# Data Structures:
## Binary Search Trees, Height-Balanced Trees, AVL Tree

Won Kim
(Lecture by Youngmin Oh)
Spring 2022

# Binary Search

# Binary Search

- Prerequisite
  - An ordered list
- $O(\log_2 n)$

# Algorithm

- Take the midpoint of the sorted list.
- If the search key matches the key at the midpoint, search ends in success.
- If the list contains only one key and the search key does not match it, search ends in failure.
- If the search key is < the key at the midpoint, take the list to the left of the midpoint, and repeat from the start.
- If the search key is > the key at the midpoint, take the list to the right of the midpoint, and repeat from the start.

# Binary Search: Examples

7 9 2 5 15 3 14 1 8 11 12 4 13 6 10

# Binary Search: Example

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

search for 13

# Binary Search: Solution

take the midpoint of the list and compare with 13

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

take the list after 8

9 10 11 12 13 14 15

take the midpoint of the list and compare with 13

9 10 11 12 13 14 15

take the list after 12

13 14 15

# Binary Search: Solution (cont.)

take the list after 12

13 14 15

take the midpoint of the list and compare with 13

13 14 15

take the list before 14

13

take the midpoint of the list and compare with 13

13

# Binary Search: Examples

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

search for 5

1 3 4 5 7 8 9 10 12 13 14 16 20 21 25

search for 4
search for 22

# Binary Search: Examples

1 2 3 4 5

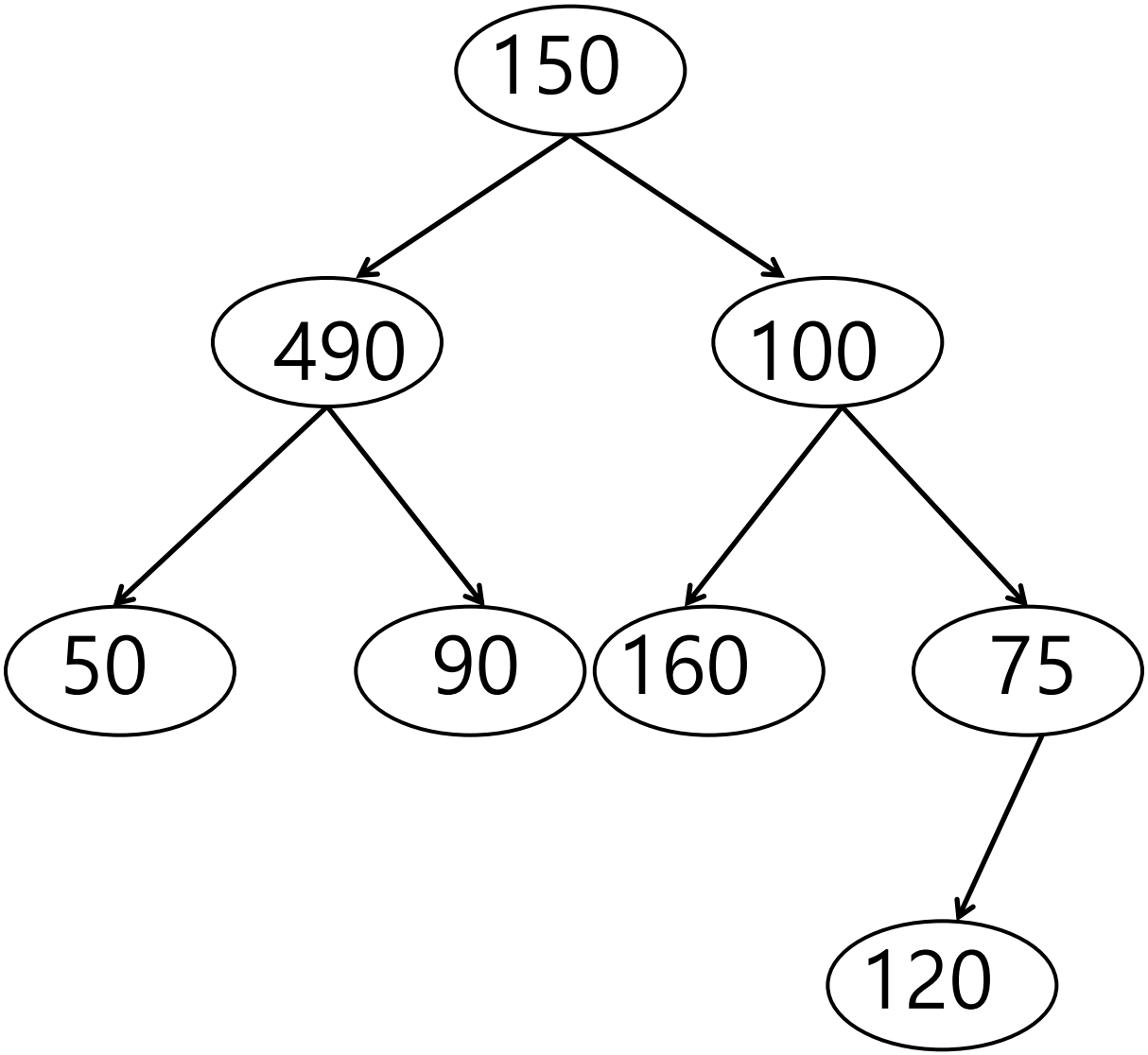
search for 3, 1, 5, 9

1 2 3 4

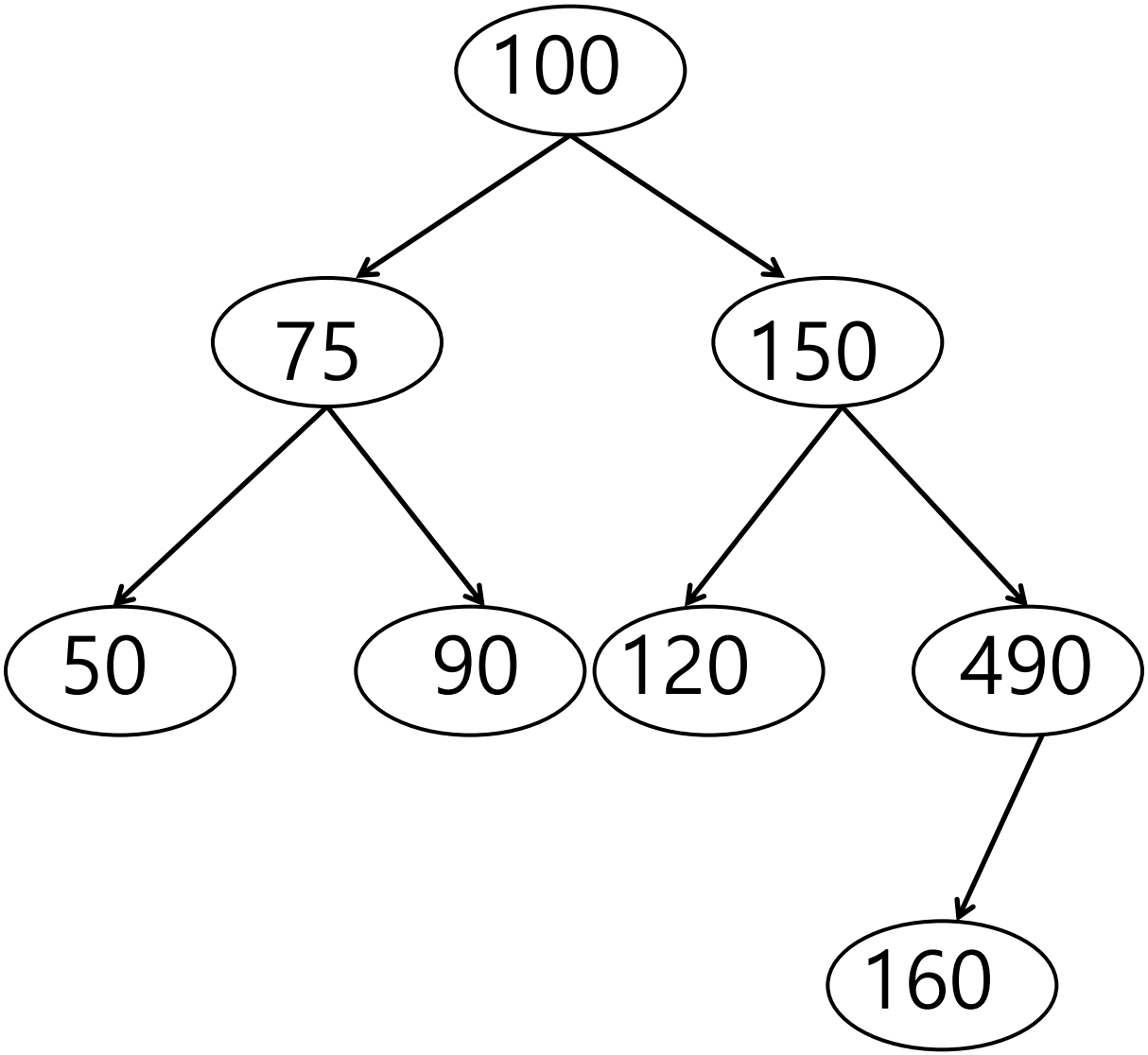
search for 1, 3, 2, 4, 9

# Binary Search Trees

# Binary Tree (no order)



# Binary Search Tree (ordered)



# Summary Definition of a Tree

- A tree is a linked list of data, where a data item is linked to other data items by a certain relationship.
- A search for a data item on a tree proceeds from one data item to another data item that satisfies a certain relationship.
- The hierarchy of a tree is a convenient visualization of the relationships among the data items.

# Properties

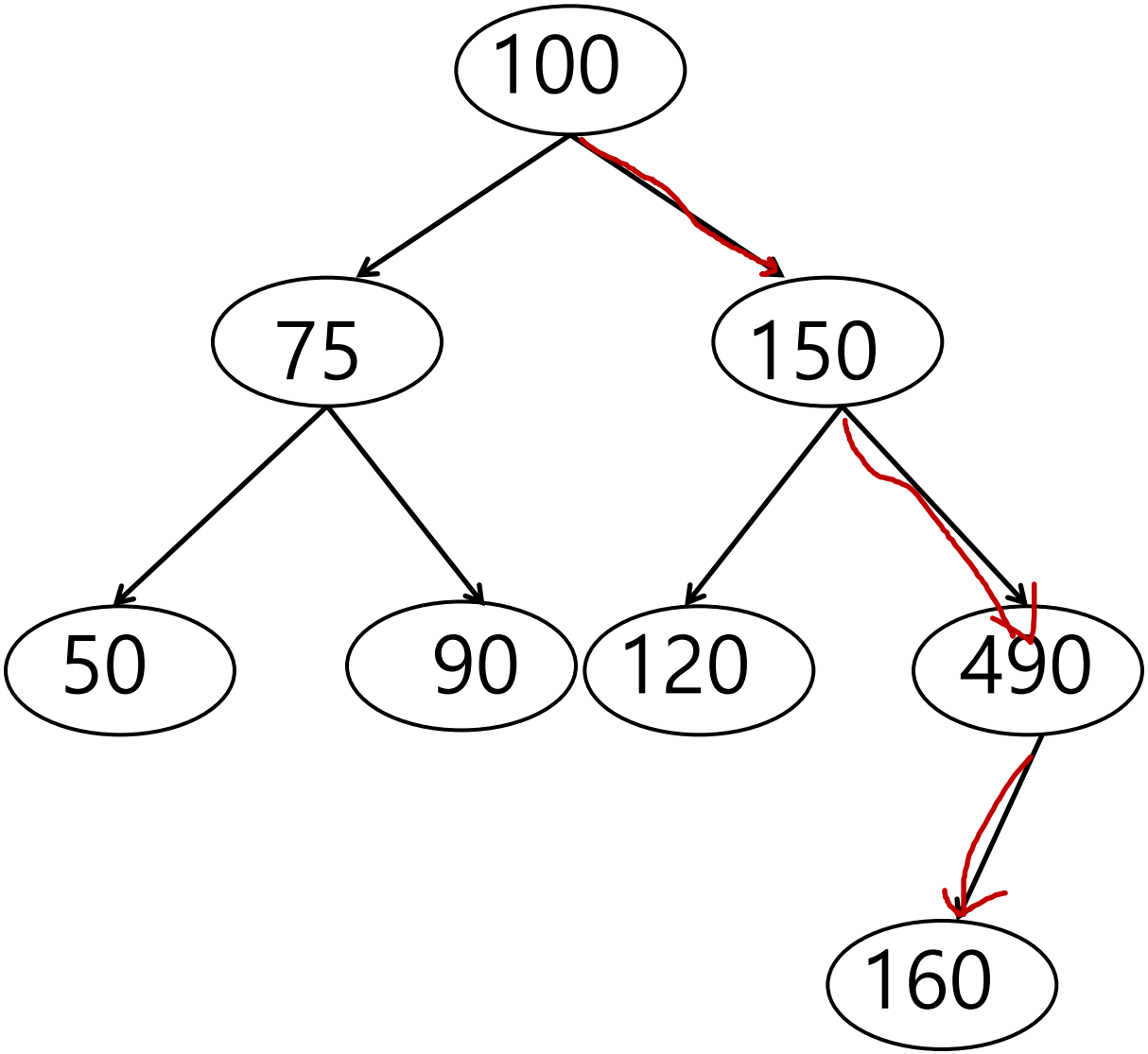
- Each Node Has a Unique Key (Data)
- The Key < The Key of Any Node in the Right Subtree
- The Key > The Key of Any Node in the Left Subtree

# Smaller to the Left, Larger to the Right

# Searching a Binary Search Tree

- Compare the Key to Be Searched with the Key of the Root Node.
- If the Search Key = the Root Key, Success.
- If the Search Key < the Root Key, Search the Left Subtree.
- If the Search Key > the Root Key, Search the Right Subtree.
- If No Match, Failure.

