

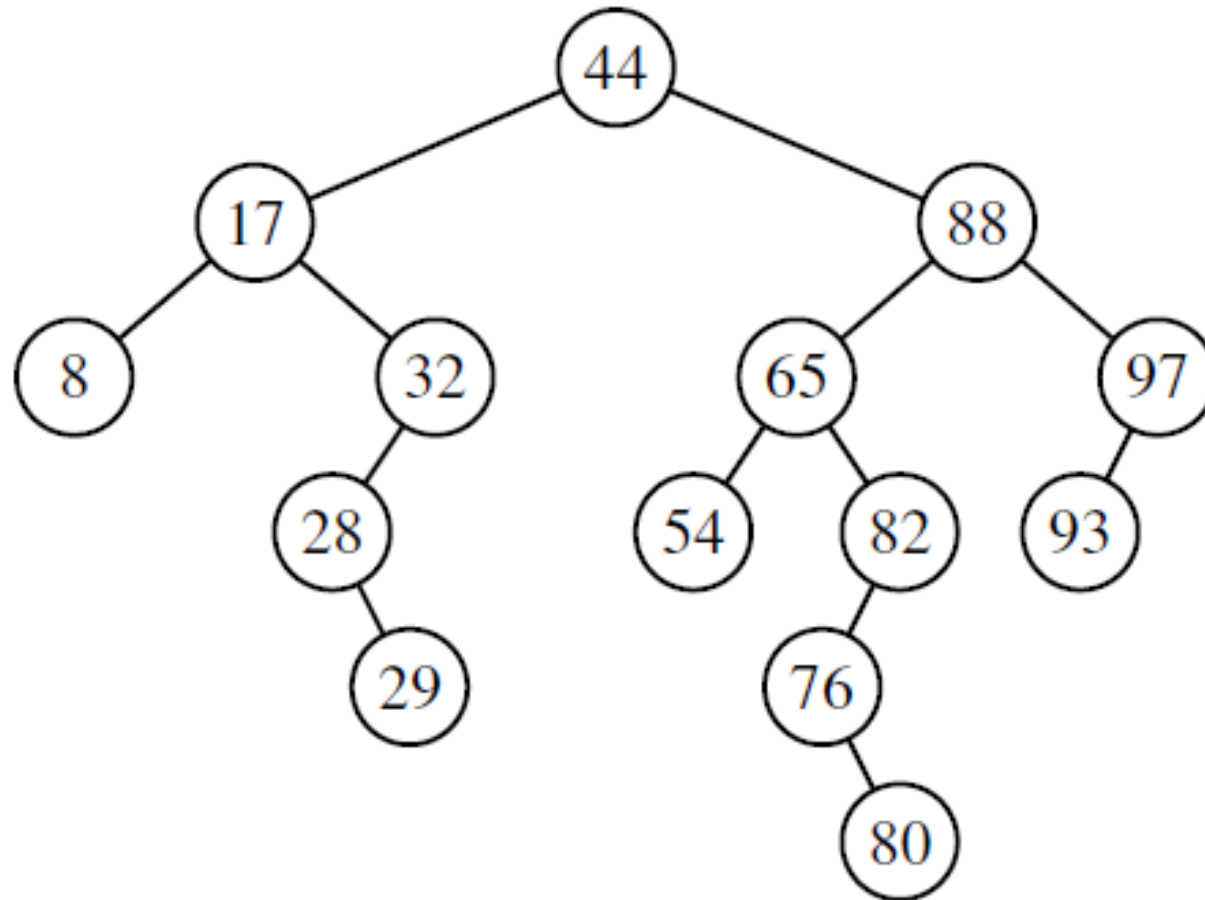


DET102 Data Structures and Algorithms

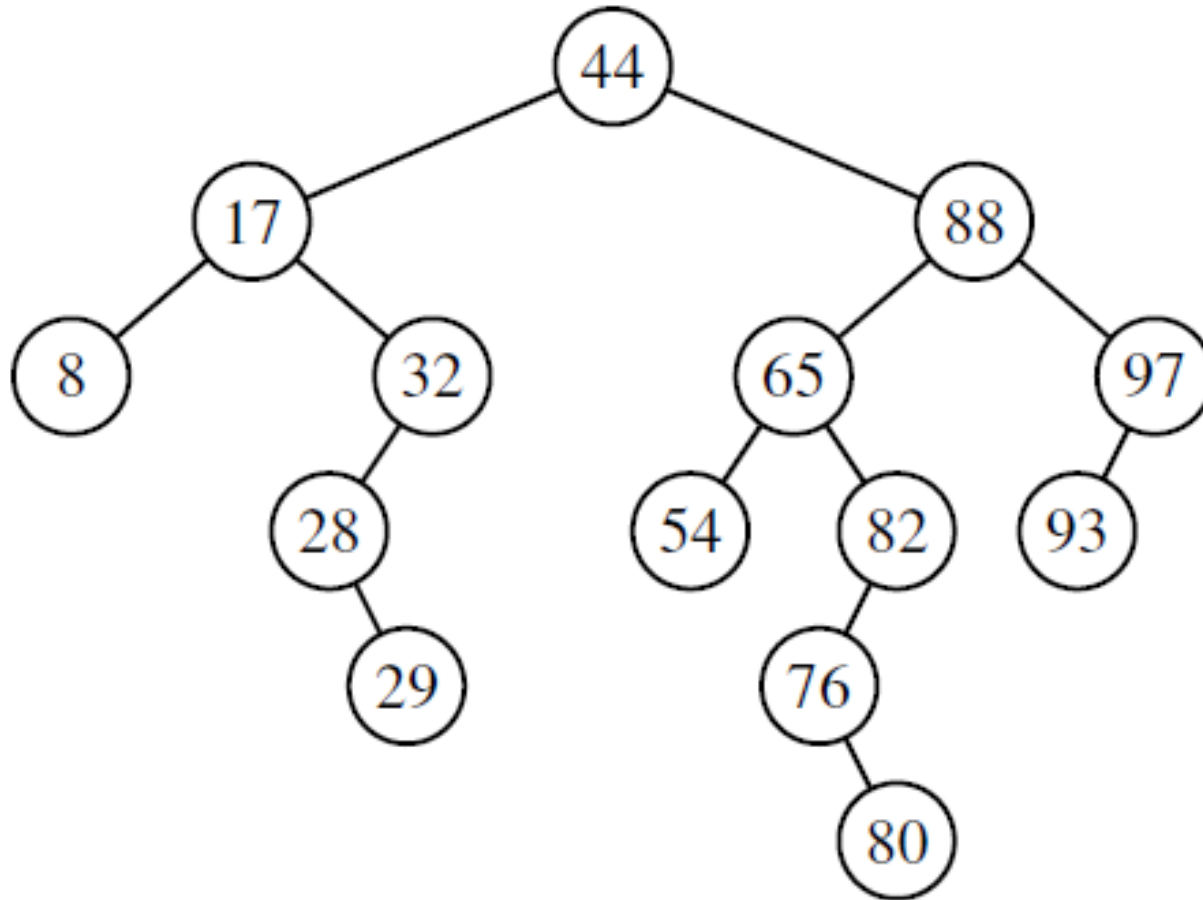
Lecture 07: Binary Search Tree

Binary Search Tree

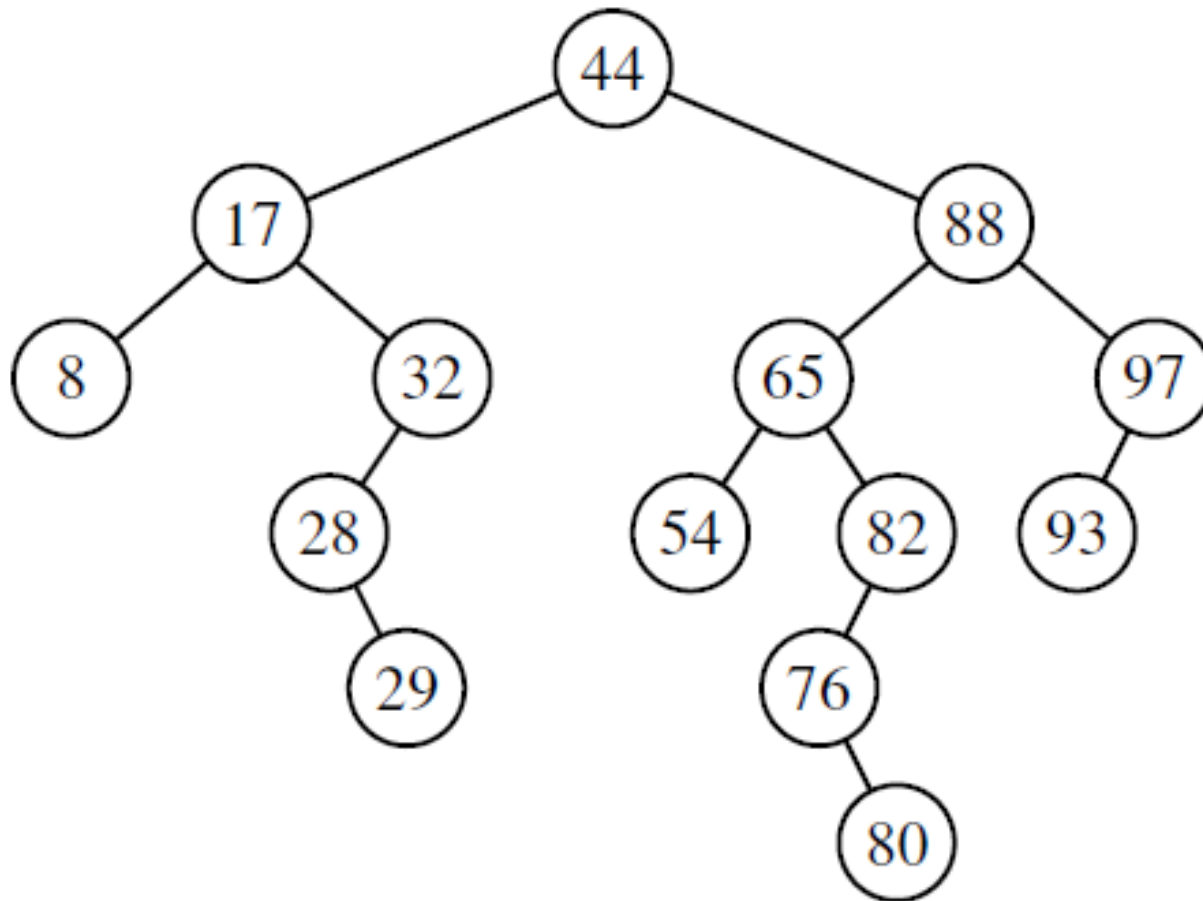
- It is a binary tree.
- For each node v in this tree
 - all the nodes in its left subtree have **smaller** data than v
 - all the nodes in its right subtree have **larger** data than v
- If we traverse the tree in inorder, then all the data are visited in **increasing** order.



What is the inorder traversal of the above binary search tree?



Is 80 in the above binary search tree?



Is 81 in the above binary search tree?

ADT of Binary Search Tree

➤ Attributes of BST node

- Data
- Left child
- Right child
- Count (optional)

➤ Operations

- Create (initialize)
- Search (find)
- Insert
- Delete
- Traverse
- Copy
- Find the height
- Find the number of nodes
- Find the number of leaves.

We can inherit all of these operations from the binary tree.

