CMPT 225: Data Structures & Programming – Unit 24 – AVL Trees

Dr. Jack Thomas
Simon Fraser University
Spring 2021

Today's Topics

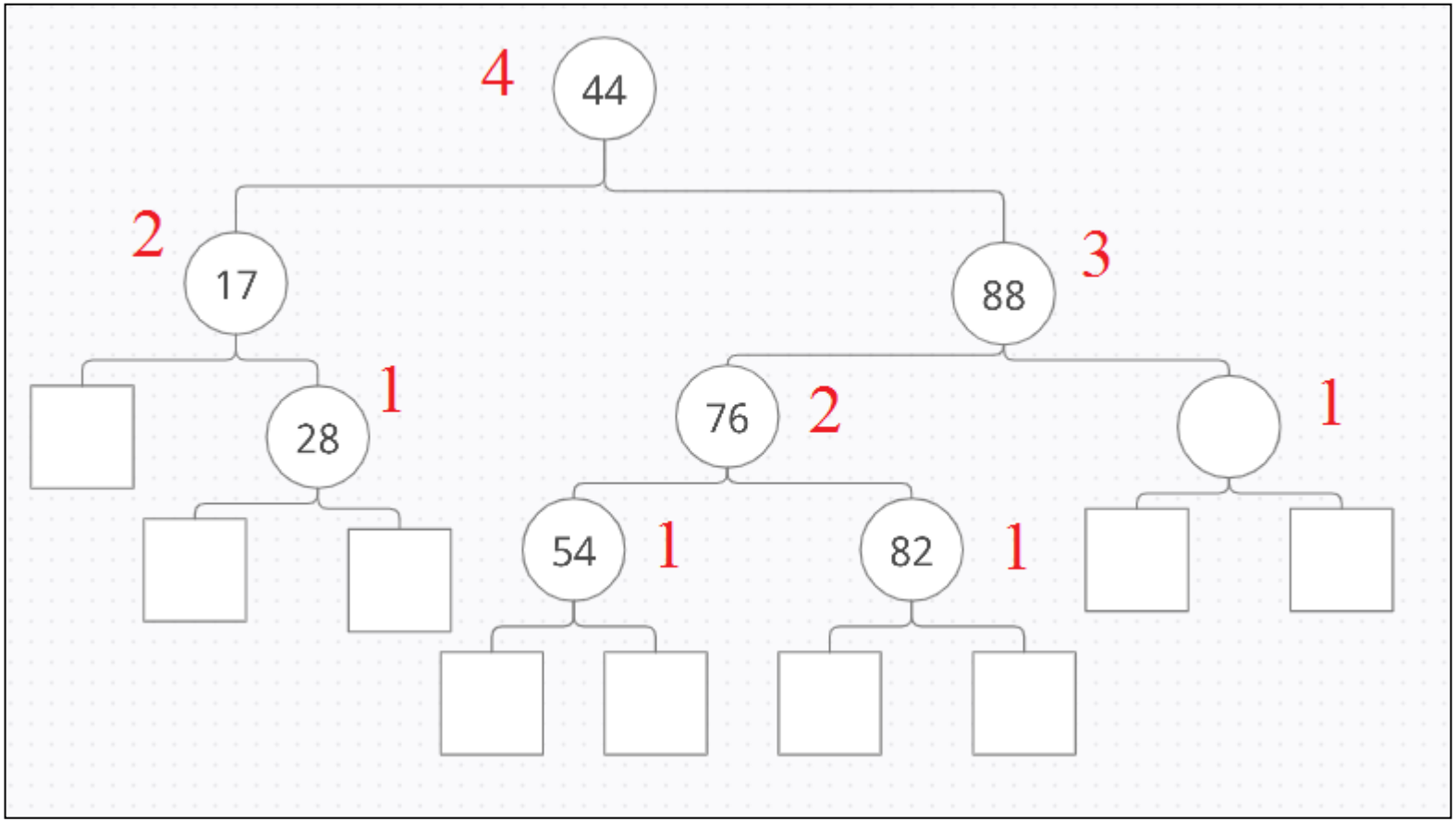
- AVL Trees and the Height-Balancing Property
- Updating Operations
- Performance of AVL
- Java Implementation

What's an AVL Tree?

- AVL stands for Adelson-Velsky and Landis, the inventors, which is kind of impressive getting your name in on a fundamental data structure.
- AVL Trees are a way to address the issue in BSTs that while we can bound our search time to the height of the tree, there's nothing bounding the height from the number of inputs.
- AVL Trees introduce the Height-Balancing Property to rebalance the tree whenever the height starts growing too fast.

The Height Balancing Property

- "For every internal node v of T, the heights of the children of v differ by at most 1."