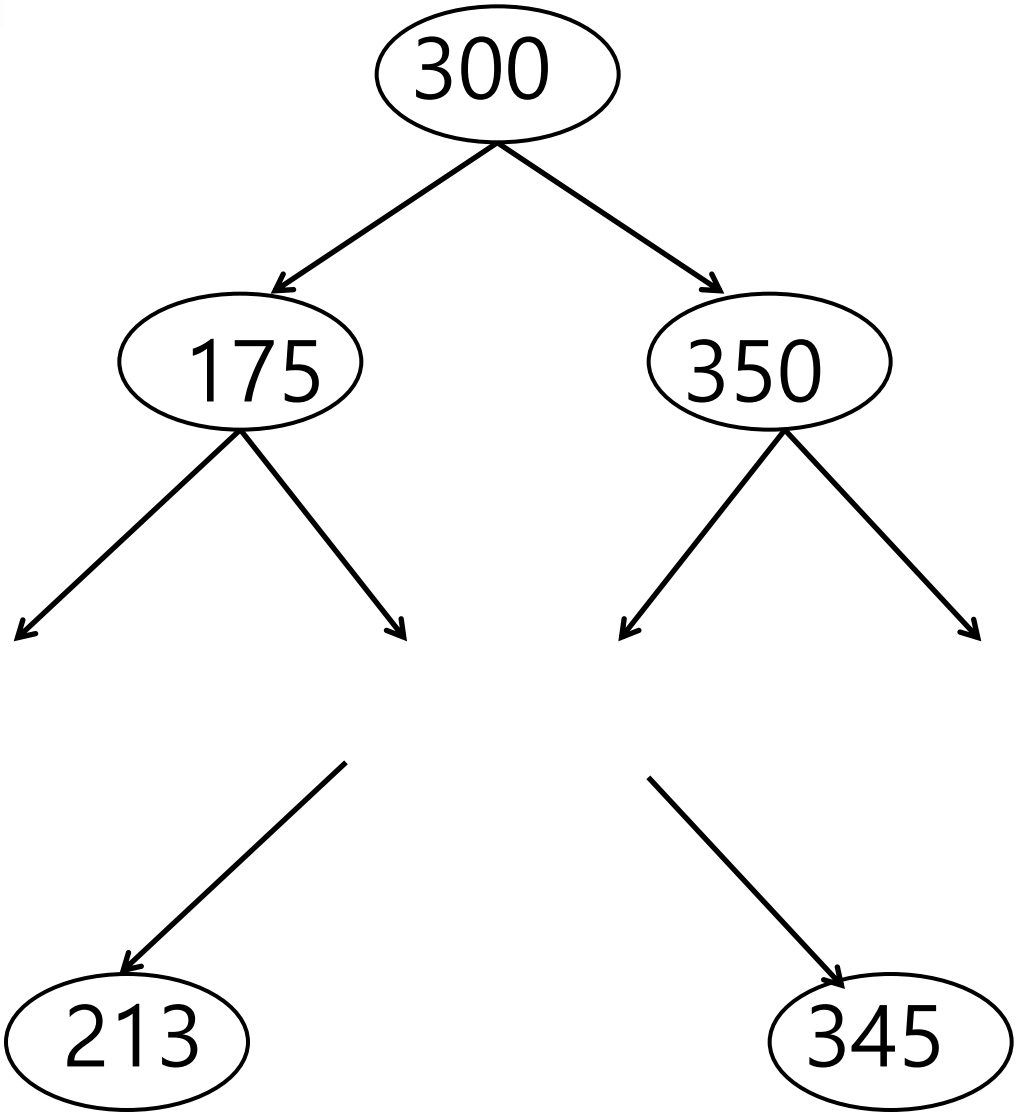
# Example search 120, search 400



# Caution (1/2): complex “key”, large number of nodes

- In Textbooks
  - “key”: a small number or short string
    - e.g., 250, “Hong Gil Dong”
  - “number of nodes”: 5-15
  - “number of levels”: 3-5
- In Real Software
  - “key”: a structure, array, array of structures
    - e.g., [xxxxxxxxx, “Hong Gil Dong”, 3xxxxx, “Suwon”]
  - “number of nodes”: 100, 1000, 10000, 1000000,…