Search

Algorithm TreeSearch(T, p, k):

if $k == p.key()$ then

 return p //successful search

else if $k < p.key()$ and T.left(p) is not None then

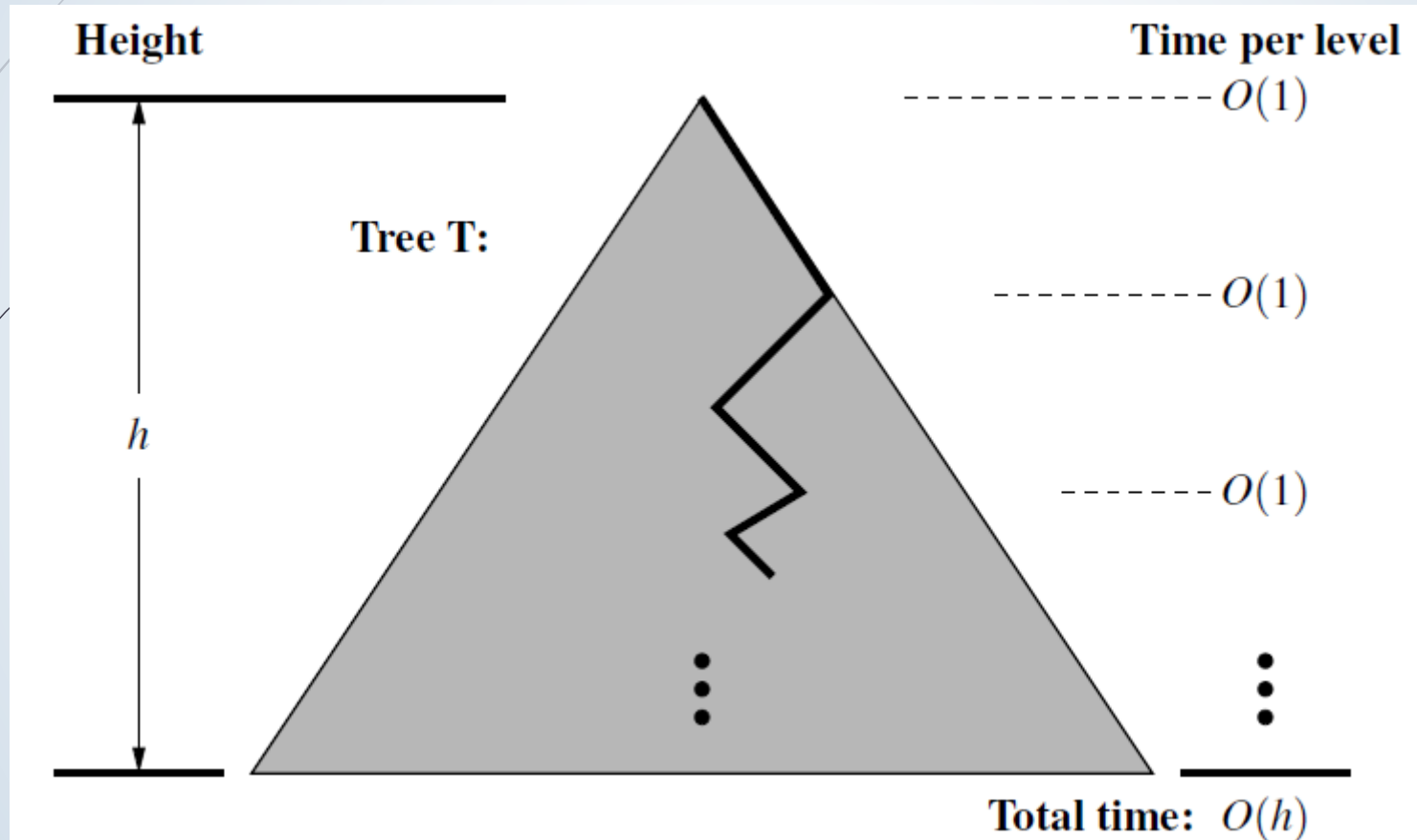
 return TreeSearch(T, T.left(p), k) //recur on left subtree

else if $k > p.key()$ and T.right(p) is not None then

 return TreeSearch(T, T.right(p), k) //recur on right subtree

return p //unsuccessful search

Search complexity



Insertion

➔ **Algorithm** TreeInsert(T, k, v):

