

# Data Structures and Algorithms

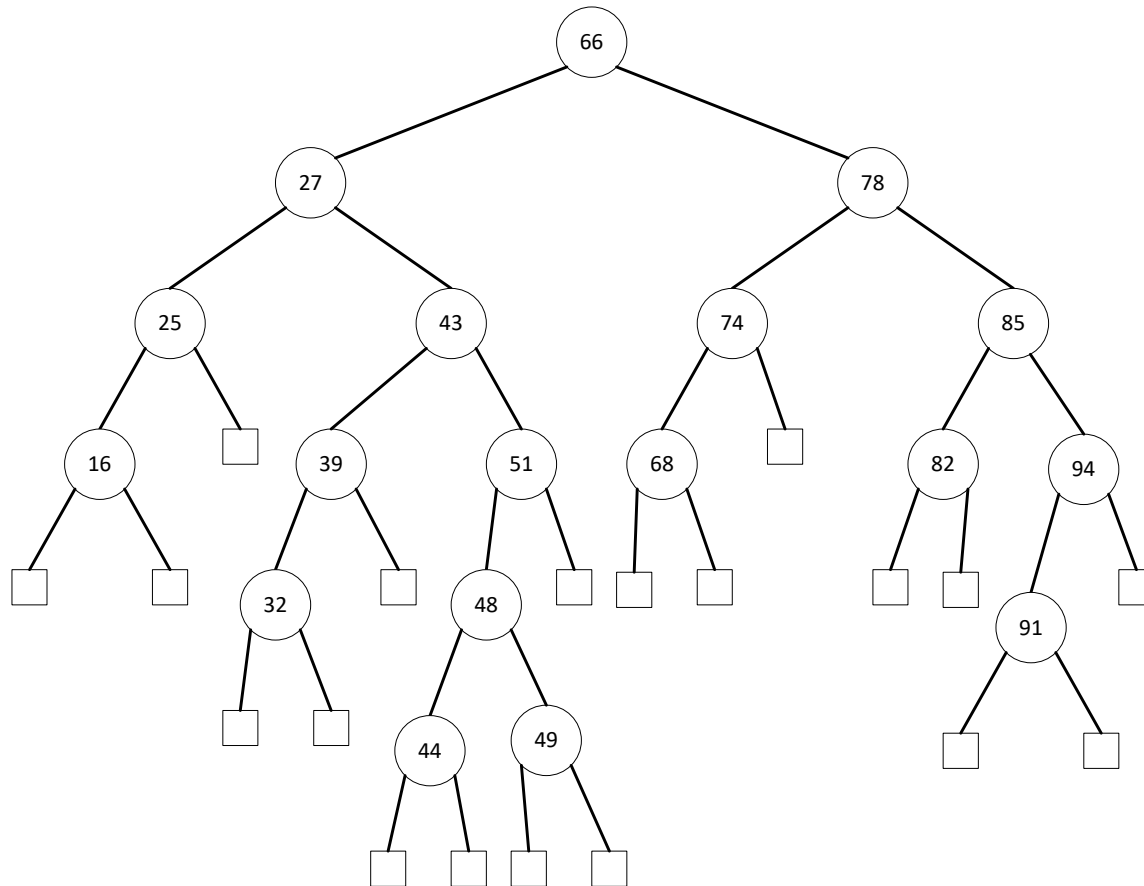
## Chapter 11

# Binary Search Trees

- Can be used to implement a *sorted map*.
- Each internal position  $p$  in a binary search tree stores  $(k, v)$  pair.
- Binary search tree is a *proper binary tree* with the following properties:
  - For each internal position  $p$  with entry  $(k, v)$  pair,
    - Keys stored in the left subtree of  $p$  are less than  $k$ .
    - Keys stored in the right subtree of  $p$  are greater than  $k$ .
- Note: In the above definition leaves are “placeholders,” which is shown as small squares in the graph.

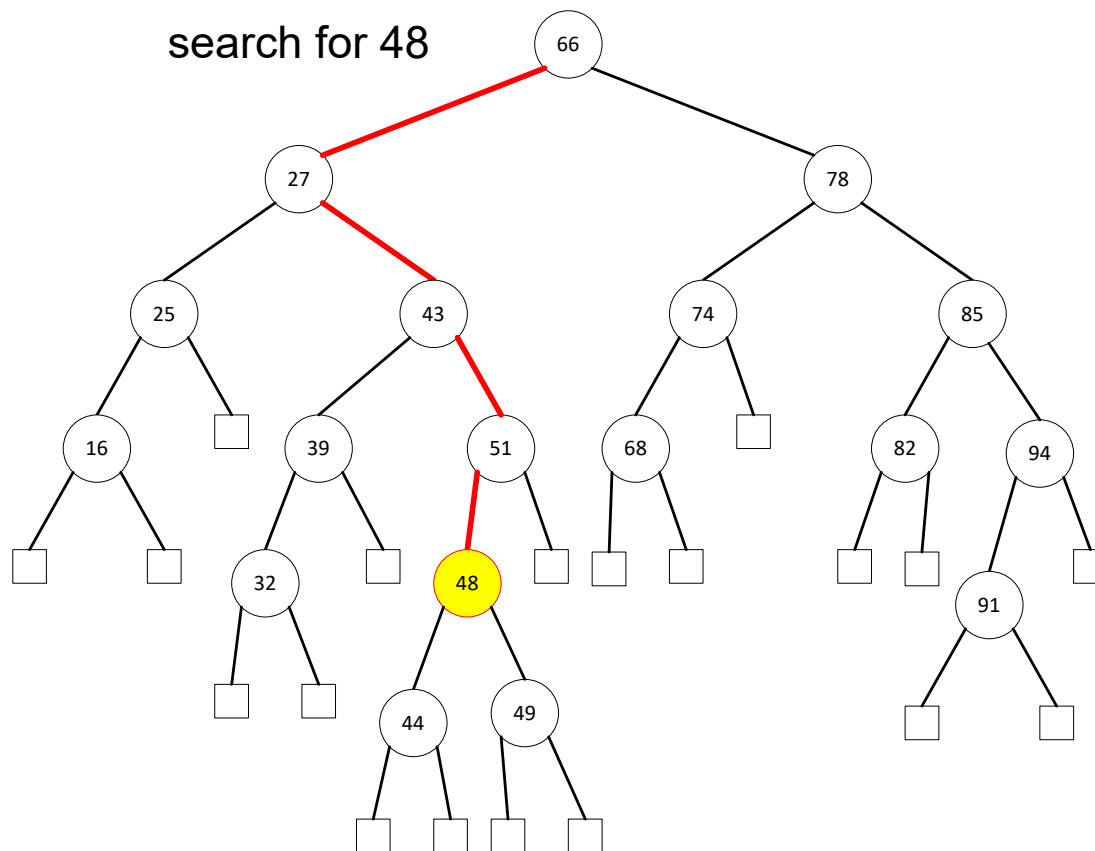
# Binary Search Trees

- Example (only keys are shown):