# Caution (2/2): large number of levels




level 1

level 2

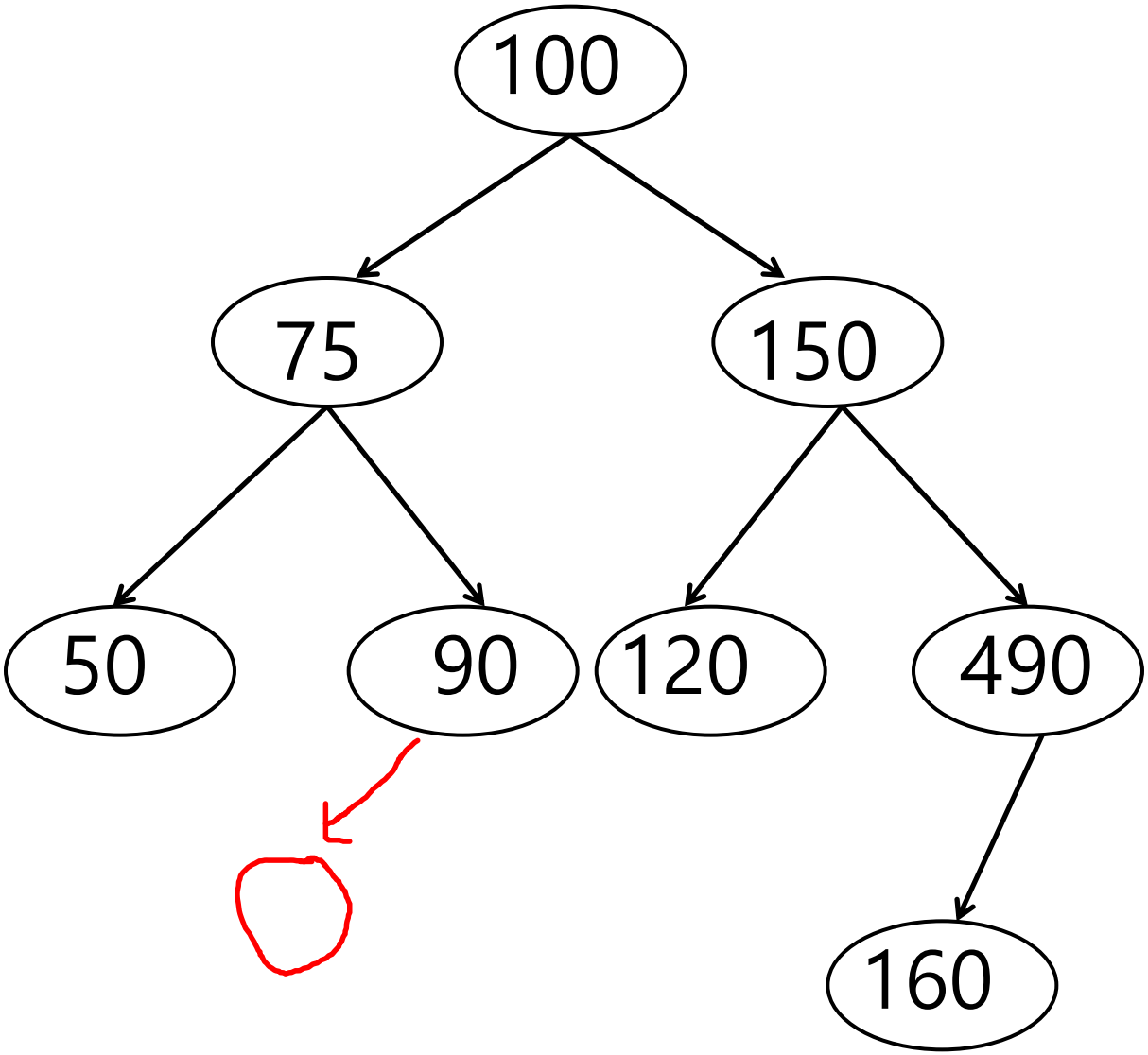
level 123

# Inserting into a Binary Search Tree

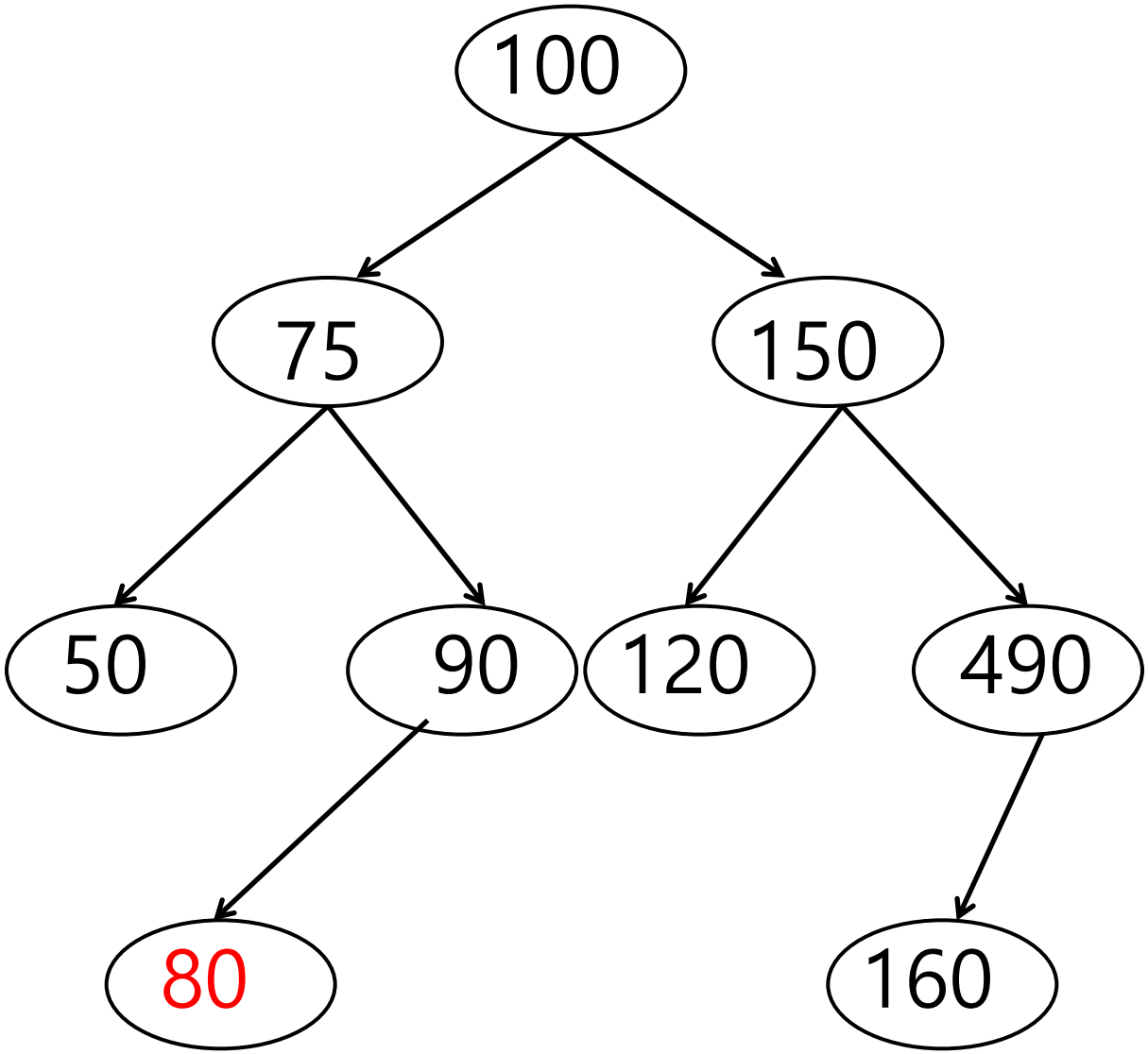
- Search the Key to Be Inserted.
- If the Search Fails, Insert the Key Where the Search Failed.

