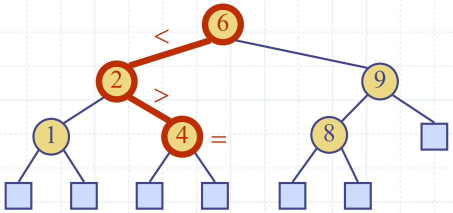
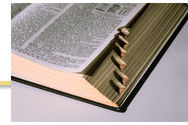
Binary Search Trees



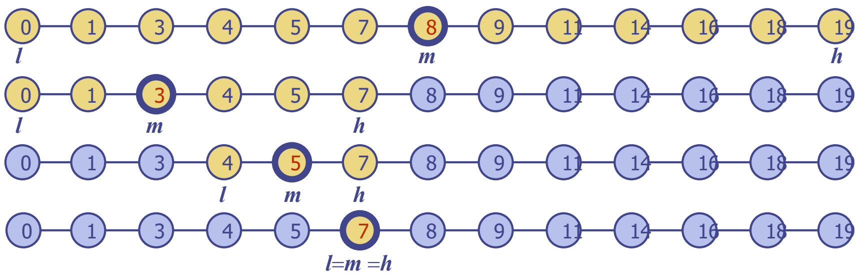
Recall: Ordered Maps



- Keys come from a total order
- New operations:
  - Each returns an iterator to an entry:
  - firstEntry(): smallest key in the map
  - lastEntry(): largest key in the map
  - floorEntry(k): largest key ≤ k
  - ceilingEntry(k): smallest key ≥ k
  - All return end if the map is empty

Binary Search

- Binary search can perform operations get, floorEntry and ceilingEntry on an ordered map implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)

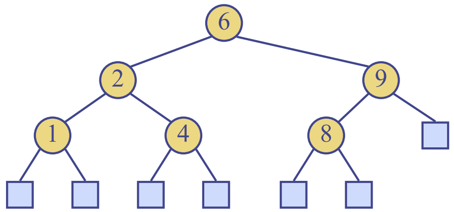


Search Tables

- A search table is an ordered map implemented by means of a sorted sequence
  - We store the items in an array-based sequence, sorted by key
  - We use an external comparator for the keys (for any arbitrary comparison)
- Performance:
  - get, floorEntry and ceilingEntry take $O(\log n)$ time, using binary search
  - get takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - erase take $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal

Binary Search Trees

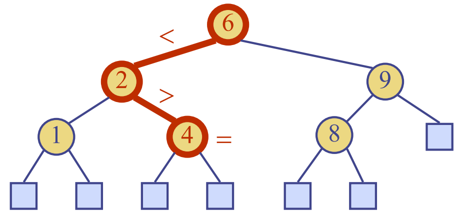
- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \le key(v) \le key(w)$
- External nodes do not store items
- An inorder traversal of a binary search trees visits the keys in increasing order



Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for floorEntry and ceilingEntry are similar
- Recursive

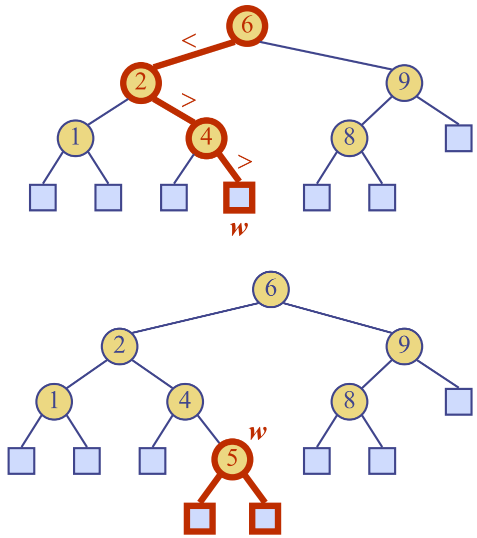
```
Algorithm TreeSearch(k, v)
    if v.isExternal ()
        return v
    if k < v.key()
        return TreeSearch(k, v.left())
    else if k = v.key()
        return v
    else { k > v.key() }
        return TreeSearch(k, v.right())
```