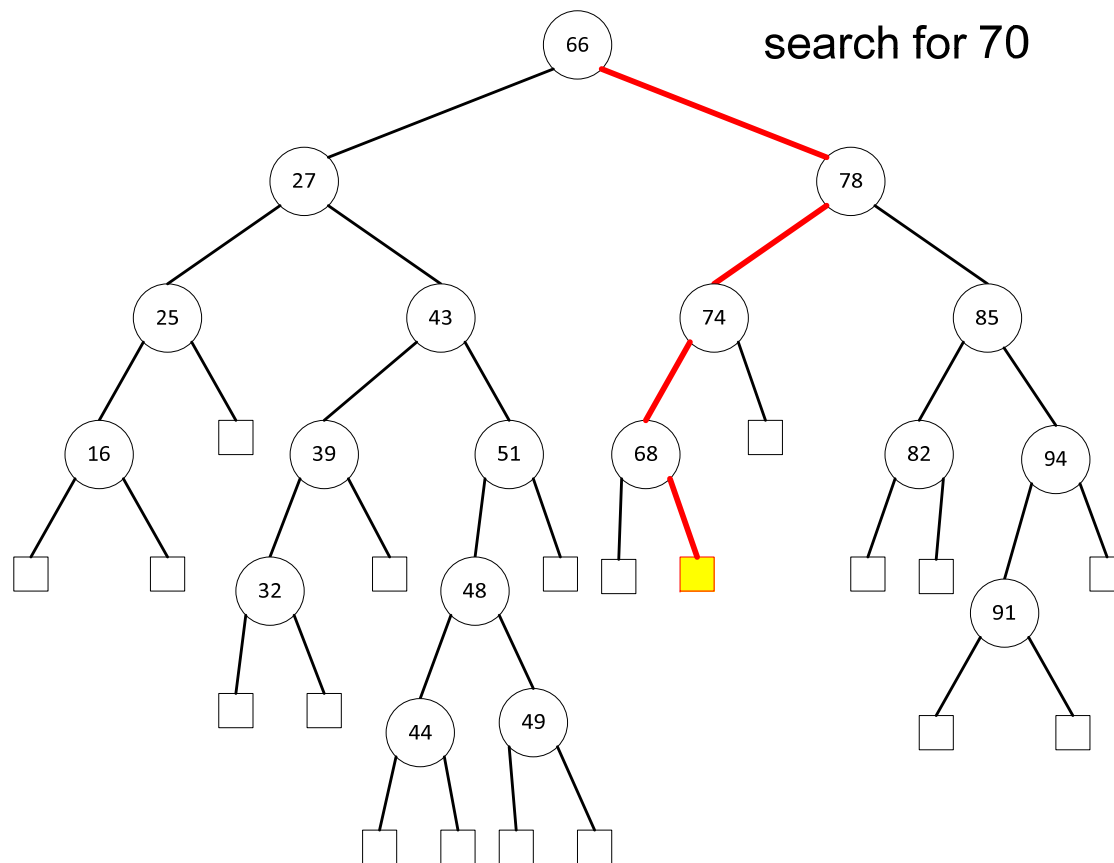
# Binary Search Trees

- Search (successful search)



# Binary Search Trees

- Search (unsuccessful search)



# Binary Search Trees

- Search pseudocode

Algorithm TreeSearch(p, k)

if p is external then                      // unsuccessful search

    return p

else if k == key(p)                      // successful search

    return p

else if k < key(p)

    return TreeSearch(left(p), k)      // recurse on left subtree

else

    return TreeSearch(right(p), k)    // recurse on right subtree

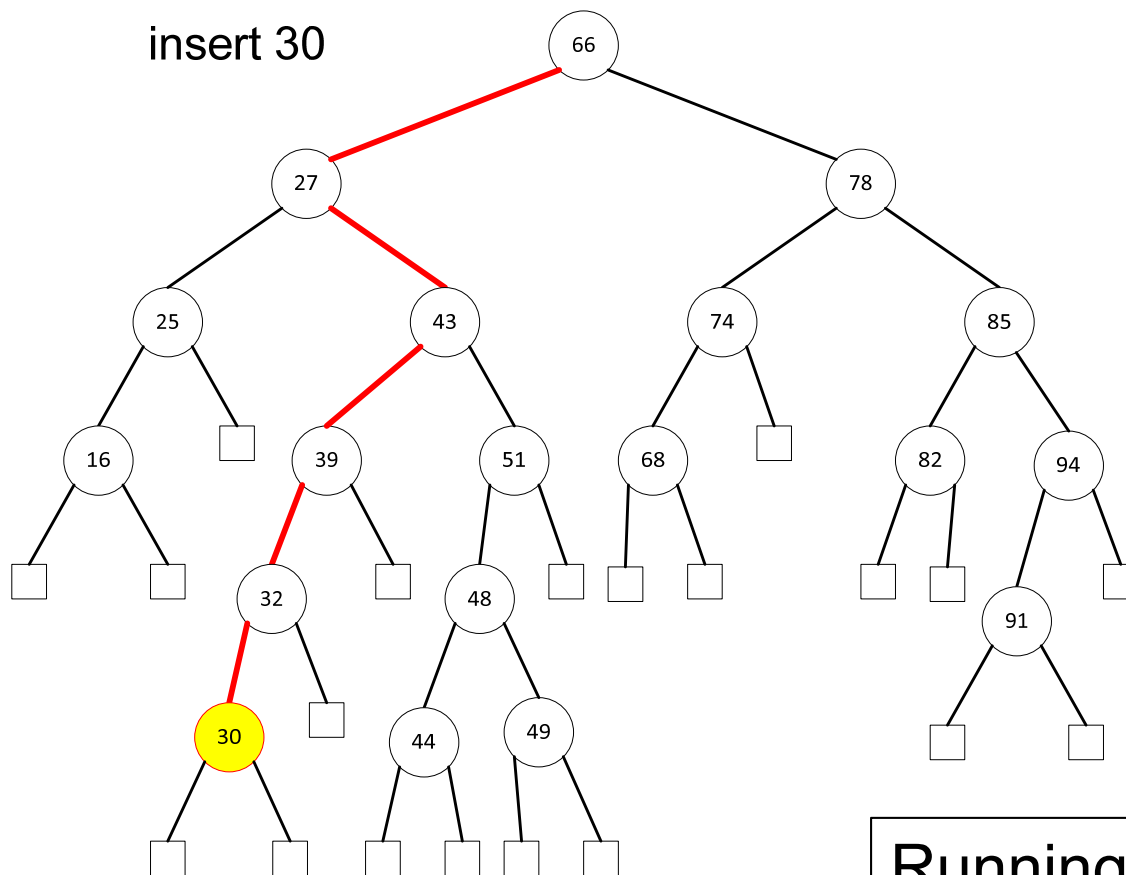
- Running time:  $O(h)$

# Binary Search Trees

- Inserting an entry with  $(k, v)$ 
  - Perform a search operation.
  - If an entry with key  $k$  is found (i.e., successful search), the existing value is replaced with the new value  $v$ .
  - If there is no entry with key  $k$ , then we add an entry at the leaf node where the unsuccessful search ended up.