# Example

insert 80



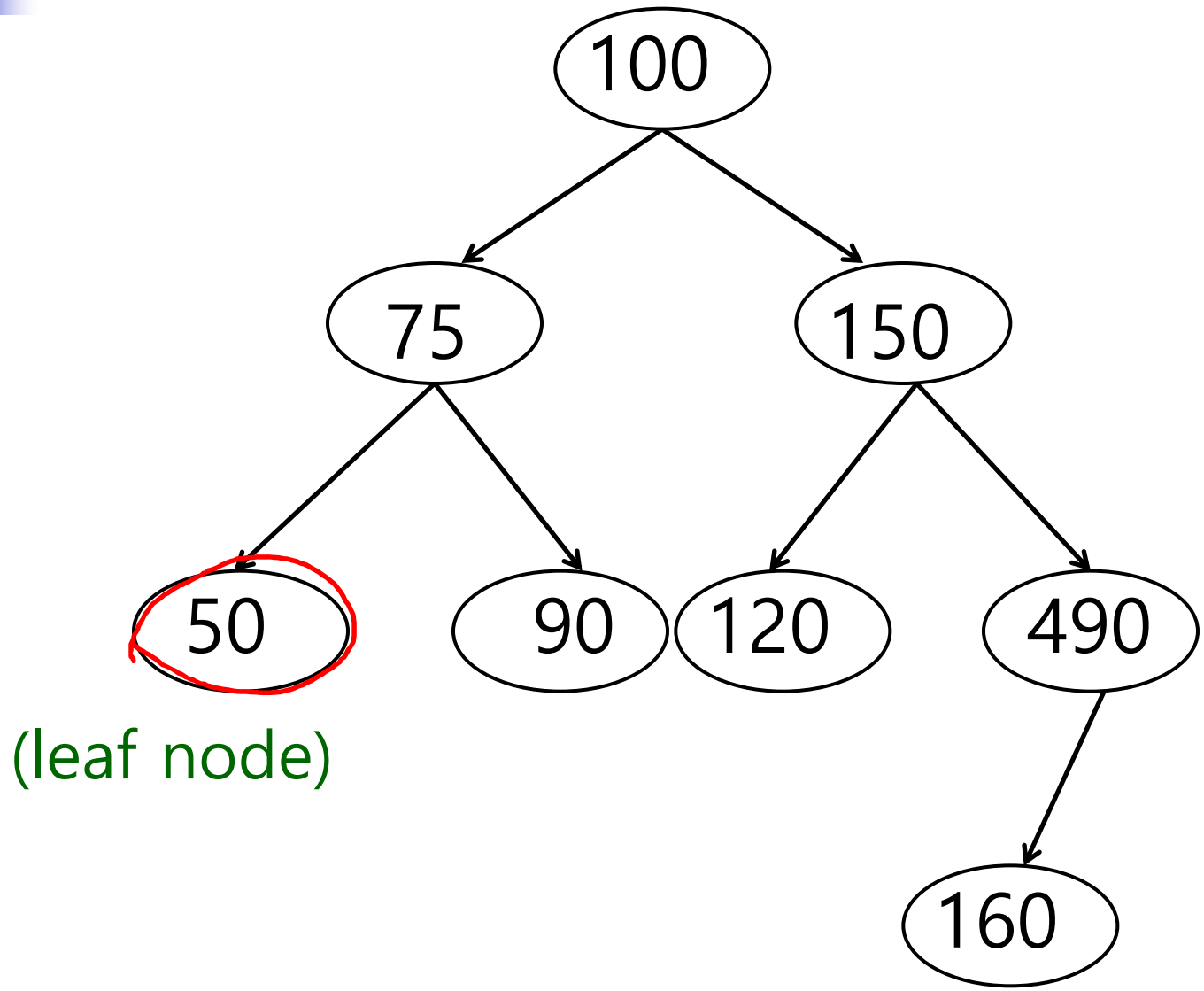
# Example: Result



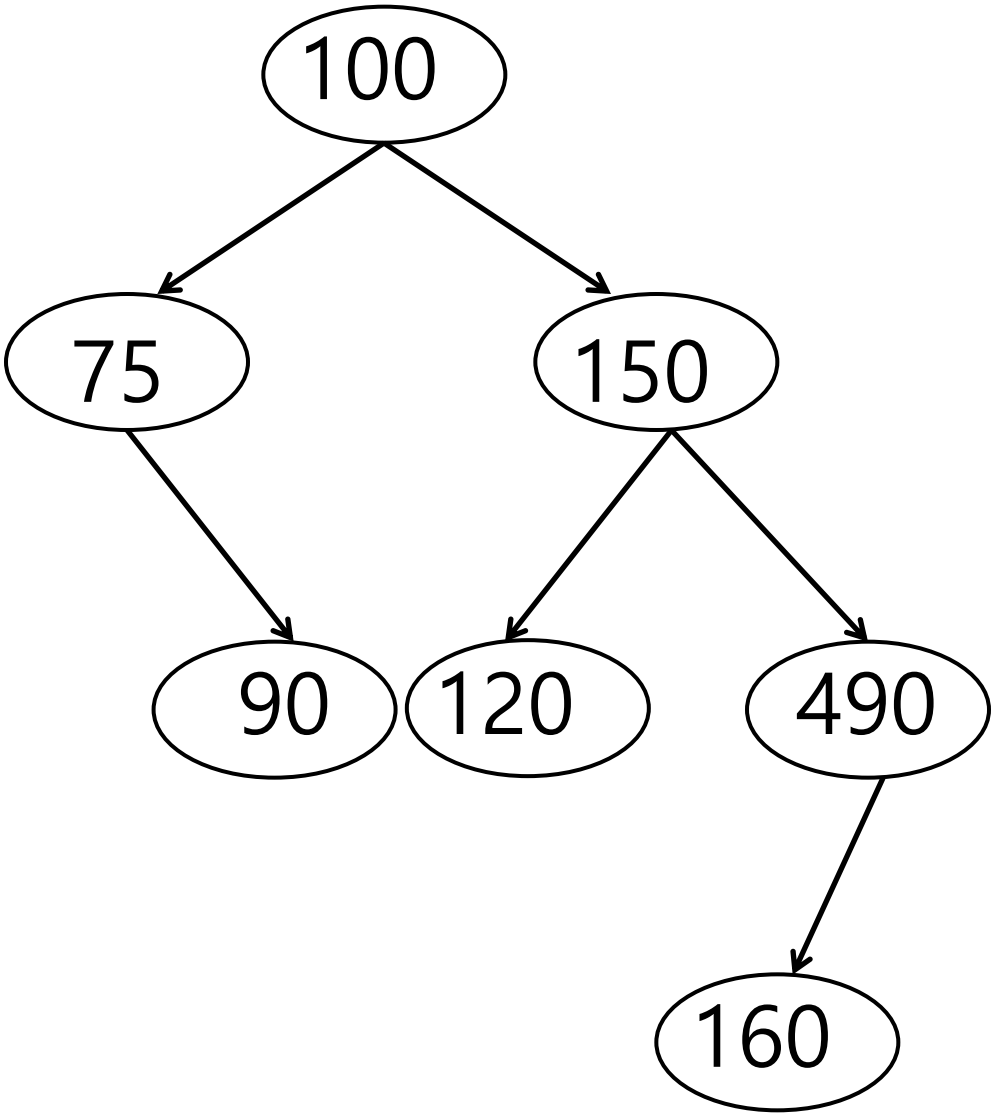
# Deleting from a Binary Search Tree

- Search the Key to Be Deleted.
- If the Search Fails, No Deletion.
- If the Search Succeeds
  - If the key is a leaf node, just delete it.
  - If the key is an interior node with one child node, delete the key, and simply promote the child node.
  - If the key is an interior node with two child nodes, delete the key, and
    - Promote the largest node in its left subtree or
    - Promote the smallest node in its right subtree.