Input: A search key k to be associated with value v

$p = \text{TreeSearch}(T, T.\text{root}(), k)$

if $k == p.\text{key}()$ **then**

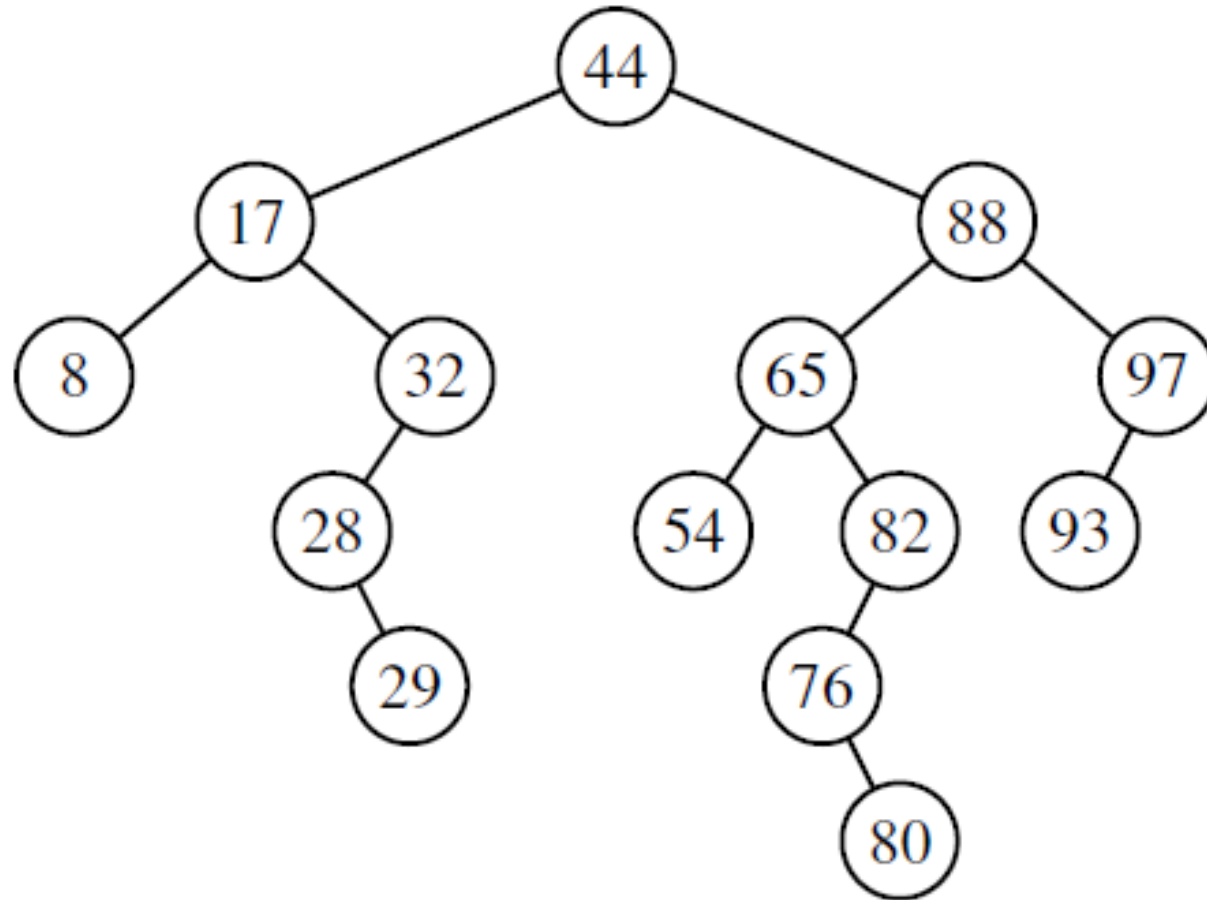
 Set p 's value to v

else if $k < p.\text{key}()$ **then**

 add node with item (k, v) as left child of p

else

 add node with item (k, v) as right child of p



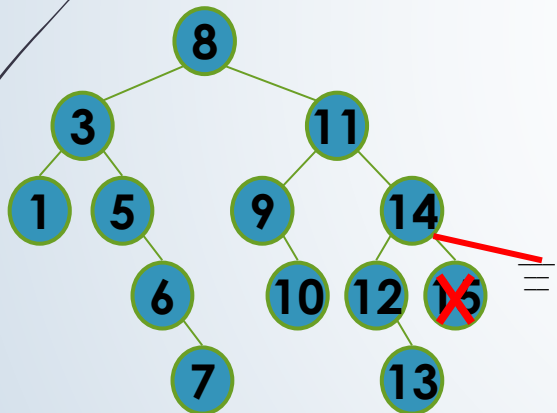
How to insert 81 in the above binary search tree?

Binary Search Tree - Deletion

Delete a node in a BST:

Case 1:

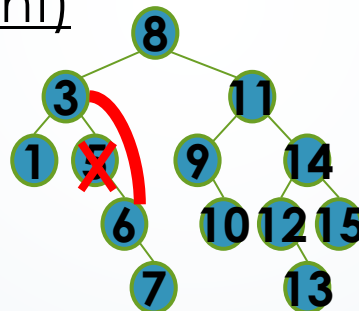
To delete a leaf node



- Set the parent node's child pointer to NULL

Case 2:

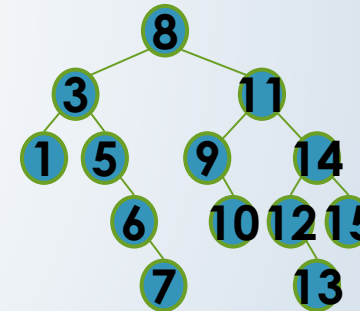