# Binary Search Trees

- Insert illustration



Running time:  $O(h)$

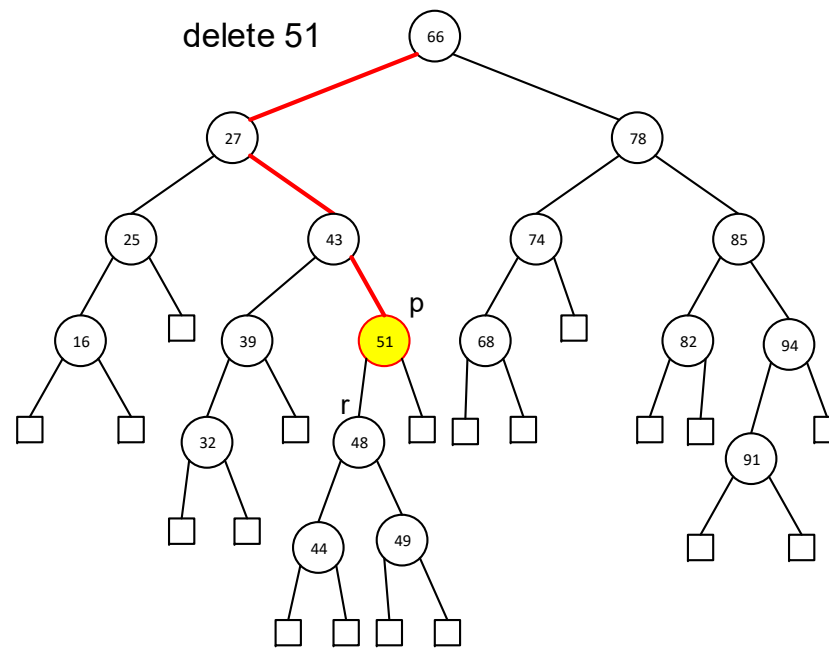


# Binary Search Trees

- Deleting an entry with  $(k, v)$ 
  - Slightly more complex
  - Perform search
    - If we reach a leaf node, do nothing
    - If we find the entry at position  $p$ 
      - Case 1: at most one child of  $p$  is an internal node
      - Case 2:  $p$  has two children, both of which are internal

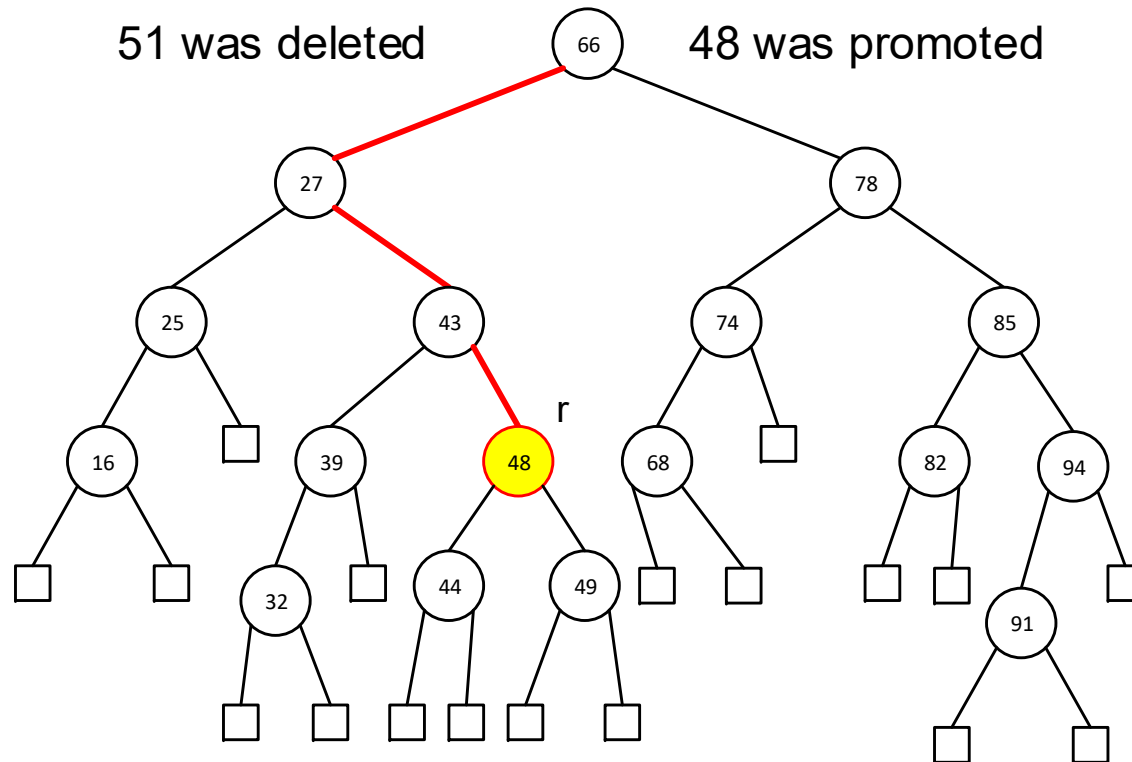
# Binary Search Trees

- Deletion Case 1
  - If both children are leaf nodes, then  $p$  is replaced with a leaf node.
  - If  $p$  has one internal-node child, then that child node replaces  $p$



# Binary Search Trees

- Deletion Case 1
  - If  $p$  has one internal-node child (continued)