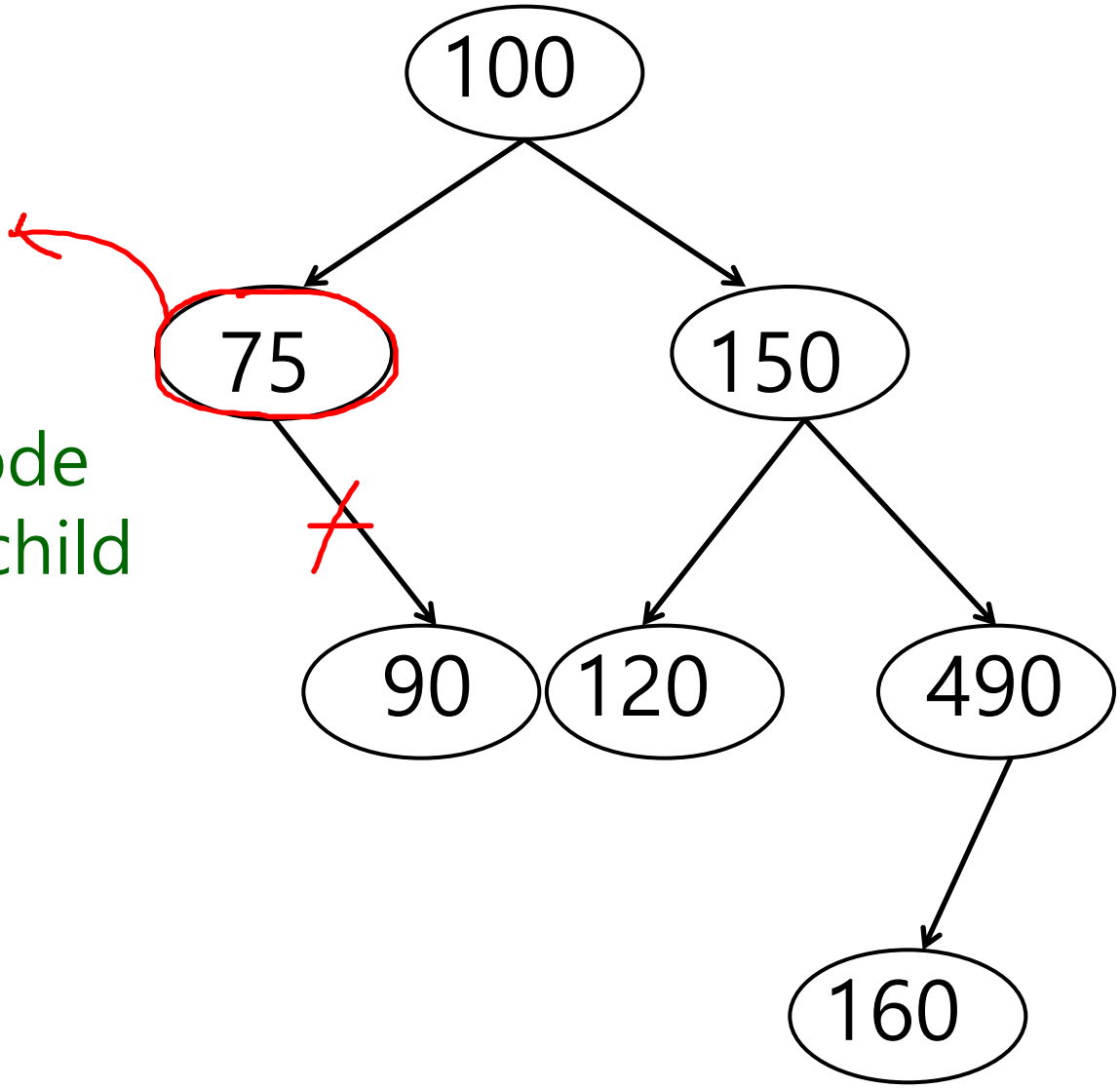
# Example (1/3) delete 50



# Example (1/3): Result

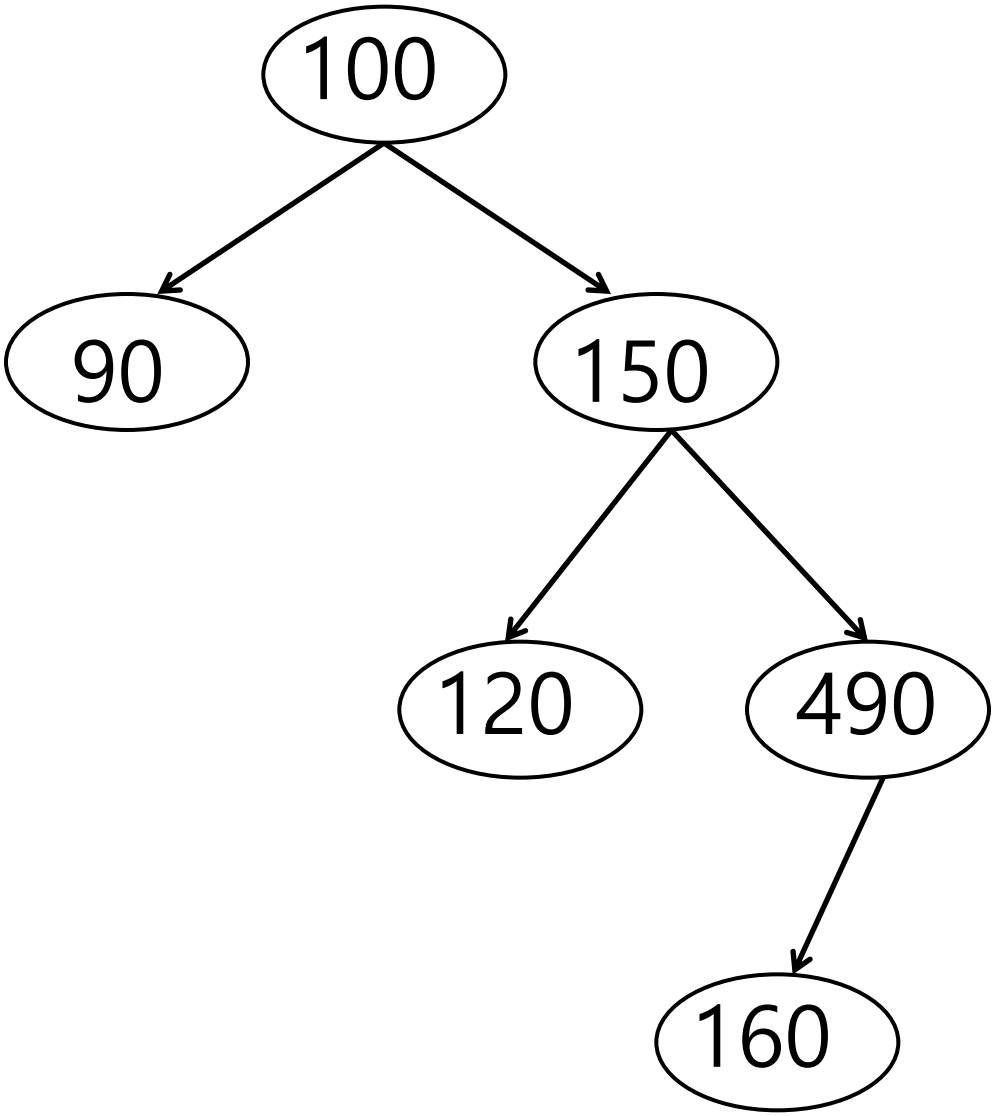


# Example (2/3) delete 75

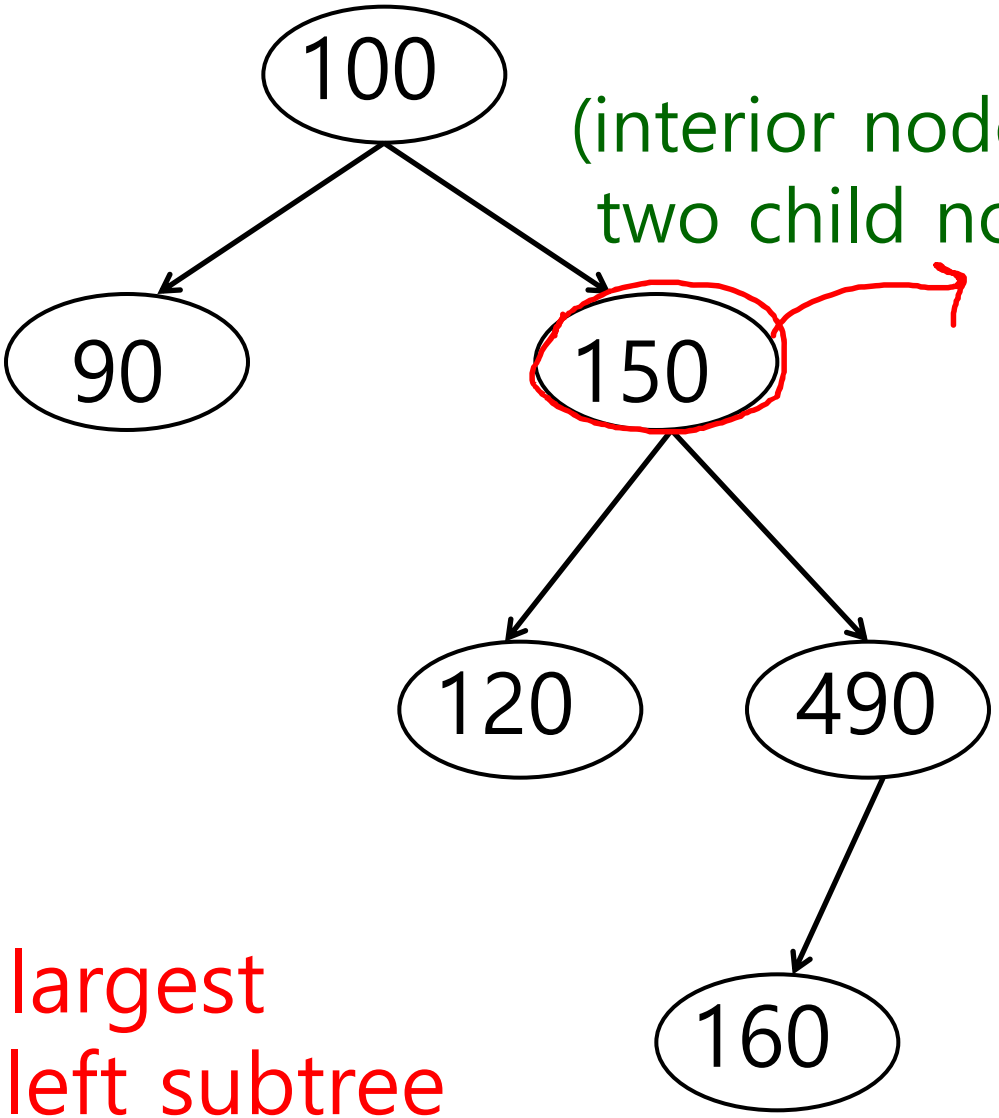



(interior node with one child node)