To delete a node that has 1 subtree (left or right)



- Set the parent's child pointer to root of unwanted node's subtree (left or right).

Case 3:

To delete a node that has 2 subtrees



More complicated

Binary Search Tree - Deletion

Delete a node in a BST:

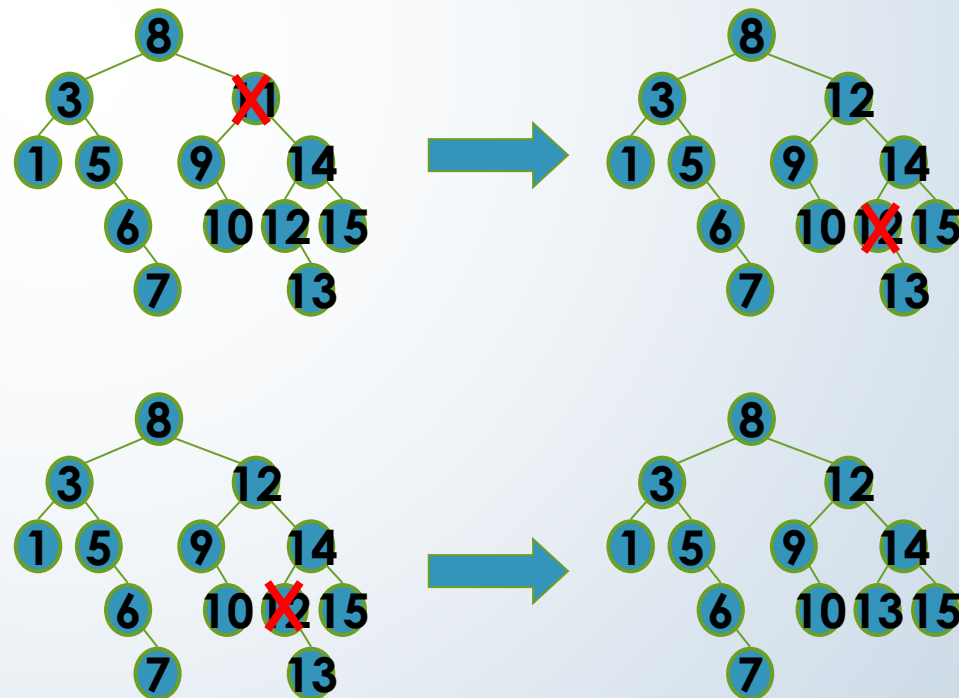
Case 3:

To delete a node
that has 2 subtrees

Replace its value by its
inorder successor (or
predecessor):

**The inorder successor is the
leftmost node of right subtree.**

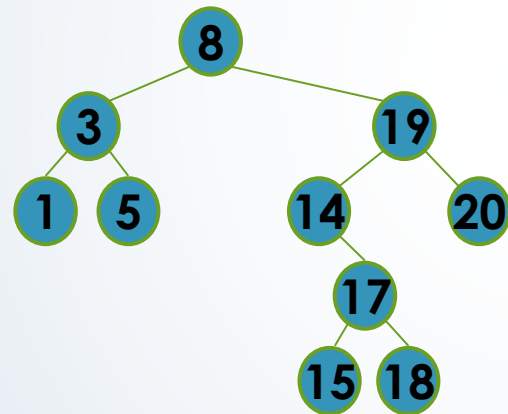
Delete the successor in turn:



Binary Search Tree - Deletion

Exercise:

What is the result after the node containing 8 in the following tree is deleted?