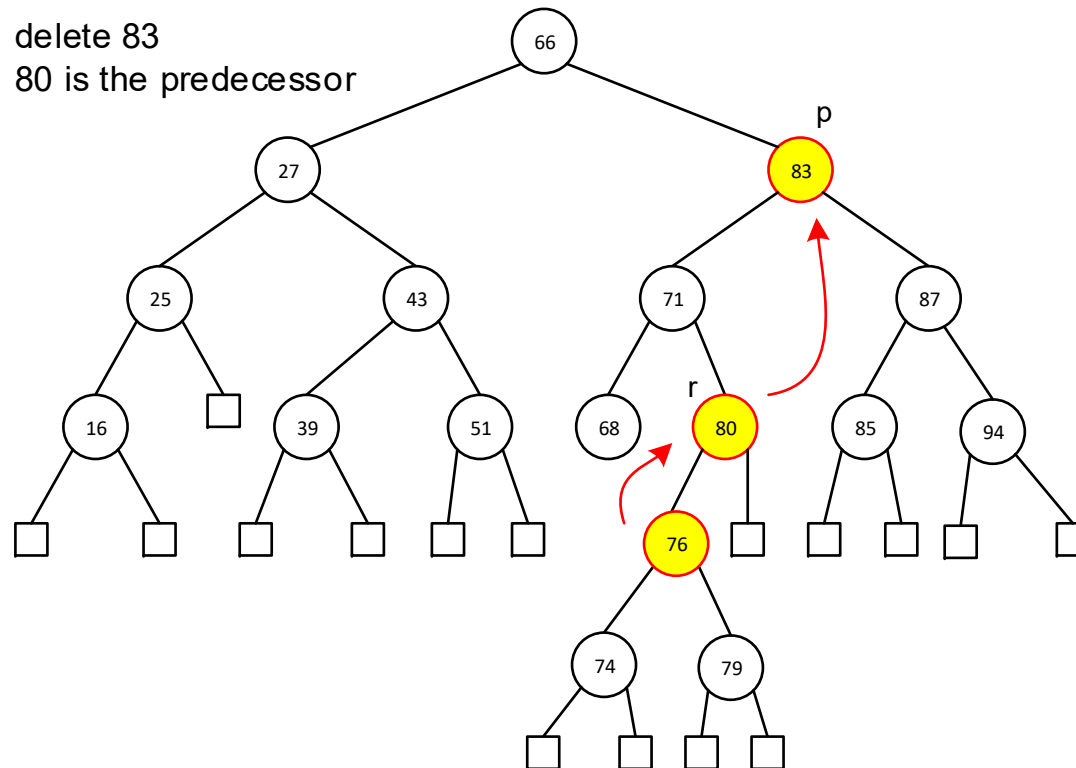


# Binary Search Trees

- Deletion Case 2
  - First, we find the node  $r$  that has the largest key that is strictly less than  $p$ 's key. This node is called the *predecessor* of  $p$  in the ordering of keys, which is the rightmost node in  $p$ 's left subtree.
  - We let  $r$  replace  $p$ .
  - Since  $r$  is the rightmost node in  $p$ 's left subtree, it does not have a right child. It has only a left child.
  - The node  $r$  is removed and the subtree rooted at  $r$ 's left child is promoted to  $r$ 's position.

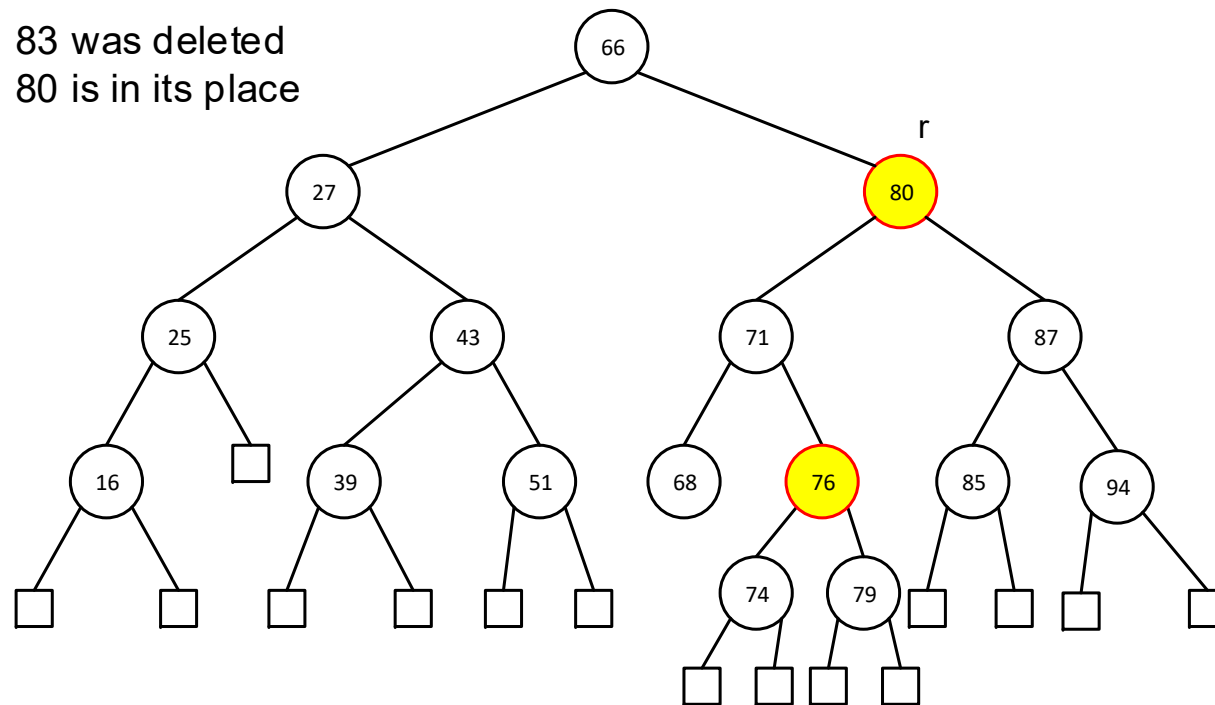
# Binary Search Trees

- Deletion Case 2



# Binary Search Trees

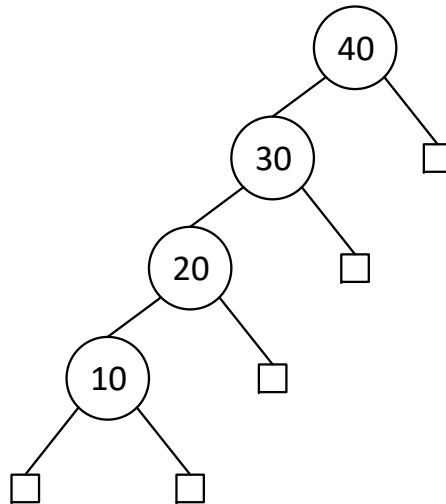
- Deletion Case 2



- Running time:  $O(h)$

# Binary Search Trees

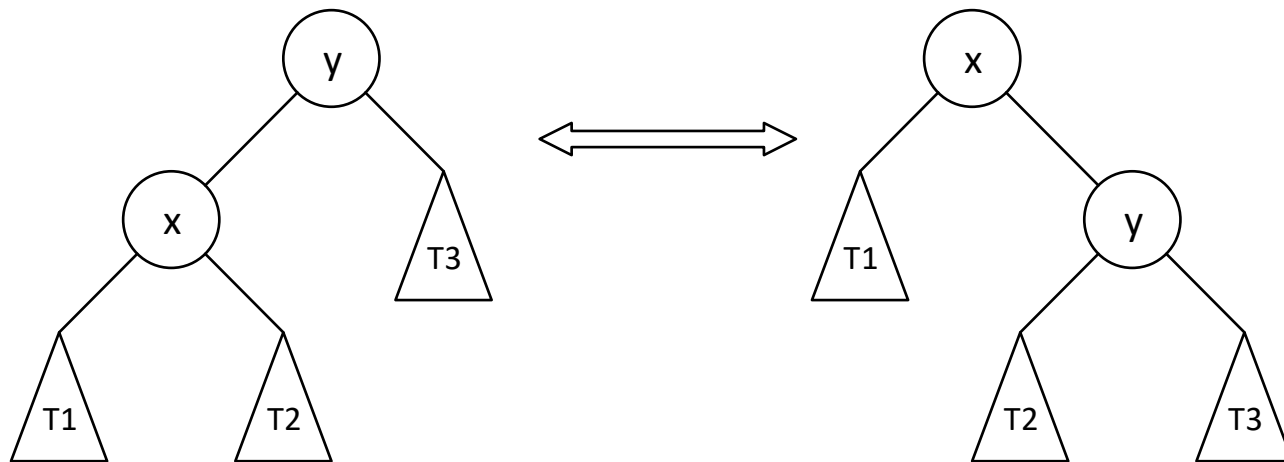
- Most binary search tree operations run in  $O(h)$ .
- In the worst case, a tree is just a linked list. In this case, running times are  $O(n)$ .



- To guarantee  $O(h)$ , a tree needs to be balanced.

# Balanced Search Trees

- When a binary search tree is unbalanced, it is necessary to *rebalance* the tree.
- Primary operation for rebalancing a binary search tree is *rotation*.



- Can rotate in either direction.
- Binary search tree property is maintained after rotation.



# Balanced Search Trees

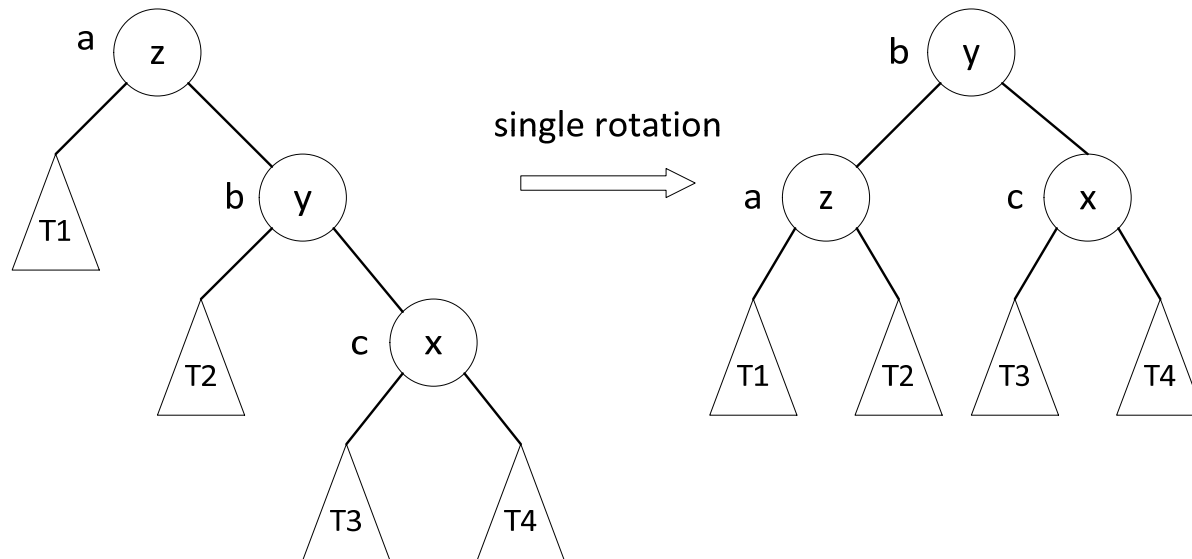
- A *trinode restructuring* performs a broader rebalancing.
- It involves three positions:  $x$ ,  $y$ , and  $z$
- $y$  is the parent of  $x$  and  $z$  is the grandparent of  $x$ .
- Goal: Restructure the subtree rooted at  $z$  to reduce the path length from  $z$  to  $x$  and its subtrees.
- Use secondary labels,  $a$ ,  $b$ , and  $c$ , for the three positions such that  $a$  comes before  $b$  and  $b$  comes before  $c$  in an inorder tree traversal of the tree.
- There are four different configurations. These secondary labels allow us to describe the trinode restructuring operations in a uniform way.

# Balanced Search Trees

- Outline of the algorithm:
  - $(T_1, T_2, T_3, T_4)$  are left-to-right listing of subtrees of  $x$ ,  $y$ , and  $z$ .
  - The subtree rooted at  $z$  is replaced with the subtree rooted at  $b$ .
  - Make  $a$  the left child of  $b$ .
  - Make  $T_1$  and  $T_2$  the left and right subtree of  $a$ , respectively.
  - Make  $c$  the right child of  $b$ .
  - Make  $T_3$  and  $T_4$  the left and right subtree of  $c$ , respectively.

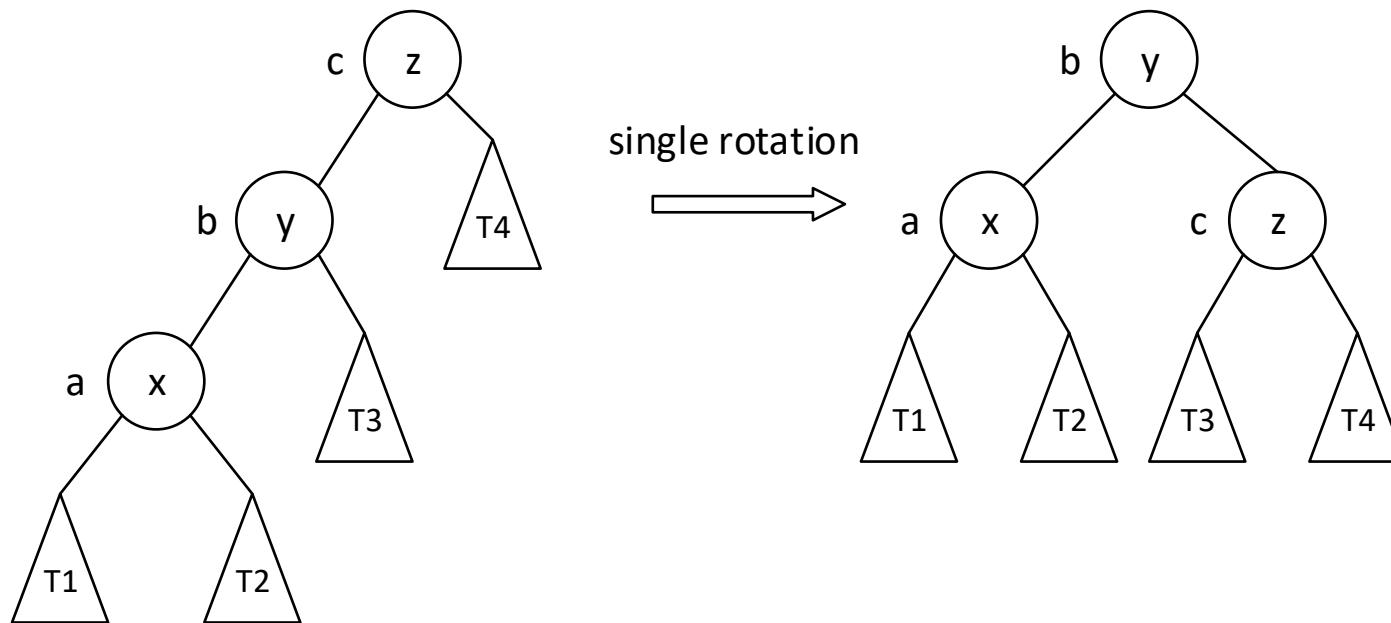
# Balanced Search Trees

- Trinode restructuring: single rotation 1



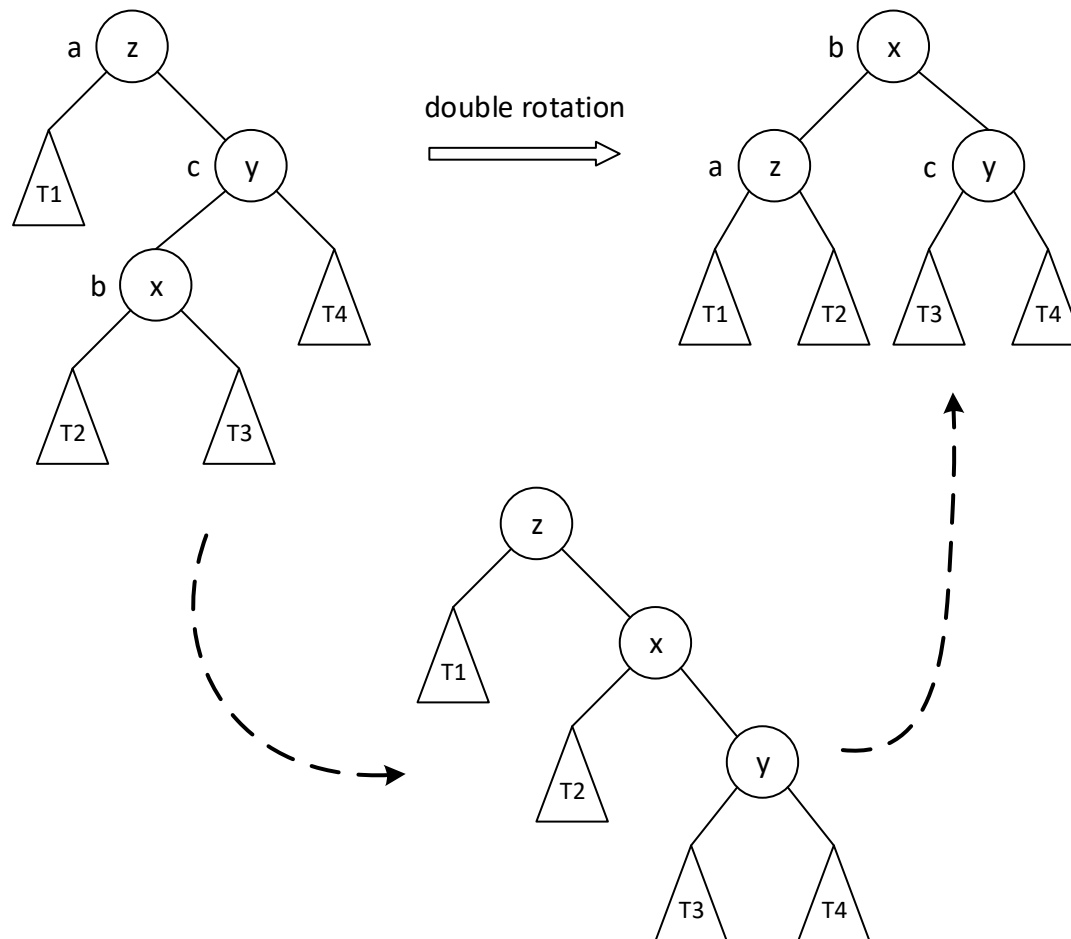
# Balanced Search Trees

- Trinode restructuring: single rotation 2



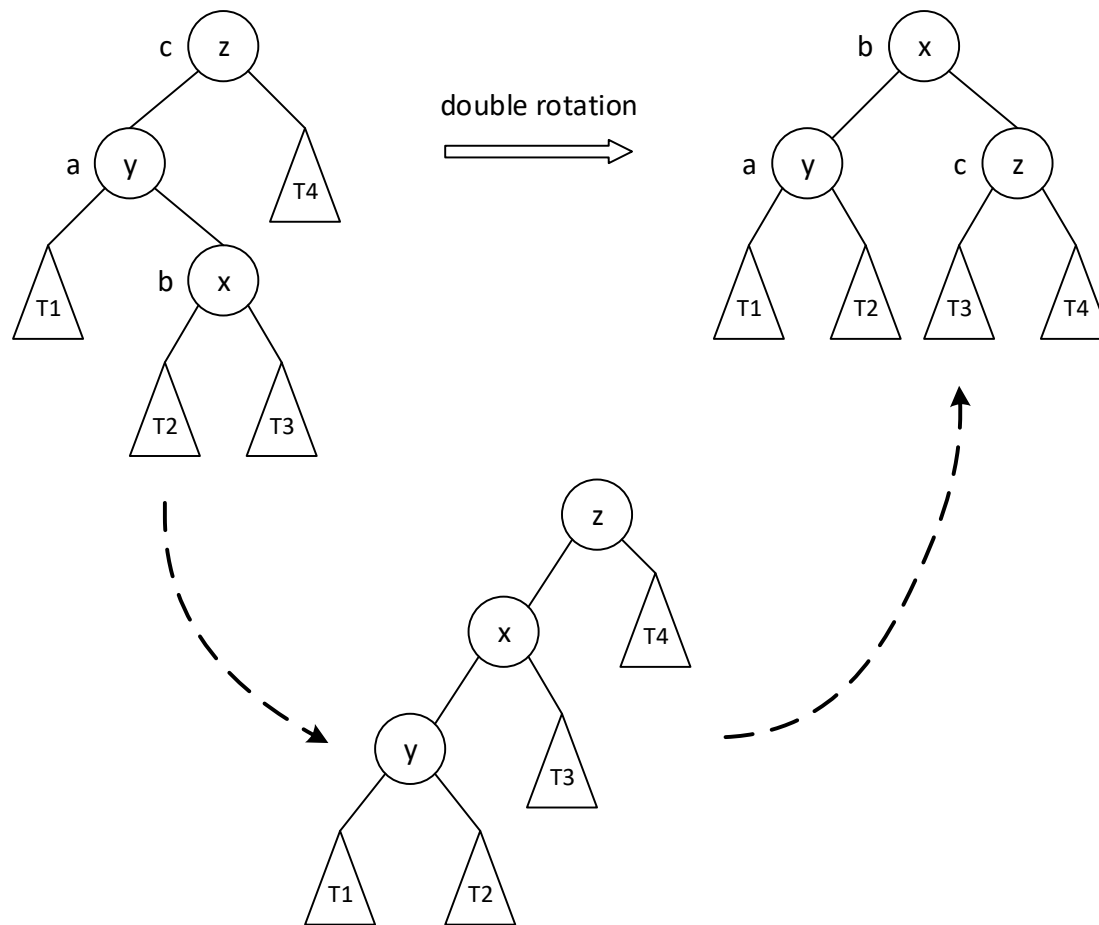
# Balanced Search Trees

- Trinode restructuring: double rotation 1



# Balanced Search Trees

- Trinode restructuring: double rotation 2



# AVL Trees

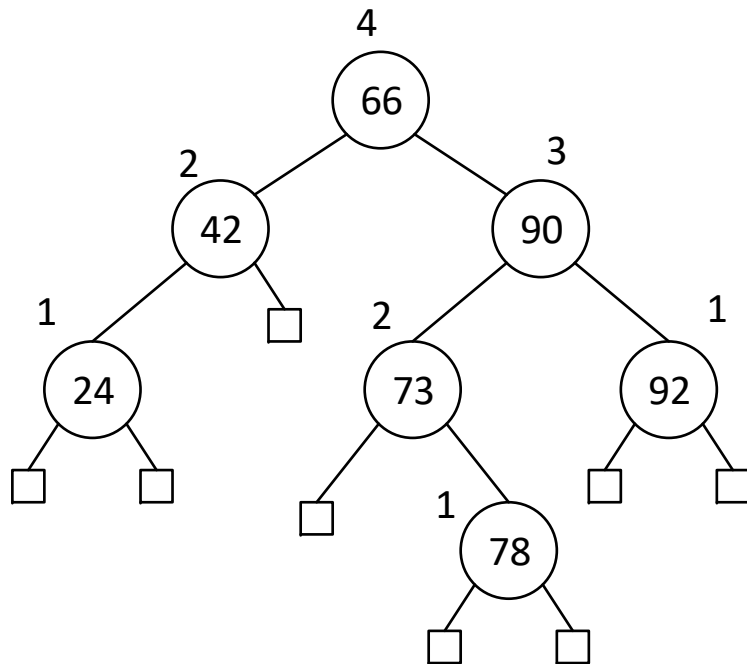
- Recall
  - The *height of a node* is the number of edges on the longest path from that node to a leaf node.
  - The *height of a tree* (or a subtree) is the height of the root of the tree (or a subtree).
  - The height of a leaf node is zero.
- An AVL tree is a binary search tree that satisfies the following *height-balance property*:

