EECS 2011 M: Fundamentals of Data Structures

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/2011M`
Also on Moodle

Binary Search Trees

Ch. 11.1-11.4

- Motivation
- Binary Search Trees
- Increasing efficiency: height-balanced binary trees

Note: Some slides in this presentation have been adapted from the author's slides.

Ordered Maps

Consider the problem of storing a map where

- Keys are assumed to come from a total order
- Items are stored in order by their keys
- This allows us to support nearest neighbor queries:
  - Item with largest key less than or equal to k
  - Item with smallest key greater than or equal to k

What is a good data structure for supporting such queries?

Array Implementation: Lookup Tables

aka Search Tables

- Items stored in an array-based sequence, sorted by key (using an appropriate comparator)
- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - at each step, the number of candidate items is halved
  - terminates after $O(\log n)$ steps
- Insertion, Removal: $\Omega(n)$ time in the worst case, because of shifting involved

OK for small maps, or when insertions, deletions are rare

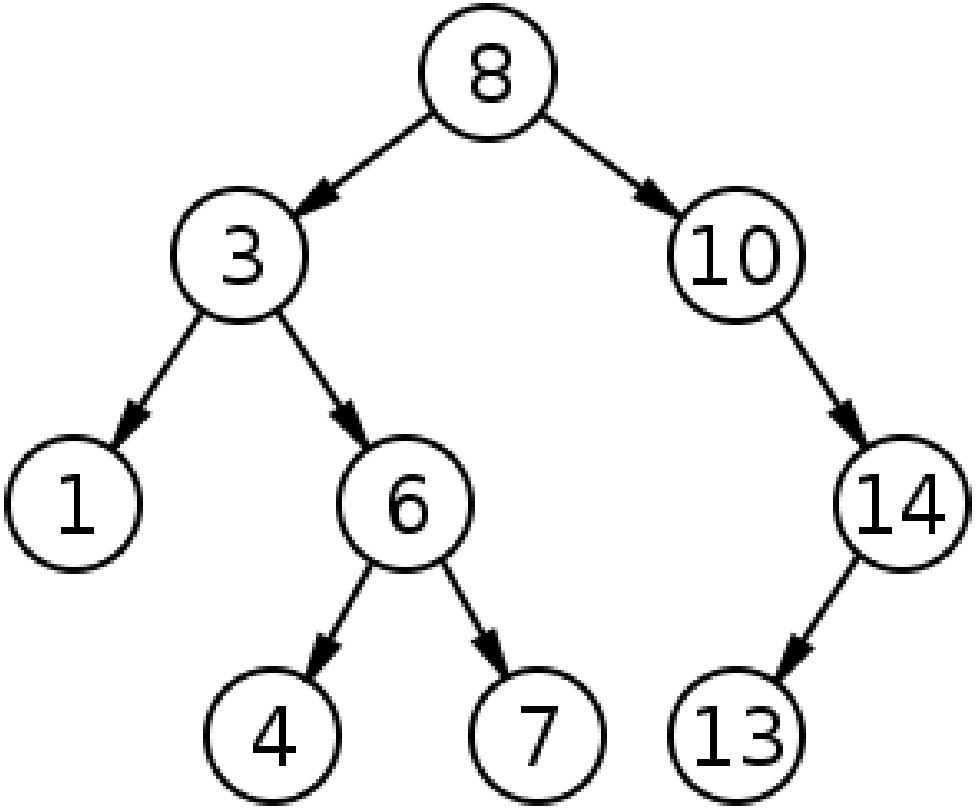
Binary Search Trees (BST)

More efficient alternative to lookup tables
A BST is a binary tree that

- stores keys (or key-value entries) at its internal nodes and satisfies 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 in-order traversal of a BST lists the keys in increasing order

Binary Search Trees (BST) - Example

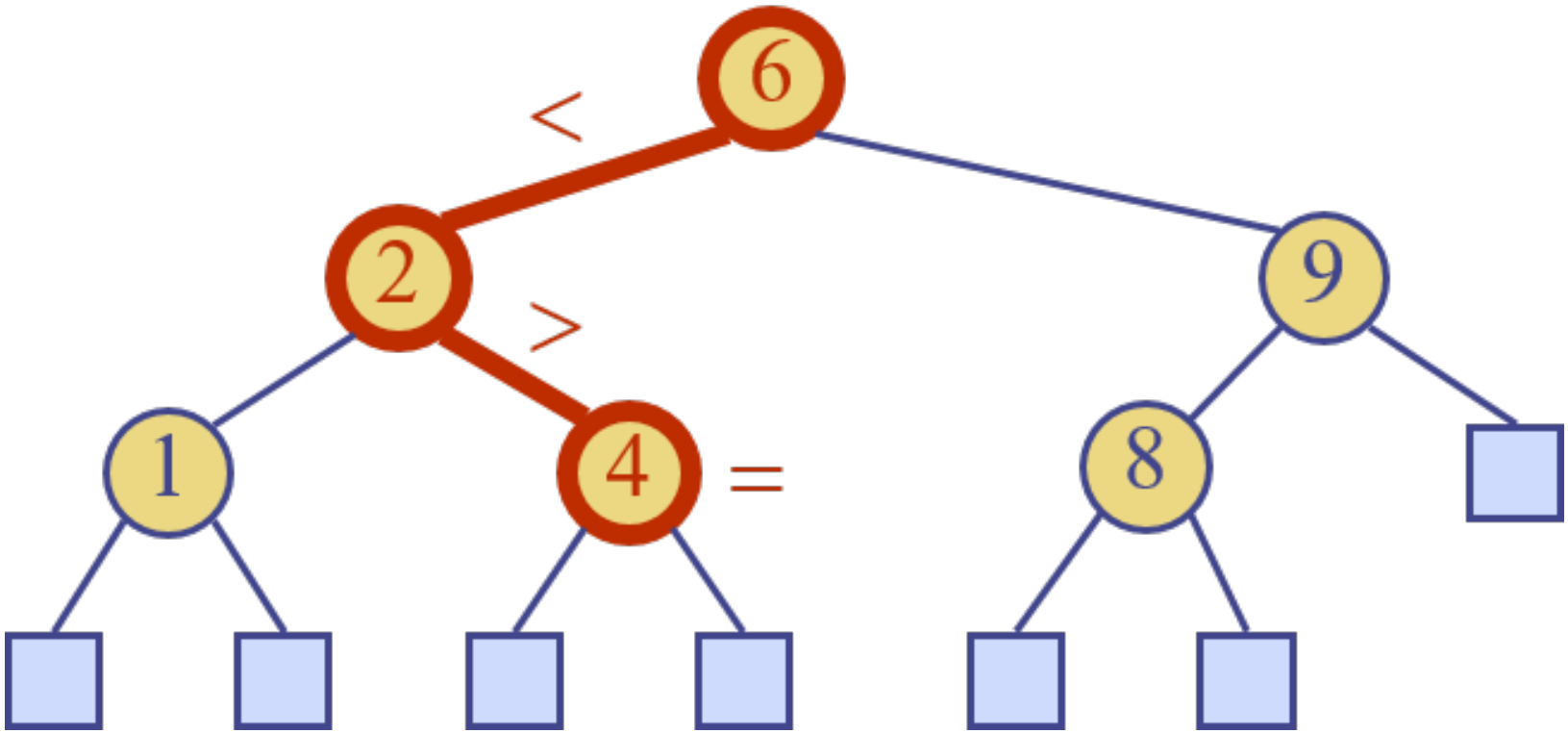
Leaf nodes are not shown



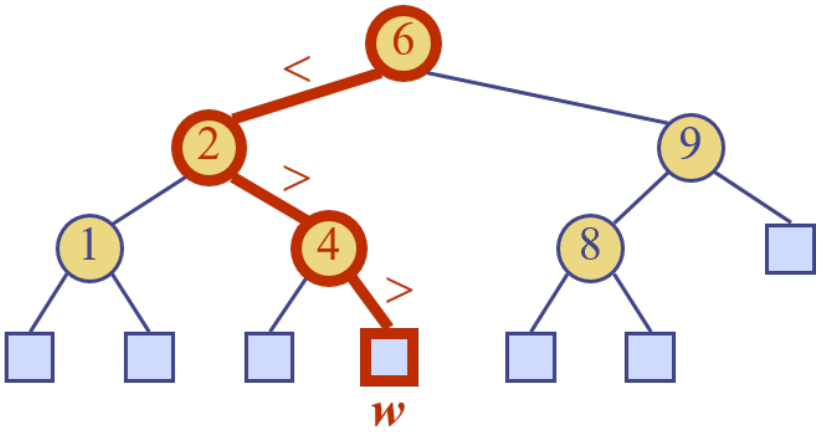
Public Domain, https://commons.wikimedia.org/w/index.php?curid=488330

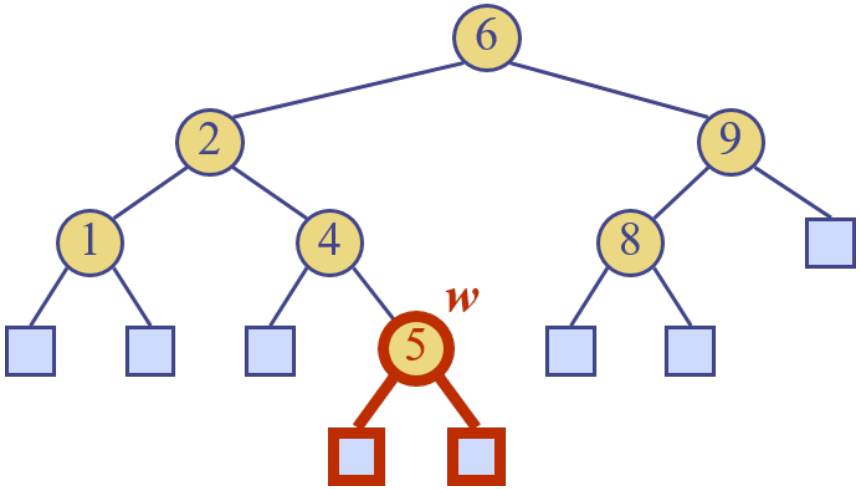
Binary Search Trees (BST) - Searching

Note the similarity with binary search (hence the name)

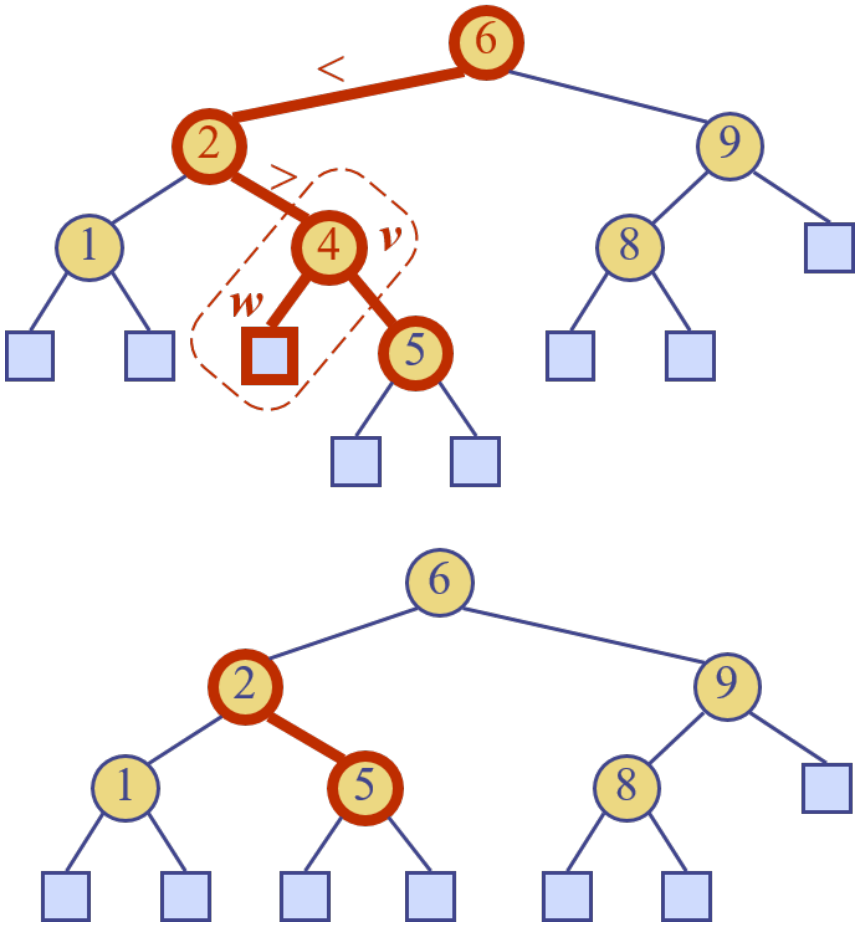


Binary Search Trees (BST) - Insertion

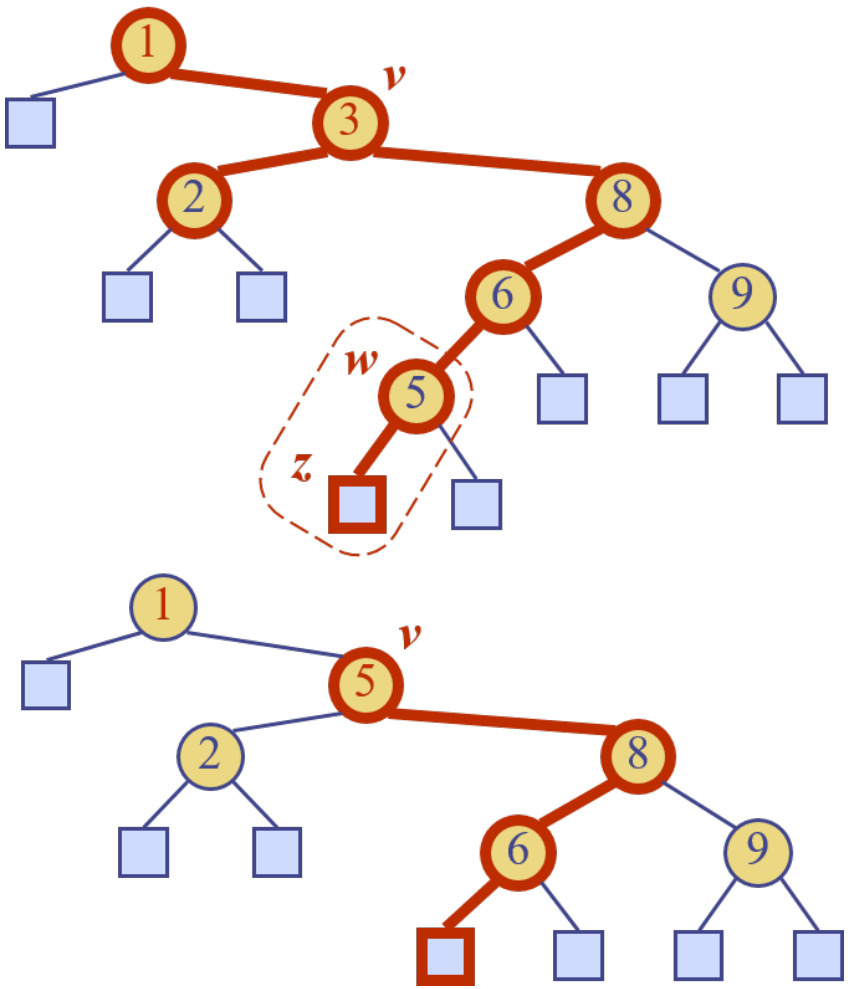




Binary Search Trees (BST) - Deletion 1



Binary Search Trees (BST) - Deletion 2

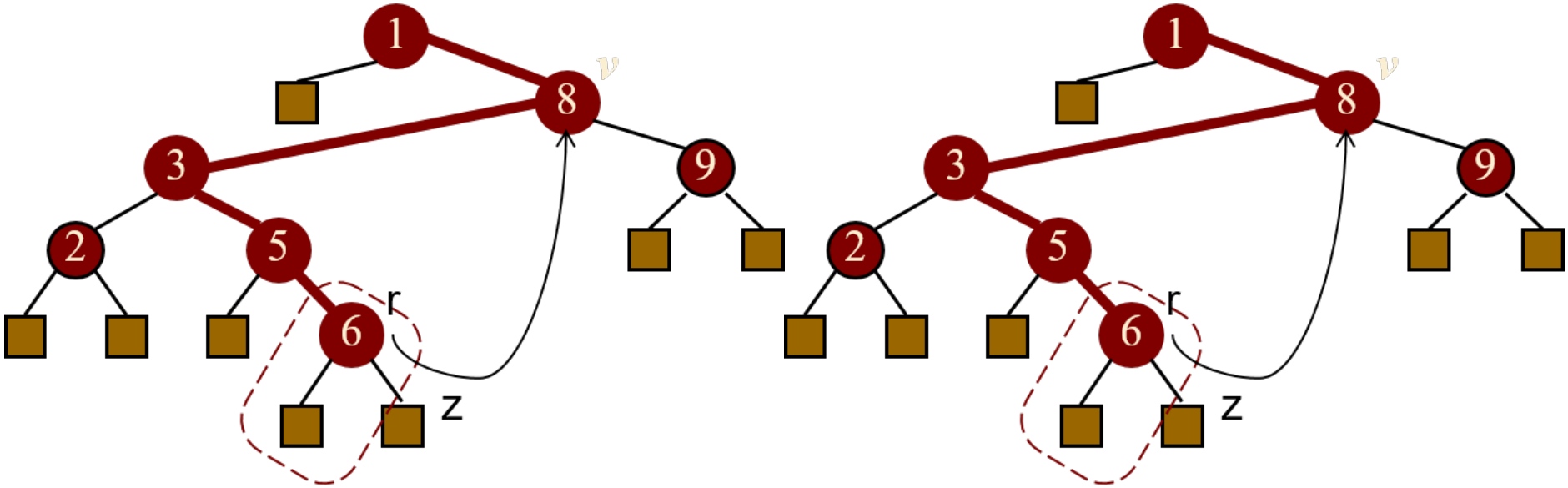


p has 2 internal children

- we find the internal node r that follows p in an in-order traversal
- we copy the entry stored at r into p, and delete the node at position r (which cannot have a right child) using the previous method

Binary Search Trees (BST) - Deletion 3

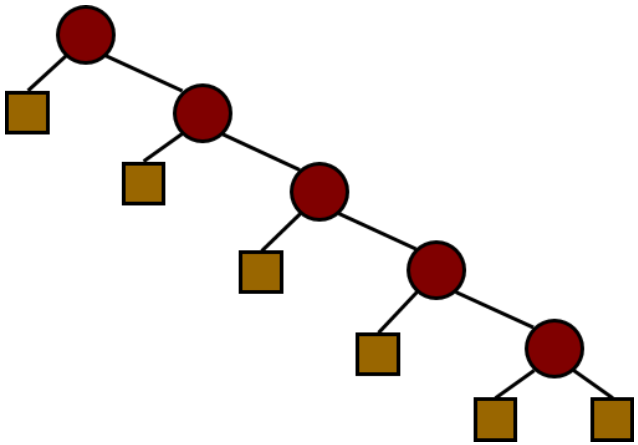
Can also use the node that precedes p: remove(8)



Binary Search Trees (BST) - Performance

- the space used is $\theta(n)$
- methods get, put and remove take $\theta(h)$ time
- The height h is $\Omega(n)$ in the worst case and $O(logn)$ in the best case
- Easy way to generate a skewed tree: insert in sorted order

Height-balanced Binary Search Trees

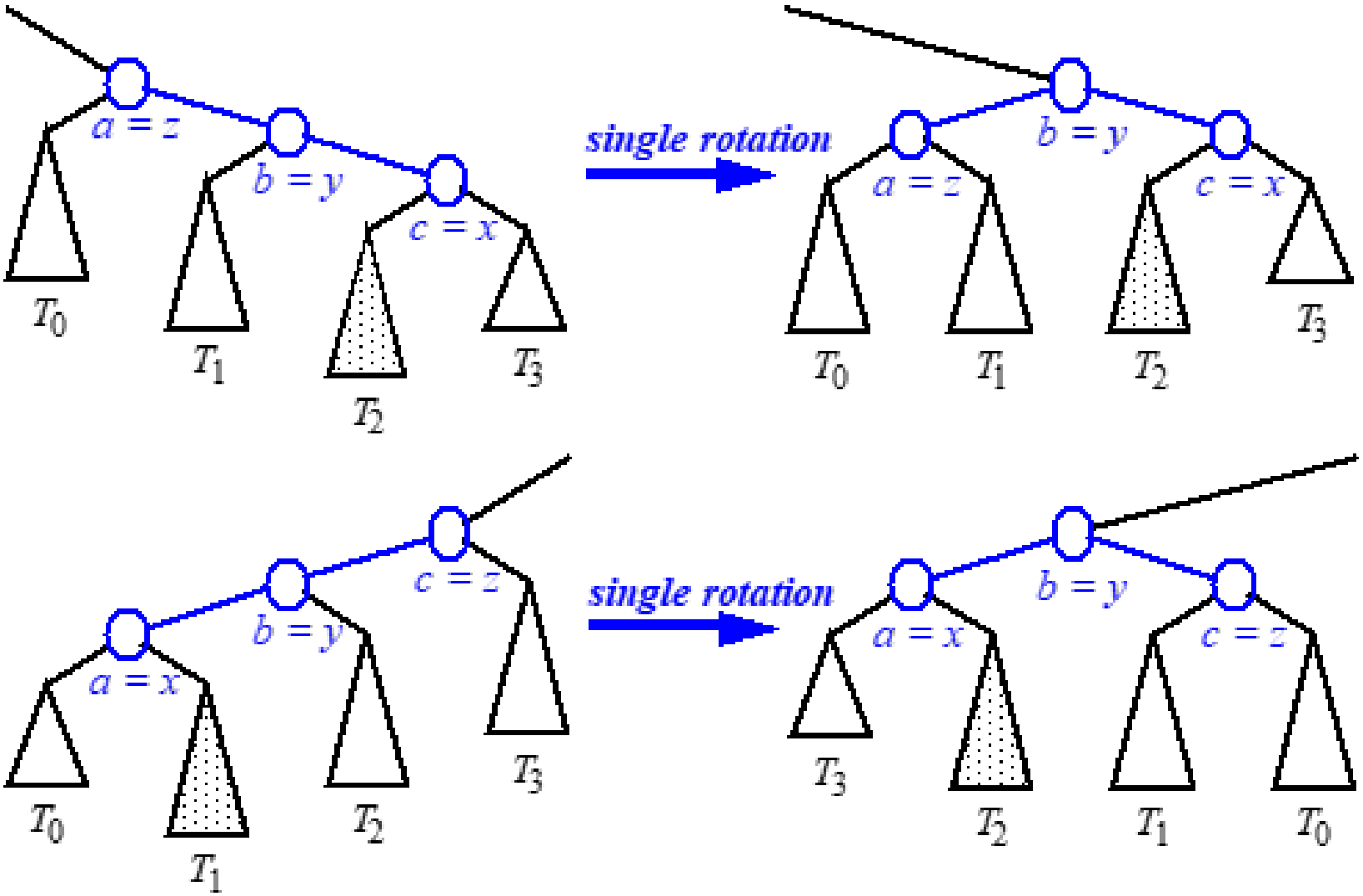


Q: Can this be made more balanced?

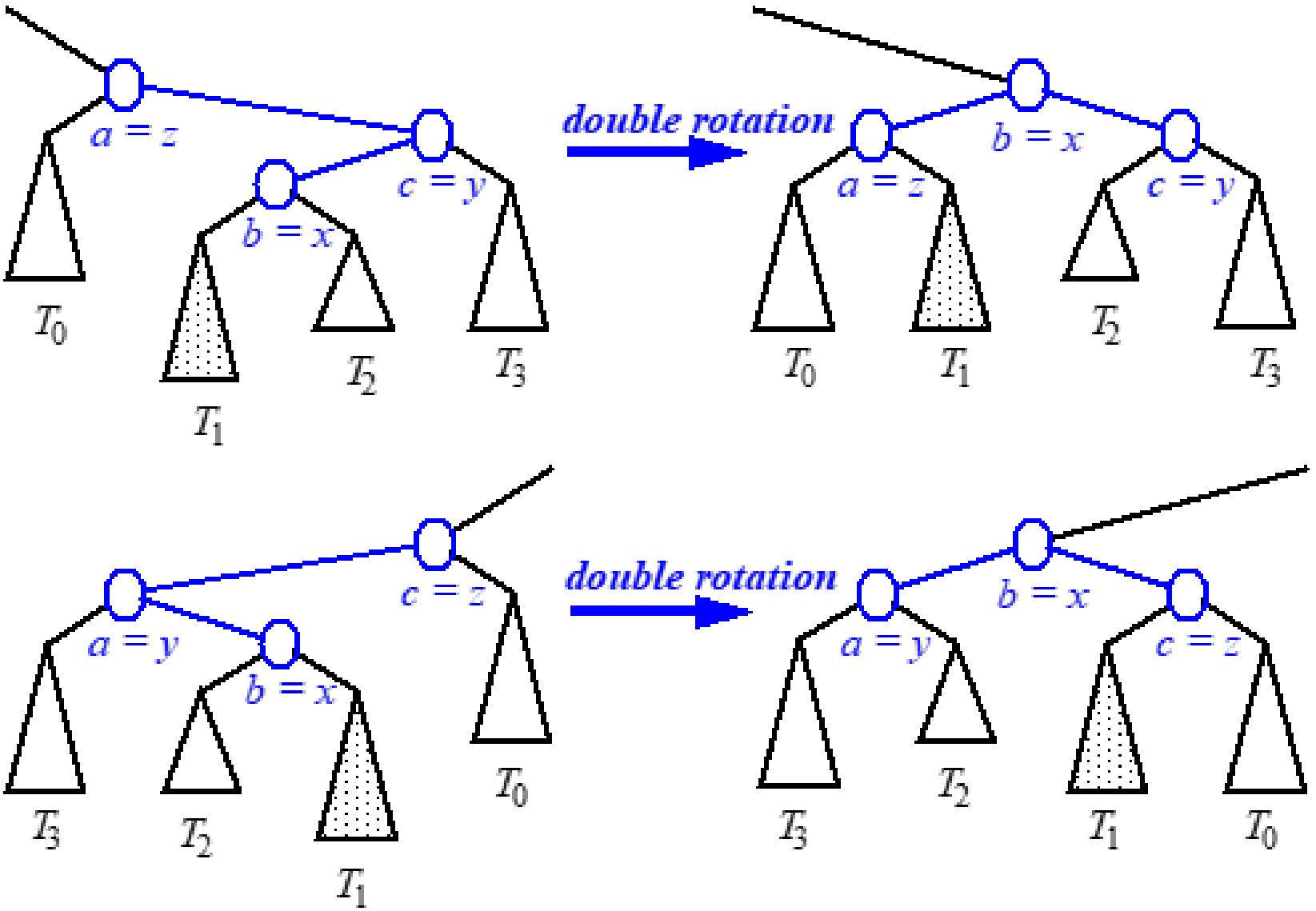
A: Yes. Many different algorithms exist. We will look at

- AVL trees
- Splay Trees

General Rebalancing Techniques - 1

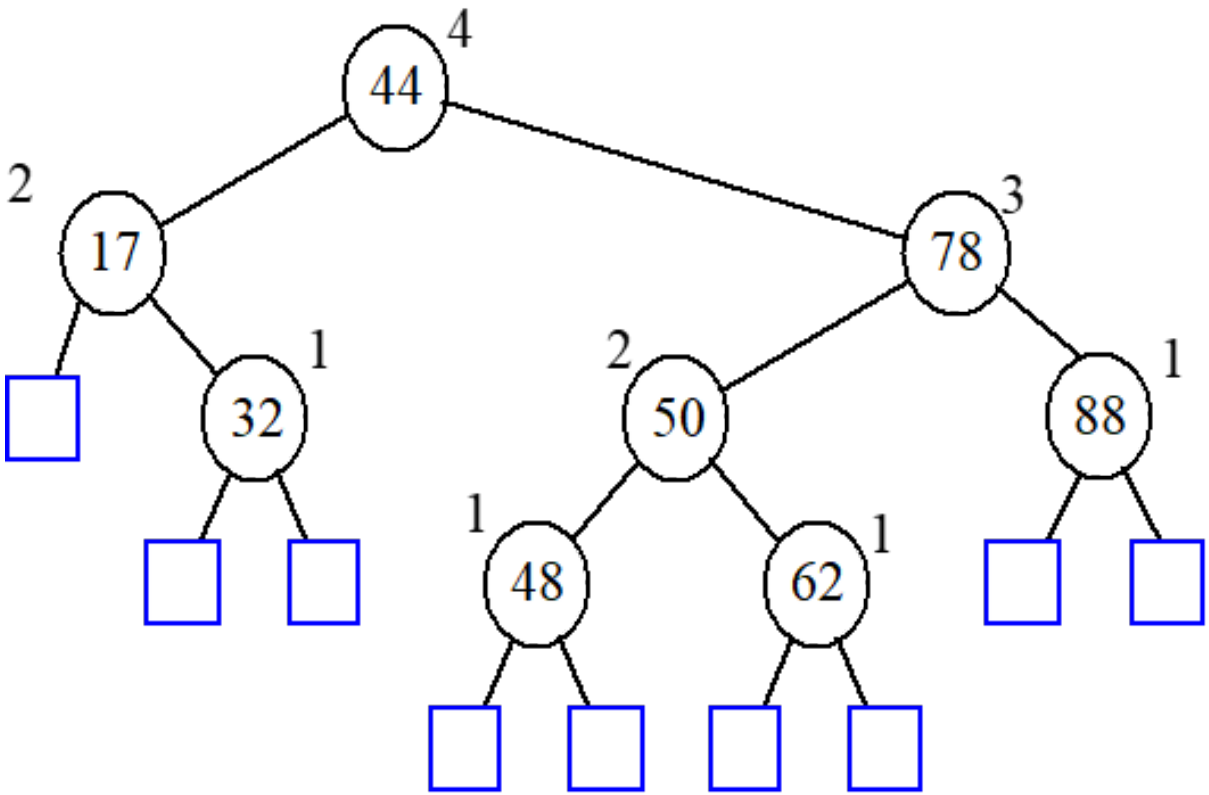


General Rebalancing Techniques - 2



AVL Trees

An AVL Tree is a BST such that for every internal node v, the heights of the children of v can differ by at most 1



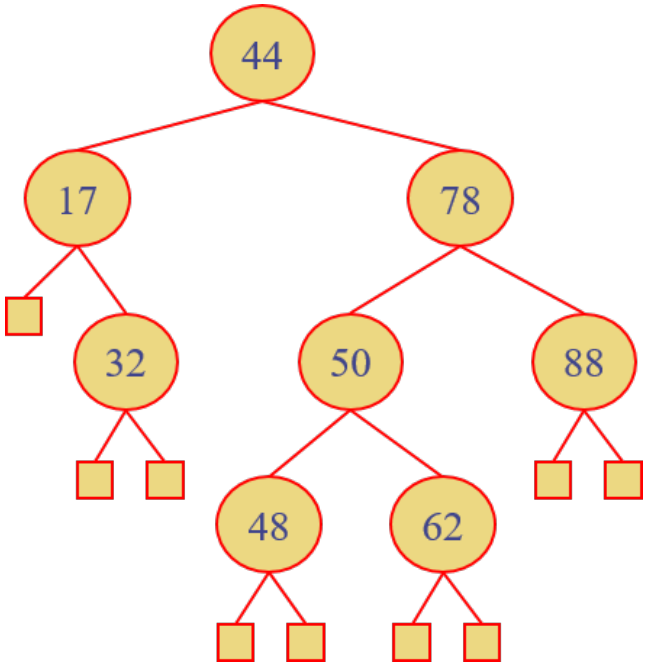
Height of an AVL Tree in $O \log n)$

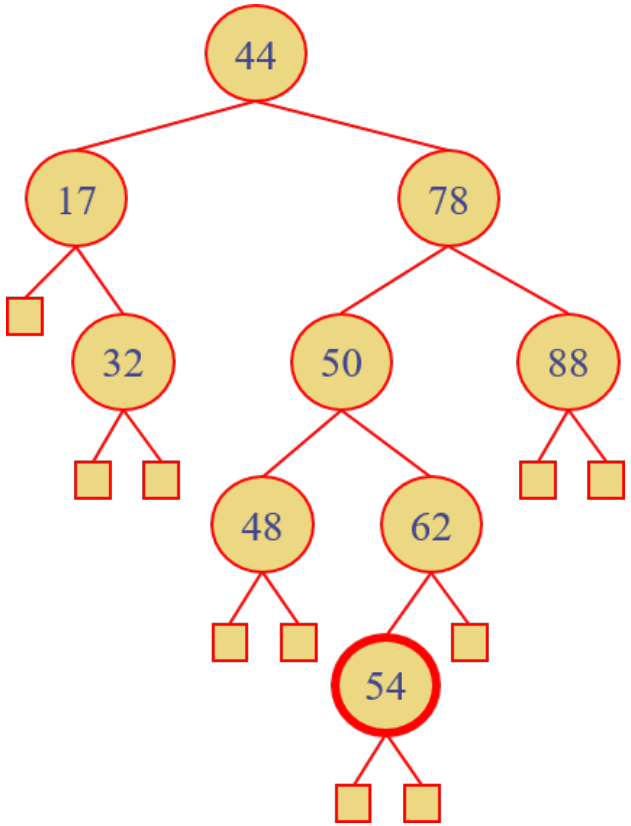
Proof (by induction). bound $n(h)$: the minimum number of internal nodes of an AVL tree of height h.

- We 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, $n(h) = 1 + n(h - 1) + n(h - 2)$
- Since $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$
- Solving the base case we get: $n(h) > 2^{h/2-1}$
- Taking logs: $h < 2 \lg n(h) + 2$, so $h = O(\log n)$

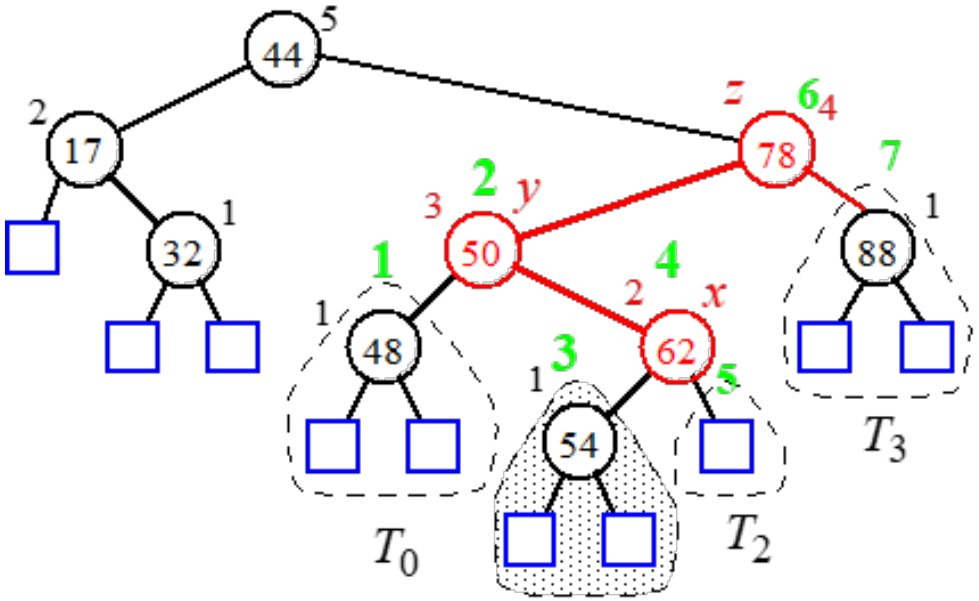
AVL Tree: Insertion

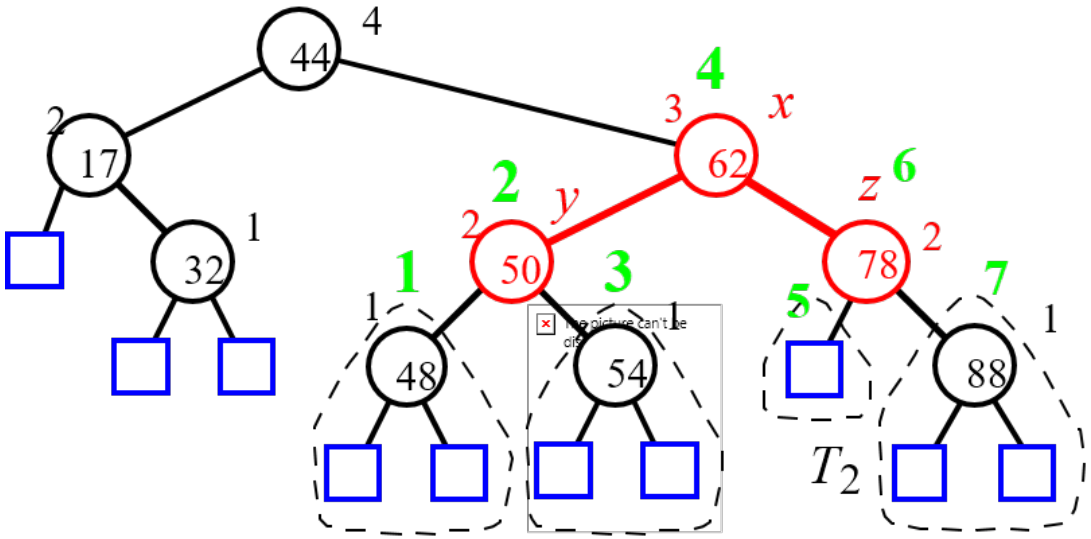
- Insertion is like that in any BST
- However, this may make the tree unbalanced





AVL Tree: Balancing After Insertion





Tri-node restructuring

- traverse toward the root until an imbalance is discovered
- Identify x, y, z such that
  - y : high sibling
  - x : high child of y
  - z : parent of y

AVL Tree: Balancing After Insertion - 2

Can we stop here? Yes, because

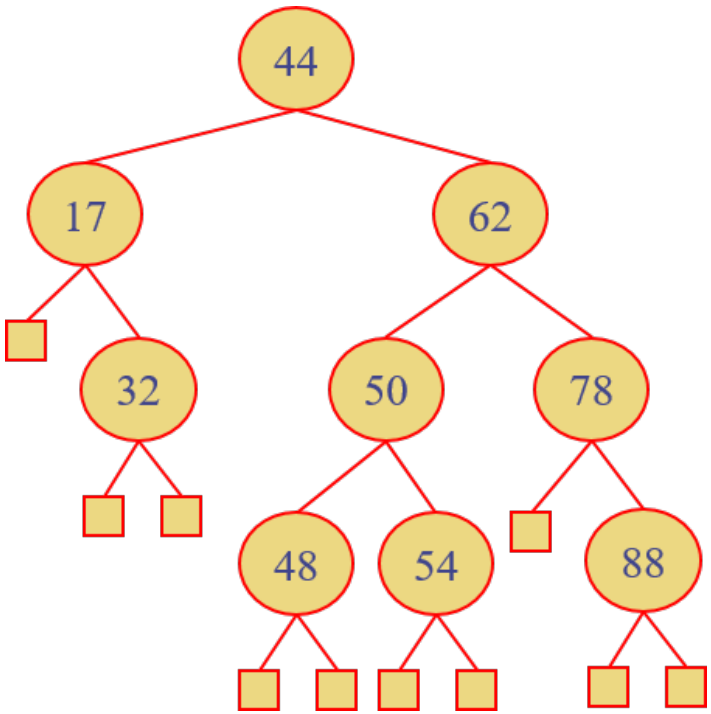
- The tree was balanced before the insertion
- Insertion raised the height of the subtree by 1
- Rebalancing lowered the height of the subtree by 1
- Thus the whole tree is still balanced

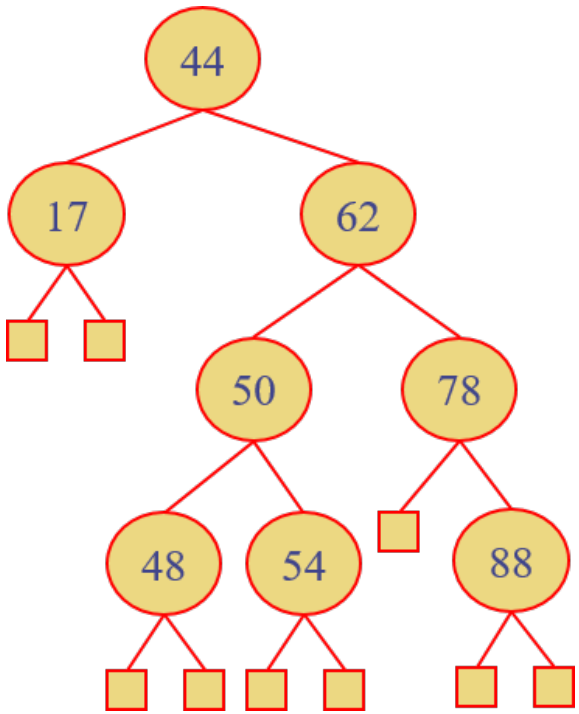
One restructuring is enough

AVL Tree: Deletion

delete(32)

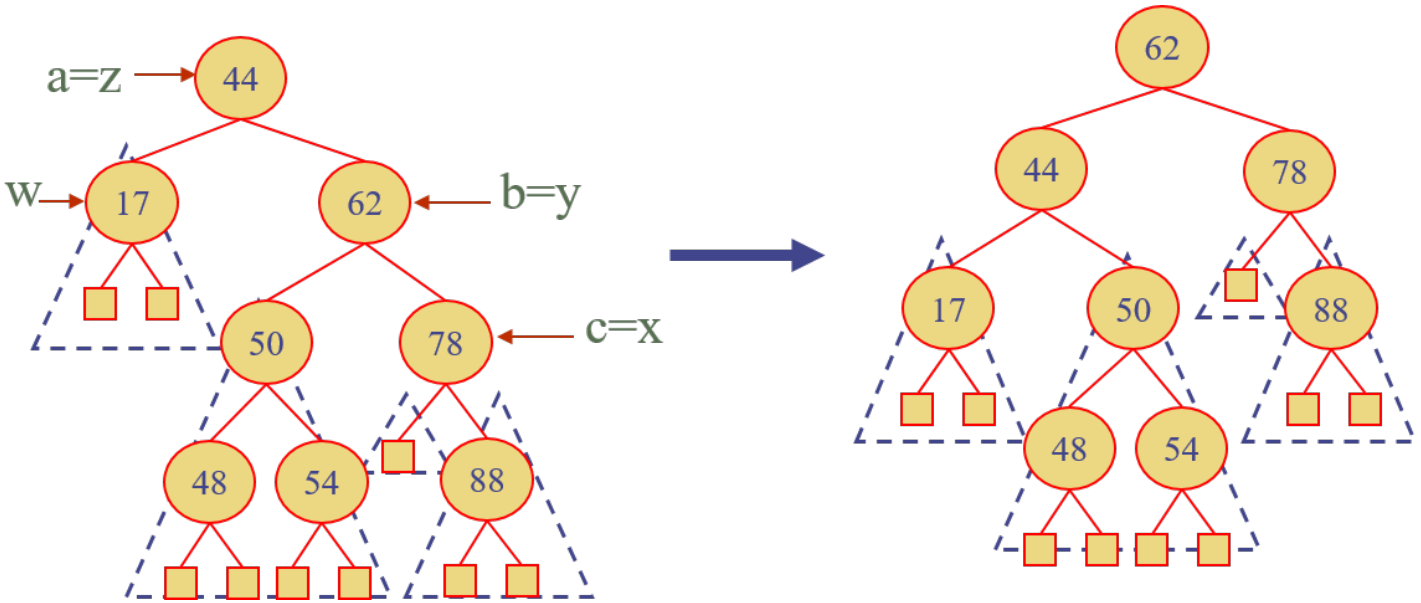
- Deletion is like that in any BST
- However, this may make the tree unbalanced





AVL Tree: Balancing After Deletion

- Let z be the first unbalanced ancestor of w, y be the child of z with the larger height, and let x be the child of y with the larger height
- perform trinode restructuring to restore balance at z
- check for balance all the way to the root



AVL Tree: Balancing After Deletion - 2

Can we stop after one restructuring? No, because

- trinode restructuring may reduce the height of the subtree, causing another imbalance further up the tree
- Thus this search and repair process must be repeated in the worst case until we reach the root
- $\Omega(\log n)$ balancing moves may be needed

AVL Tree: Performance

For a n item tree,

- The tree uses $\theta(n)$ space
- A single restructuring takes $O(1)$ time using a linked-structure binary tree
- Searching takes $\theta(\log n)$ time
- Insertion takes $\theta(\log n)$ time
  - initial find is $\theta(\log n)$
  - restructuring up the tree, maintaining heights is $\theta(\log n)$
- Removal takes $\theta(\log n)$ time
  - initial find is $\theta(\log n)$
  - restructuring up the tree, maintaining heights is $\theta(\log n)$

Splay Trees

- Self-balancing BST [D. Sleator and R. Tarjan]
- Allows quick access to recently accessed elements
- Bad: worst-case $\Omega(n)$ for one operation
- Good: guaranteed amortized $O(\log n)$ performance
- Often perform better than other BSTs in practice
- Used in the gcc compiler, GNU C++ library, the most popular implementation of Unix malloc, Linux loadable kernel modules

These slides are adapted from Prof Elder's slides

Splay Trees - 2

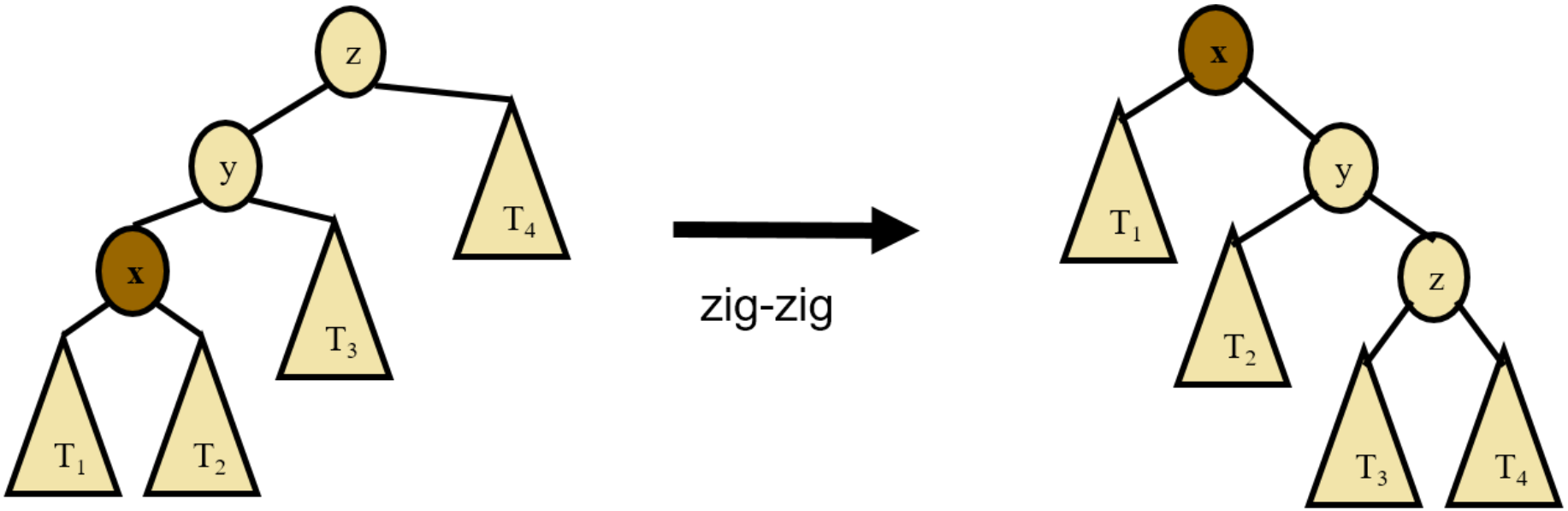
- Still BSTs – same insertion, deletion techniques
- each BST operation (find, insert, remove) is augmented with a splay operation
- Splaying is an operation performed on a node that iteratively moves the node to the root of the tree
- recently searched and inserted elements are near the top of the tree, for quick access

Splay Operation

- Each splay operation on a node consists of a sequence of splay steps
- Each splay step moves the node up toward the root by 1 or 2 levels
- There are 3 kinds of steps:
  - Zig-Zig
  - Zig-Zag
  - Zig
- These steps are iterated until the node is moved to the root.

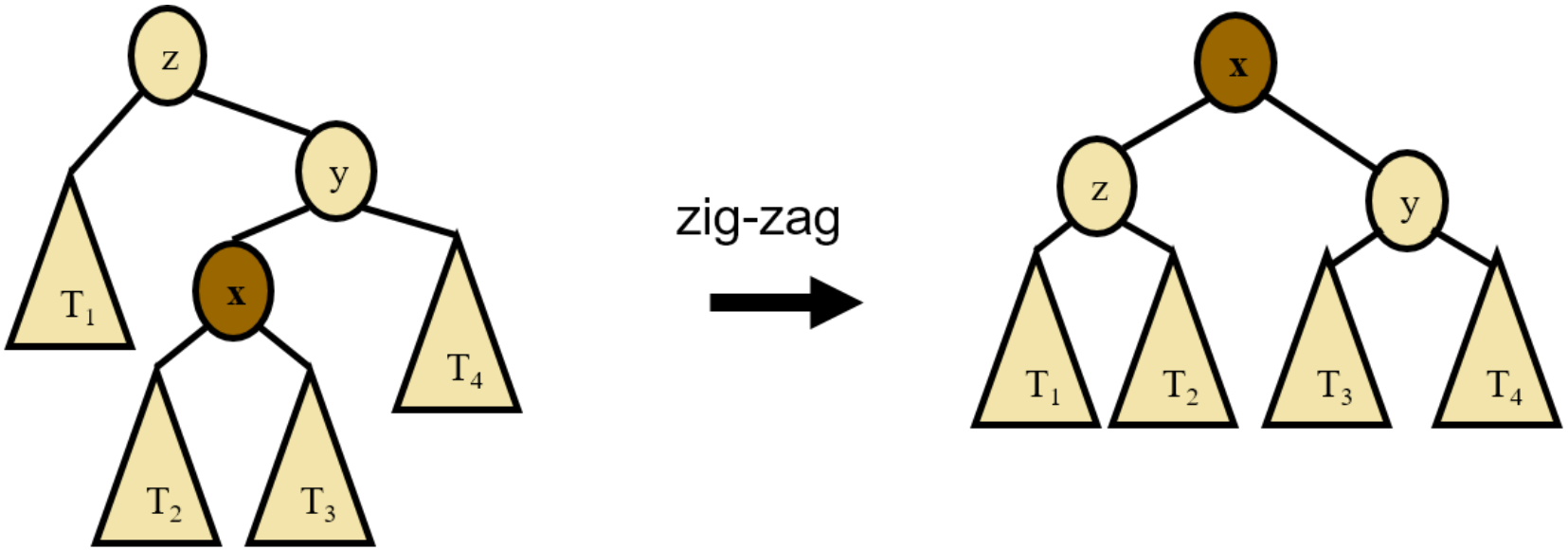
Zig-Zig

Performed when the node x forms a linear chain with its parent and grandparent, i.e., right-right or left-left



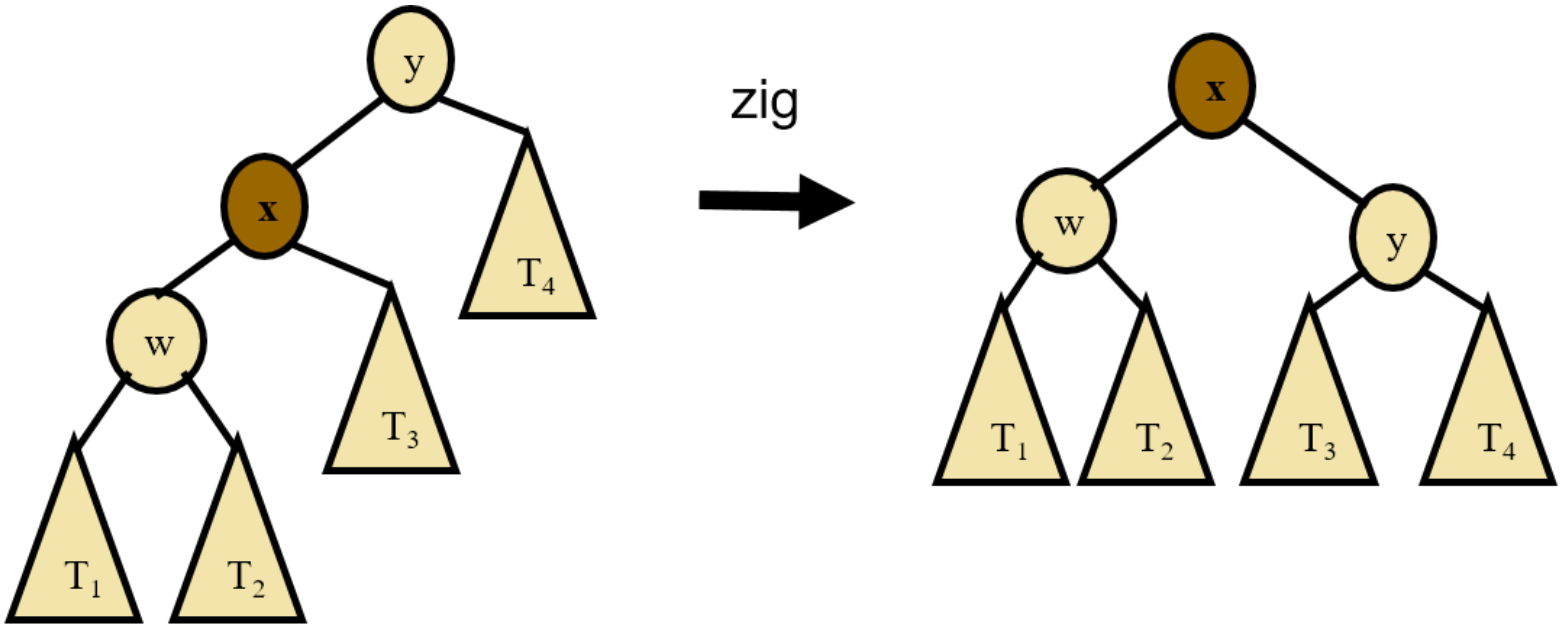
Zig-Zag

Performed when the node x forms a non-linear chain with its parent and grandparent, i.e., right-left or left-right



Zig

Performed when the node x has no grandparent, i.e., its parent is the root



Which Nodes are Splayed?

- find:
  - if key found, use that node
  - else, use parent of external node where search terminated
- insert: use the new node containing the entry inserted
- delete: use the parent of the internal node w that was actually removed from the tree. (If the node with key k had two internal children, this is the parent of the node it was swapped with.)

Splay Trees: Performance

- Worst-case $\Omega(n)$ operation, e.g.,
  - Find all elements in sorted order
  - This will make the tree a left linear chain of height n, with the smallest element at the bottom
  - Subsequent search for the smallest element will be $\Omega(n)$
- Average-case is $O(\log n)$: Advanced use of amortized analysis
- Operations on more frequently-accessed entries are faster. Given a sequence of m operations on an initially empty tree, the running time to access entry i is: $O(\log m/f(i))$, where $f(i)$ is the number of times entry i is accessed.

Other Height-Balanced Trees

- $(2, 4)$ Trees
  - These are multi-way search trees (not binary trees) in which internal nodes have between 2 and 4 children
  - Have the property that all external nodes have exactly the same depth
  - Worst-case $O(\log n)$ operations
  - Somewhat complicated to implement
- Red-Black Trees
  - Binary search trees
  - Worst-case $O(\log n)$ operations
  - Somewhat easier to implement
  - Requires only $O(1)$ structural changes per update