Deletion

- ▶ Deleting a tree node is a complicated business. One way to minimize this complication is to use “*Lazy Delete*”.
 - ▶ Update the “count” information without deleting the corresponding tree node


Lazy Delete

- ▶ Use traversal to lazy delete a value.
 - ▶ first, find the node using tree traversal
 - ▶ if the **count** field=0 then report “error, doesn’t exist”
 - ▶ else reduce the count field by one

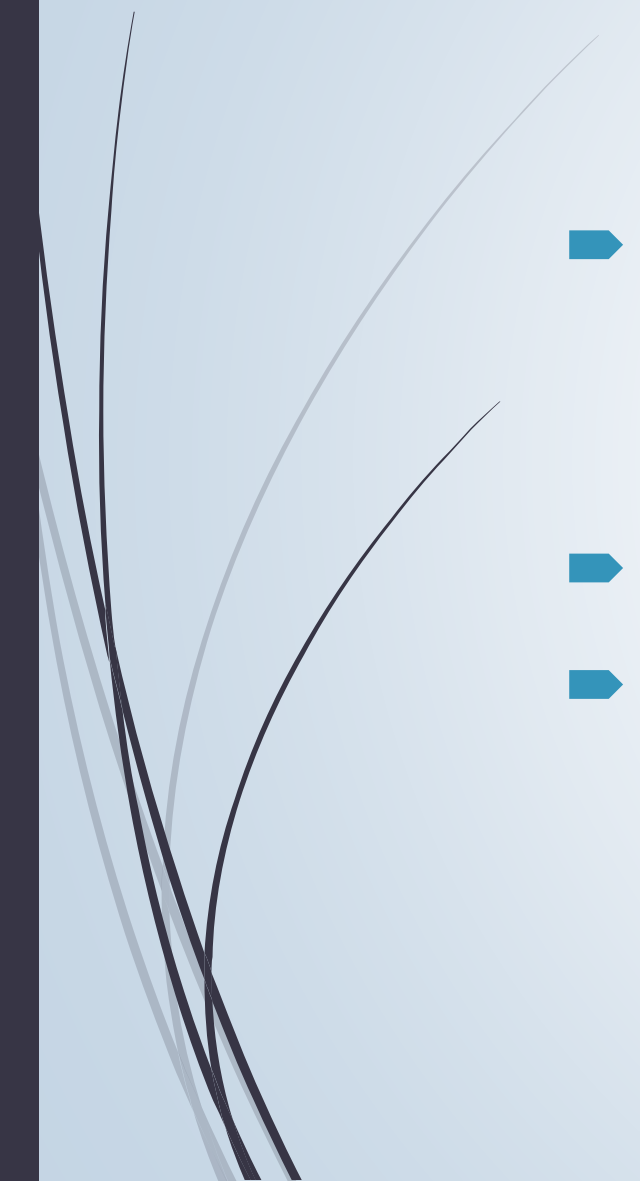
If a node's count=0,
then it means that the
node has already
been deleted.

Worst Case Analysis

- The worst case running time for search in BST is:
- The worst case running time for insertion in BST is:
- The worst case running time for deletion in BST is:
- What is the worst structure of BST?
- What kind of BST is good?
- What is the running time for good BST?
- How to maintain good BST?



Other Trees

- Balanced Binary Search Tree
 - AVL tree
 - Red-black tree
 - B tree
 - B+ tree
- 

AVL Tree

AVL Tree

- AVL tree was introduced in 1962 by Adelson-Velskii and Landis.
- Definition
 - An empty tree is an AVL tree.
 - If B is the nonempty binary tree with L and R as its left and right subtrees, then B is an AVL tree if and only if
 - L and R are AVL trees, and
 - $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of L and R subtrees, respectively.

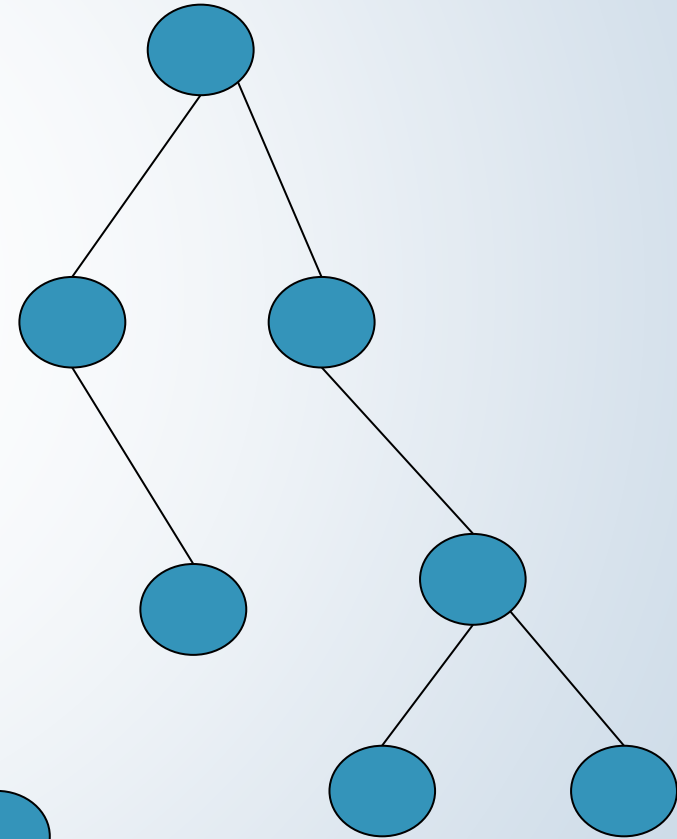
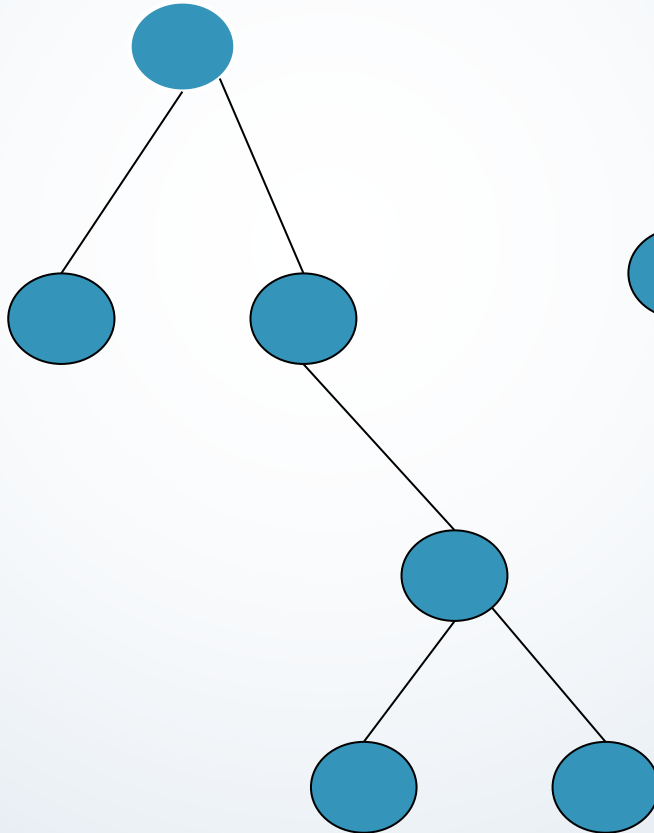
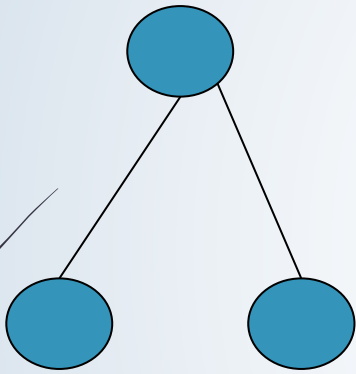


