# Properties of Binary Trees

Size vs Height

## Binary Tree: Recursive definition

- A Binary Tree is either
  - empty, or
  - consists of:
    - a distinguished node called the root, which contains an element and two disjoint subtrees
    - A left subtree $T_L$, which is a binary tree
    - A right subtree $T_R$, which is a binary tree

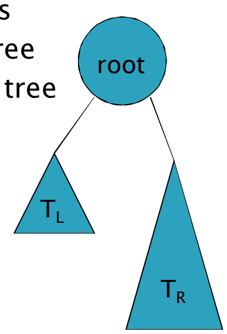
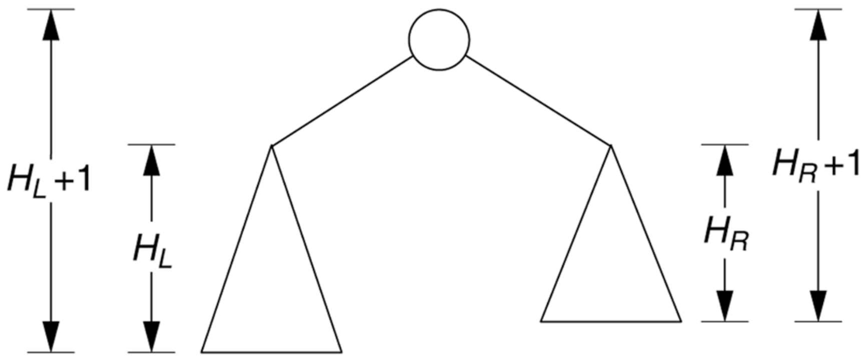


**Figure 18.20**
Recursive view of the node height calculation:
$H_T = \text{Max}\ (H_L + 1, H_R + 1)$






## Size and Height of Binary Trees

- If T is a tree, we'll often write h(T) for the height of the tree, and N(T) for the number of nodes in the tree
- For a particular h(T), what are the upper and lower bounds on N(T)?
  - **Lower:** N(T) ≥ _______ (prove it by induction)
  - **Upper** N(T) ≤ _______ (prove it by induction)
  - Thus _______ ≤ N(T) ≤ _________
- Write bounds for h(T) in terms of N(T)
  - Thus _______ ≤ h(T) ≤ _________

## Extreme Trees

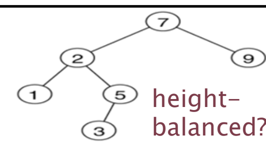
- A tree with the maximum number of nodes for its height is a **full** tree.
  - Its height is O(log N)
- A tree with the minimum number of nodes for its height is essentially a ______ ______
  - Its height is O(N)
- Height matters!
  - We saw that the algorithms for search, insertion, and deletion in a Binary Search Tree are O(h(T))

# Introduction to Balanced Trees

# BST algorithms and their efficiency

- Review:
  - Efficiency of insertion, deletion, find for
    - Array list
    - Linked list
- BST insertion algorithm – O(height of tree)
- BST deletion algorithm – O(height of tree)
- BST search algorithm – – O(height of tree)
- Efficiency (worst case)?
  - Can we get a better bound?
  - What about balancing the tree each time?
  - What do we mean by completely balanced?
  - Insert E C G B D F A into a tree in that order.
  - What is the problem?
  - How might we do better? (less is more!)

# What have we discovered about BSTs so far?




height-balanced?

- We'd like the worst-case time for find, insert, and delete to be O(log N).
- The running time for find, insert, and delete are all proportional to the height of the tree.
- Height of the tree can vary from log N to N.
- Keeping the tree completely balanced is too expensive. Can require O(N) time to rebalance after insertion or deletion.
- Height-balanced trees may provide a solution.
  - A BST T is ***height balanced*** if T is empty, or if
    - $|\ height(T_L) - height(T_R)\ | \leq 1$, and
    - $T_L$ and $T_R$ are both height-balanced.
- What can we say about the maximum height of an height-balanced tree with N nodes?

**Why do we use absolute value in the formula?**

**Details: next slide**

## Height-Balanced trees

- We want to calculate the maximum height of a height-balanced tree with N nodes.
- It's not the shortest possible tree, but how close is it?
- We first look at the dual concept: find the minimum number of nodes in a HB tree of height h.
- Make a table of heights and # of nodes.
- What can we say in general about height as a function of number of nodes?

## What is an AVL tree?

- Named for authors of original paper, Adelson-Velskii and Landis (1962).
- It is a height-balanced Binary Search Tree.
- Recall: A BST T is *height balanced* if
  - T is empty, or if
    - | height( $T_L$ ) - height( $T_R$ ) | ≤ 1, and
    - $T_L$ and $T_R$ are both height-balanced.
- Maximum height of an AVL tree with N nodes is O(log N).

# Recap: Why we study AVL trees

- For a Binary Search Tree (BST), worst-case for *insert*, *delete*, and *find* operations are all O(height of tree).
- Height of tree can vary from O(log N) to O(N).
- We showed that the height of a height-balanced tree is always O(log N).
- Thus all three operations will be O(log N) if we can rebalance after insert or delete in time O(log N)

# More on AVL trees

- An AVL tree is
  1. height-balanced
  2. a Binary search tree
- We saw that the maximum height of an AVL tree with N nodes is O(log n).
- We want to show that after an insertion or deletion (also O(log n) since the height is O(log n)), we can rebalance the tree in O(log n) time.
  - If that is true, then find, insert, and remove, will all be O(Log N).
- An extra field is needed in each node in order to achieve this speed. Values: / = \
  We call this field the ***balance code.***
- The balance code could be represented by only two bits.

## Balancing an AVL tree after insertion

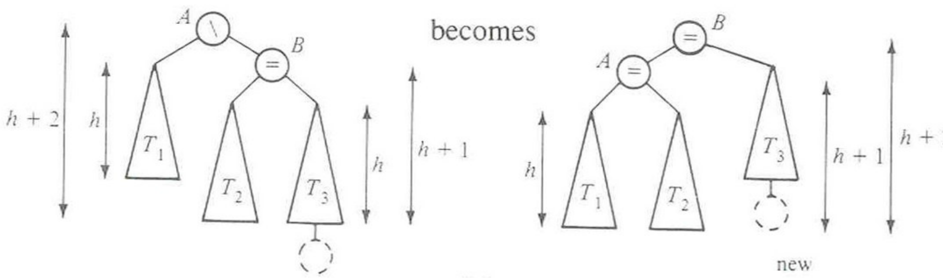
- Assume that the tree is height-balanced before the insertion.
- Start at the inserted node (always a leaf).
- Move back up the tree to the **first** (lowest) node (if any) where the heights of its subtrees now differ by more than one.
  - We'll call that node **A** in our diagrams.
- Do the appropriate single or double rotation to balance the subtree whose root is at this node.
- If a rotation is needed, we will see that the combination of the insertion and rotation leaves this subtree with the same height that it had before insertion.
- So why is the algorithm O(log N)?

## Which kind of rotation to do?

Depends on the first two links in the path from the node with the imbalance (A) down to the newly-inserted node.

| **First link** (down from A) | **Second link** (down from A's child) | **Rotation type** (rotate "around A's position") |
|---|---|---|
| Left | Left | Single right |
| Left | Right | Double right |
| Right | Right | Single left |
| Right | Left | Double left |

Single left rotation (right is the mirror image of this picture)




Diagrams are from *Data Structures* by E.M. Reingold and W.J. Hansen.