Data Structures

Search Trees

Shin Hong

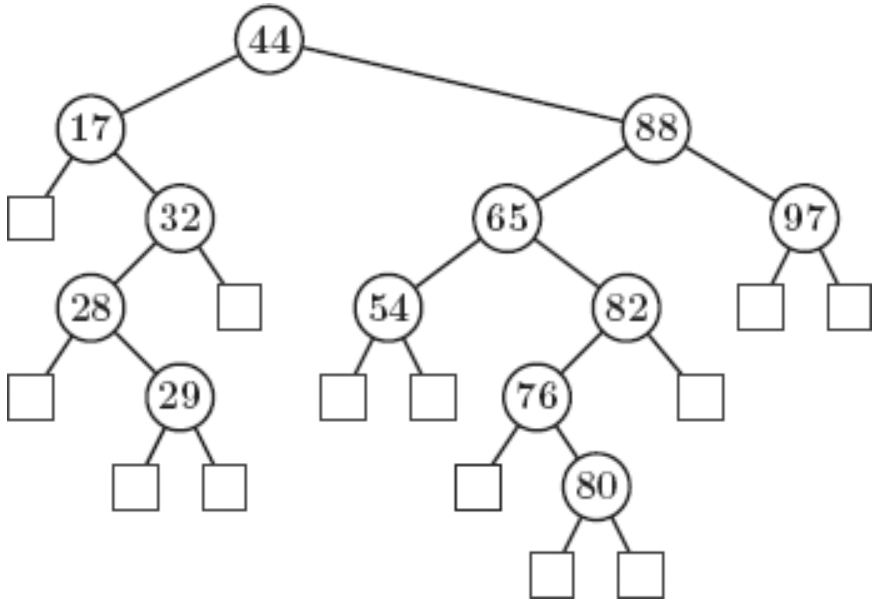
30 May 2023

DS&A. Chapter 10.1 Binary Search Trees
DS&A. Chapter 10.2 AVL Trees
DS&A. Chapter 10.5 Red-black Trees

Binary Search Trees

- a search tree can be used to implement an ordered map

- a binary search tree is a binary tree *T* where each internal node *v* stores an entry (*k*, *x*) such that
  - keys stored at nodes in the left subtree are less than equal to *k*
  - keys stored at nodes in the right subtree are greater than equal to *k*



Binary Search Tree with Proper Binary Tree

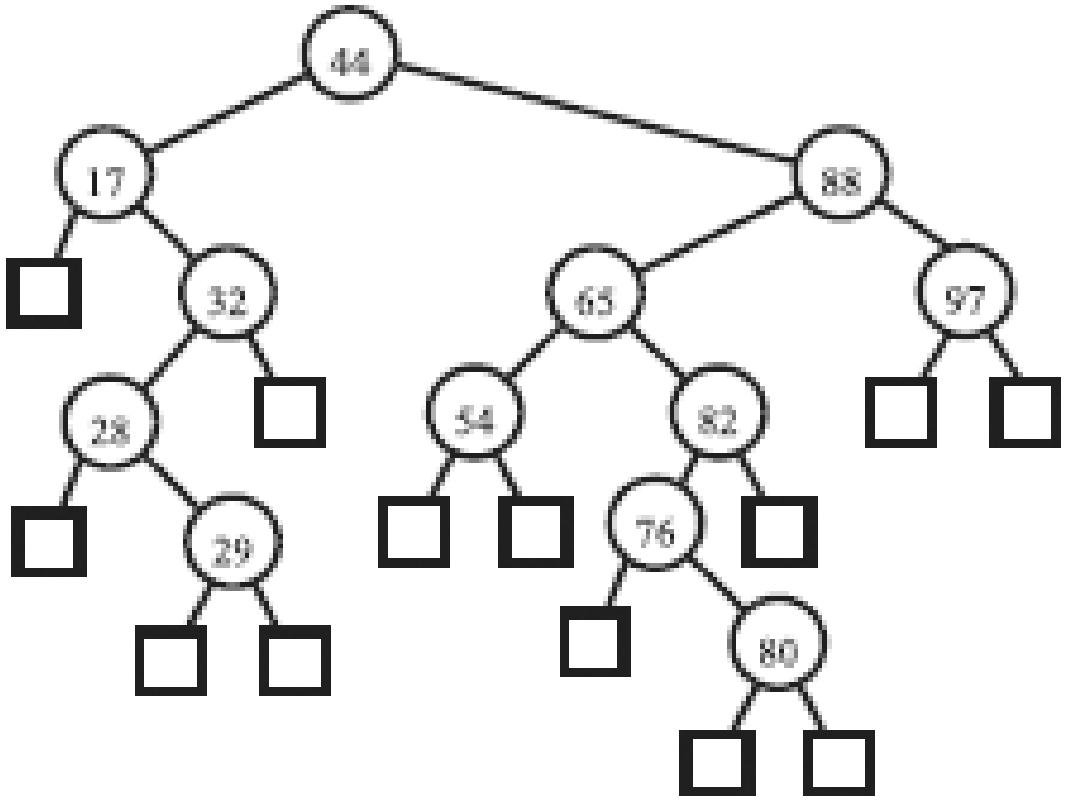
- A binary search tree can be represented as a proper binary tree
  - a tree has the same number of the internal nodes as entries such that each internal node represents an entry
  - an external node is used a placeholder

- Operations
  - find(k)
  - erase(k)
  - insert(k, x)

Searching

- Starting from the root node, compare the search key *k* with the key stored at an internal node *v*, *key*(*v*)
  - if *k* is equal to *key*(*v*), terminate the search successfully
  - if *k* is less than *key*(*v*), recursively search on the left subtree
  - if *k* is greater than *key*(*v*), recursively search on the right subtree
- If an external node is reached, terminate the search unsuccessfully
- The worst-case running time is O(*h*) where *h* is the height of a binary search tree

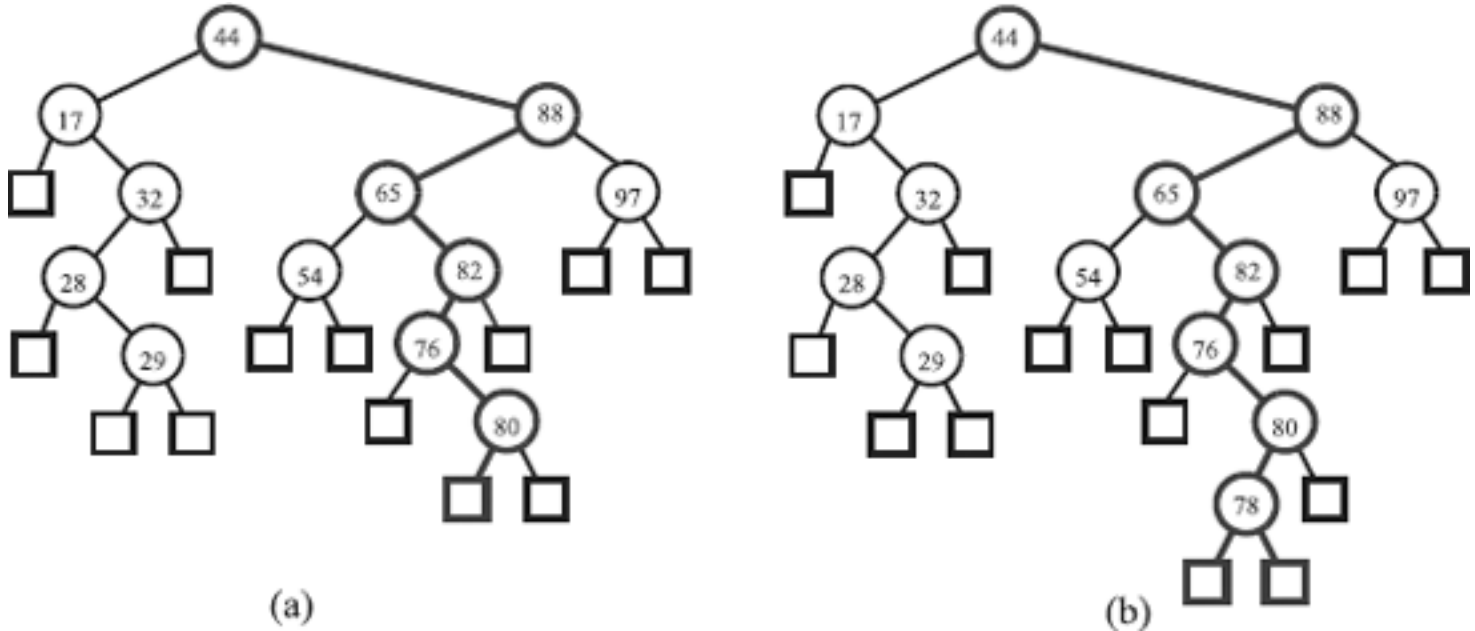
Example



- find(76)
- find(25)

Insertion

- First, perform a searching operation to find the external node *w* at the location of a newly given key *k*
- Extend w to a proper internal node by creating two children
- e.g., insert an entry with key 78



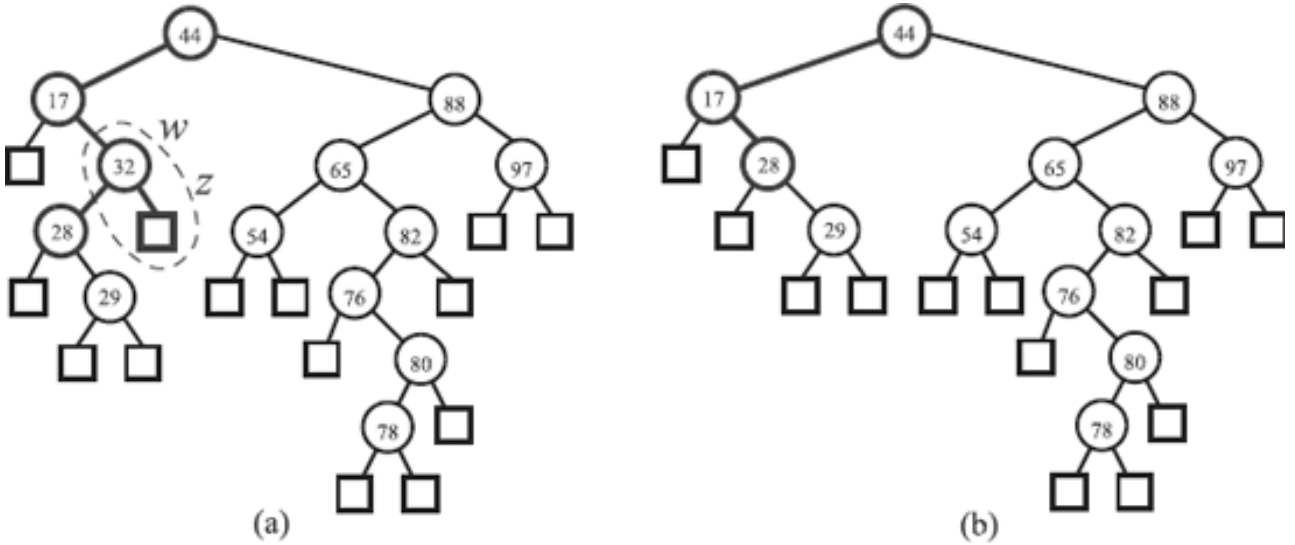
Removal

- First, find an internal node *w* that has an entry with key *k*

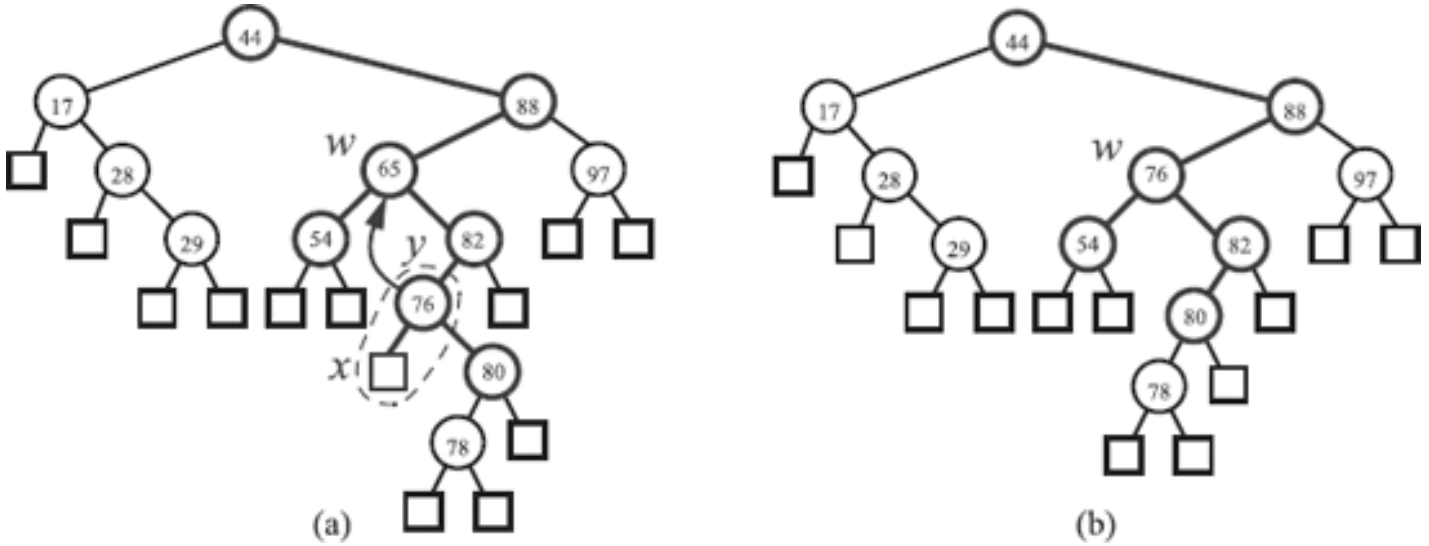
- If both children of *w* are external, remove *w*
- If only one child of *w* is an external node, replace *w* with the remaining children, and then remove *w*
- If both children of *w* are internal:
  - find the first internal node *y* that comes after *w*
    - *y* is the left-mode internal node in the right subtree of w
  - move the entry of *y* to w
  - remove *y*

Examples

Remove 32

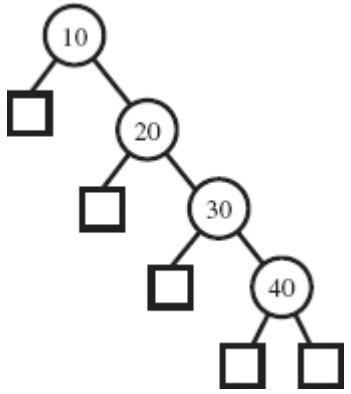


Remove 65



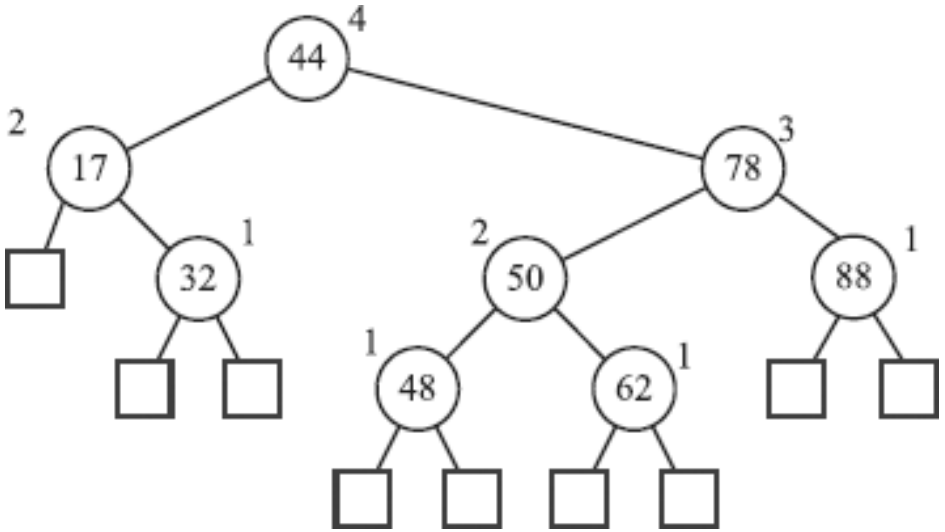
Runtime Complexity

- A binary search tree *T* with height *h* for *n* key-value entries uses O(n) space and executes the operations with the following running times:
  - Operations size and empty each take O(1) time.
  - Operations find, insert, and erase each take O(*h*) time.



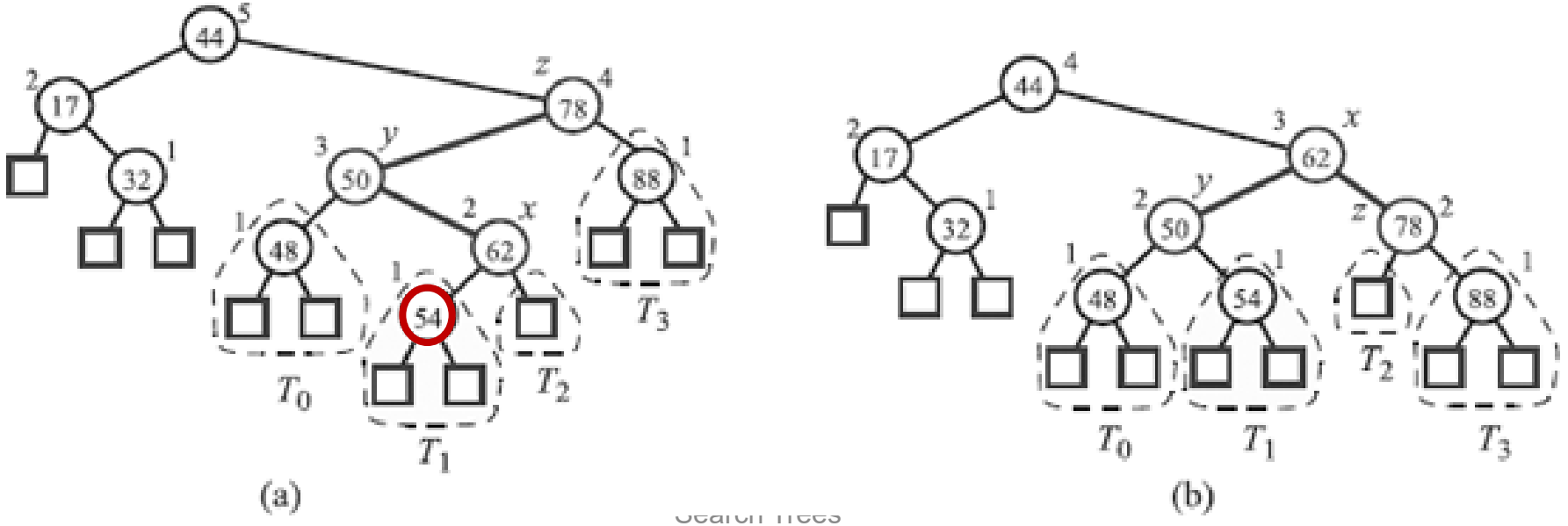
AVL Tree

- Use a balanced binary search tree to provide a O(log n) search
- AVL tree is a binary tree with the height-balance property
- **Height-balance property**: for every internal node *v* of *T*, the height of the children of *v* differ by at most 1
  - every subtree of a AVL tree is a AVL tree



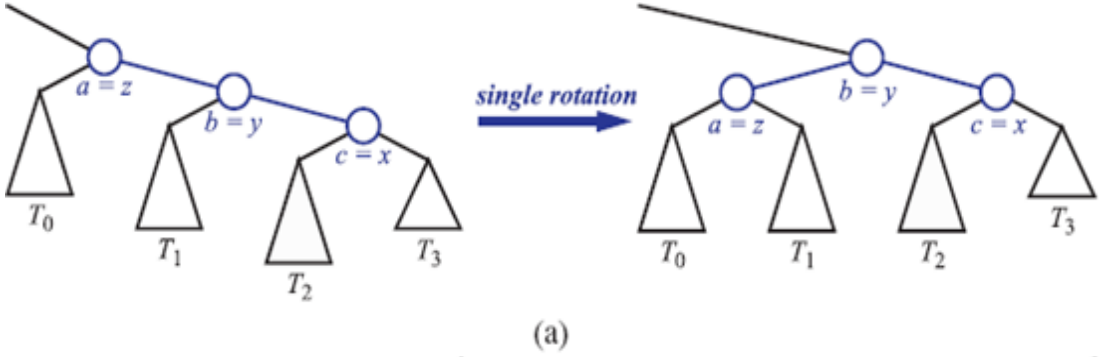
Insertion

- Two steps: search-and-repair
  - add a new node like does for a binary search tree
  - Identify the minimal subtree that violates the height balance property if exists, and restructure the subtree to restore its height balance

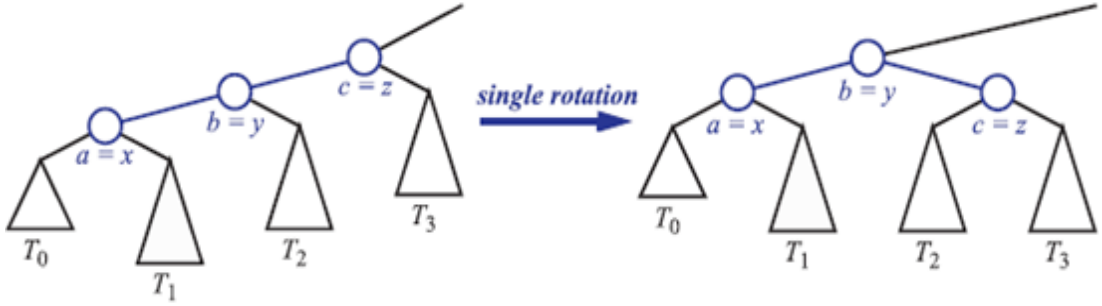


Four Cases

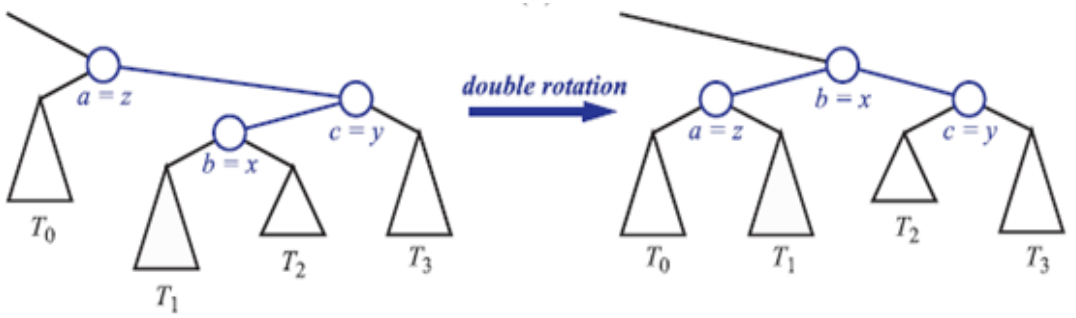
- *w*: the node where a new node is added
- *z*: the root of the minimal unbalanced subtree
- *y*: the child of *z* with a higher height (must be an ancestor of *w*)
- *x*: the child of *y* with a higher height (can be the same as *w*)



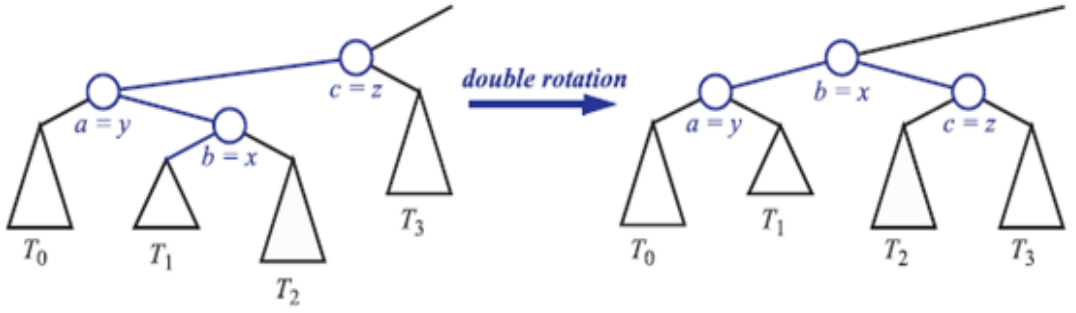
(a)



(b)

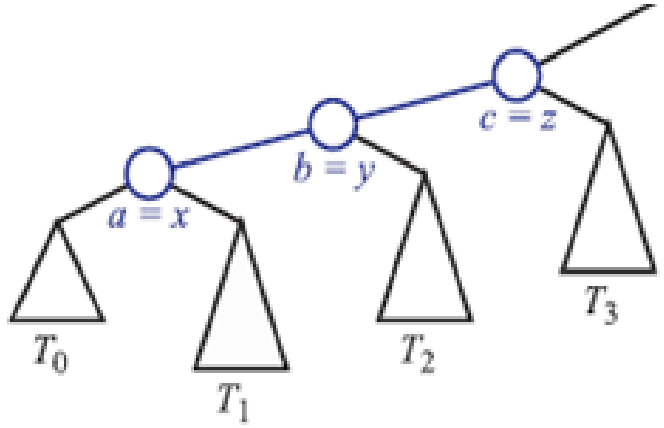


(c)



(d)

Trinode Restructuring



- Given nodes *x*, *y* and *z*, let *a*, *b*, *c* be a left-to-right listing of *x*, *y*, and *z,* and T_0, T_1, T_2, and T_3 be a left-to-right listing of four subtrees of x, y, and *z*.
- Replace *z* with a new subtree having *b* as the root, *a* as the left-child, and *c* as the right-child nodes
- Let T_0 and T_1 be the two subtrees of *a*
- Let T_2 and T_3 be the two subtrees of c

Removal

- Two steps: search-and-repair
  - remove the target node like does for a binary search tree
  - Identify the minimal subtree that violates the height balance property if exists, and restructure the subtree to restore its height balance

- Restructuring
  - Let *z* be the minimal unbalanced subtree, and let *y* be the child of *z* with a higher height
  - Let x be a child of *y* with a higher height. Or, if both children have the same height, *x* is child of y on the same side as *y*
  - Perform the trinode restructuring recursively, since the restructuring may reduce the height of the subtree rooted by *z*

Example

- Remove 32