Height-
balance
property

Balance factor

- The balance factor of a node in a binary tree is defined as the heights h_L minus h_R .
- AVL tree may have the following balance factor for a node $\{-1, 0, 1\}$.
- If a node x has balance factor other than $\{-1, 0, 1\}$, then it is said to be unbalanced.
- If balance factor of any node is -1 , then it is said to be right balanced, and if balance factor is 1 , then it is said to be left balanced.
- For balance factor 0 , the tree is said to be completely balanced.

Example

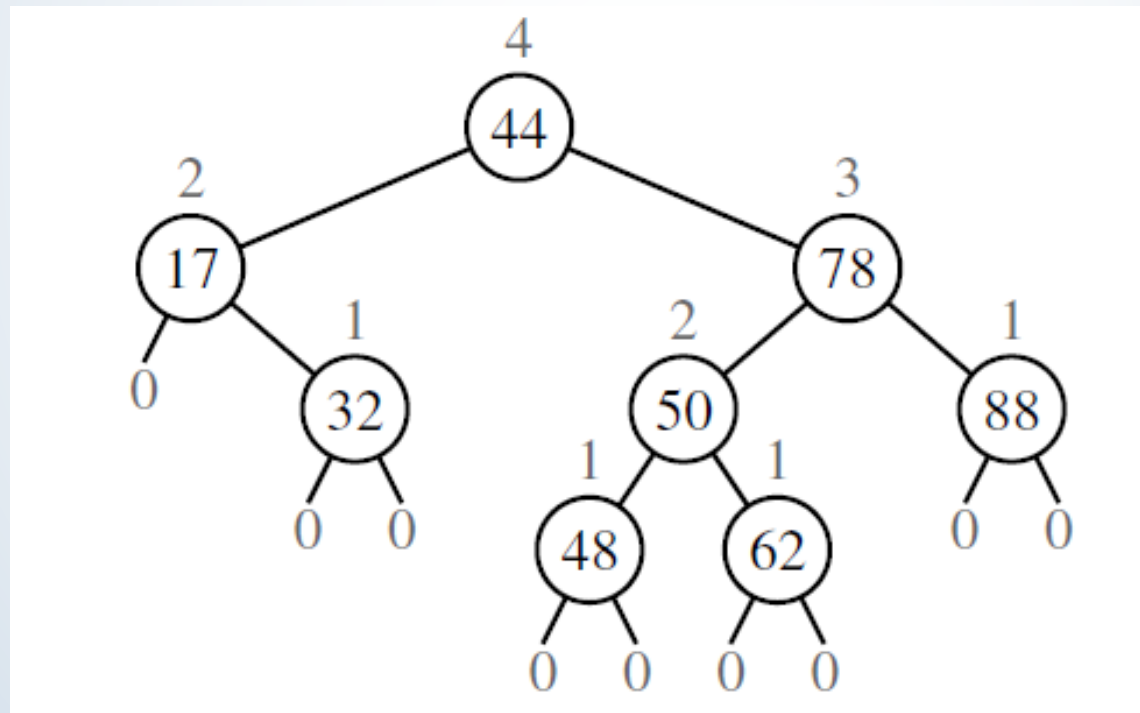


AVL Search Tree

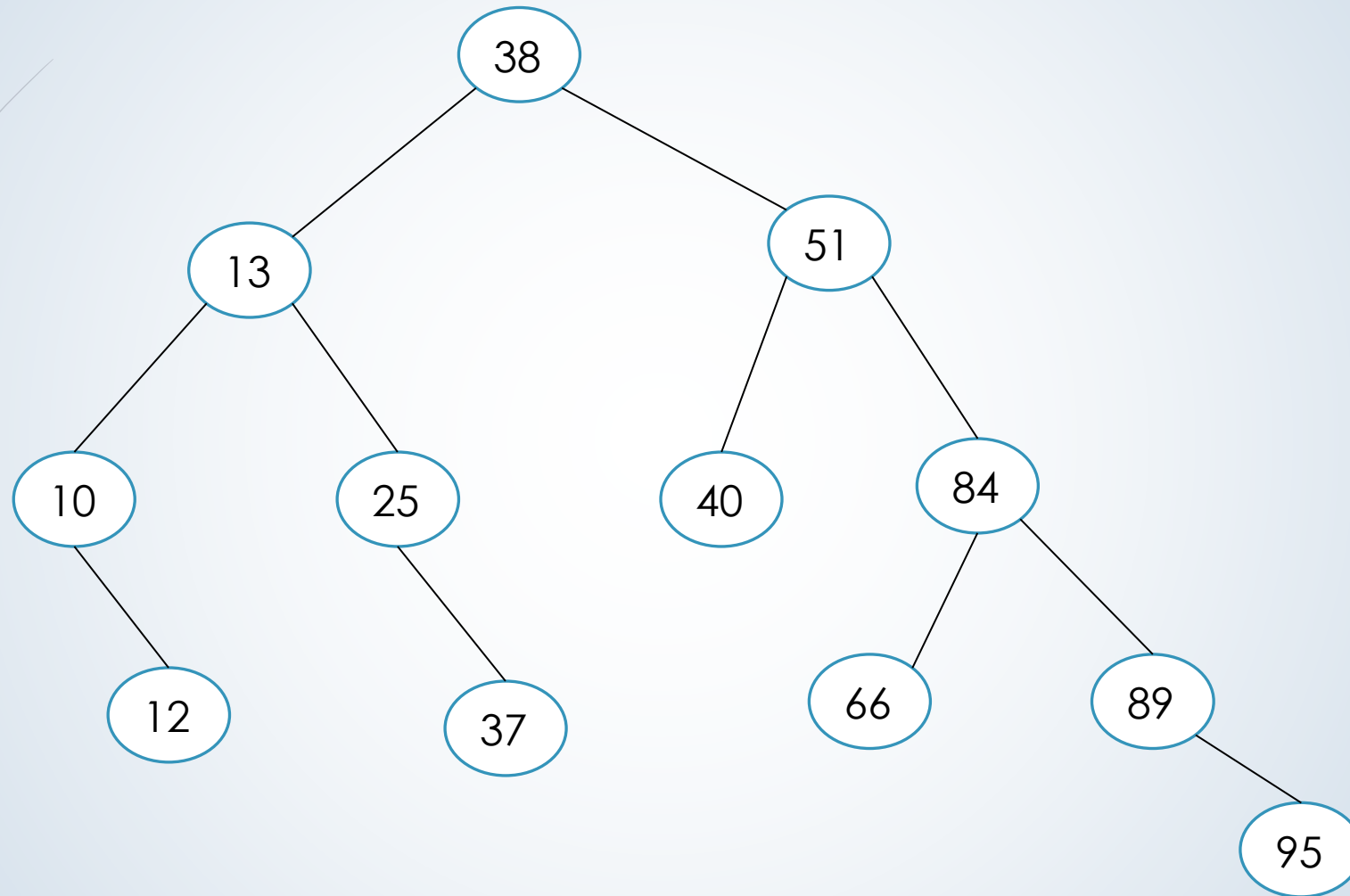
- An empty binary search tree is an AVL search tree.
- If Bst is the non empty binary search tree with Lst and Rst as its left and right subtrees, then Bst is an AVL seartree if and only if
 - Lst and Rst are AVL trees, and
 - $|h_{Lst} - h_{Rst}| \leq 1$ where h_{Lst} and h_{Rst} are the heights of Lst and Rst subtrees, respectively.
- For AVL search tree to be balanced, the balance factor $bf(x)$ of node x must be $bf(x) = h_{Lst} - h_{Rst} = \{-1, 0, 1\}$

AVL tree

- Any binary search tree T that satisfies the height-balance property is said to be an **AVL tree**.

