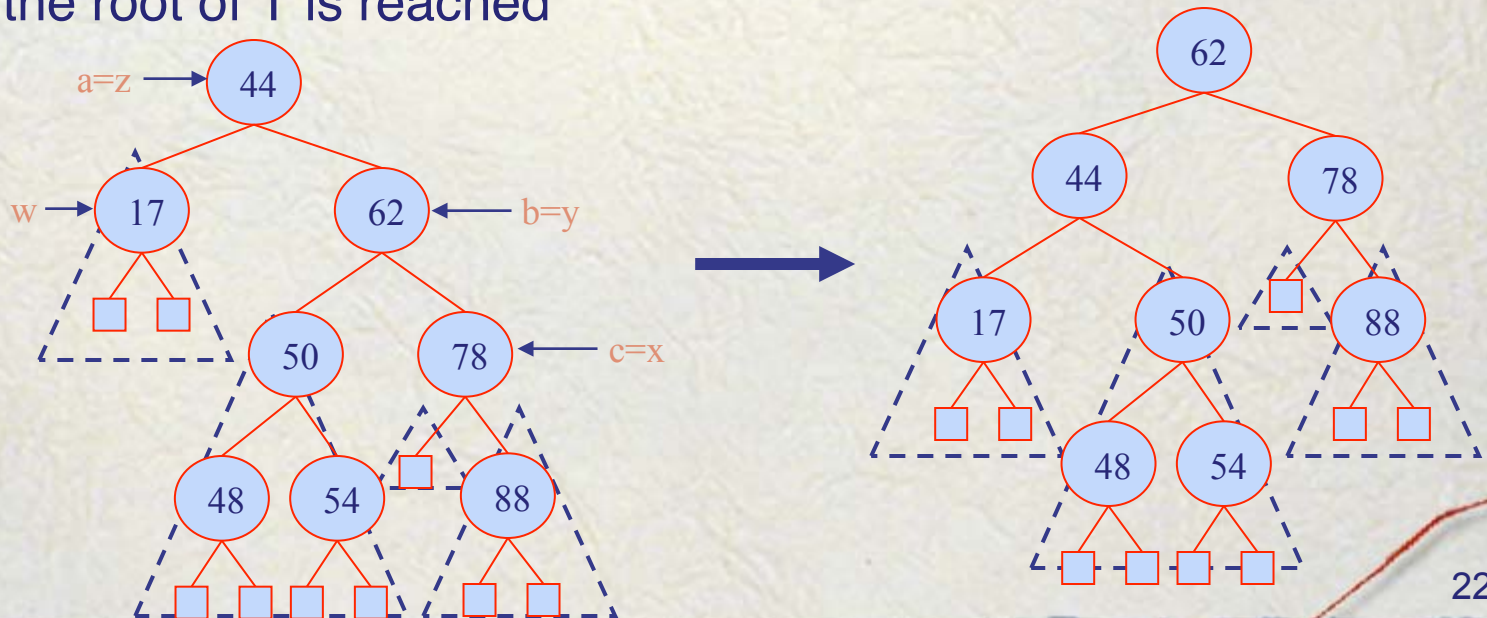
Removal in an AVL Tree

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:



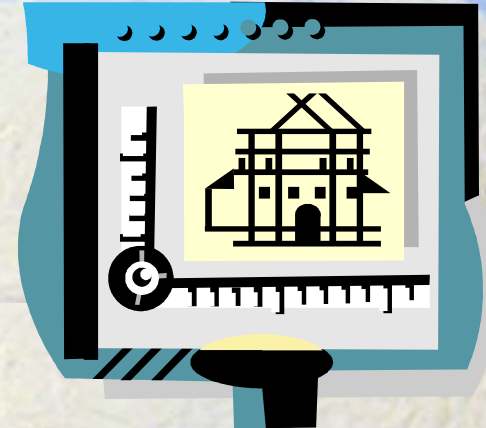
Rebalancing after a Removal

- Let z be the **first unbalanced** node encountered while travelling up the tree from w . Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.
- We perform **restructure**(x) to restore balance at z .
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached

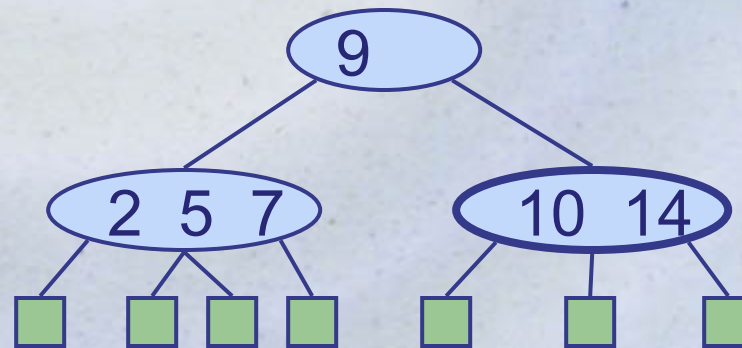


Running Times for AVL Trees

- 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 $O(\log n)$

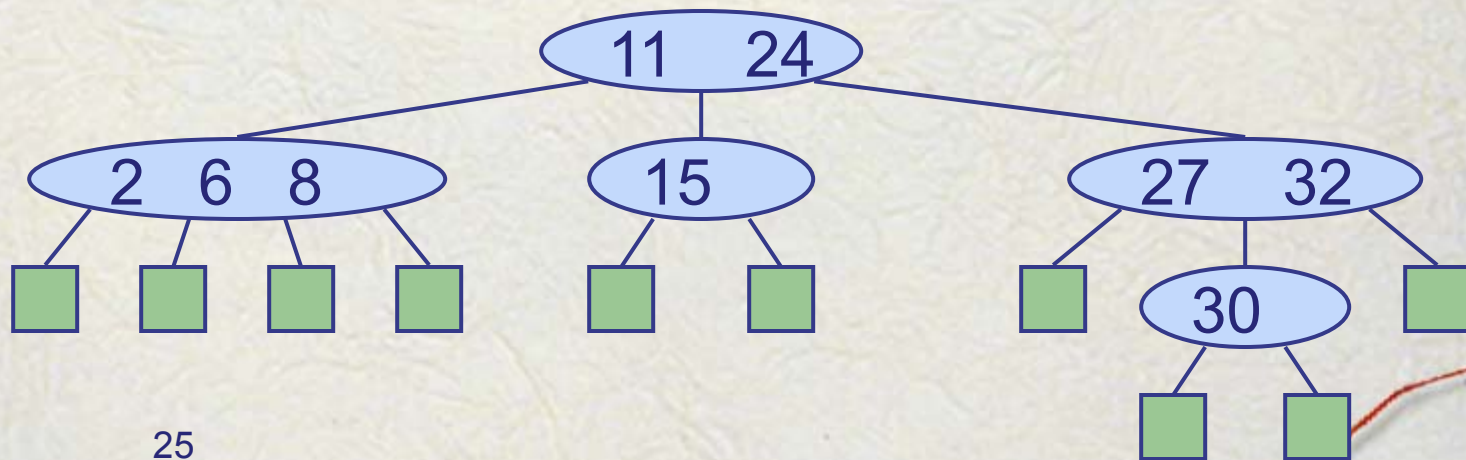


(2,4) Trees



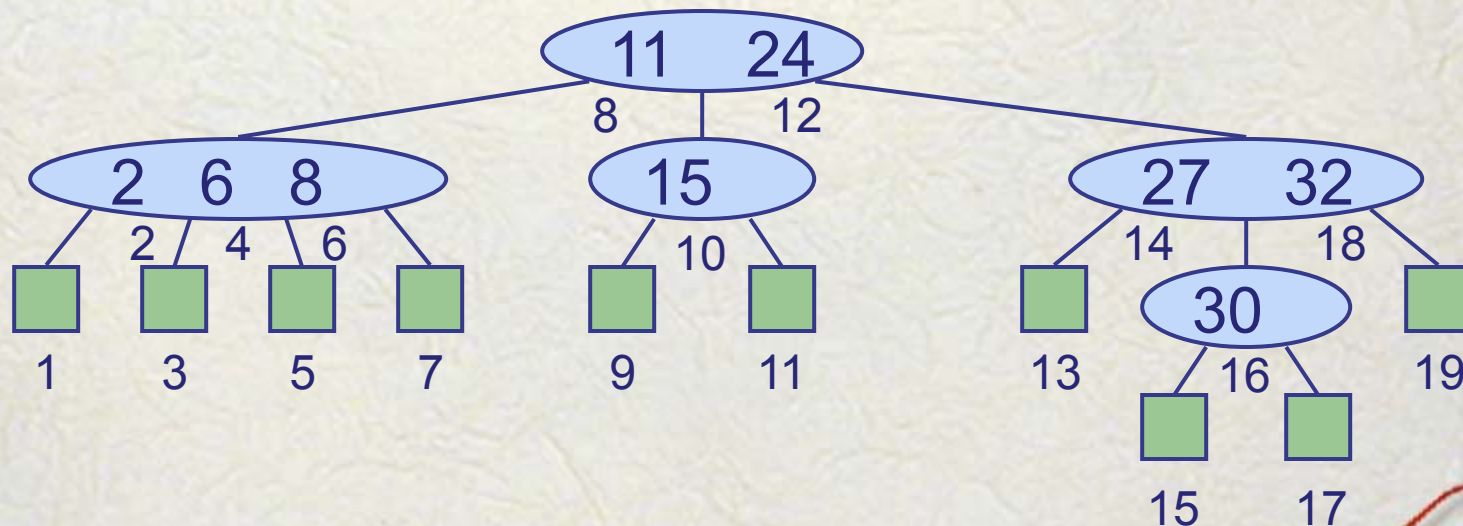
Multi-Way Search Tree

- 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 $v_1 v_2 \dots v_d$ storing keys $k_1 k_2 \dots k_{d-1}$
 - 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 ($i = 2, \dots, d - 1$)
 - keys in the subtree of v_d are greater than k_{d-1}
 - The leaves store no items and serve as placeholders



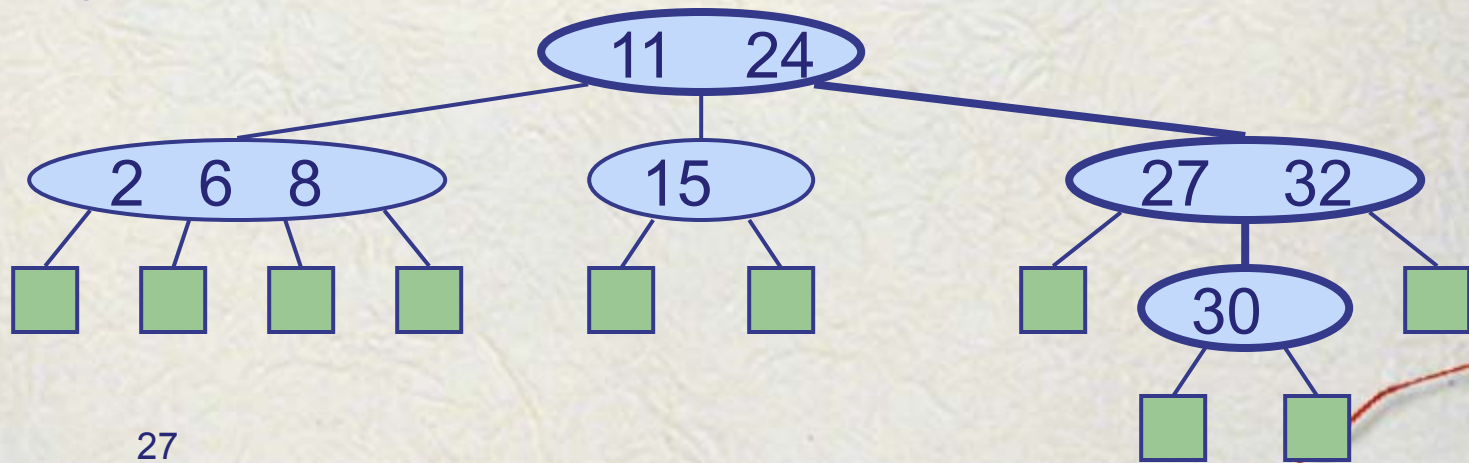
Multi-Way Inorder Traversal

- 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 rooted at children v_i and v_{i+1}
- An inorder traversal of a multi-way search tree visits the keys in increasing order



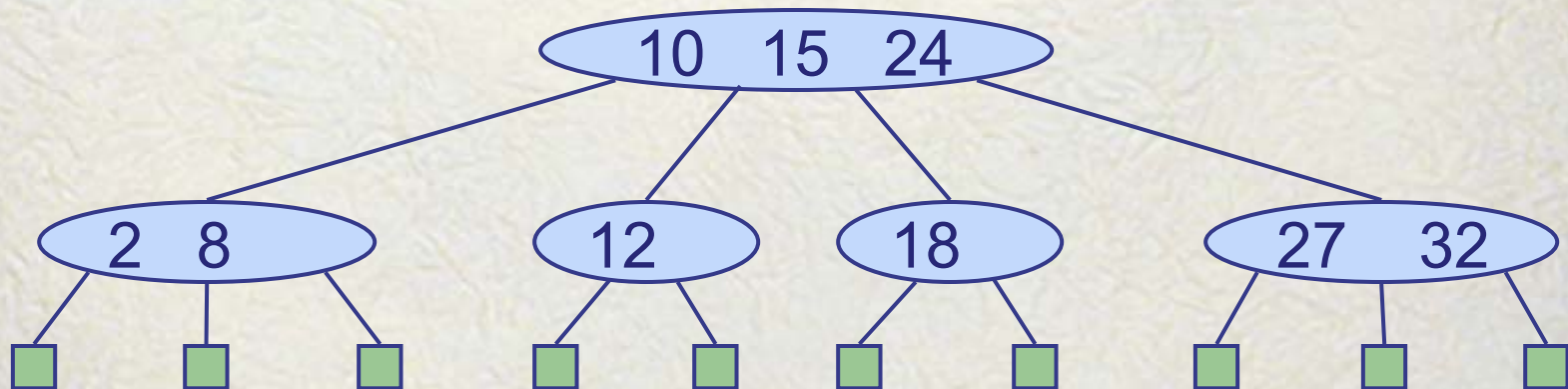
Multi-Way Searching

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



Height of a (2,4) Tree

- Theorem: A (2,4) tree storing n items has height $O(\log n)$

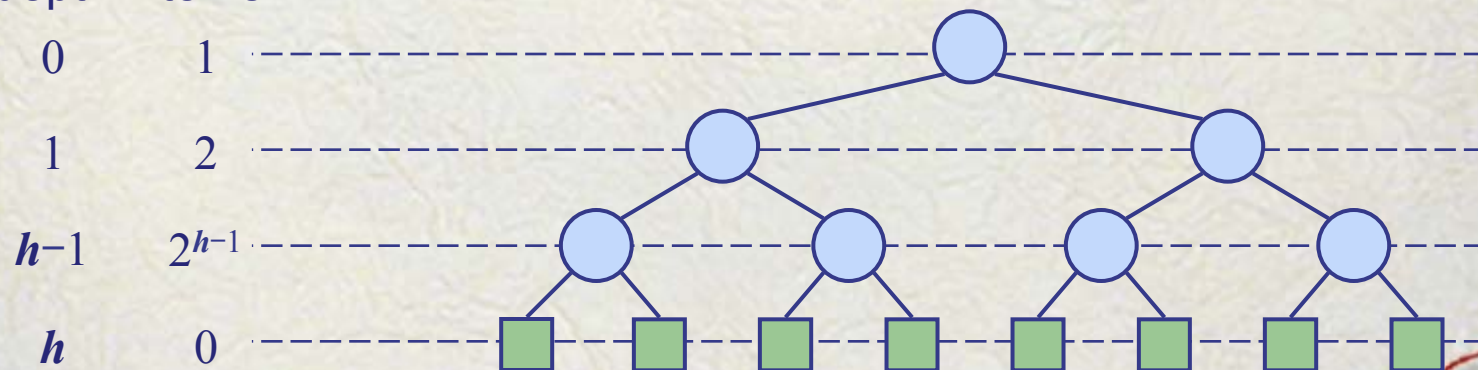
Proof:

- Let h be the height of a (2,4) tree with n items
- Since there are at least 2^i items at depth $i = 0, \dots, h-1$ and no items at depth h , we have

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

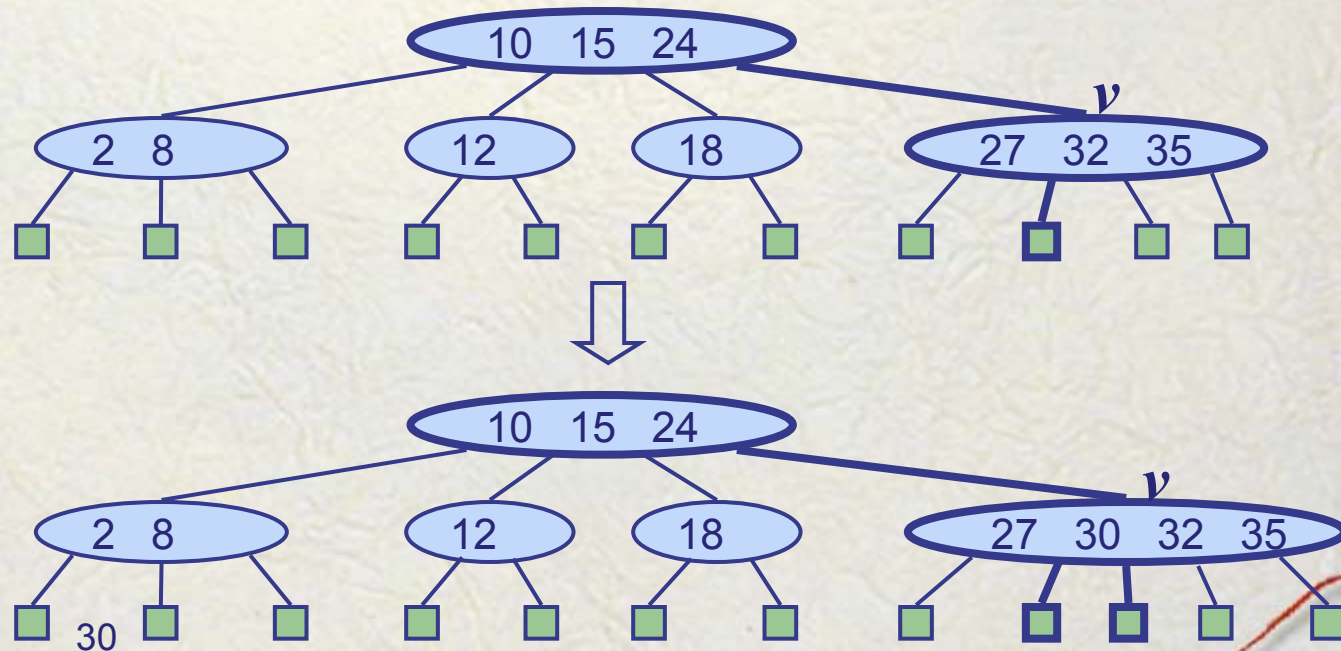
- Thus, $h \leq \log(n + 1)$
- Searching in a (2,4) tree with n items takes $O(\log n)$ time

depth items



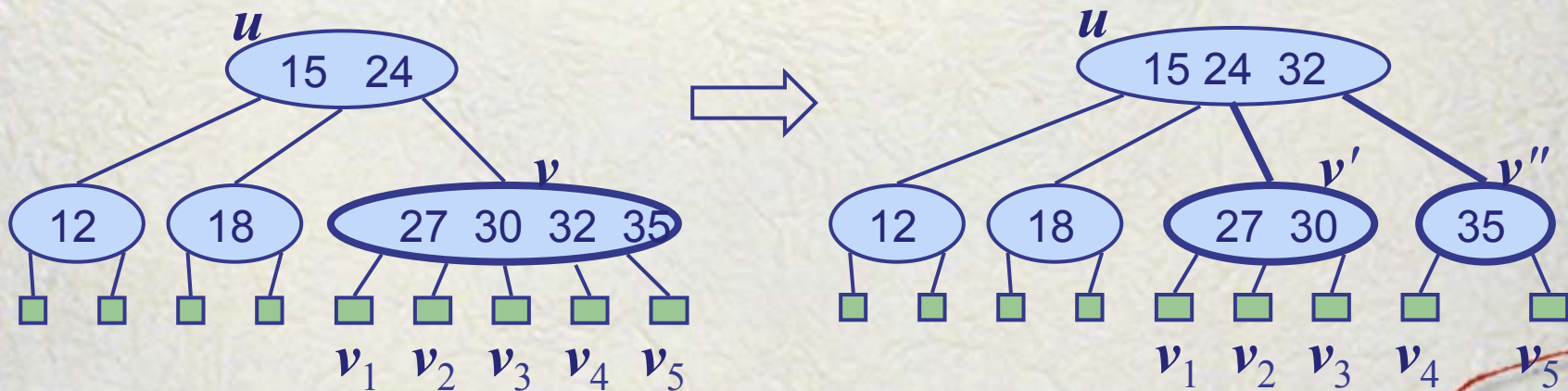
Insertion

- We insert a new item (k, o) at the parent ν of the leaf reached by searching for k
 - We preserve the depth property but
 - We may cause an overflow (i.e., node ν may become a 5-node)
- Example: inserting key 30 causes an overflow



Overflow and Split

- We handle an overflow at a 5-node v with a split operation:
 - let $v_1 \dots v_5$ be the children of v and $k_1 \dots k_4$ be the keys of v
 - node v is 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



Analysis of Insertion

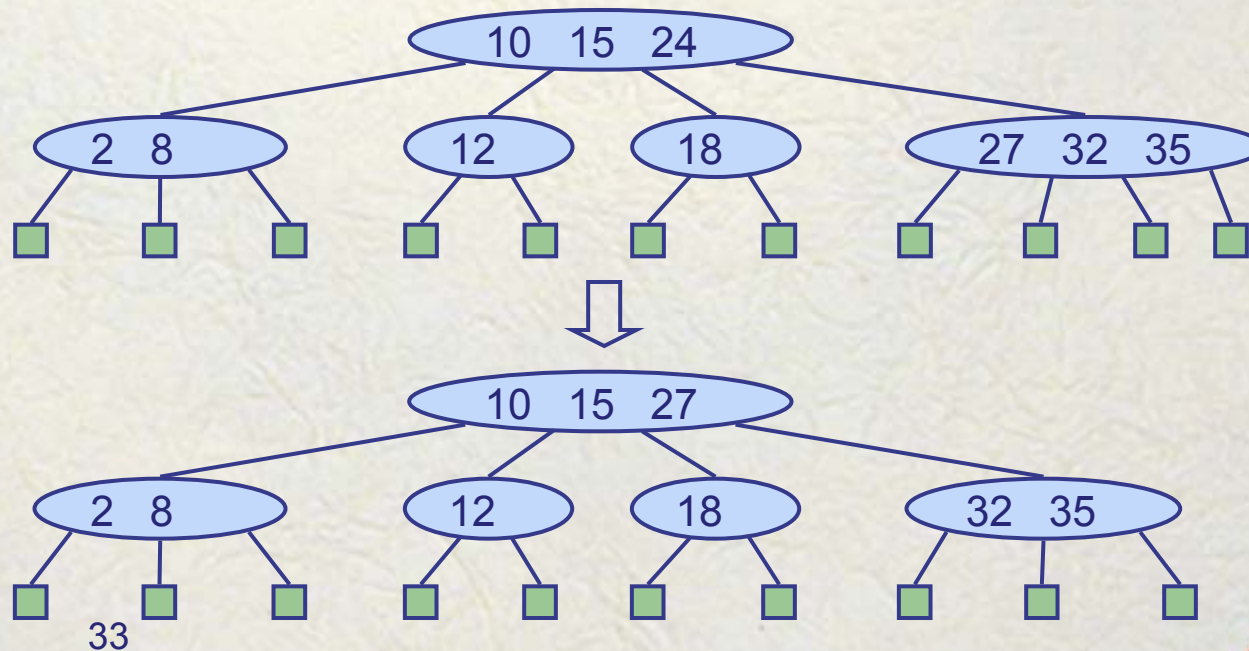
Algorithm *insert(k, o)*

1. We search for key k to locate the insertion node v
2. We add the new entry (k, o) at node v
3. **while** *overflow*(v)
 if *isRoot*(v)
 create a new empty root above v
 $v \leftarrow \textit{split}(v)$

- Let T be a (2,4) tree with n items
 - Tree T has $O(\log n)$ height
 - Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
 - Step 2 takes $O(1)$ time
 - Step 3 takes $O(\log n)$ time because each split takes $O(1)$ time and we perform $O(\log n)$ splits
- Thus, an insertion in a (2,4) tree takes $O(\log n)$ time

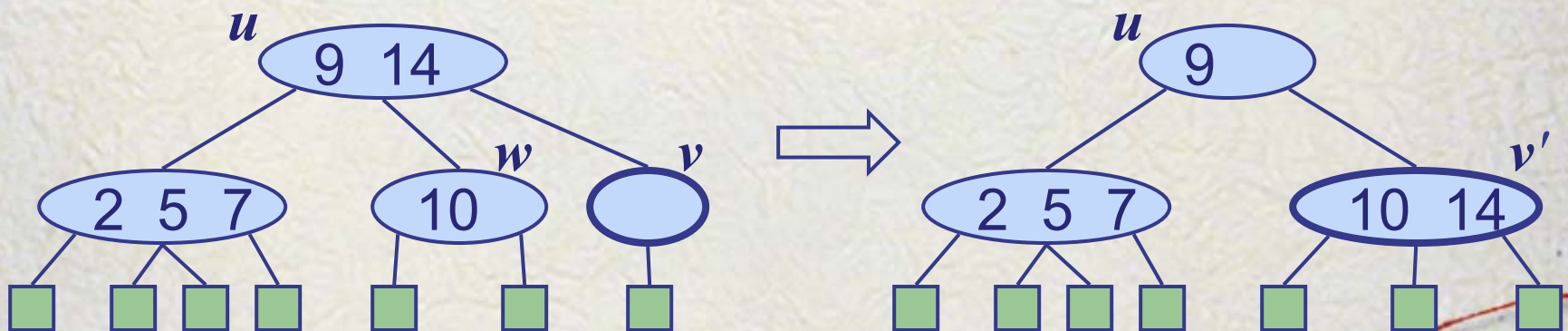
Deletion

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



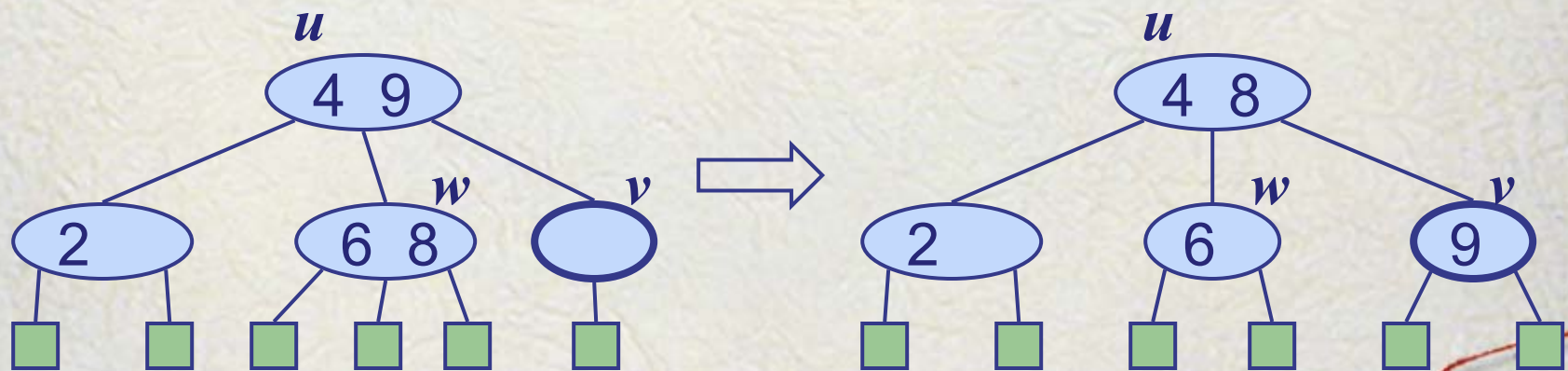
Underflow and Fusion

- Deleting an entry 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 entry from u to the merged node v'
 - After a fusion, the underflow may propagate to the parent u



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



Analysis of Deletion

- Let T be a (2,4) tree with n items
 - Tree T has $O(\log n)$ height
- In a deletion operation
 - We visit $O(\log n)$ nodes to locate the node from which to delete the entry
 - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
 - Each fusion and transfer takes $O(1)$ time
- Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time

References

- Chapter 10: Data structures and Algorithms by Goodrich and Tamassia.