Insertion

- To perform operation put(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node

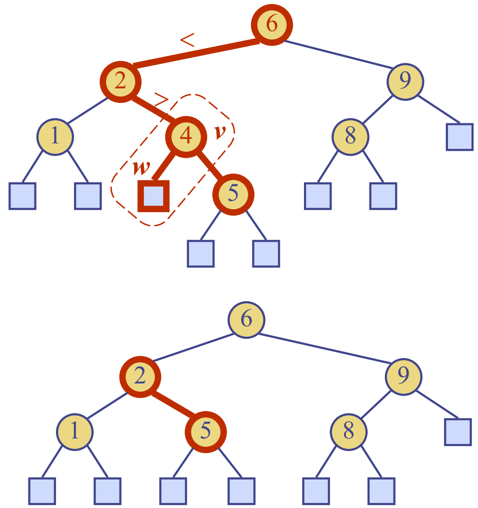
Example: insert(5)



Deletion

- To perform operation erase(k), we search for key k
- Assume key k is in the tree, and let v be the node storing k
- Basic method
  - removeAboveExternal(w): removes w and its parent
- If node v has a leaf child w, we remove v and w from the tree with removeAboveExternal(w)
- What about "remove 1"?

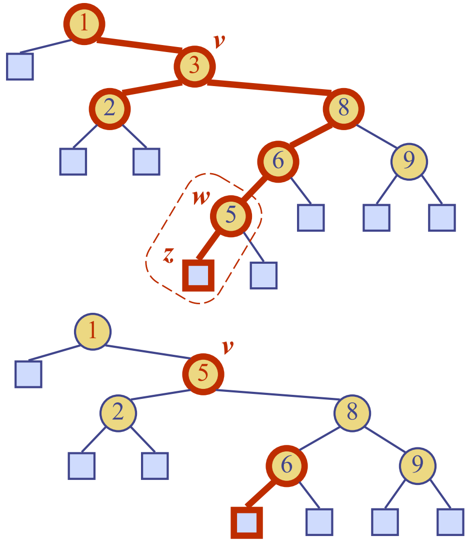
Example: remove 4



Deletion (cont.)

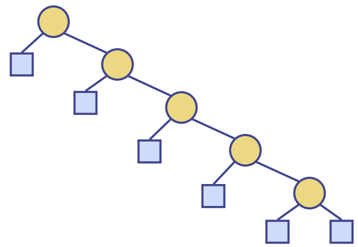
- Key k to be removed is stored at a node v whose children are both internal
- 1. Find the internal node w that follows v in an inorder traversal (find the smallest w larger than v)
- 2. Copy $key(w)$ into node v
- 3. Remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)
  - Why left child z?
- *No other cases?*

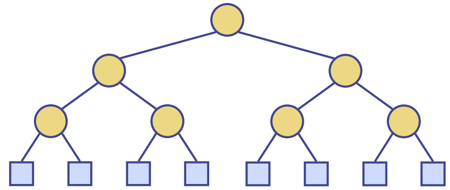
Example: remove 3



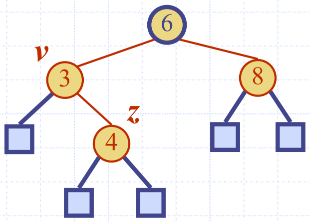
Performance

- Consider an ordered map with n items implemented by a binary search tree of height h
  - Space: $O(n)$
  - methods get, floorEntry, ceilingEntry, put and erase take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case
- Question: Can we find the algorithm with worst-case $O(\log n)$
  - Idea??? Balancing



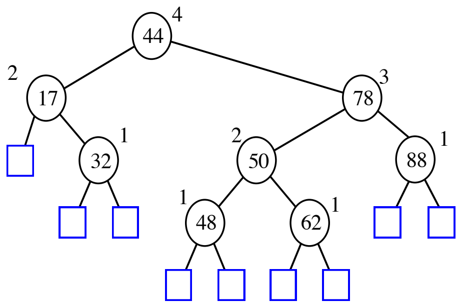


AVL Trees



Adelson-Velskii, G.; E. M. Landis (1962). “"An algorithm for the organization of information"”. *Proceedings of the USSR Academy of Sciences* ***146****: 263–266.* (Russian) English translation by Myron J. Ricci in *Soviet Math. Doklady*, 3:1259–1263, 1962.

AVL Tree Definition



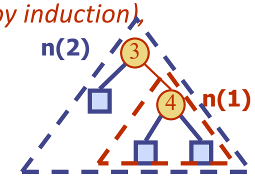
- An AVL Tree T is a binary search tree with the following property
  - *Height-Balance:* For every internal node v of T, the heights of the children of v can differ by at most 1
- This tree seems to be well-balanced
  - Height: O(log n)

Height of an AVL Tree (1)

- Fact: The height of an AVL tree storing *n* keys is *O(log n)*.
- Proof
  - *n(h)*: the minimum number of internal nodes of an AVL tree of height h.
  - Easily see that *n(1) = 1* and *n(2) = 2*
  - For *h > 2*, an AVL tree of height *h* and the minimum number of nodes contains (i) the root node, (ii) one AVL subtree of height *h-1* and (iii) another AVL subtree of height *h-2*.
  - That is, *n(h) = 1 + n(h-1) + n(h-2)*
- Knowing *n(h-1) > n(h-2)*, we get *n(h) > 2n(h-2)*. So

  n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

  n(h) > $2^i n(h-2i)$



Height of an AVL Tree (2)

- *n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),*

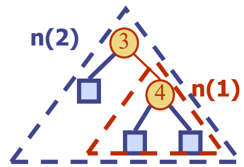
  n(h) > $2^i n(h-2i)$ *(for any integer i, such that h-2i ≥ 1)*
- We pick *i* so that *h-2i = 1* or *2* (base case)

$$i = \left\lceil \frac{h}{2} \right\rceil - 1.$$

- Then, we have

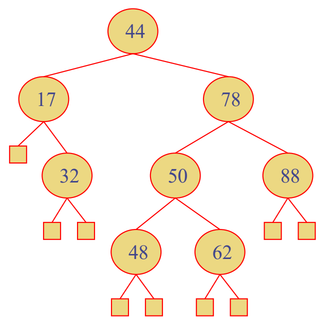
$$\begin{aligned} n(h) &> 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n\left(h - 2\left\lceil \frac{h}{2} \right\rceil + 2\right) \\ &\geq 2^{\lceil \frac{h}{2} \rceil - 1} n(1) \\ &\geq 2^{\frac{h}{2} - 1}. \end{aligned}$$

- Taking logarithms: *h < 2log n(h) +2*
- Thus, the height of an AVL tree is *O(log n)*

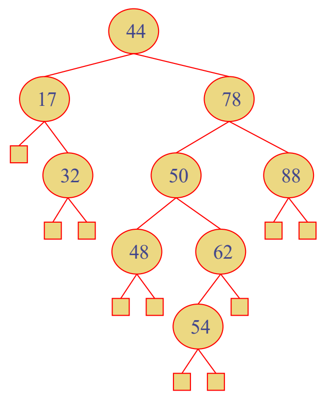


Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example of insertion 54. What's the problem?



before insertion 54



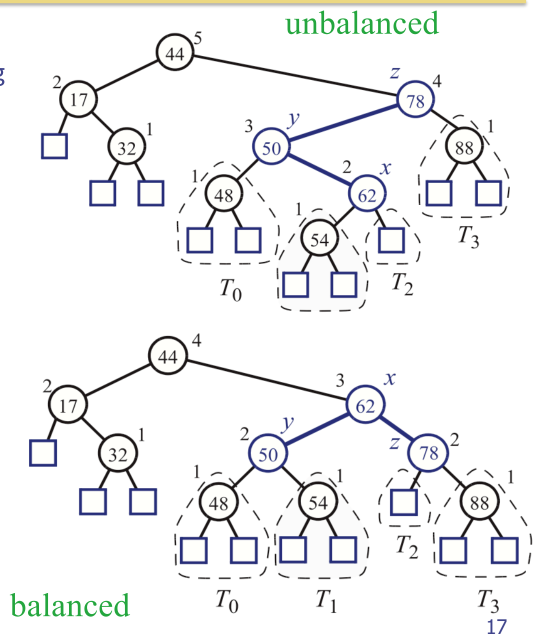
after insertion 54

Rebalancing Needed

- How should we do this?
  - (1) Take some examples
  - (2) Find difference cases
  - (3) Make each sub-algorithm for each case
  - (4) Make an entire algorithm
  - (5) Run it with some inputs
  - (6) Find out it is not working perfectly, and say "What the hell is this?" "How should I do?"
- Lessons
  - Let's summarize them later

Rebalancing Example: Insertion of w=54

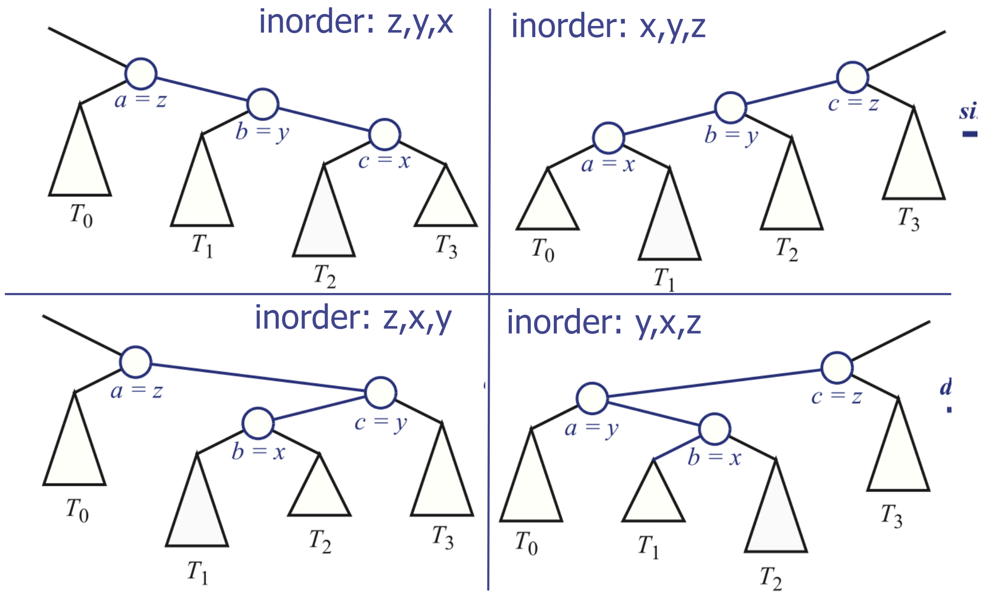
- "Search-and-Repair" strategy
- ***z***: first node we encounter in going up from w toward the root such that ***z*** is unbalanced
- ***y***: the child of ***z*** with higher height (note that ***y*** must be an ancestor of w)
- ***x***: the child of ***y*** with higher height (there cannot be a tie and node ***x*** must be an ancestor of ***w***)
- What are we doing for balancing?
- Can we do this systematically?
- What are other cases?



Please remember the notations! z, y, z

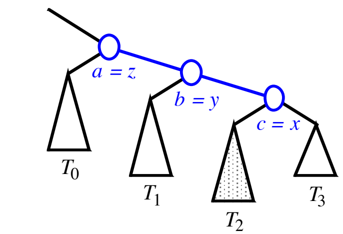
- ***z***: first node we encounter in going up from w toward the root such that ***z*** is unbalanced
  - "w에서 위로 쭉 올라가서, balance깨지는 첫 놈"
- ***y***: the child of ***z*** with higher height
  - "그 놈의 자녀 중 키가 큰 놈"
- ***x***: the child of ***y*** with higher height
  - "그 키 큰 자녀의 자녀(손주) 중 키가 큰 놈"
- Rename ***x,y,z*** as ***a,b,c*** so that ***a*** precedes ***b*** and ***b*** precedes ***c*** in "inorder traversal"
  - We can make many combinations

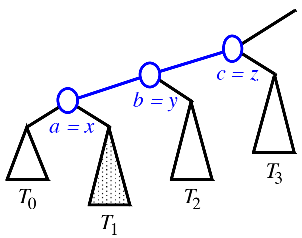
4 Combinations



Restructuring (as Single Rotations)

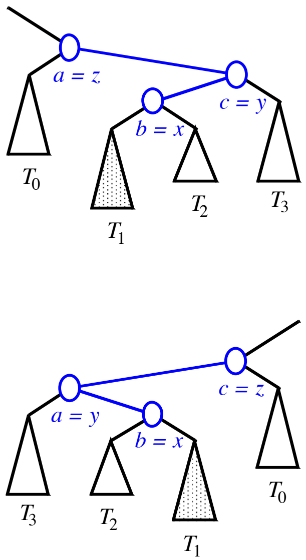
- Single Rotations:





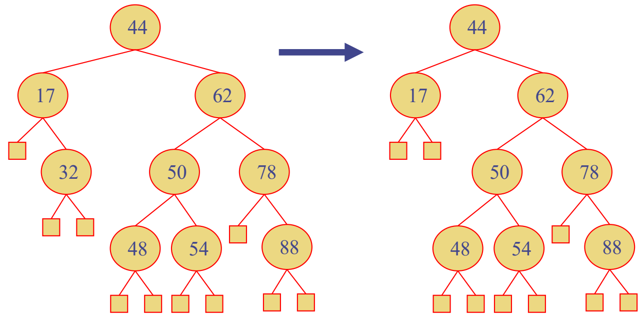
Restructuring (as Double Rotations)

- double rotations:



Removal

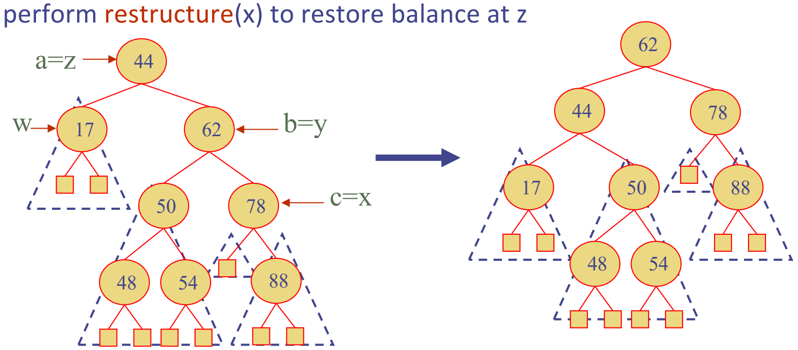
- Removal begins as in a binary search tree, which means the node removed (after copying the in-order successor) will become an empty external node. Its parent, w, may cause an imbalance.
- Example:



before deletion of 32 after deletion

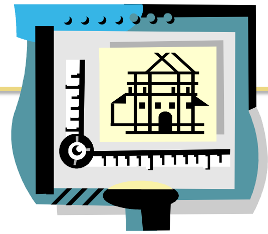
Rebalancing after a Removal

- Let z be the first unbalanced node encountered while travelling up the tree from w. Also, 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



- What happens if z is an internal node, not the root?
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached

AVL Tree Performance



- a single restructure takes O(1) time
  - using a linked-structure binary tree
- find takes O(log n) time
  - height of tree is O(log n), no restructures needed
- put takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- erase takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

Recall: Rebalancing Needed

How should we do this?

- (1) Take some examples
- (2) Find difference cases
- (3) Make each sub-algorithm for each case
- (4) Make an entire algorithm
- (5) Run it with some inputs
- (6) Find out it is not working perfectly, and say "What the hell is this?" "How should I do?"

Lessons

- Sometimes, we need to do case-by-case handling to complete the algorithm
- People often rely on "Half-assed (대충) algorithm design first " and "Complete it using example inputs". Not recommended.
  - Same as "Roughly make the code, and debug it later". Bad coding behavior

Questions?