CSE 212: Data Structures and Algorithms

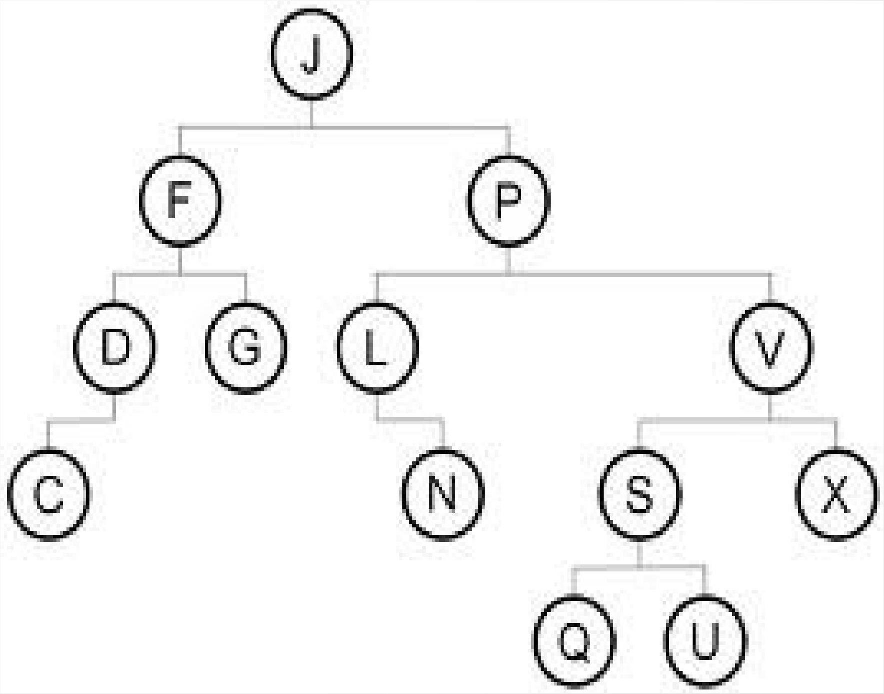
Lecture 8.3: Search Trees

Dr. Vidhya Balasubramanian

Amrita School of Engineering
Amrita Vishwa Vidyapeetham

Search Trees

- Trees can be used to organize data such that searching for elements is easy
- Different search trees
  - Binary Search Trees
  - AVL Trees
  - Multi-way search trees
  - (2,4) trees
  - Red-black trees

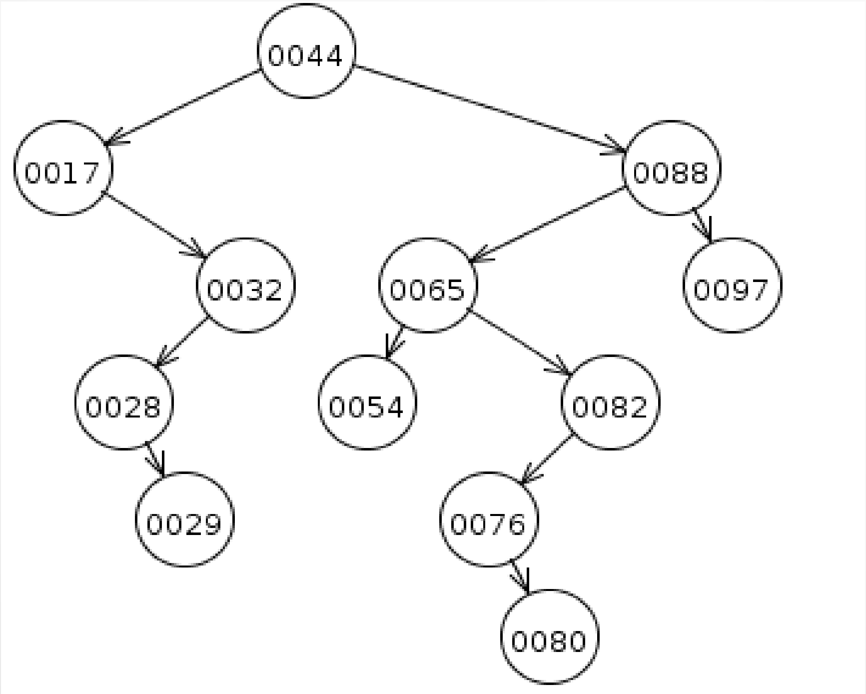


Binary Search Trees

- Is a binary tree storing keys (or key-element pairs) at its nodes and satisfying the following properties:
  - The left subtree of a node contains only nodes with keys less than the node's key
  - The right subtree of a node contains only nodes with keys greater than the node's key.
    - 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. key(u) ≤ key(v) ≤ key(w)
  - Both the left and right subtrees must also be binary search trees
  - Values are stored only in internal nodes (in the text book)
- Also called ordered or sorted binary tree

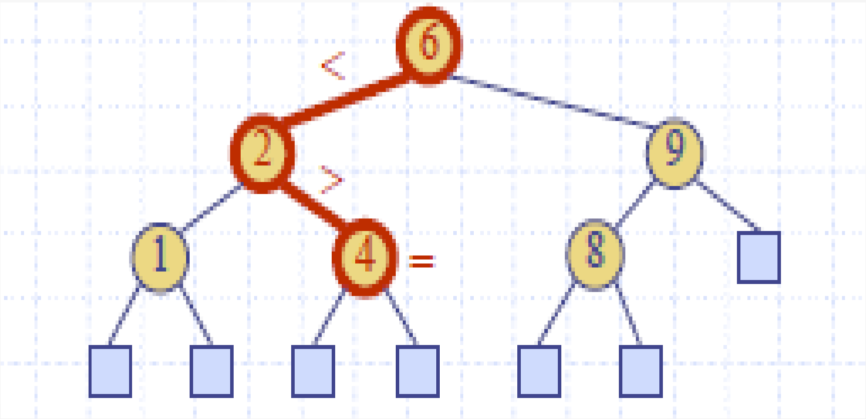
Binary Search Trees

- Binary trees are very efficient for sorting and searching
- Fundamental data structure used to construct more abstract data structures
  - e.g sets, multisets, and associative arrays



Searching

- Can be recursive or iterative
- Start by examining the root and traverse
- If the key is less than the root, search the left subtree else search the right subtree
- Repeat until the key is found or remaining subtree is null
- Complexity :O(h)



Src: Goodrich notes

Searching: Iterative Algorithm

- **Algorithm** *find*(*k*, *root*):

    curnode ← *root*

    while *curnode* is not null:

        if *curnode*.*key* == k:

            return *curnode*

        else if *k* < *curnode.key*:

            curnode ← *curnode.left*

        else

            curnode ← *curnode.right*

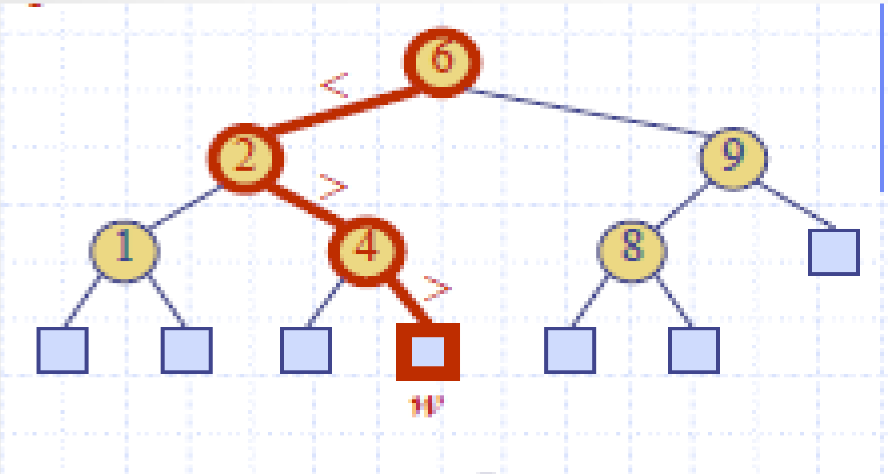
Searching: Recursive Algorithm

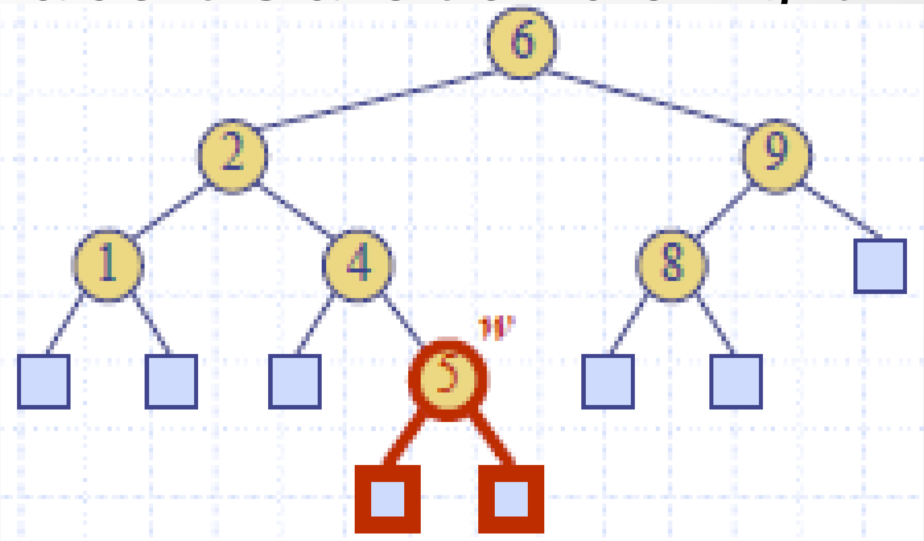
- **Algorithm** find-recursive(*k*, *node*): // call initially with node = root

```
if node.key == k:
    return node
else if k < node.key:
    find-recursive(k, node.left)
else
    find-recursive(k, node.right)
```

Insertion

- insertItem(k,n) inserts a node with key k, into the tree with root node n
- Assume k is not already in the tree, and let let w be the leaf reached by the search
- We insert k at node w or add it as a child of w
  - Depending on the relative value it is a left child or right child





Insertion

- **Procedure** InsertItem(k,n) :

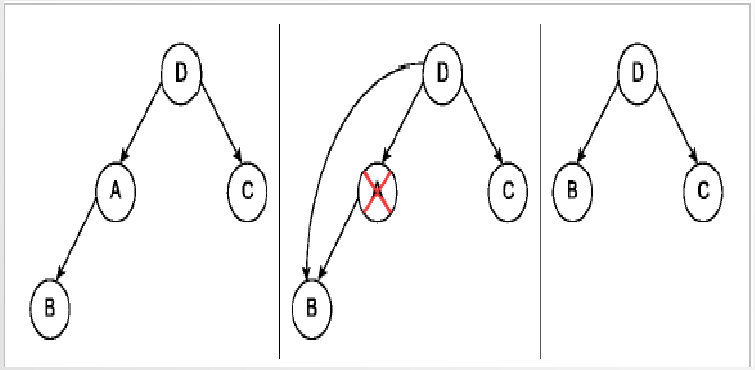
```
if (k < n.key):
    if (n.left == null):
        n.left = Node(k)
    else:
        InsertItem(k,n.left)
else if (k > n.key):
    if (n.right == null):
        n.right = Node(k)
    else:
        InsertItem(k,n.right)
```

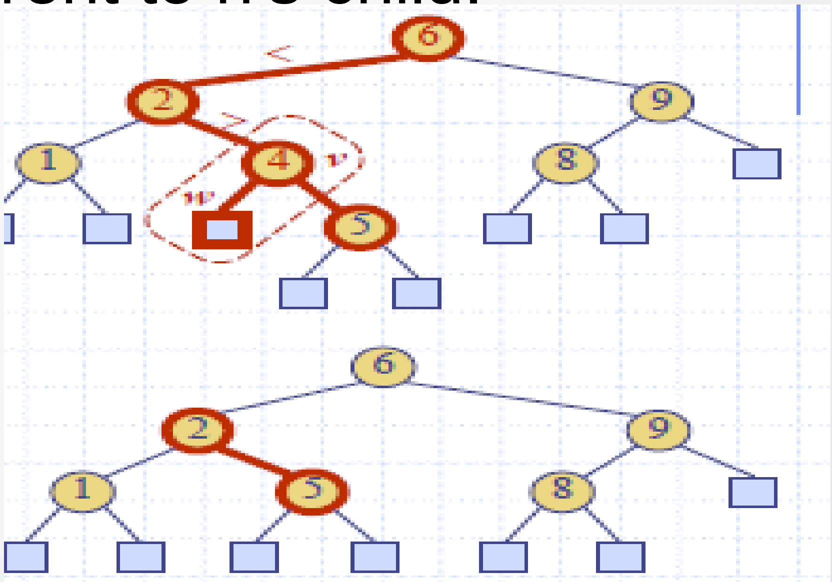
Deletion

- Three cases
  - Deleting a leaf or external node:
    - Just remove the node
  - Deleting a node with one child
    - Remove the node and replace it with its child
  - Deleting a node with two children
    - Instead of deleting the node replace with its
      - inorder successor node
      - Inorder predecessor node

Deleting node with one child

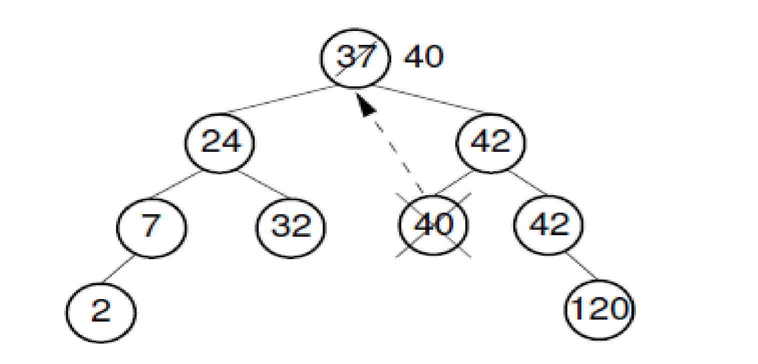
- removeElement(k):
  - First find the node n with key k using the search method
    - Remove using removeAboveExternal(n.child)
      - set the parent of n's child to n's parent
      - set the child of n's parent to n's child.



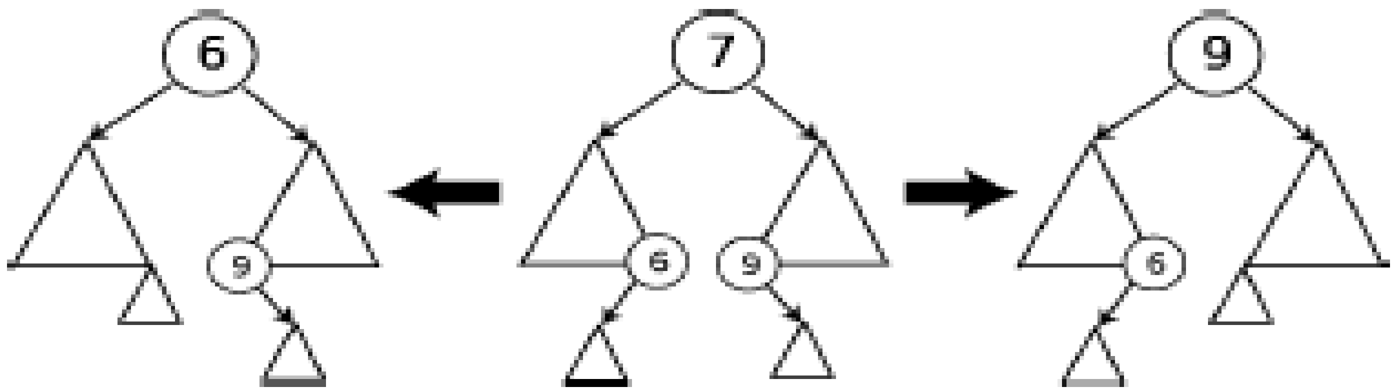


Deleting node with two children

- Naive Approach:
  - to set n's parent to point to one of R's subtrees, and then reinsert the remaining subtree's nodes one at a time
- Find the best values in one of the subtrees to replace n
  - The least key value greater than (or equal to) the one being removed or
  - the greatest key value less than the one being removed



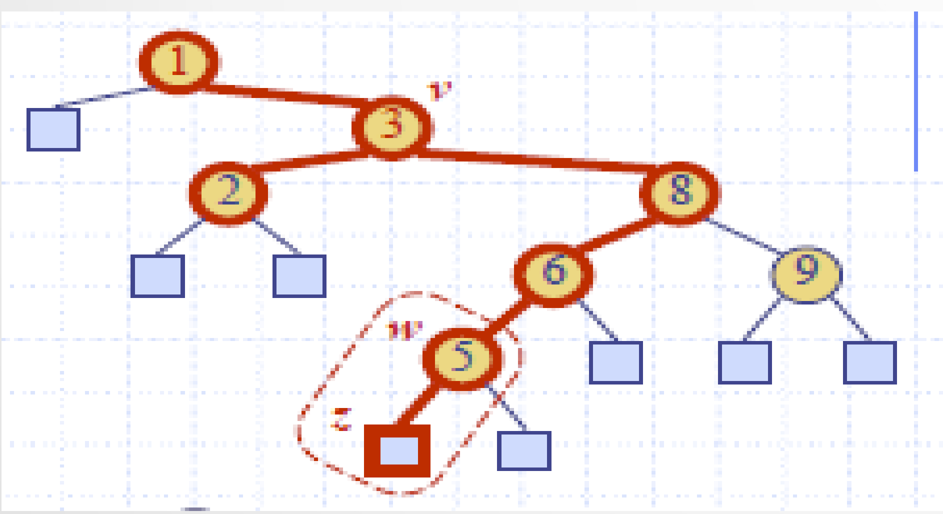
Deleting a node with two children

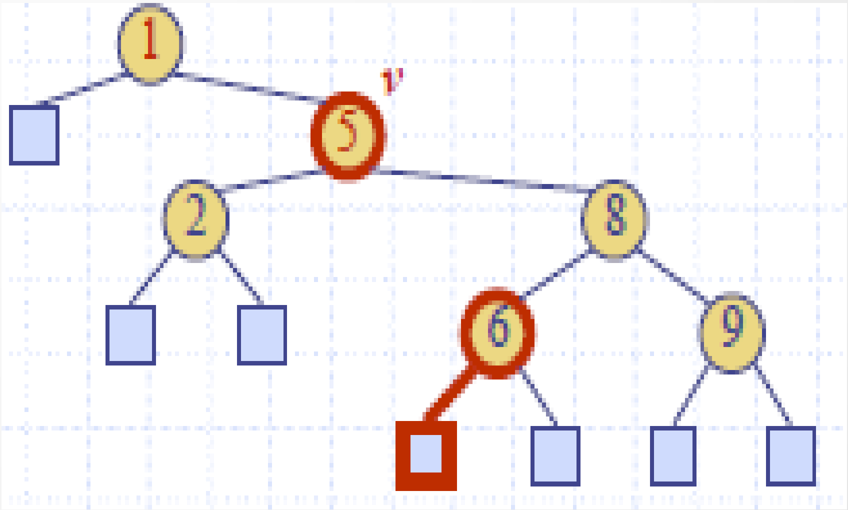


Deleting a node with two children from a binary search tree. The triangles represent subtrees of arbitrary size, each with its leftmost and rightmost child nodes at the bottom two vertices.

Deletion of node with two children

- find the node w that follows v in an inorder traversal
- copy key(w) into node v
- we remove node w and its left child z
  - Using the removeAboveExternal(z) method



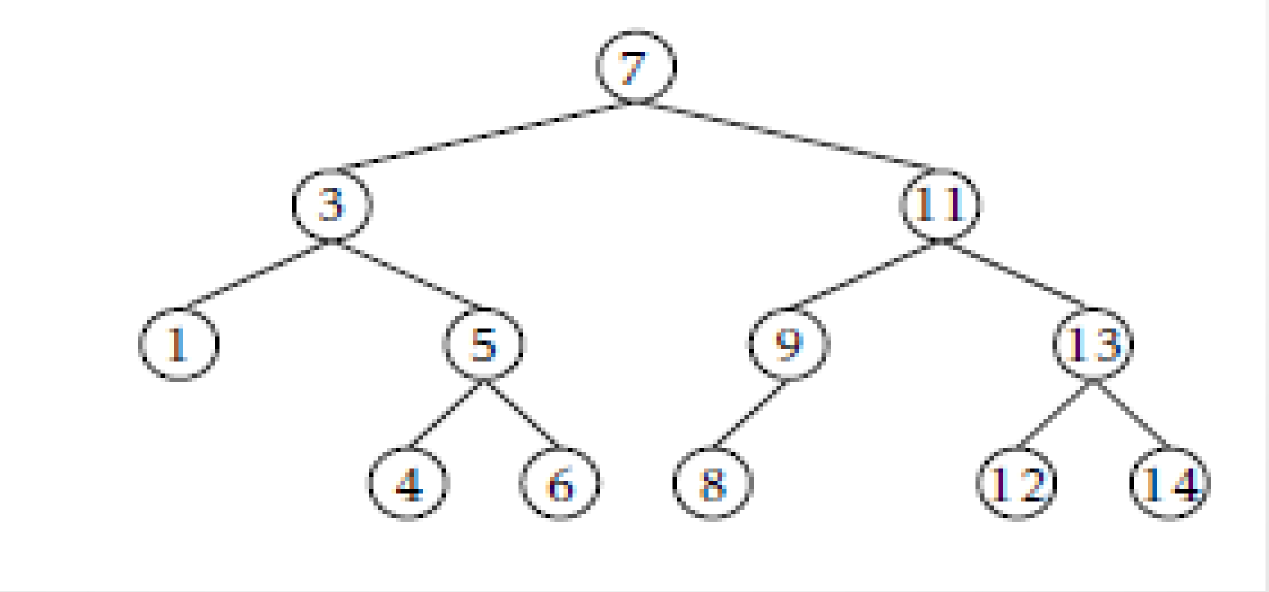


Exercise

- Insert into an initially empty binary search tree, items with the following keys (in the same order)
  - 30, 40, 24, 58, 48, 26, 11, 13
  - What happens if the values are entered in ascending order starting from 11
  - Try the reverse order: 13, 11, 26, 48, 58, 24, 40, 30

Exercise

- Consider the following binary search tree
  - Illustrate what happens when we add the values 3.5 and then 4.5 to this tree
  - Illustrate what happens when we remove the values 3 and then 5 from the tree in figure



Exercise

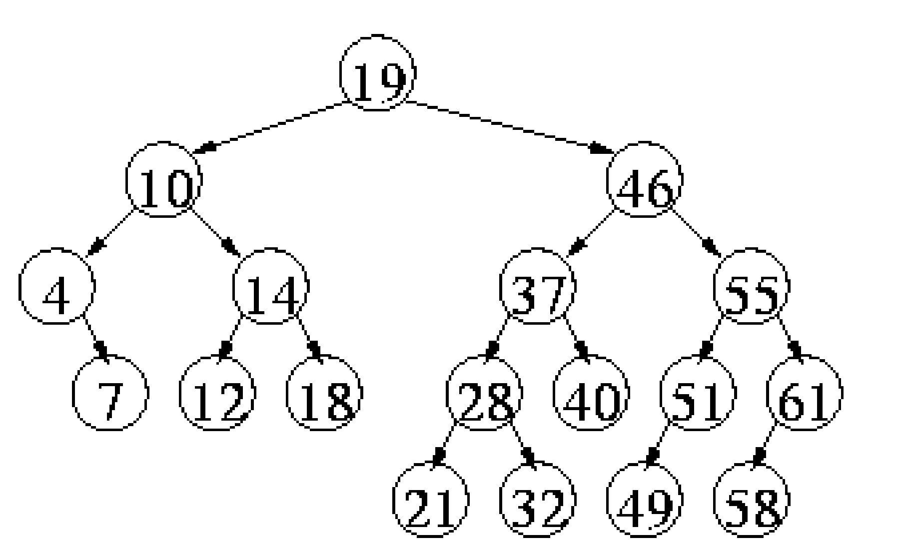
- If we have some BinarySearchTree and perform the operations add(x) followed by remove(x) (with the same value of x) do we necessarily return to the original tree?

Height Balanced Trees

- The height of a binary search tree depends on many factors
  - Order of insertion of values
  - Impact of deletions
- Worst case performance of search is linear
- Balance the height of the binary search trees so that the search cost is always O(log*n*)
  - Height Balance Property
    - For every internal node *v* of *T*, the height of the children of *v* differ by atmost 1

AVL Trees

- Is a binary search tree that satisfies the height-balance property
  - Self balancing search tree
- Named after its inventors
  - Adel'son- Vel'skii, and Landis



www.cs.uiuc.edu

AVL Trees

- Height balanced
  - Subtree of an AVL tree is also an AVL tree
- The height of an AVL tree storing *n* items is O(log *n*)
- Searching
  - As in an ordinary binary search tree
  - Cost : O(log n)

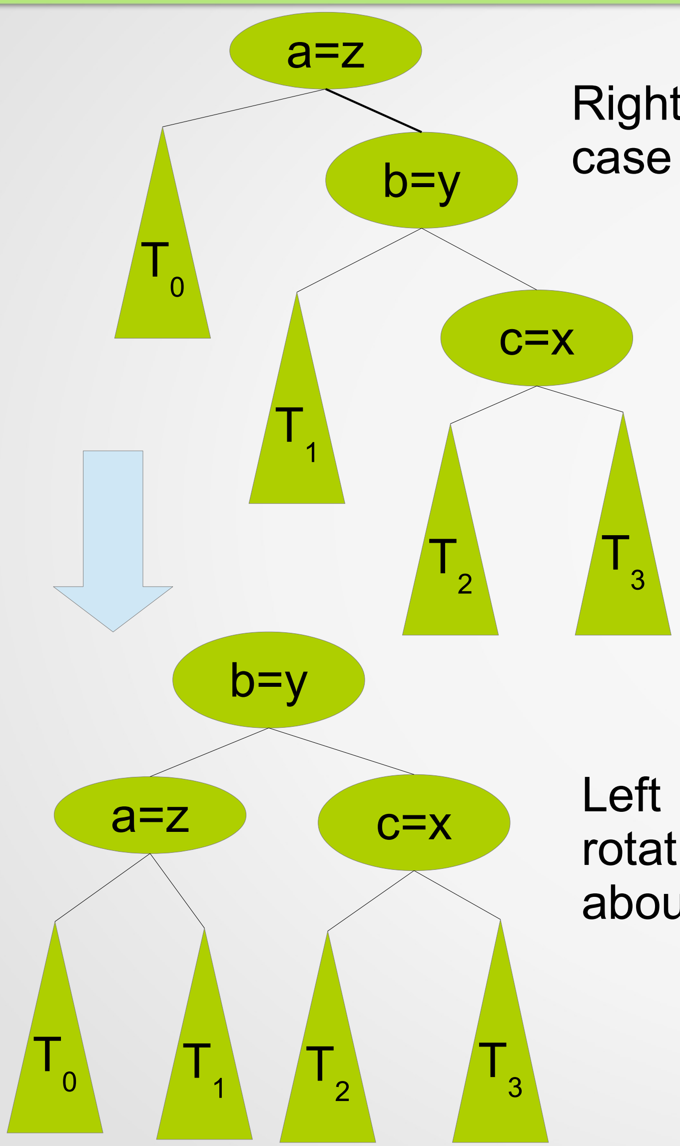
Insertion

- Node w is inserted as a leaf node as in binary search tree
- Check if height balance property still holds
  - Calculate balance factor
    - BalanceFactor = height(left-subtree) – height(right-subtree)
    - if the balance factor remains −1, 0, or +1 then no rotations are necessary, else need to rebalance
- Let z be first node going up from w towards root that is unbalanced
  - Let y be child of z with higher height, and is ancestor of w
  - Let x be child of y with higher height and is ancestor of w
  - Due to insertion height of y is higher than its sibling

Trinode Restructuring

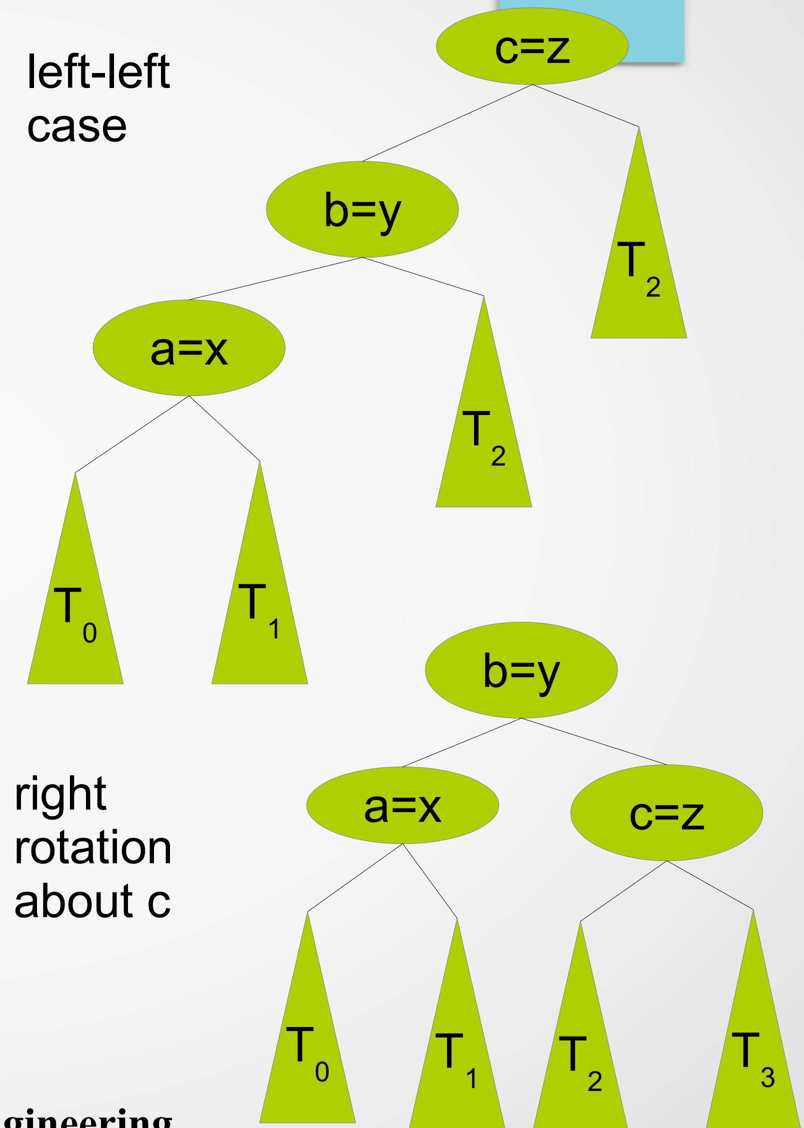
- Let the subtree that needs to be restructured be rooted at z.
  - Let (a,b,c) be an inorder listing of x, y, z
- Trinode restructure temporarily renames nodes x,y,z as a,b,c in the order of inorder listing
- Modification of T caused by trinode restructure is called rotation
- Goal is to make b the top node
  - If b=y, restructure method is called single rotation
  - If b=x, trinode restructure method is called double rotation

Single Rotations



Right-right case

Left rotation about a

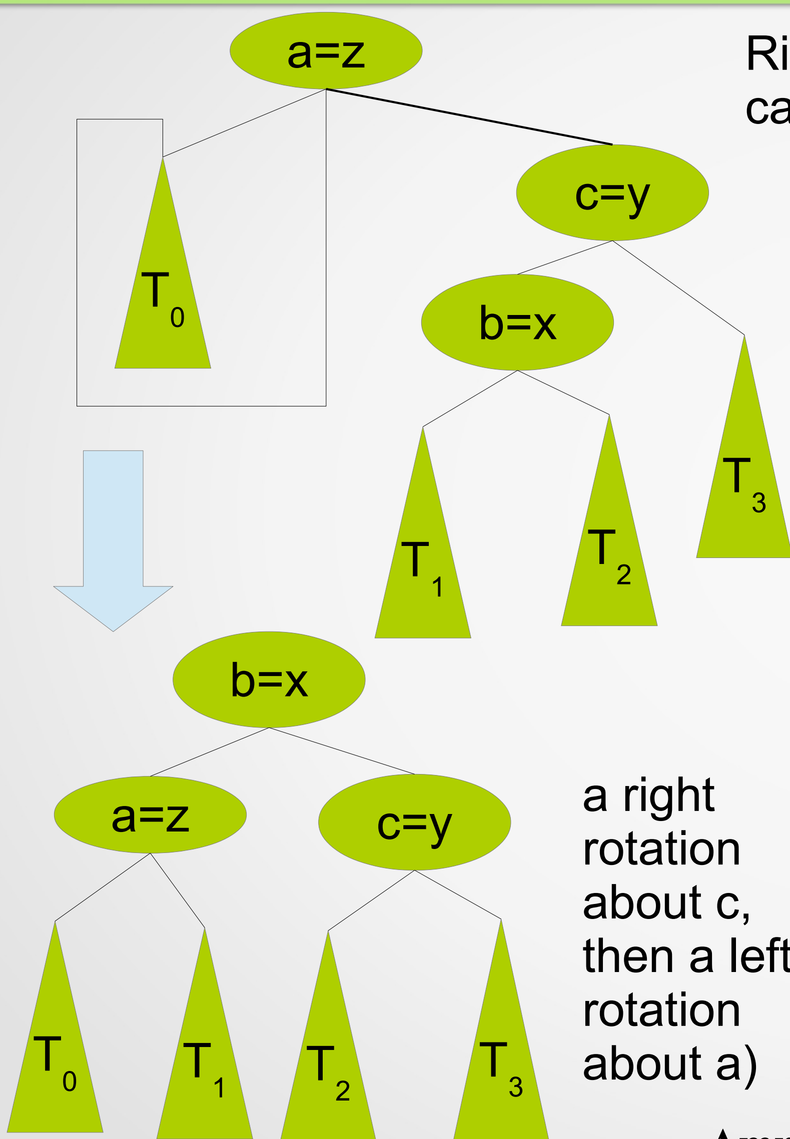


left-left case

right rotation about c

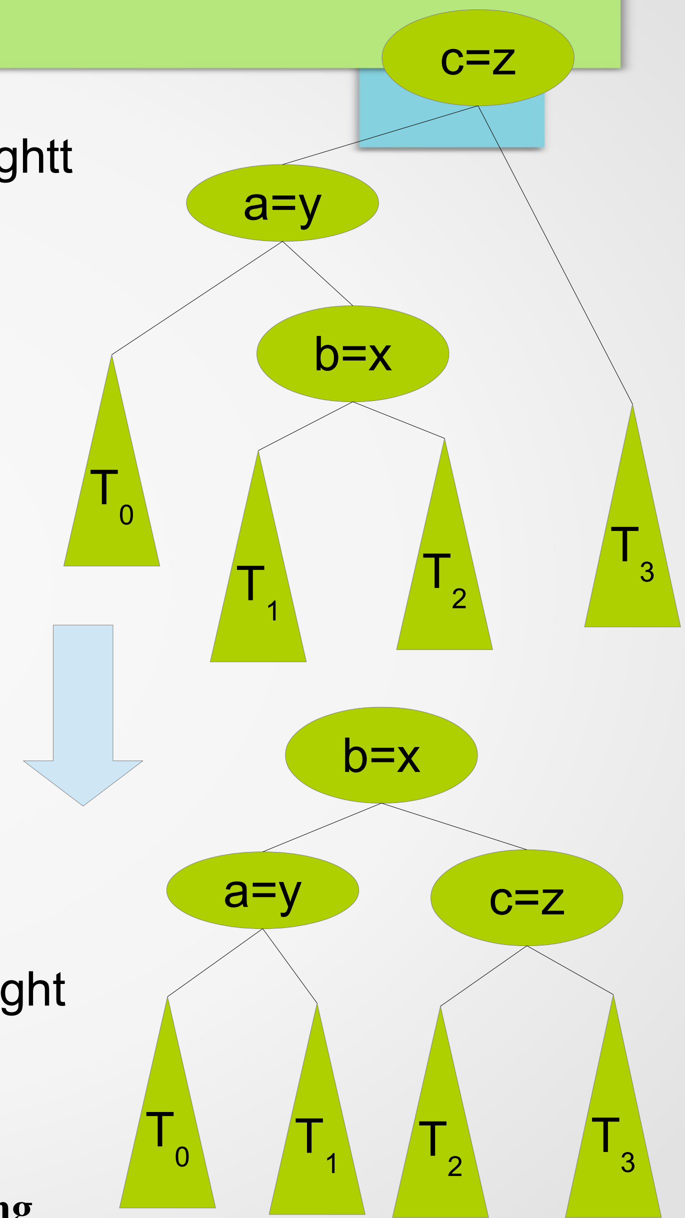
Double Rotations

Right-left case



a right rotation about c, then a left rotation about a)

left-rightt case



a left rotation about c, then a right rotation about a)

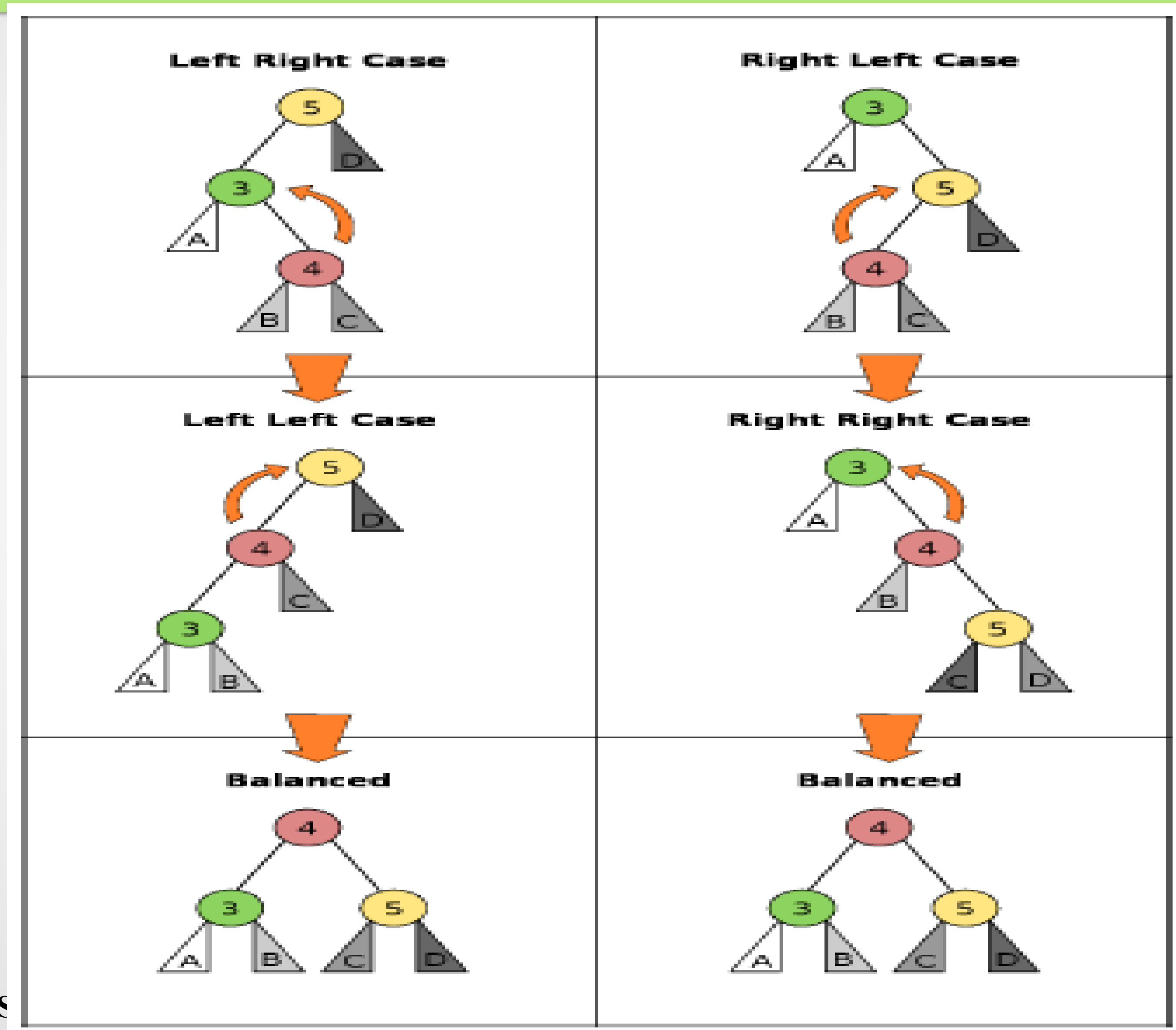
Pseudocode

- Algorithm restructure(x): //y is parent of x, and z is grandparent
- Let (a,b,c) be inorder listing of x,y,and z and let (T_0, T_1, T_2, T_3) be inorder listing of subtrees of x,y and z rooted at x,y, z
- Replace subtree rooted at z with new subtree rooted at b
- Let a be the left child of b, and let T_0 and T_1 be the left and right subtrees of a respectively
- Let c be the right child of b, and let T_2 and T_3 be the left and right subtrees of c respectively

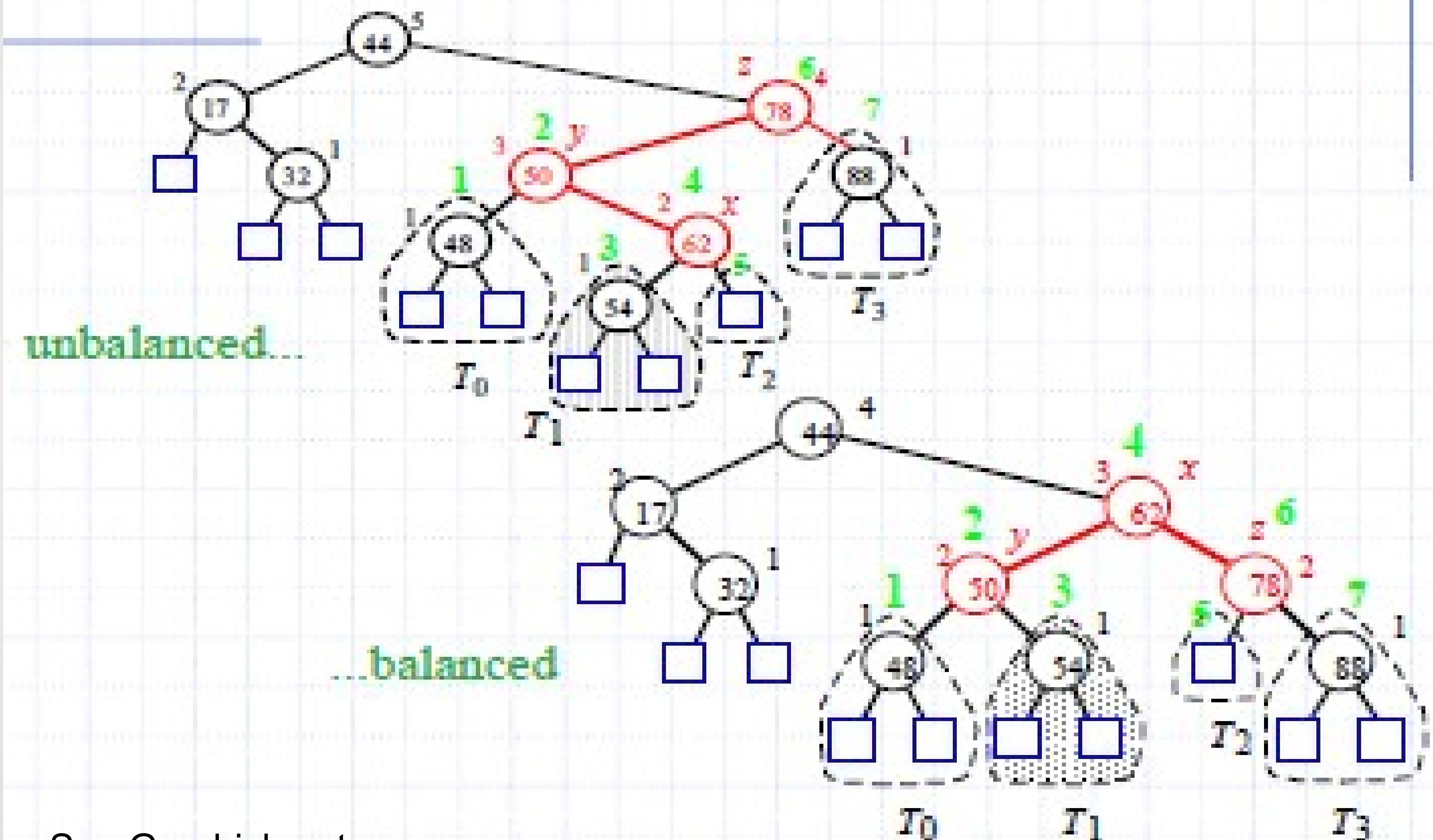
Rotations: Tree Restructuring

- If balance factor of P is -2 then the right subtree outweighs the left subtree of the given node, and the balance factor of the right child (R) must be checked.
  - The left rotation with P as the root is necessary
  - If the balance factor of R is -1
    - single left rotation with P as root is done (Right-right case)
  - If balance factor of R is 1 (right-left case)
    - first rotation is a right rotation with R as the root
    - second is a left rotation with P as the root
- Vice versa for left-left and left right case

AVL Rotations: Another Look



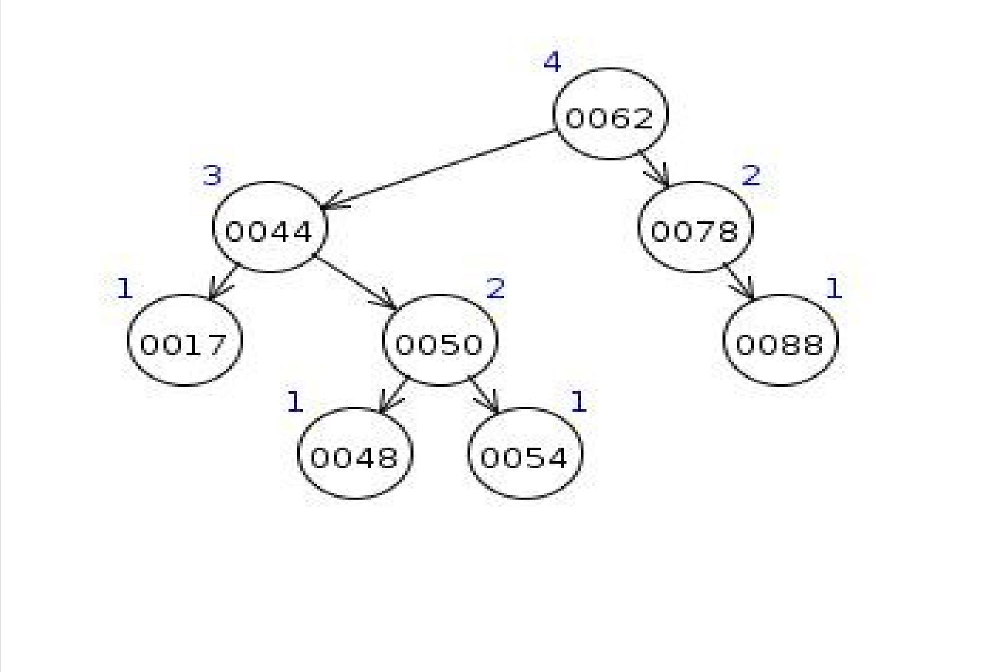
AVL Insertion: Example

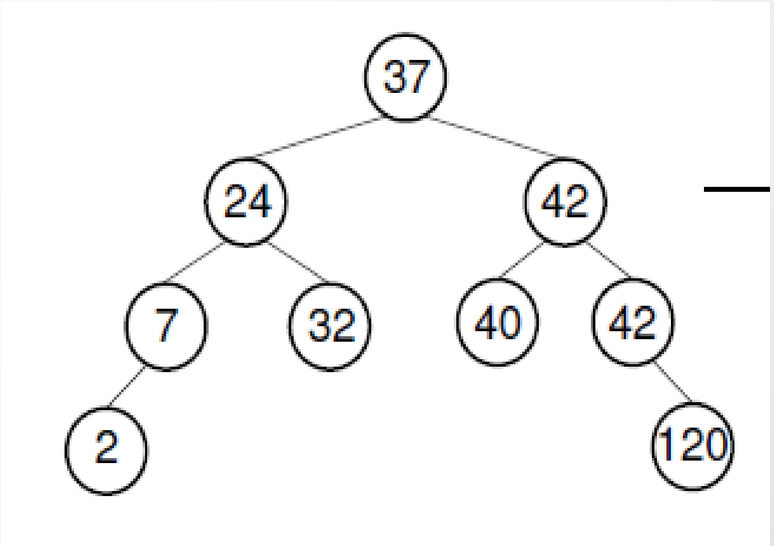


Src: Goodrich notes

Exercise

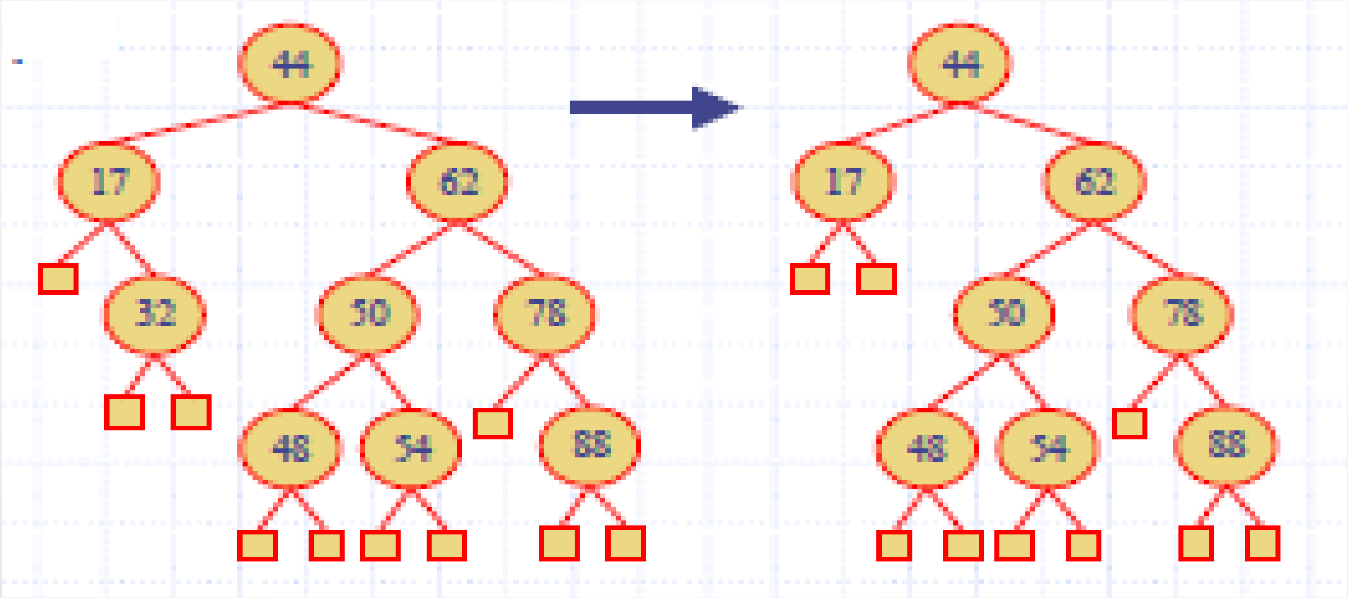
- Draw the AVL tree resulting from the insertion of an item with key 52, 95, 65 in the tree in the left given below
- Show the result (including appropriate rotations) of inserting the following values into the tree on the right
  - 39, 300,50,1





Deletion

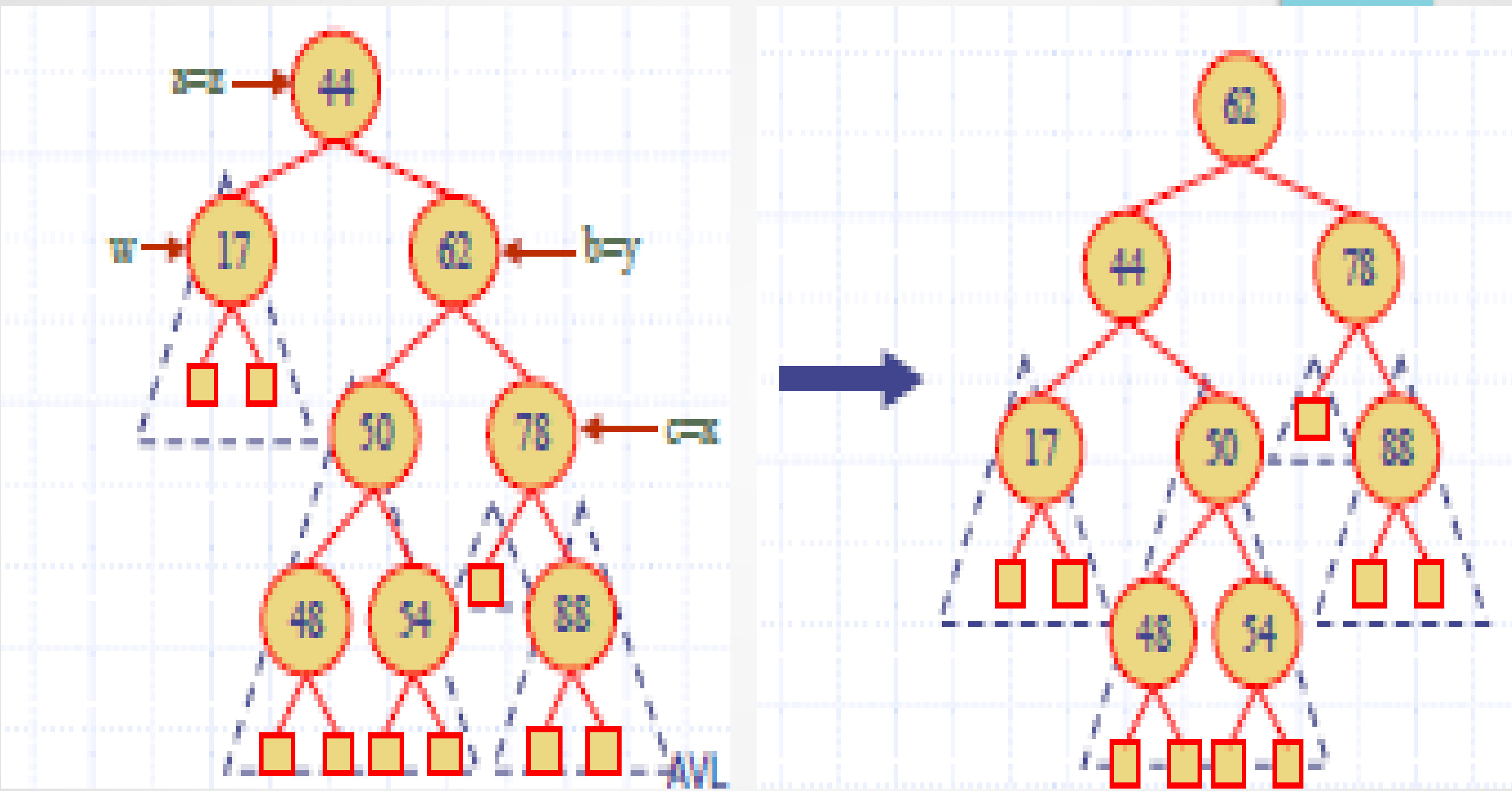
- Removal is done initially as in a binary search tree
- The node removed or replaced usually ends up in an empty external node
  - Parent may become unbalanced



Rebalancing

- Let z be the first unbalanced node encountered while travelling up the tree from w.
  - 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.
- This restructuring may upset the balance of another node higher in the tree, continue checking for balance until the root of T is reached

Restructuring after deletion: Example



Src: Goodrich notes

Analysis

- Single restructure : O(1)
  - using a linked-structure binary tree
- Finding an element: O(log n)
  - height of tree is O(log n)
- Insertion: O(log n)
  - initial find: O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- Removal: O(log n)
  - initial find: O(log n)
  - restructuring up the tree, maintaining heights is O(log n)

Exercise

- Consider the following sequence of keys
  - 5,16,22,45,2,10,18,30,50,12, 1
  - Create an AVL tree by inserting one element at a time in order
  - What happens when you delete 16, 30 from the tree