Some More Properties

- This means that every subtree of an AVL tree is also an AVL Tree.
- It also leads to the assertion that “the height of an AVL tree storing n entries is O(log n).”
- So, a BST that’s also an AVL would have a height bounded by log n, finally getting us those efficient searches!

Making a BST an AVL

- An AVL has the same methods as a BST, but now has to maintain balance the same way it maintains ordering.
- Since balance is defined as the heights of the children of each node differing by at most one, it'll now be necessary for each node (or the position storing that node) to keep track of its current height.
- That way, when the tree changes through a removal or insertion, we can quickly update the affected heights and check if the tree has become unbalanced.

Going Out Of Balance

- A change at the bottom of the tree can affect the balance all the way up to the root.
- Insertion and removal methods will need a rebalance method to traverse from the affected node up to the root, looking for unbalanced nodes.
- When an unbalanced node is found, the process to fix it is called **tri-node restructuring**, and the algorithm for it is a bit of a doozy.

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.

Woof

- Yeah I know it’s a lot.
- Don’t worry, we’ll walk it through one step at a time with insertion and removal examples.
- But first, let’s talk about how to visualize what the restructure algorithm is trying to accomplish.
- Essentially, trinode restructuring is about taking three unbalanced nodes and rotating them.

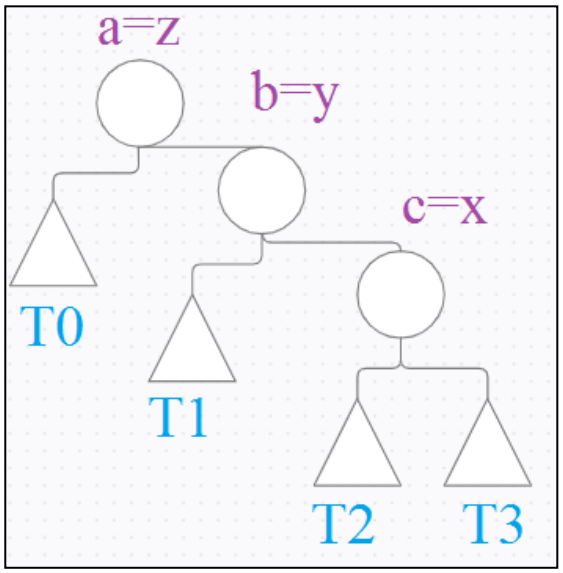
Finding Our Rotation: Who are Z, Y, and X?

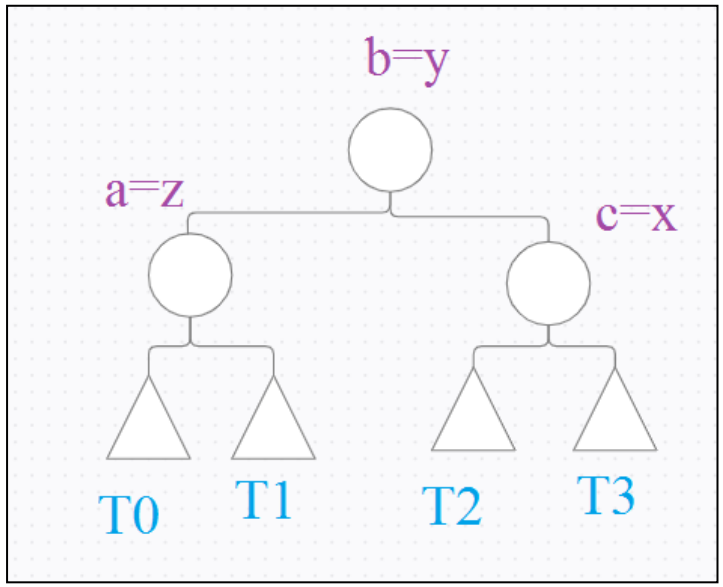
- In all rotations, z will be the first node we encounter going up from w (the node we added or removed) toward the root of T such that z is unbalanced.
- Then y will be the child of z with higher height (an ancestor of w).
- Finally, x will be the child of y with higher height (no ties, or else this wouldn't be unbalanced).

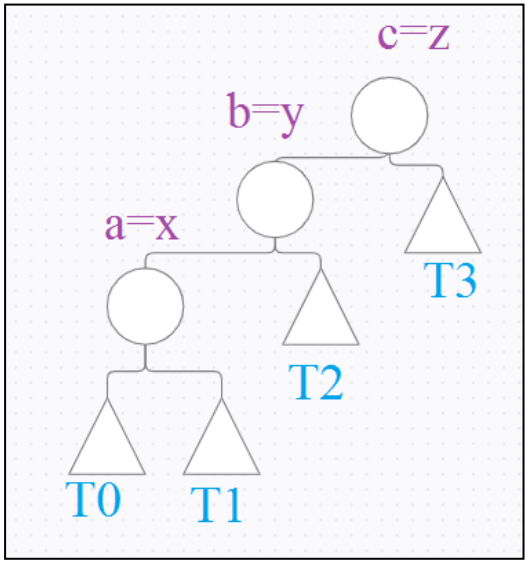
Single or Double Rotation

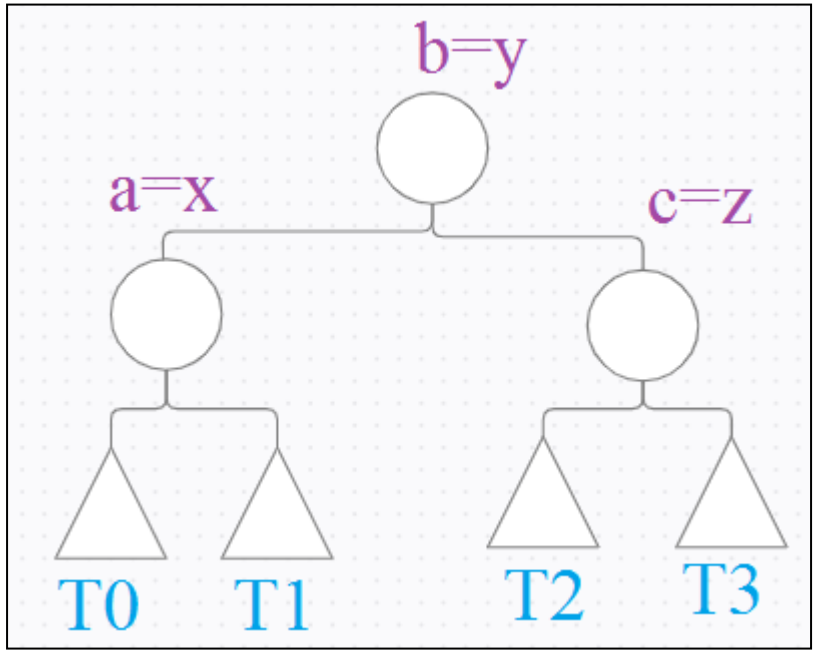
- If b = y, then this is called a Single Rotation, because we can imagine we're "rotating" y over z.

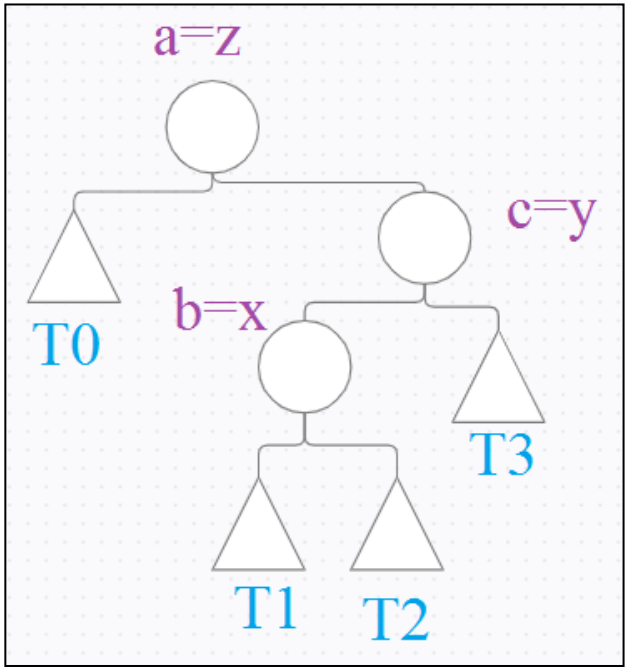
- Otherwise, if b = x, it's a double rotation, where we first rotate x over y and then over z.

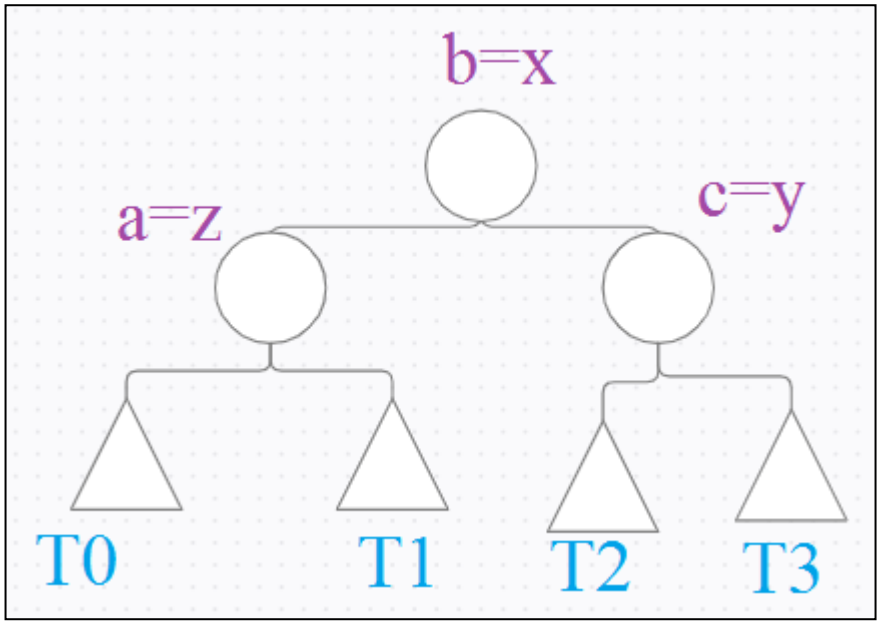

a=z
b=y
c=x
T0
T1
T2
T3

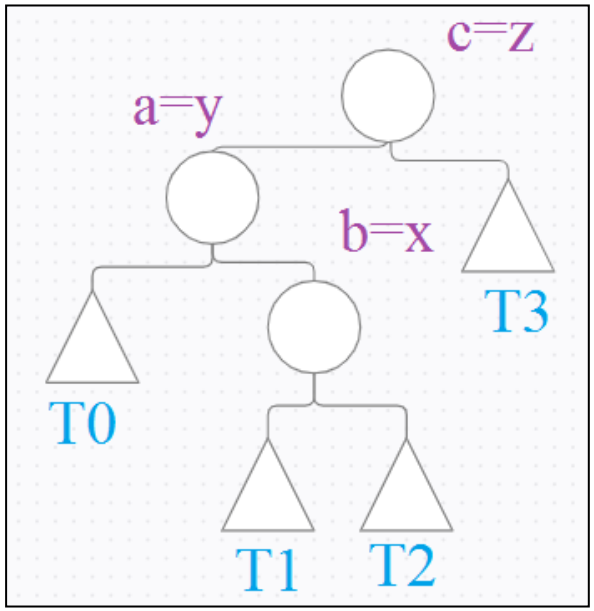

b=y
a=z
c=x
T0
T1
T2
T3

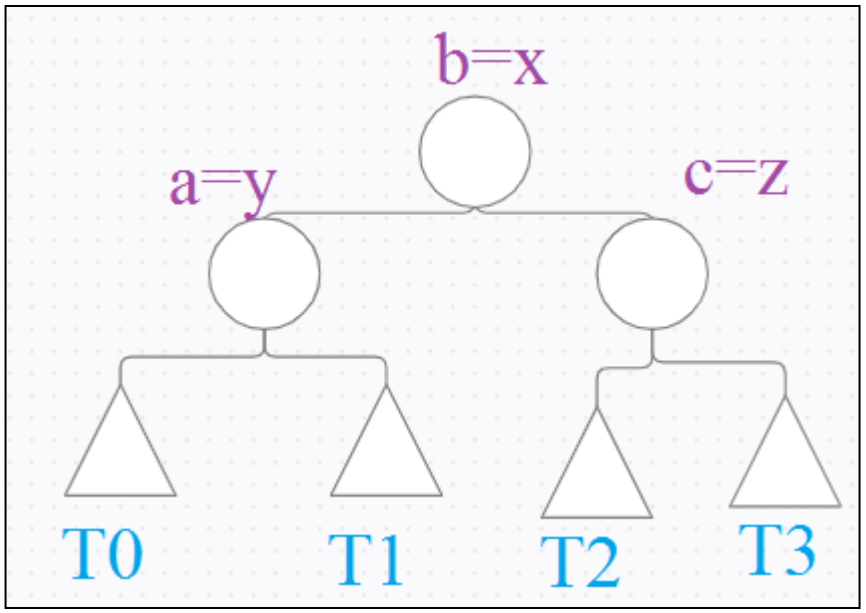

c=z
b=y
a=x
T3
T2
T0
T1


b=y
a=x
c=z
T0
T1
T2
T3


a=z
c=y
T0
b=x
T3
T1
T2

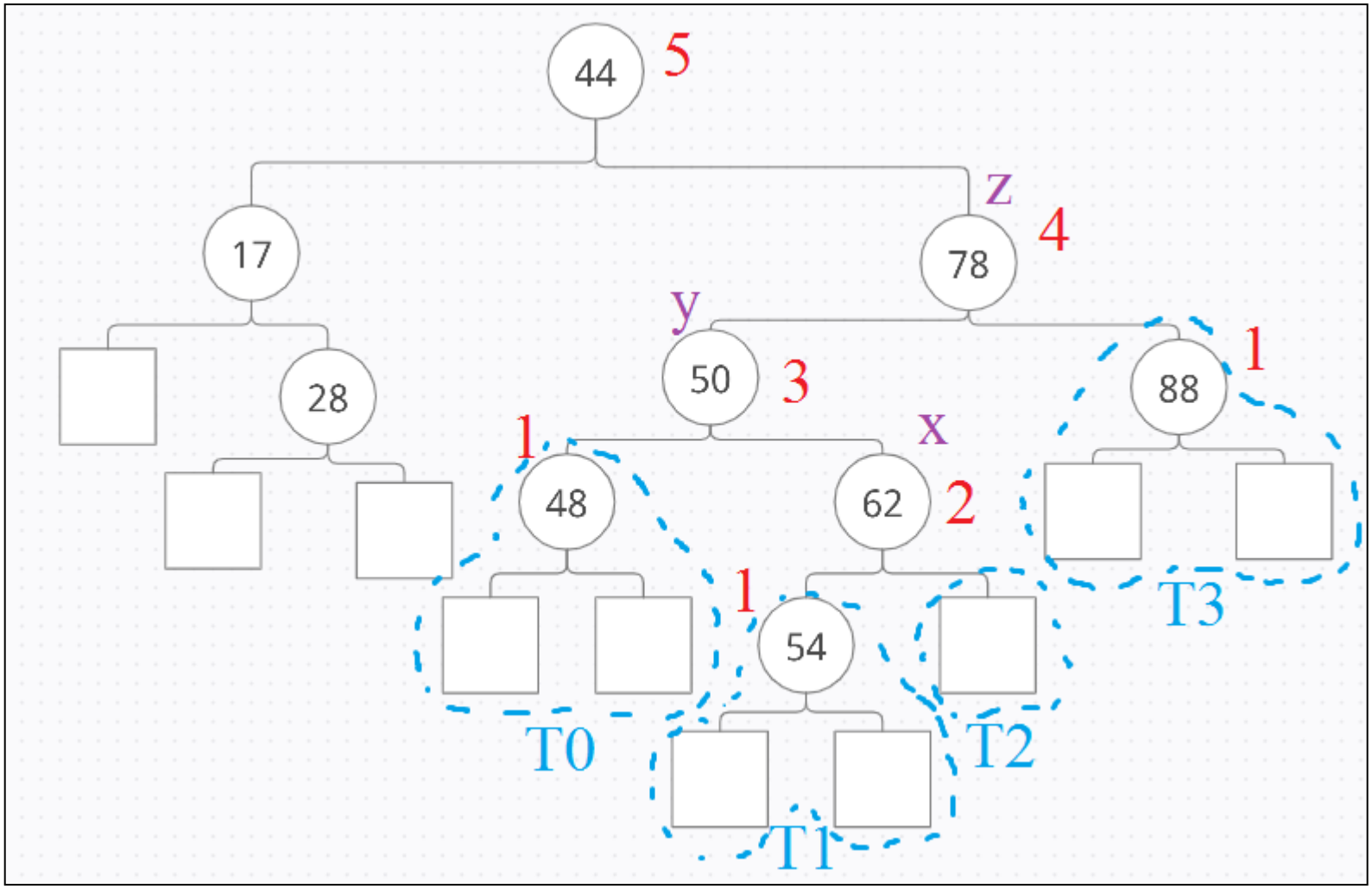

b=x
a=z
c=y
T0
T1
T2
T3


c=z
a=y
b=x
T3
T0
T1
T2

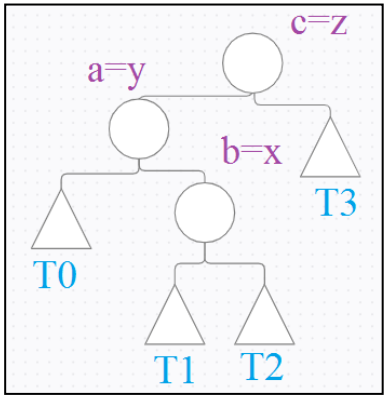

b=x
a=y
c=z
T0
T1
T2
T3

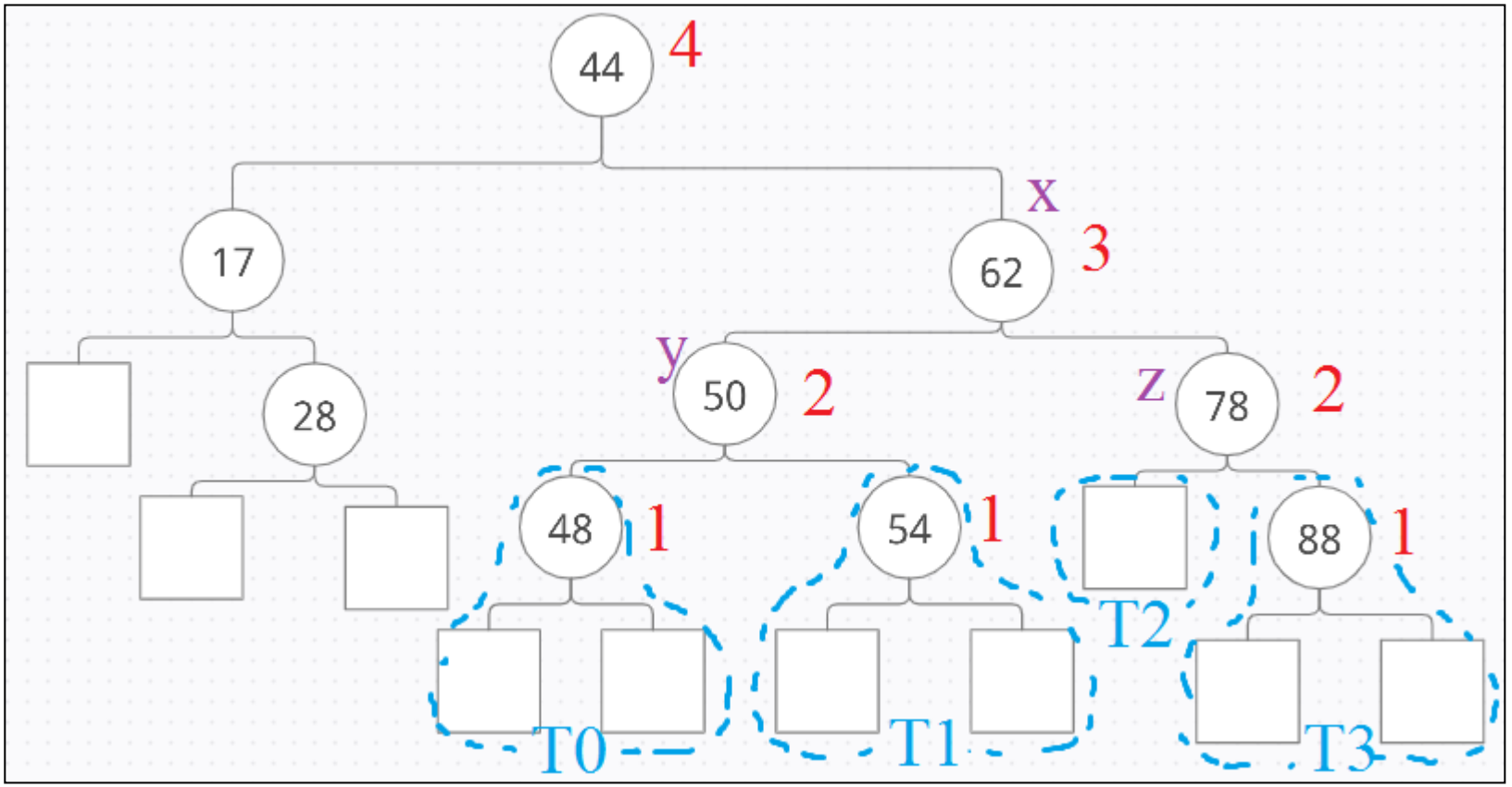
Let’s See Some Examples

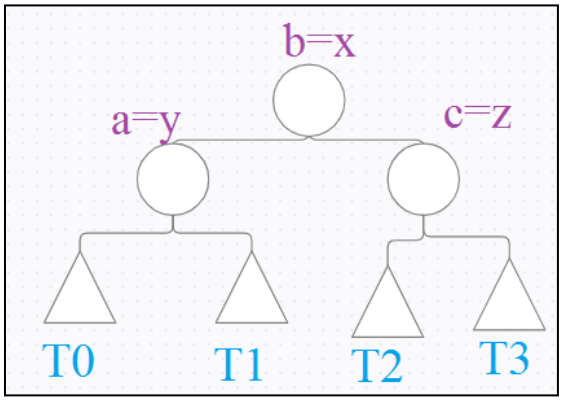
- We’ll start by taking our tree from earlier and adding a node with a key of 54.
- This will lead to a double-rotation.



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.



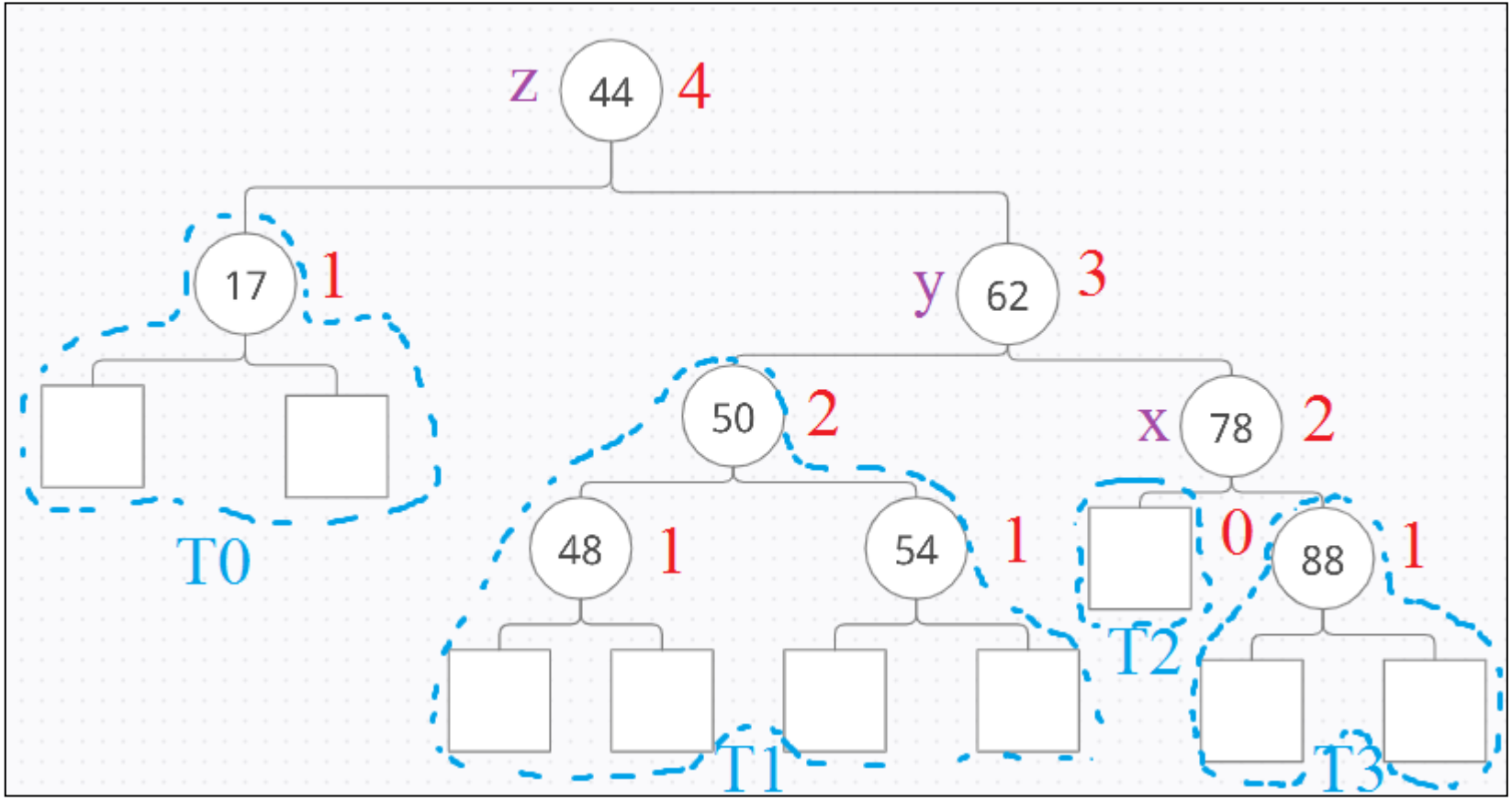

44
4
x
17
62
3
y
50
2
z
78
2
28
48
1
54
1
88
1
T2
T0
T1
T3


b=x
a=y
c=z
T0
T1
T2
T3

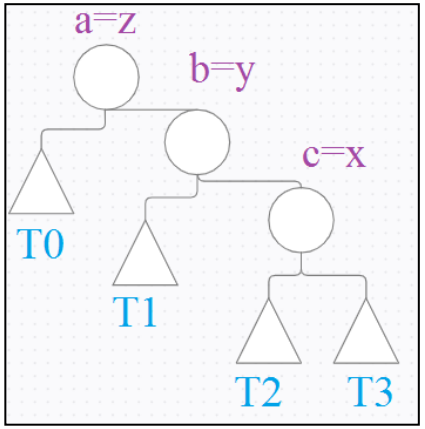
Now For Removal

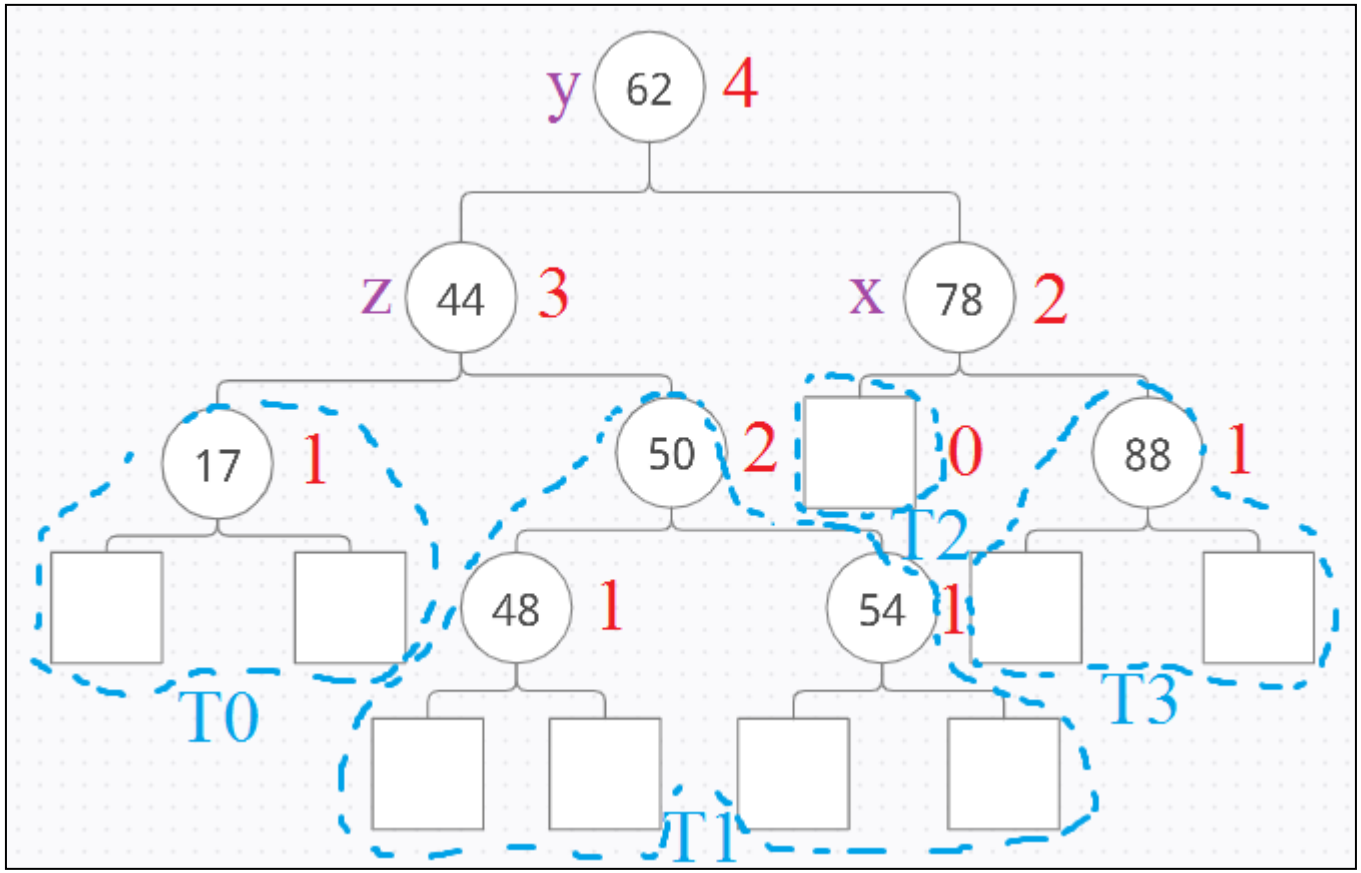
- We'll take our updated tree and remove the node with a key of 28 from it.

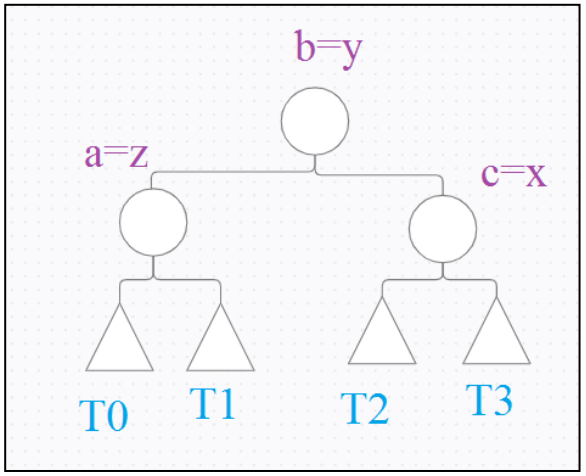
- This will require a single rotation.



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.




y
62
4
z
44
3
x
78
2
17
1
50
2
0
88
1
T2
48
1
54
1
T0
T3
T1


b=y
a=z
c=x
T0
T1
T2
T3

Is There A Java Version, Or…?

- Nah.
- “Oh come on, really?” I hear you saying, especially since I said it can give us some pretty efficient implementations for certain map functions.
- There IS a balanced tree used to implement a map built into Java, but we’re not there yet.
- Otherwise, you’ll have to implement all of this manually as part of any BST you build if you want to ensure it’s as efficient as possible.

Recap – A Balanced Perspective

- An **AVL Tree** is a self-balancing variety of Binary Search Tree.
- AVL Trees are constrained such that **the height of each node's children may only differ by one**, which keeps the height of the tree overall bound to log n, and searches to O(log n).
- By adding a **rebalancing** step to restructure the tree after every insertion or removal, we can maintain this height bound.
- The process of **trinode restructuring** involves matching the unbalanced node with the right kind of **rotation**.