For every internal node  $p$  of  $T$ , the heights of the children of  $p$  differ by at most one.

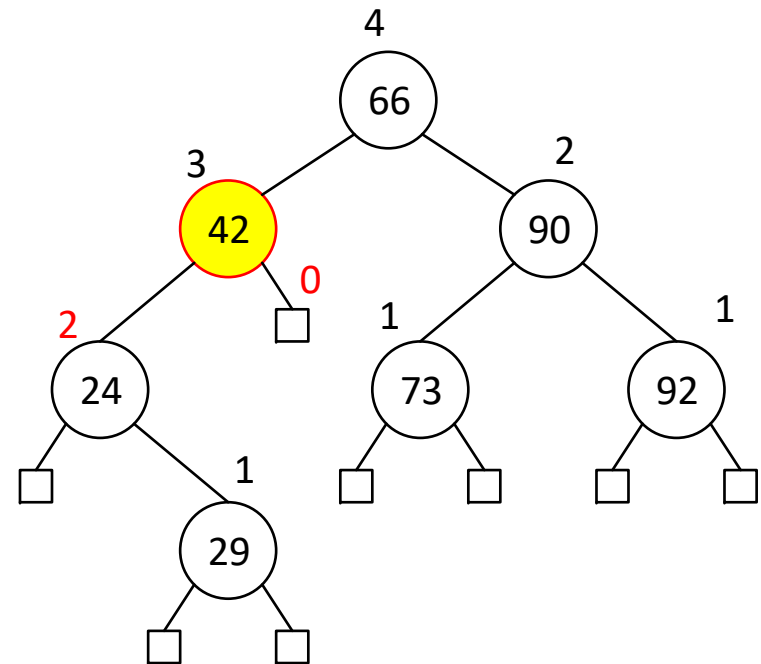
# AVL Trees

- AVL tree example:

AVL tree



Not an AVL tree



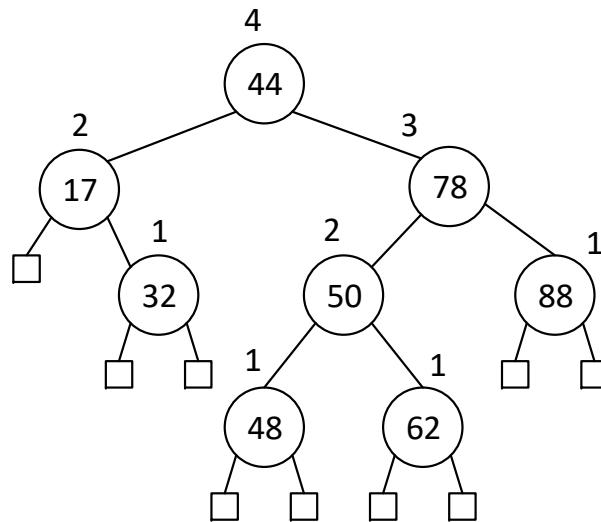


# AVL Trees

- Updating an AVL tree
  - A node  $p$  in a binary search tree is said to be *balanced* if the heights of  $p$ 's children differ by at most one.
  - Otherwise, a node is said to be *unbalanced*.
  - Therefore, every node in an AVL tree is balanced.
  - When we insert a node to an AVL tree or remove a node from an AVL tree, the resulting tree may violate the height-balance property.
  - So, we need to perform *post-processing*.
  - We will discuss only insertion.