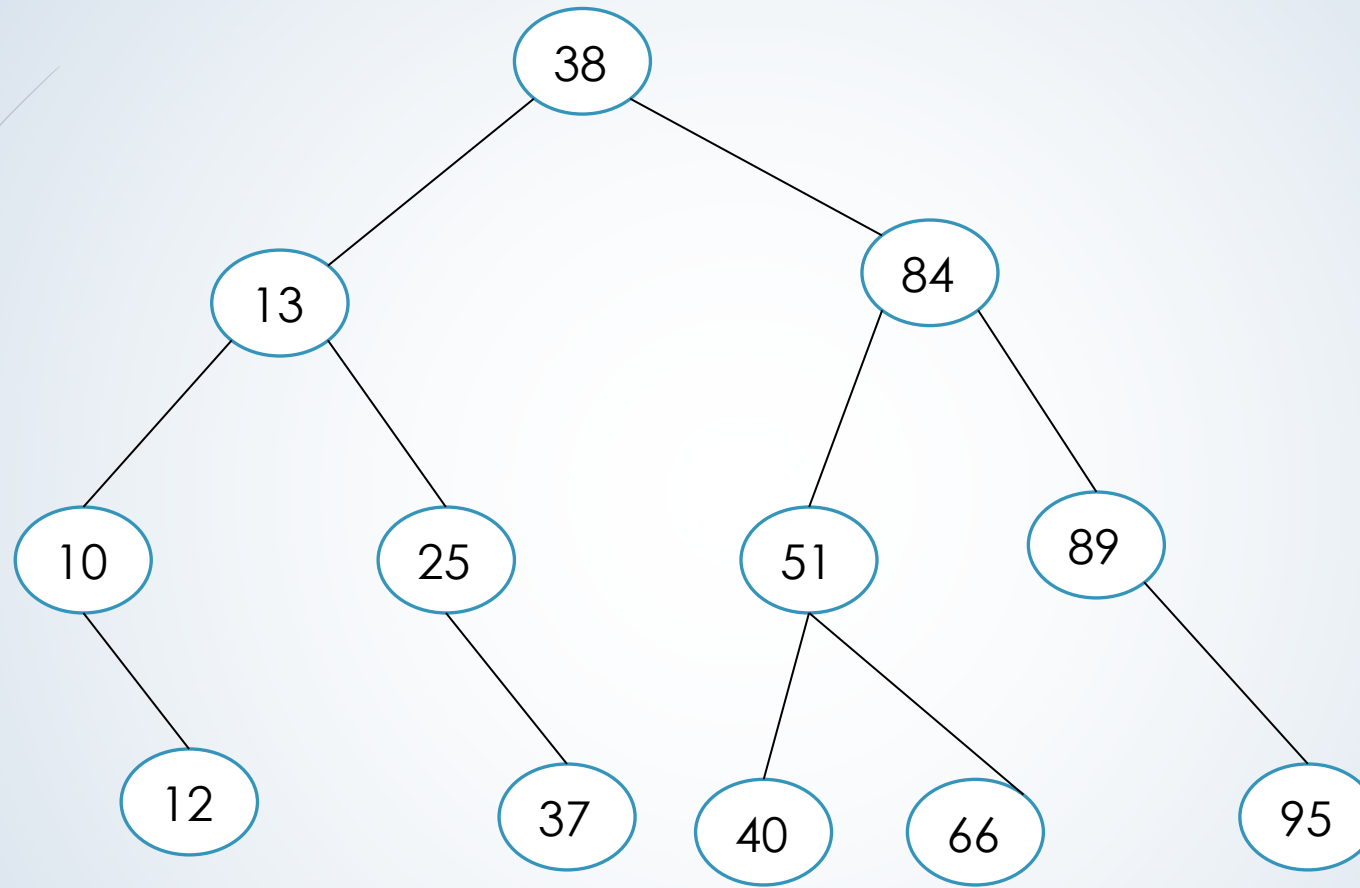


Is it an AVL tree?



Is it an AVL tree?

25



Height of an AVL Tree

- ➔ Consider $n(h)$ to be the minimum number of nodes in an AVL tree of height h .
- ➔ In the worst cast, the height of one of the subtree is $h-1$ and the height of the other subtree is $h-2$. Both are AVL tree.
 - ➔ $n(h) = n(h-1) + n(h-2) + 1$
 - ➔ $n(0) = 1, n(1) = 2$
- ➔ Height: $O(\log n)$

The height of an AVL tree storing n entries is $O(\log n)$.

Searching in an AVL search tree

- ➡ Like binary search tree.
- ➡ Function `AVLsearch (R,K)`
 - ➡ Step 1: checking, is empty?
 - ➡ Step 2: if k is equal to the value of the root node
 - ➡ Step 3: k is less than the key value of the root node, `AVLsearch(R(lchild),K)`
 - ➡ Step 4: k is greater than the key value of the root node, `AVLsearch(R(rchild),K)`.

Insertion in an AVL search tree

- If AVL search tree is empty
- If AVL search tree has only one node
- If AVL search tree has many nodes
 - Still balanced
 - Not balanced: left-right case, left-left case, right-left case, right-right case.

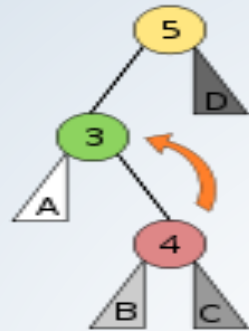
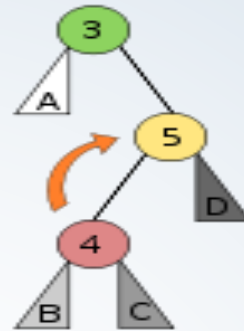
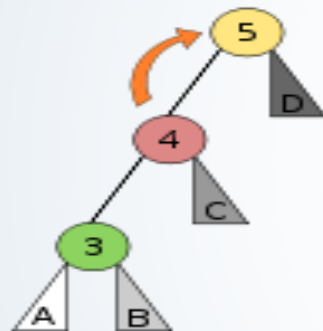
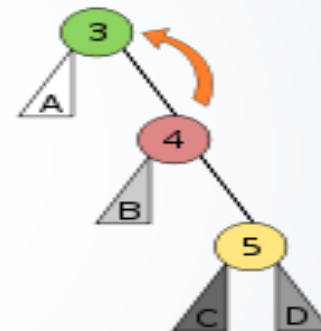
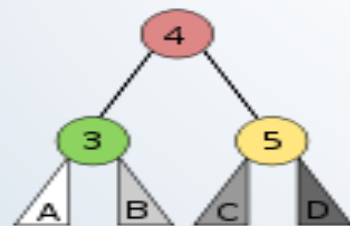
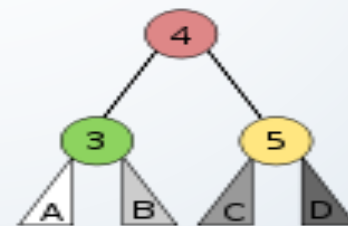
We need to restore the balance of any portions of the tree that are adversely affected by the change.

Insertion in an AVL search tree

- Let P be the root of the unbalanced subtree, with R and L denoting the right and left children of P respectively.
- **Right-Right case** and **Right-Left case**:
- If the 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 (or in case of deletion also 0), a **single left rotation** (with P as the root) is needed (Right-Right case).
 - If the balance factor of R is +1, two different rotations are needed. The first rotation is a **right rotation** with R as the root. The second is a **left rotation** with P as the root (Right-Left case).

Insertion in an AVL search tree

- **Left-Left case** and **Left-Right case**:
- If the balance factor of P is 2, then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must be checked. The right rotation with P as the root is necessary.
 - If the balance factor of L is +1 (or in case of deletion also 0), a **single right rotation** (with P as the root) is needed (Left-Left case).
 - If the balance factor of L is -1, two different rotations are needed. The first rotation is a **left rotation** with L as the root. The second is a **right rotation** with P as the root (Left-Right case).

Left Right Case**Right Left Case****Left Left Case****Right Right Case****Balanced****Balanced**

➤ L-R

➤ 30, 20, 40, 10, 25

➤ Insert 27

➤ L-L

➤ 30, 20, 40, 10, 25

➤ Insert 5

➤ R-L

➤ 30, 20, 40, 35, 50

➤ Insert 32

➤ R-R

➤ 30, 20, 40, 35, 45

➤ Insert 50



Overall Scheme for insert(x)

- Search for x in the tree; insert a new leaf for x (as in previous BST)
- If parent of x is not balanced, perform single or double rotation as appropriate
 - How do we know the height of a subtree?
 - Have a “height” attribute in each node
- Set x = parent of x and repeat the above step until x = root



Deletion

- Do normal deletion for binary search tree
- Rebalancing (need to check all the ancestors of the affected node)
- Rather complicated



AVL drawback

- ▶ extra storage/complexity for height fields
- ▶ ugly delete code

- ▶ Solution: Splay tree

- ▶ Reference

- <https://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt>