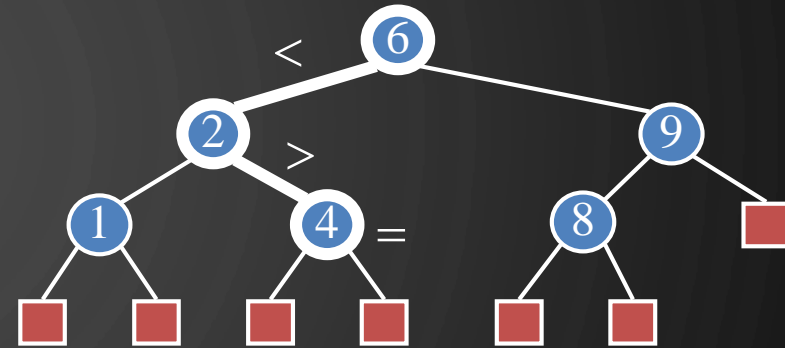


# CHAPTER 11

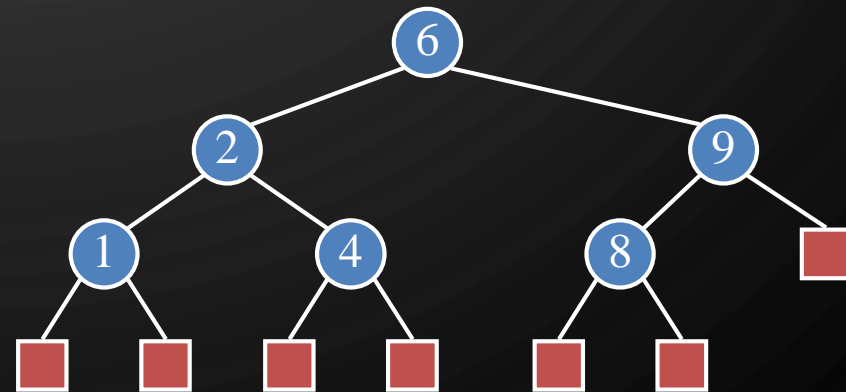
## SEARCH TREES

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

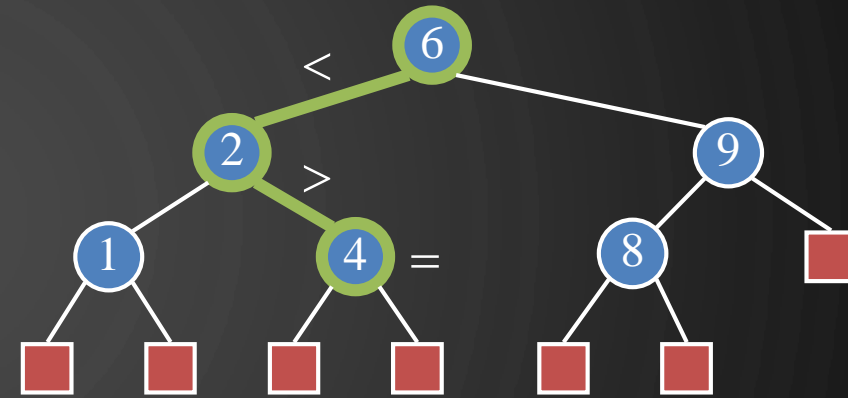


# BINARY SEARCH TREES

- A binary search tree is a binary tree storing entries  $(k, e)$  (i.e., key-value pairs) 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  $v$ . Then  $key(u) \leq key(v) \leq key(w)$
- 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
- Example: `get(4)`
  - Call `Search(4, root)`
- Algorithms for nearest neighbor queries are similar

Algorithm `Search(k, v)`

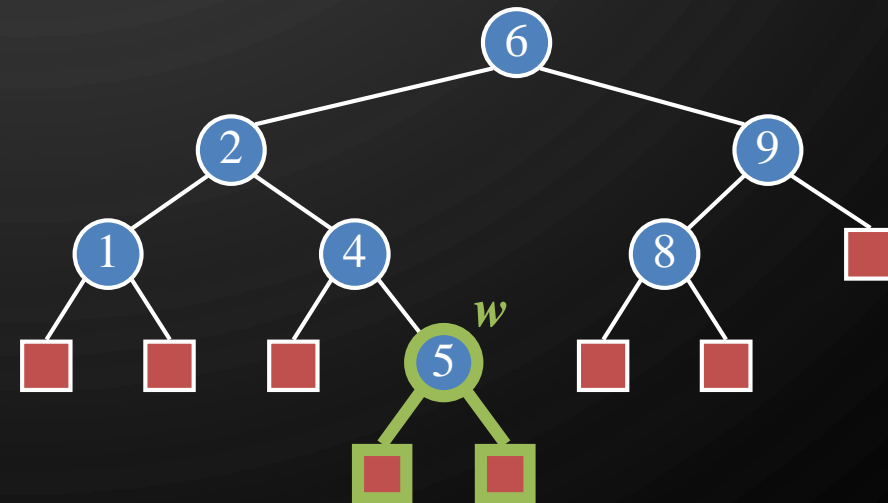
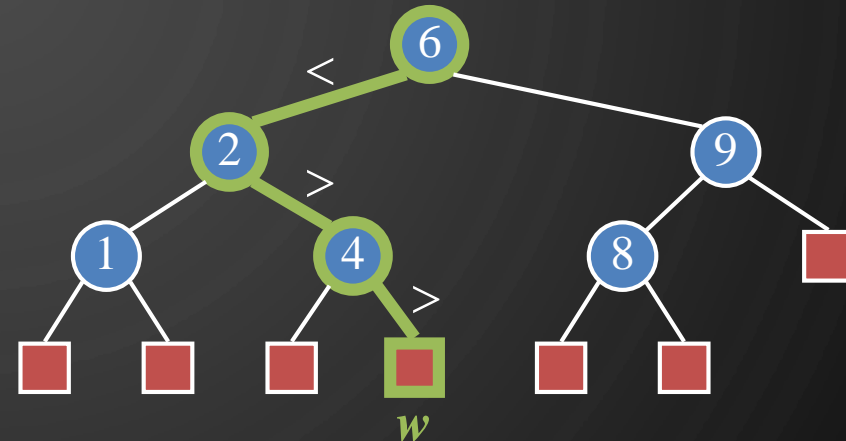
**Input:** Key  $k$ , node  $v$

**Output:** Node with key =  $k$

```
1. if  $v.isExternal()$  then
2.     return  $v$ 
3. if  $k < v.key()$  then
4.     return Search(k, v.left())
5. else if  $k = v.key()$  then
6.     return  $v$ 
7. else //  $k > v.key()$ 
8.     return Search(k, v.right())
```

# INSERTION


- To perform operation `put(k, v)`, we search for key  $k$  (using `Search(k)`)
- 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





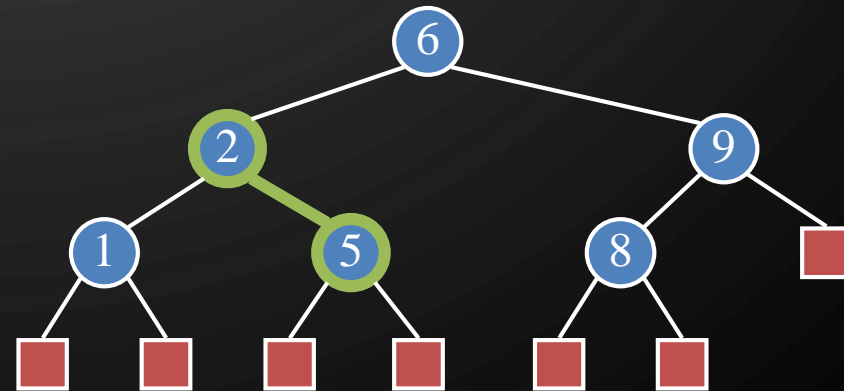
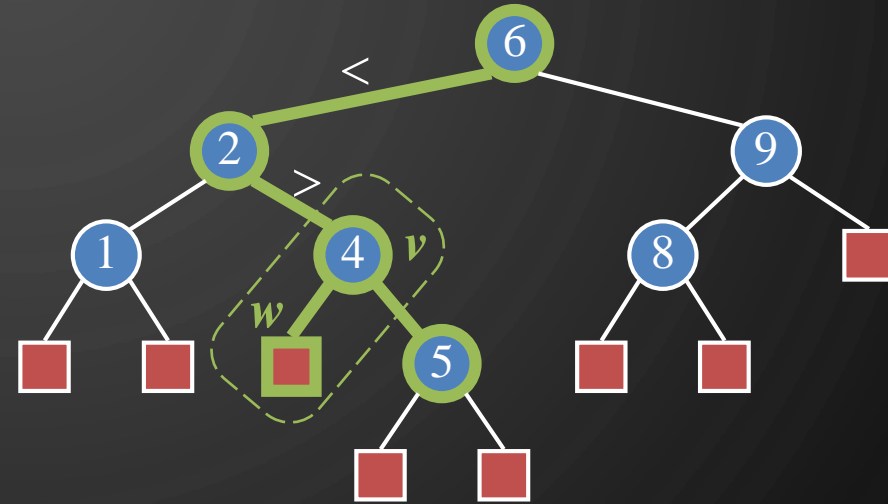
# EXERCISE

## BINARY SEARCH TREES

- Insert into an initially empty binary search tree items with the following keys (in this order). Draw the resulting binary search tree
    - 30, 40, 24, 58, 48, 26, 11, 13
- 

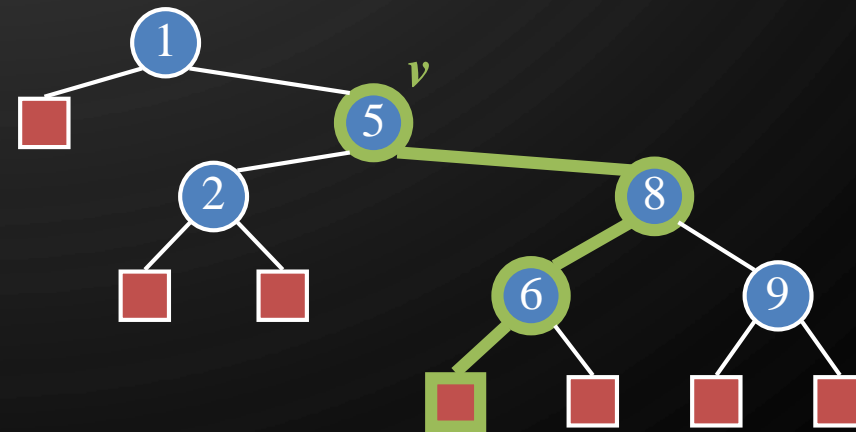
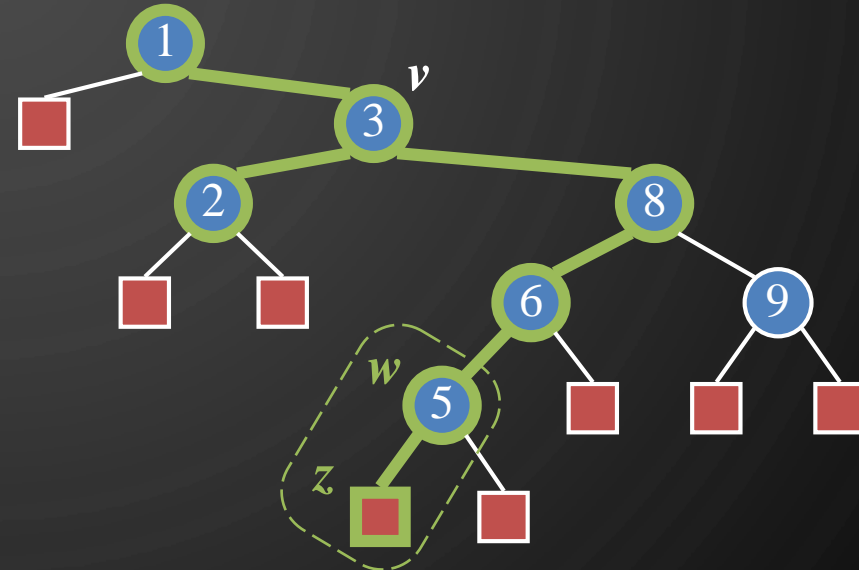
# DELETION

- To perform operation `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 `removeExternal(w)`, which removes  $w$  and its parent
- Example: remove 4



# DELETION (CONT.)



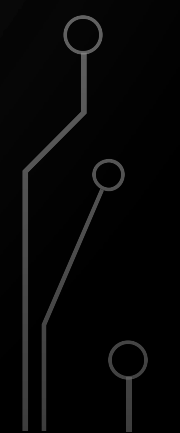
- We consider the case where the key  $k$  to be removed is stored at a node  $v$  whose children are both internal
  - we find the internal node  $w$  that follows  $v$  in an inorder traversal
  - we copy  $w.key()$  into node  $v$
  - we remove node  $w$  and its left child  $z$  (which must be a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





# EXERCISE

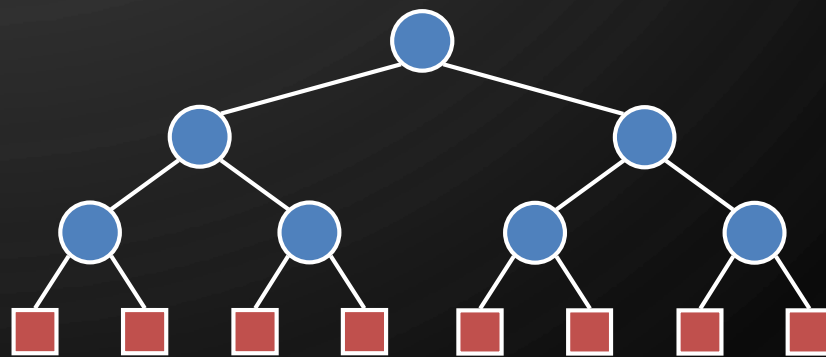
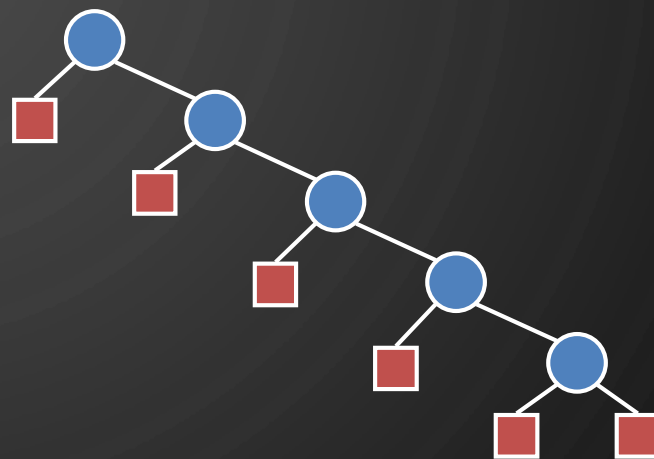
## BINARY SEARCH TREES

- Insert into an initially empty binary search tree items with the following keys (in this order). Draw the resulting binary search tree
    - 30, 40, 24, 58, 48, 26, 11, 13
  - Now, remove the item with key 30. Draw the resulting tree
  - Now remove the item with key 48. Draw the resulting tree.
- 
- 
- 

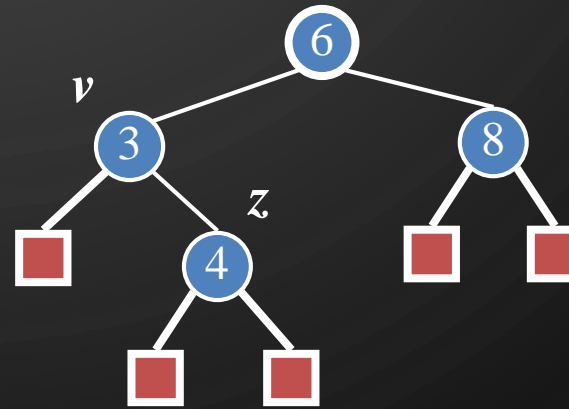


# PERFORMANCE

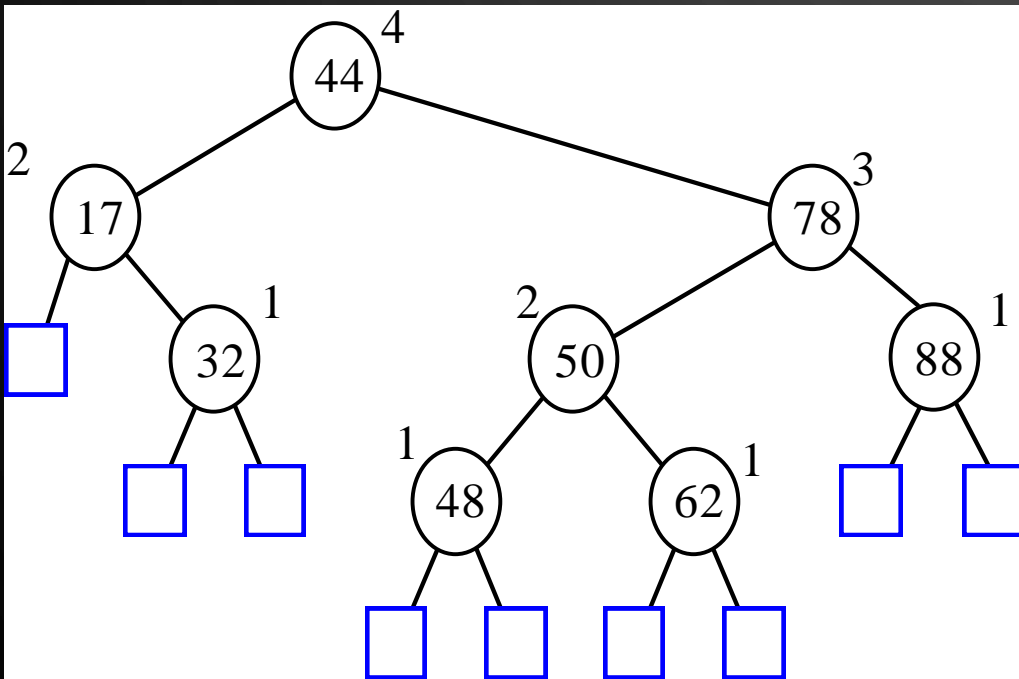
- Consider an ordered map with  $n$  items implemented by means of a binary search tree of height  $h$ 
  - Space used is  $O(n)$
  - Methods  $get(k)$ ,  $put(k, v)$ , and  $remove(k)$  take  $O(h)$  time
- The height  $h$  is  $O(n)$  in the worst case and  $O(\log n)$  in the best case



# AVL TREES



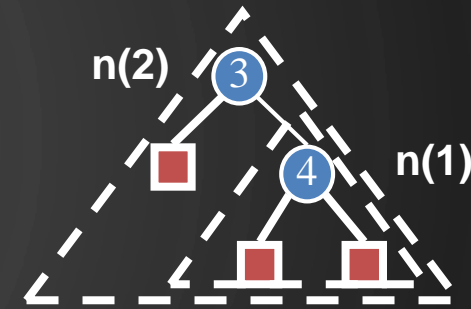
# AVL TREE DEFINITION



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

- 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

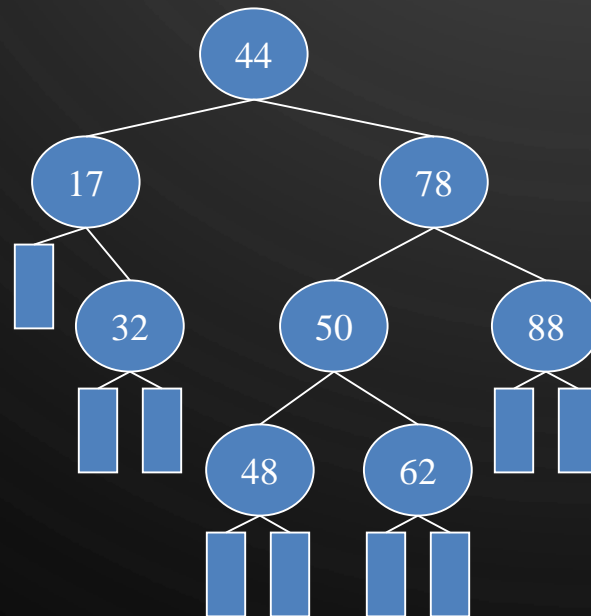
# HEIGHT OF AN AVL TREE



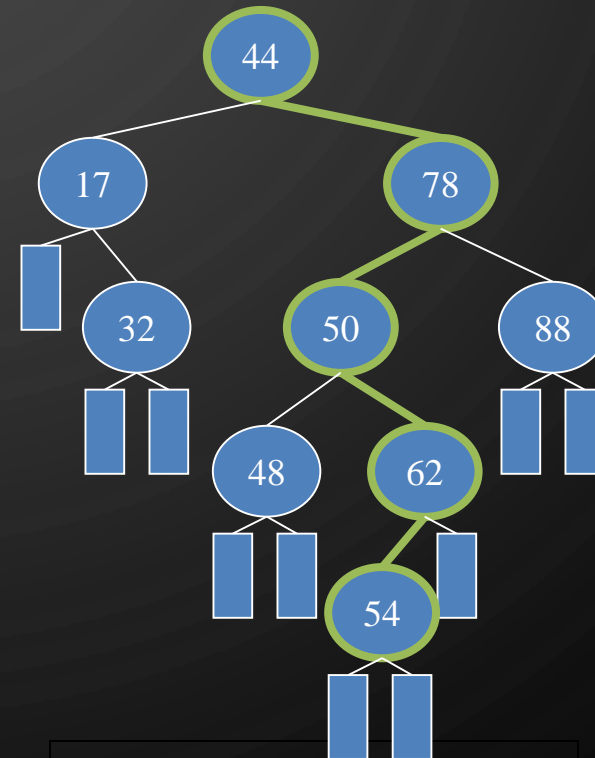
- Fact: The height of an AVL tree storing  $n$  keys is  $O(\log n)$ .
- Proof: Let us bound  $n(h)$ : the minimum number of internal nodes of an AVL tree of height  $h$ .
- We easily see that  $n(1) = 1$  and  $n(2) = 2$
- For  $n > 2$ , an AVL tree of height  $h$  contains the root node, one AVL subtree 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
  - $n(h) > 2n(h - 2) > 4n(h - 4) > 8n(h - 6), \dots$  (by induction),
  - $n(h) > 2^i n(h - 2i)$
- Solving the base case we get:  $n(h) > 2^{\frac{h}{2}-1}$
- 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 insert 54:



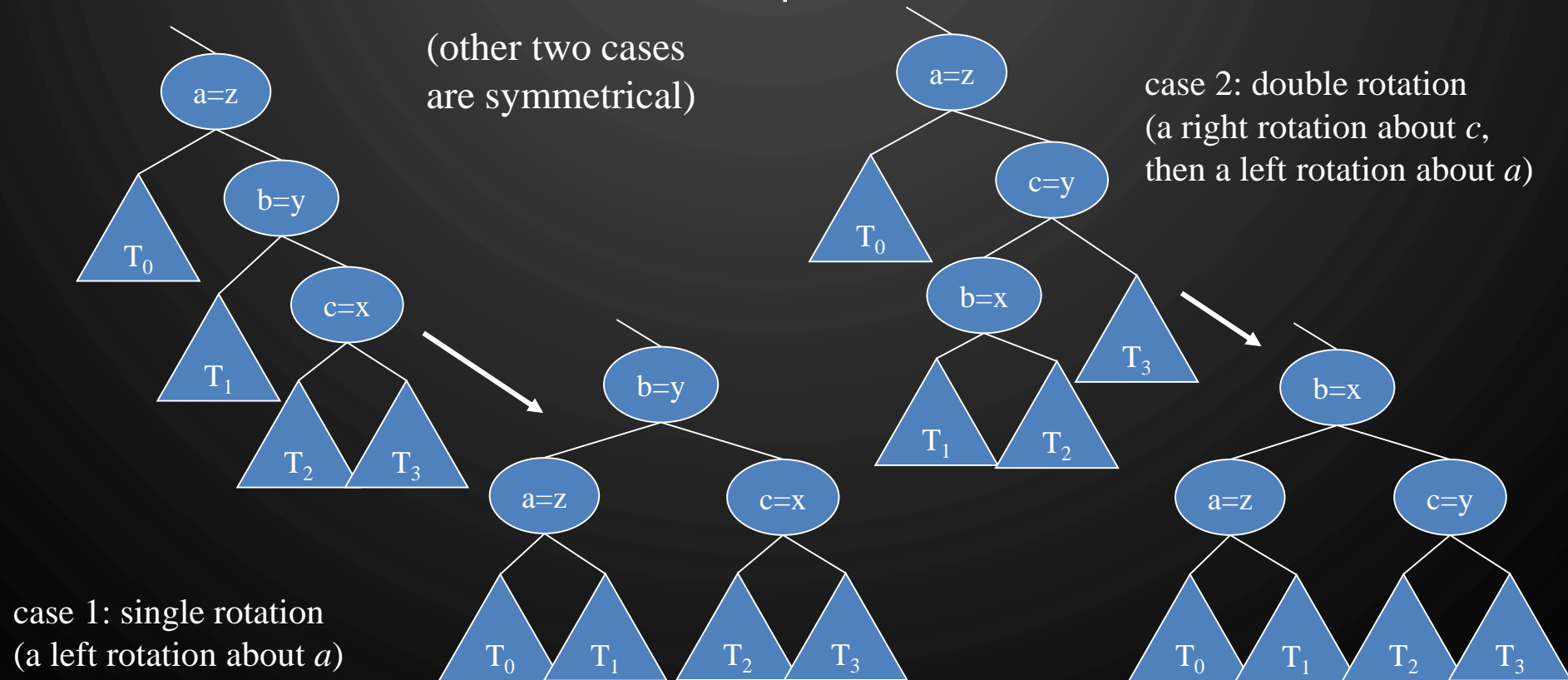
Before Insertion



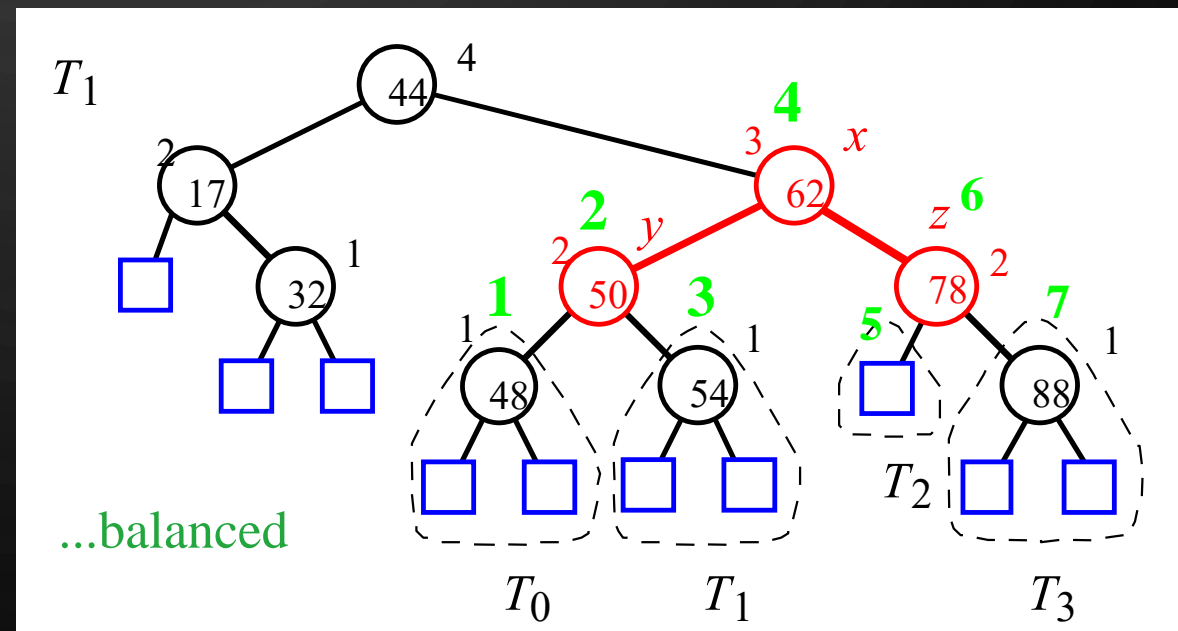
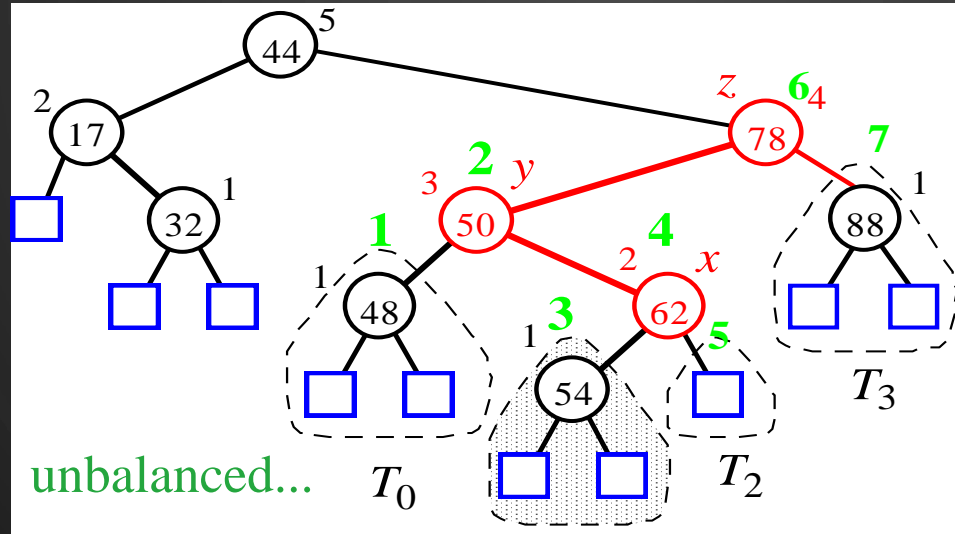
After Insertion

# TRINODE RESTRUCTURING

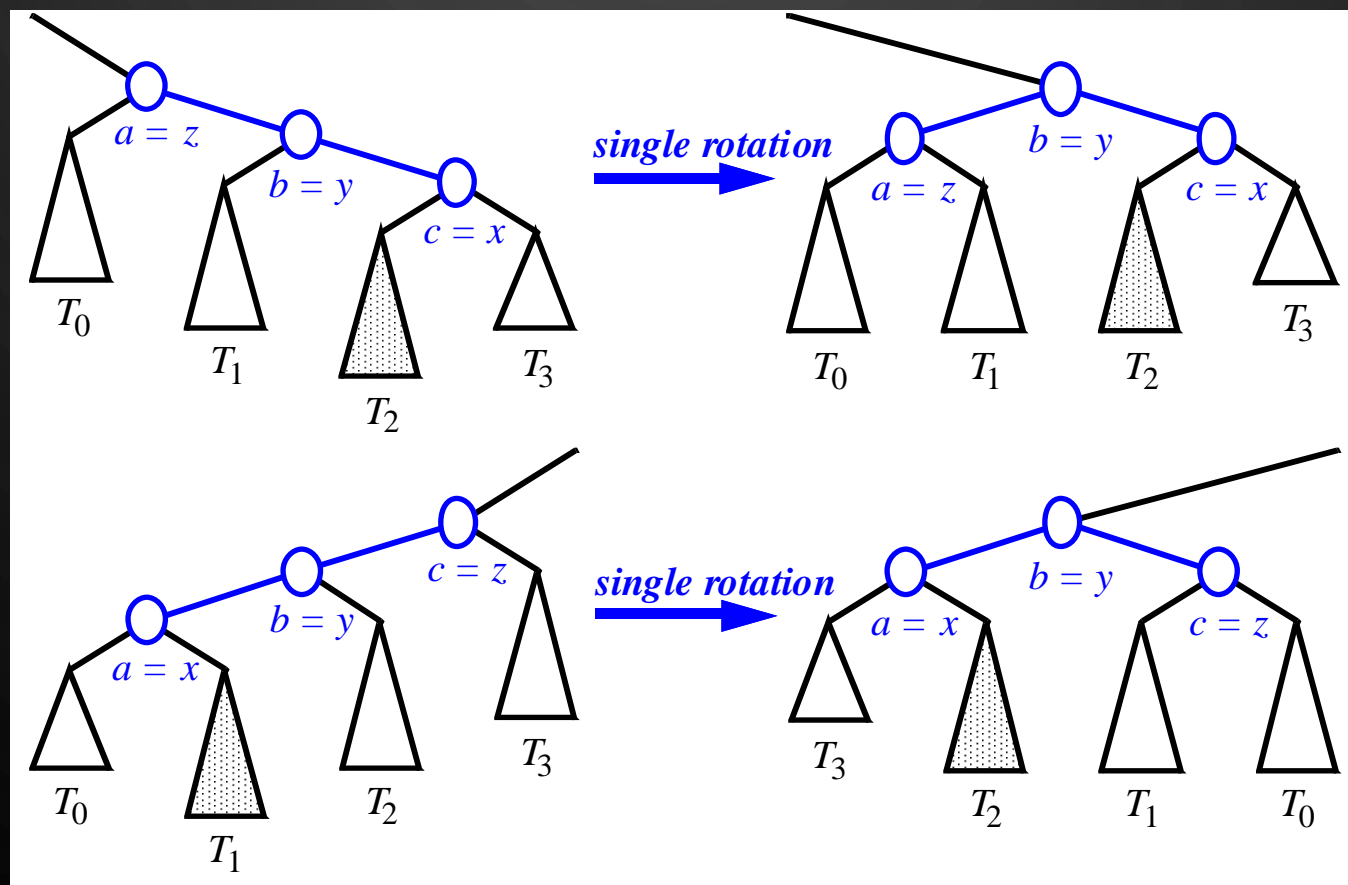
- let  $(a, b, c)$  be an inorder listing of  $x, y, z$
- perform the rotations needed to make  $b$  the topmost node of the three



# INSERTION EXAMPLE, CONTINUED

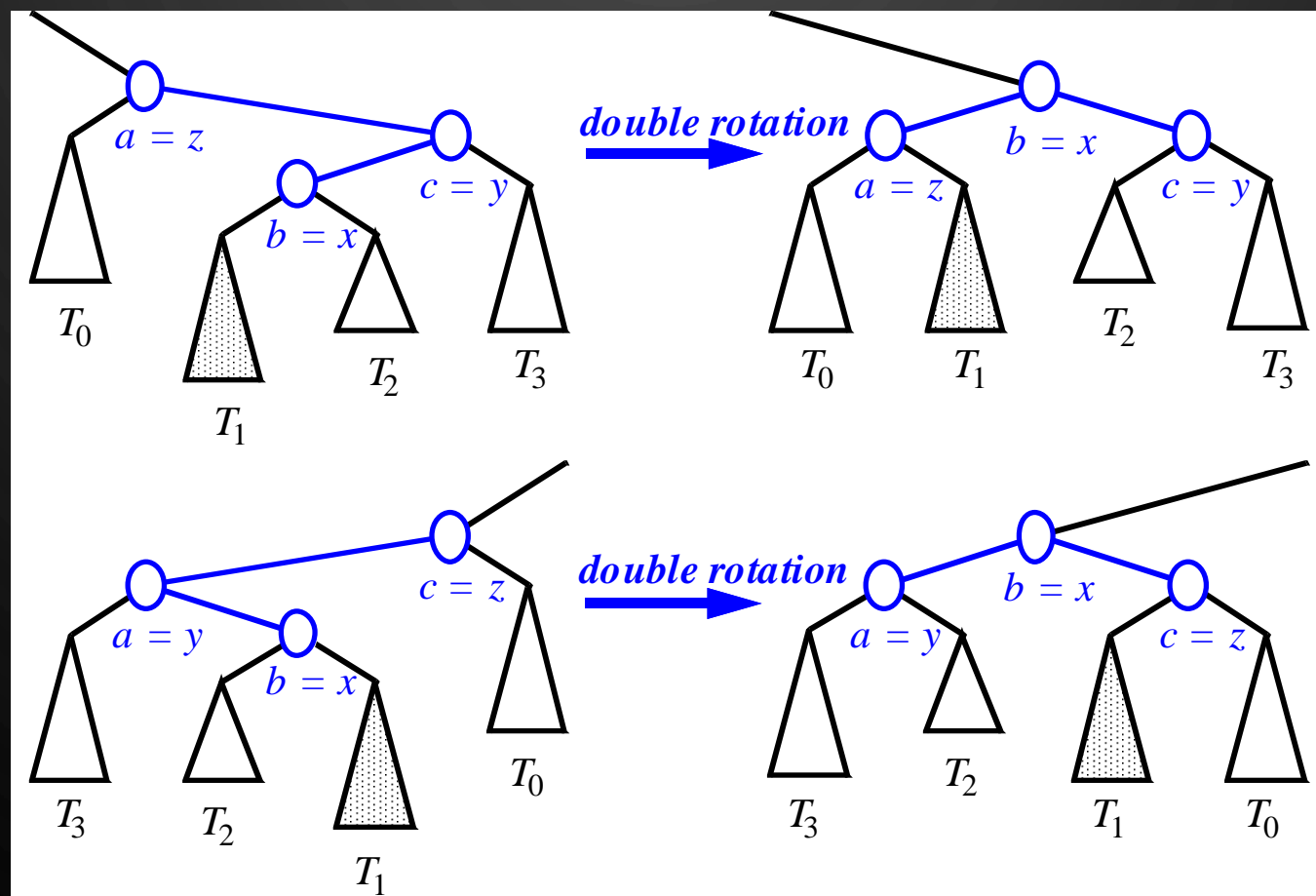


# RESTRUCTURING SINGLE ROTATIONS






# RESTRUCTURING DOUBLE ROTATIONS

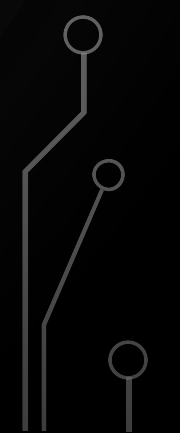




# EXERCISE

## AVL TREES

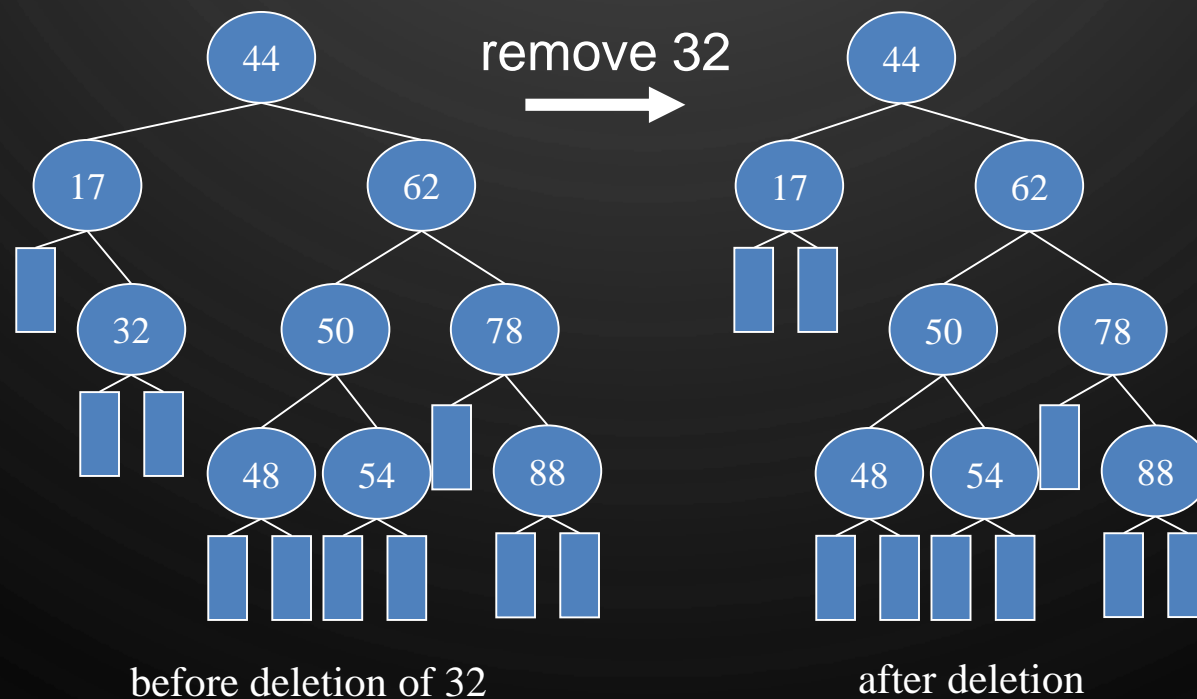
- Insert into an initially empty AVL tree items with the following keys (in this order). Draw the resulting AVL tree
    - 30, 40, 24, 58, 48, 26, 11, 13
- 



# REMOVAL IN AN AVL TREE

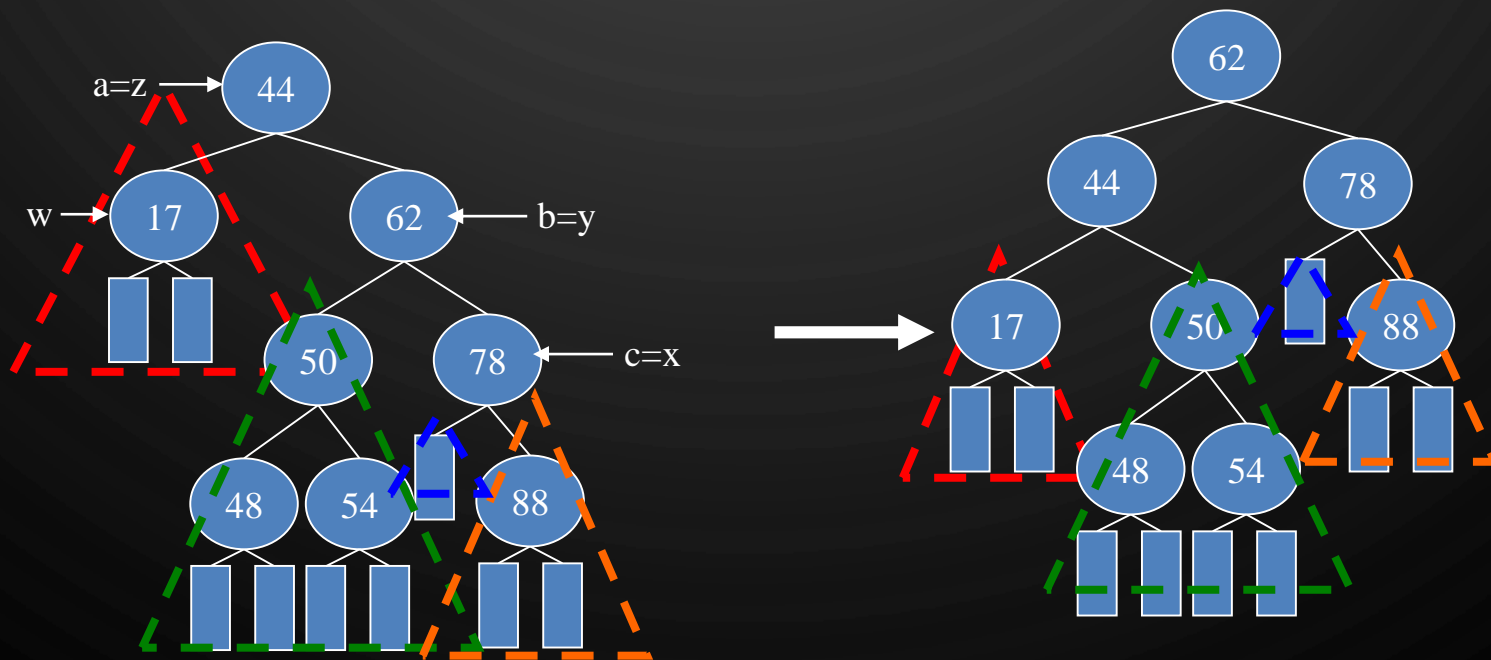
- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent,  $w$ , may cause an imbalance.

• Example:



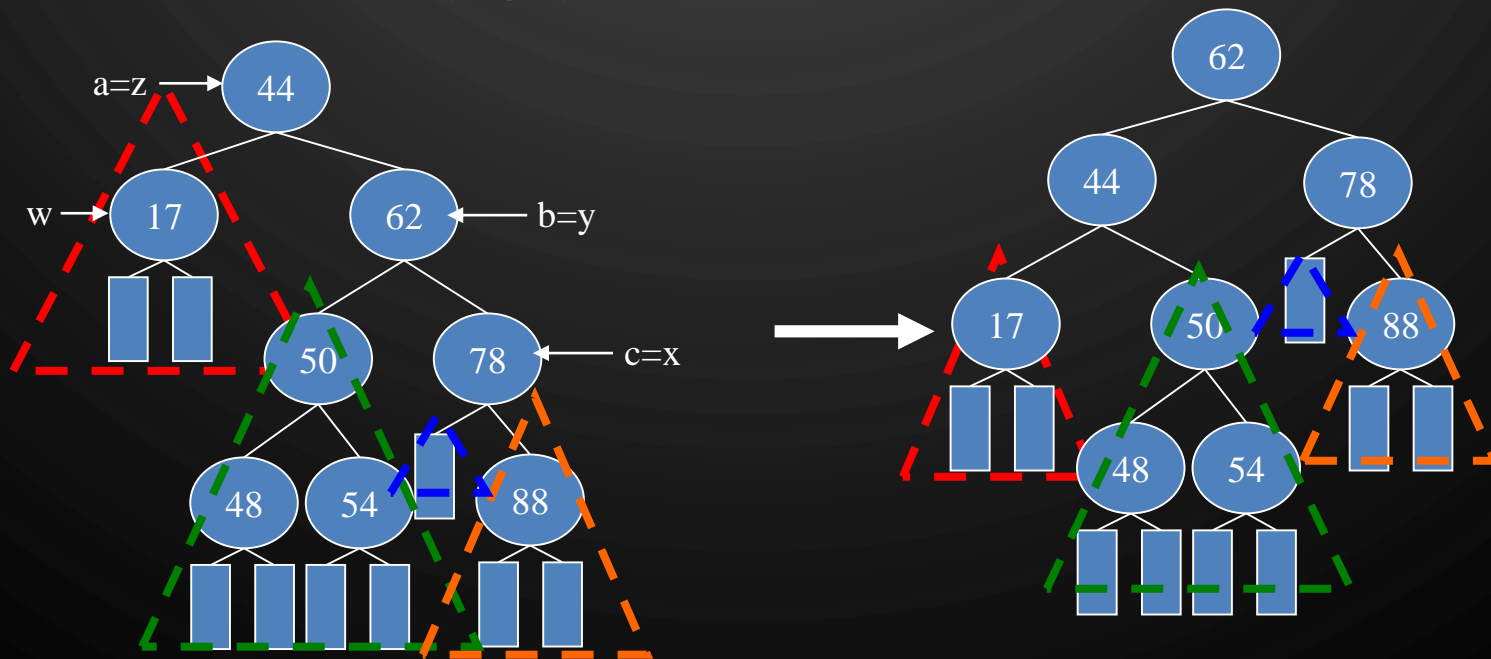
# REBALANCING AFTER A REMOVAL

- Let  $z$  be the **first unbalanced** node encountered while travelling up the tree from  $w$  (parent of removed node). 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  $z$ .



# REBALANCING AFTER A REMOVAL


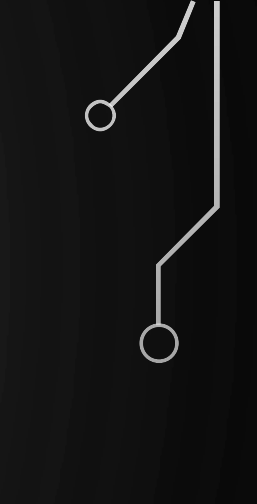
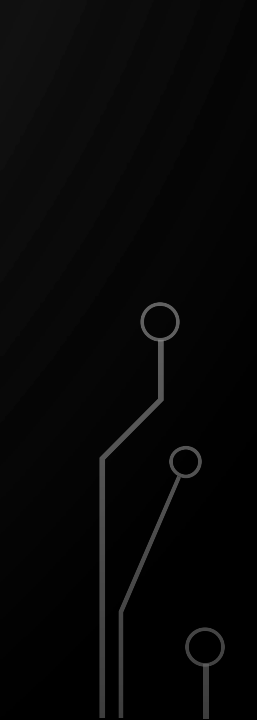
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of  $T$  is reached
  - This can happen at most  $O(\log n)$  times. Why?



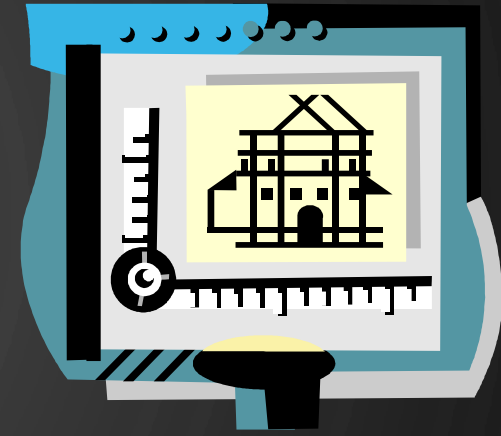


# EXERCISE

## AVL TREES

- Insert into an initially empty AVL tree items with the following keys (in this order). Draw the resulting AVL tree
    - 30, 40, 24, 58, 48, 26, 11, 13
  - Now, remove the item with key 48. Draw the resulting tree
  - Now, remove the item with key 58. Draw the resulting tree
- 
- 
- 

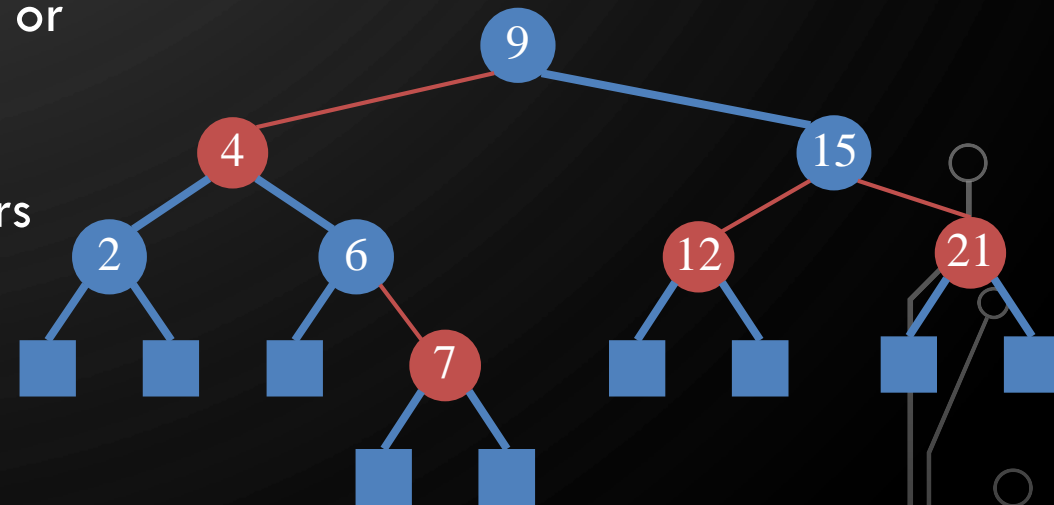
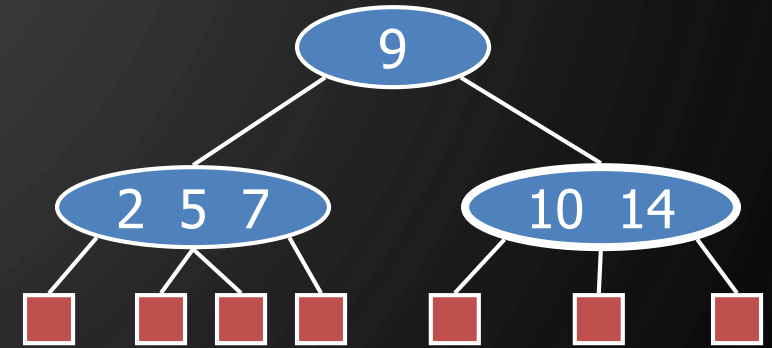
# RUNNING TIMES FOR AVL TREES



- A **single restructure** is  $O(1)$  – using a linked-structure binary tree
- `get(k)` takes  $O(\log n)$  time – height of tree is  $O(\log n)$ , no restructures needed
- `put(k, v)` takes  $O(\log n)$  time
  - Initial find is  $O(\log n)$
  - Restructuring up the tree, maintaining heights is  $O(\log n)$
- `remove(k)` takes  $O(\log n)$  time
  - Initial find is  $O(\log n)$
  - Restructuring up the tree, maintaining heights is  $O(\log n)$

# OTHER TYPES OF SELF-BALANCING TREES

- **Splay Trees** – A binary search tree which uses an operation  $\text{splay}(x)$  to allow for amortized complexity of  $O(\log n)$
- **(2,4) Trees** – A multiway search tree where every node stores internally a list of entries and has 2, 3, or 4 children. Defines self-balancing operations
- **Red-Black Trees** – A binary search tree which colors each internal node red or black. Self-balancing dictates changes of colors and required rotation operations








# INTERVIEW QUESTION 1

- Given a sorted array with unique integer elements, write an algorithm to create a binary search tree with minimal height.
  - Hint: Recursion




GAYLE LAAKMANN MCDOWELL, "CRACKING THE CODE INTERVIEW: 150 PROGRAMMING QUESTIONS AND SOLUTIONS", 5TH EDITION, CAREERCUP PUBLISHING, 2011.



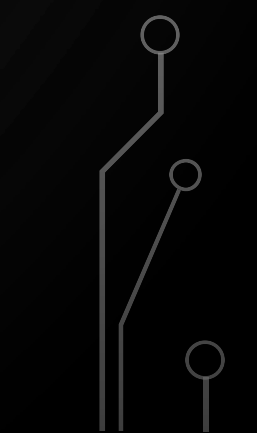


## INTERVIEW QUESTION 2

- Implement a function to check if a binary tree is a binary search tree.



GAYLE LAAKMANN MCDOWELL, "CRACKING THE CODE INTERVIEW: 150 PROGRAMMING QUESTIONS AND SOLUTIONS", 5TH EDITION, CAREERCUP PUBLISHING, 2011.



# EXAM 2

- Hack sheet – you can have a single 8 1/2" x 11" paper with handwritten notes on both sides with you during the exam. You can put anything on it, but summary slides and the generic tree traversal algorithms make great candidates.
- No Java language/programming questions. Lecture material only.
- Format – 5 questions and a bonus
  - Q1 – Fill-in-the-blank questions (similar to quizzes)
  - Q2 – Tracing Hash Maps (similar to quizzes)
  - Q3 – Tracing Binary Search Trees (similar to quizzes)
  - Q4 – Write and/or analyze algorithm using Priority Queues (similar to homework)
  - Q5 – Write and/or analyze algorithm using Maps (similar to homework)
  - Bonus – ?