# AVL Trees

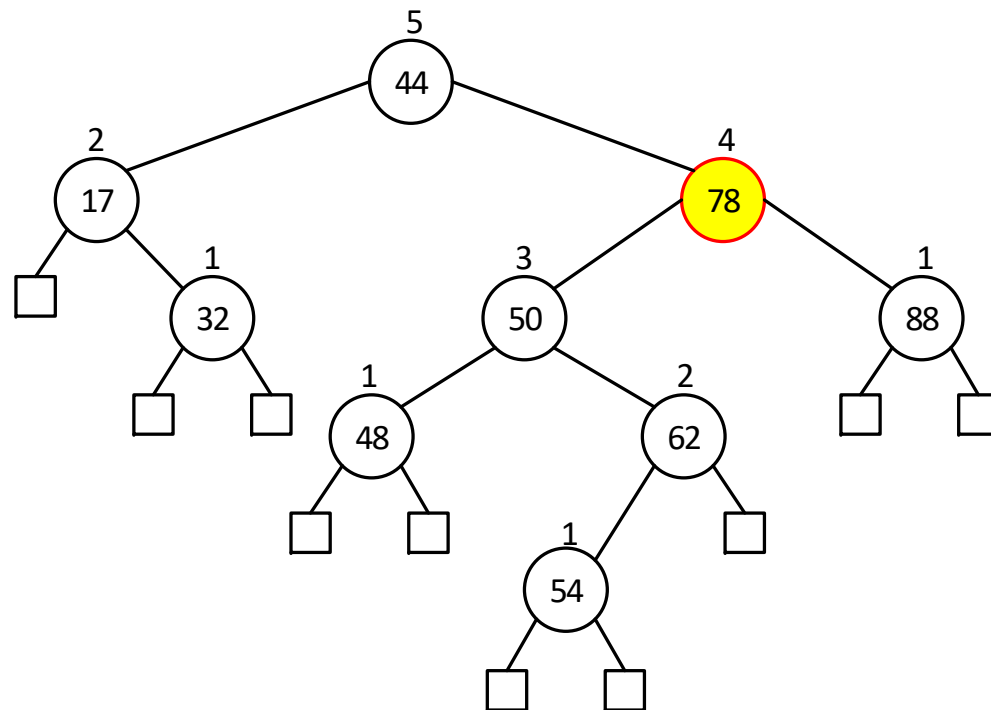
- When a node is inserted, the leaf node  $p$  where the new node is inserted becomes an internal node (with the entry of the new node).
- So, ancestors of  $p$  may be unbalanced.
- Restructuring is necessary.
- Consider the following tree:



# AVL Trees

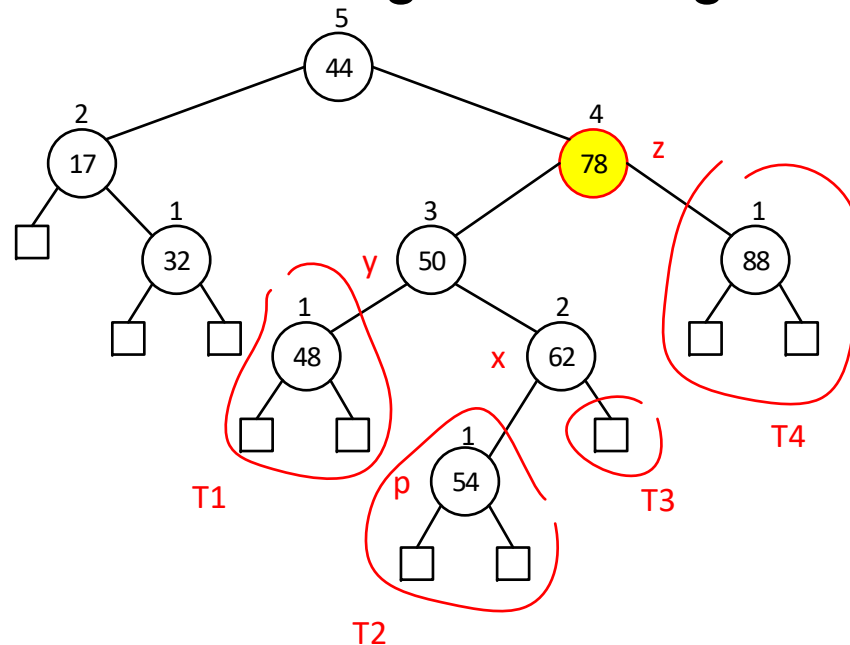
- After inserting 54, the node with 78 is unbalanced

After insertion, before rebalancing



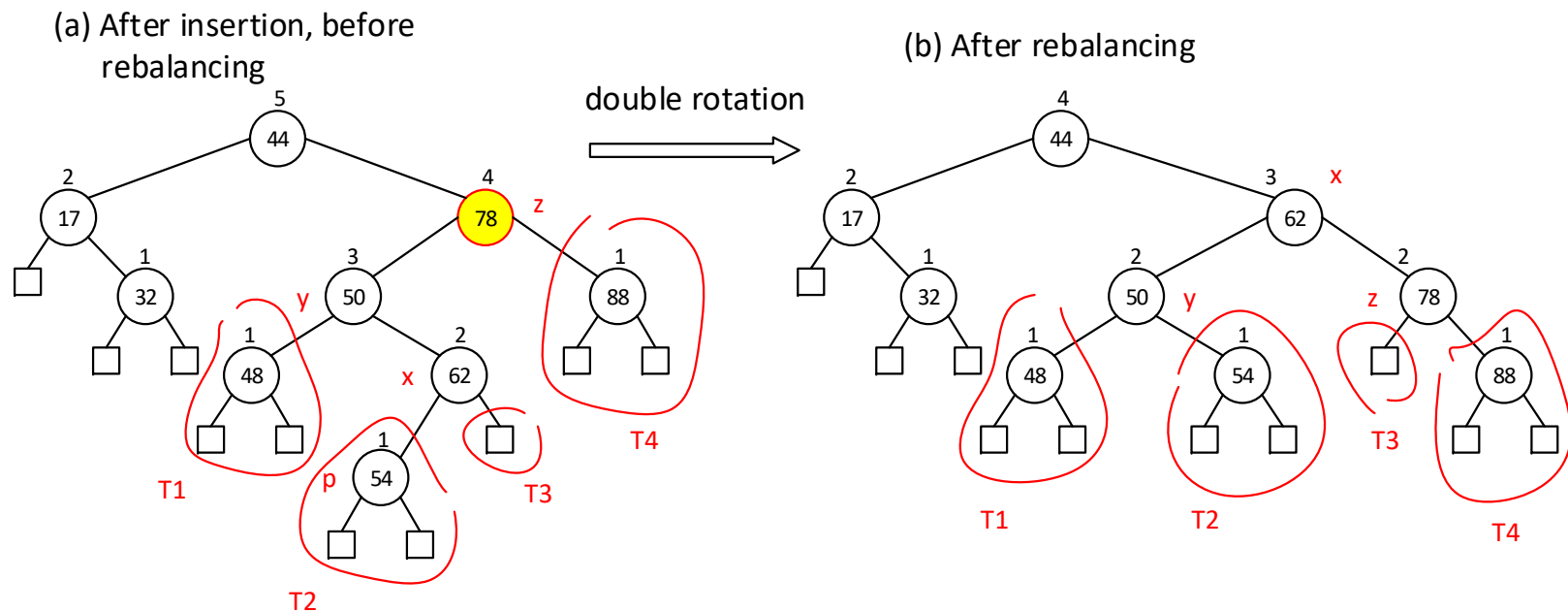
# AVL Trees

- Post-processing
  - Search-and-repair strategy
  - Search a node  $z$  that is the lowest ancestor of  $p$  that is unbalanced.
  - $y$  is  $z$ 's child with the greater height
  - $x$  is  $y$ 's child with the greater height



# AVL Trees

- Perform double rotation to rebalanced the tree



# References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.