# Example (2/3): Result

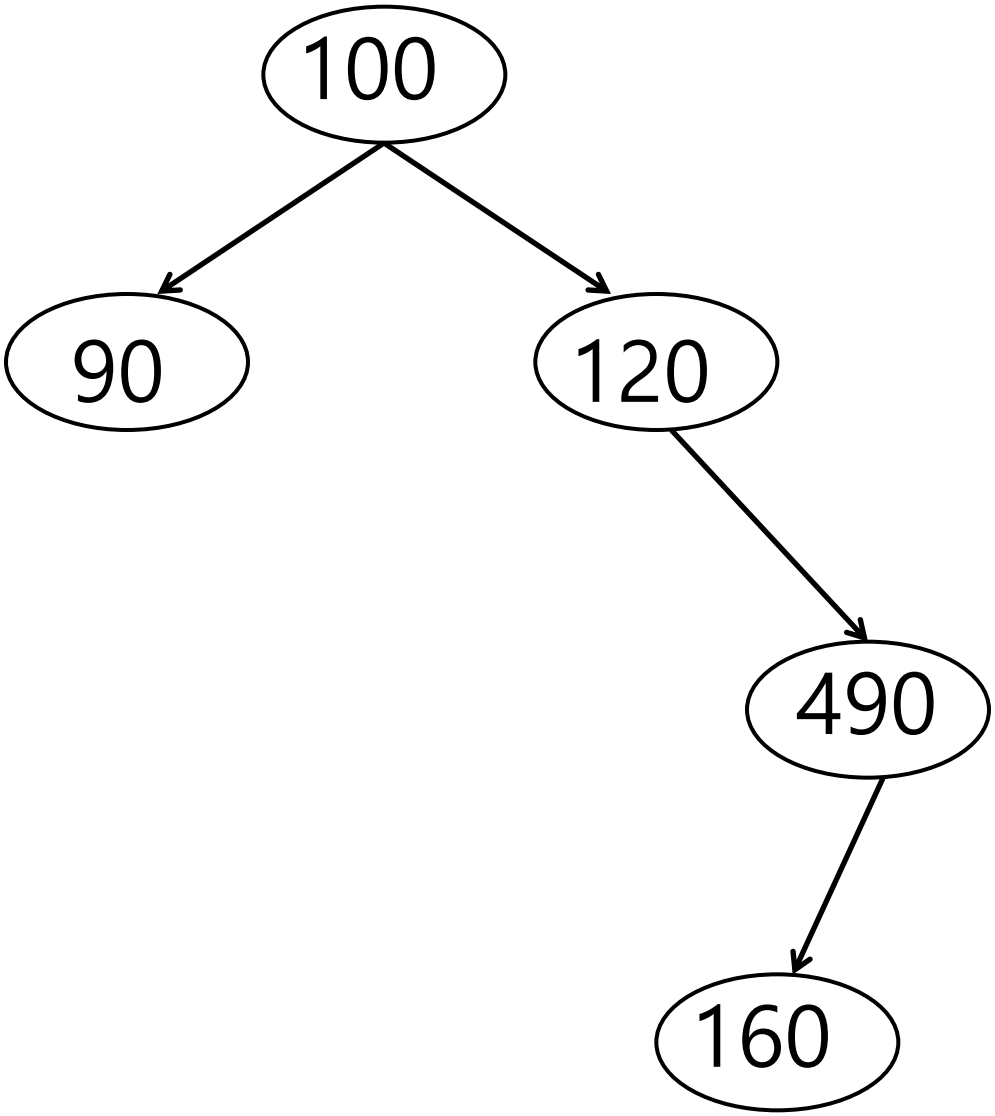


# Example (3/3) delete 150



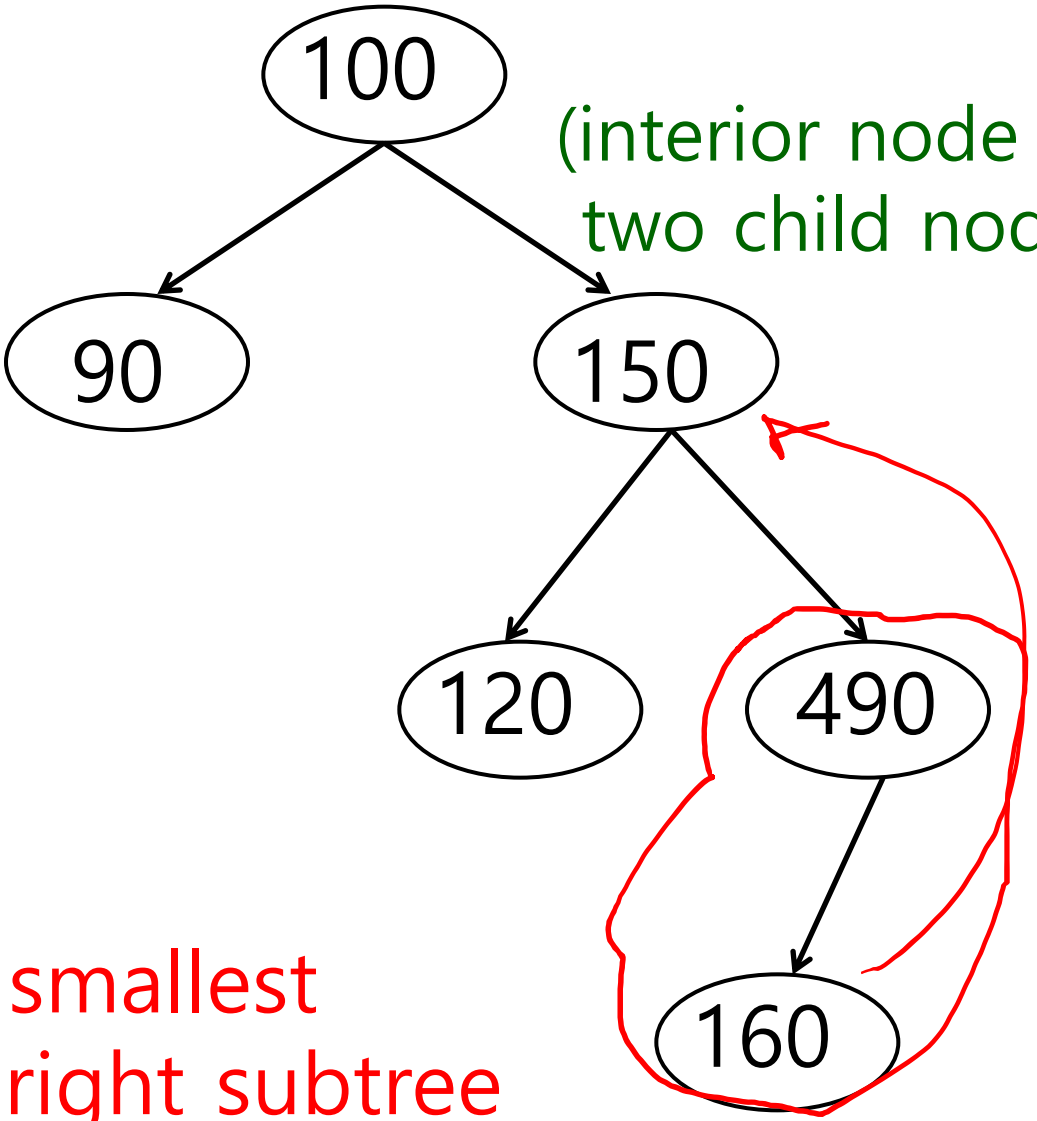

option 1:
promote the largest node on the left subtree

# Example (3/3): Result (1/2)



# Example (3/3)

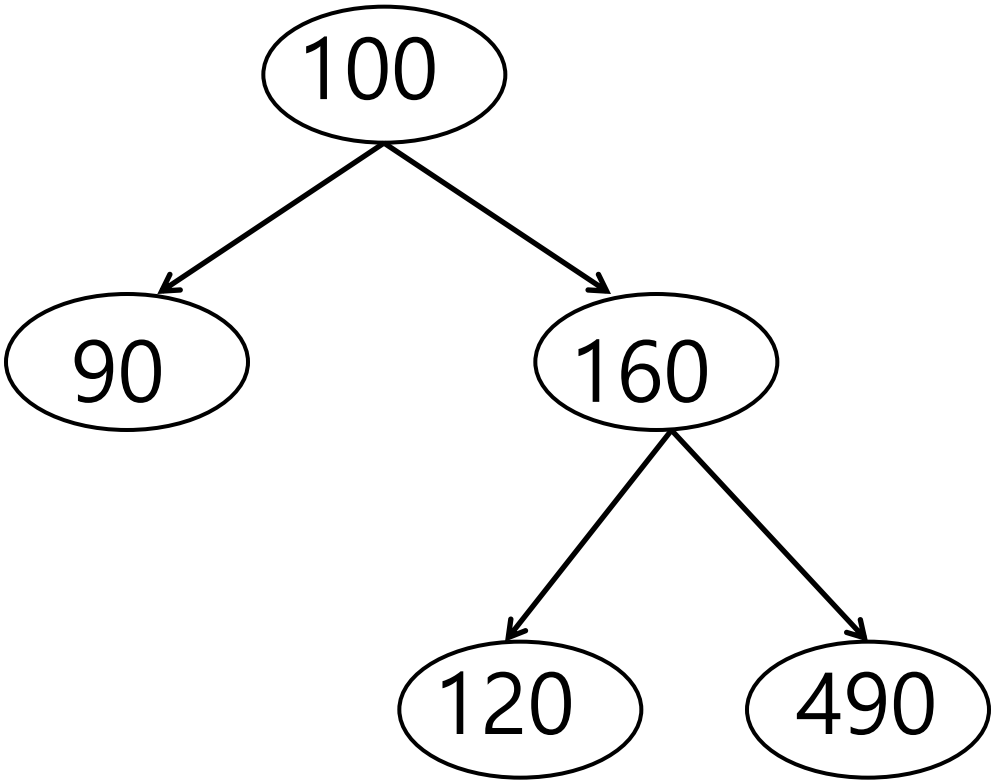
delete 150



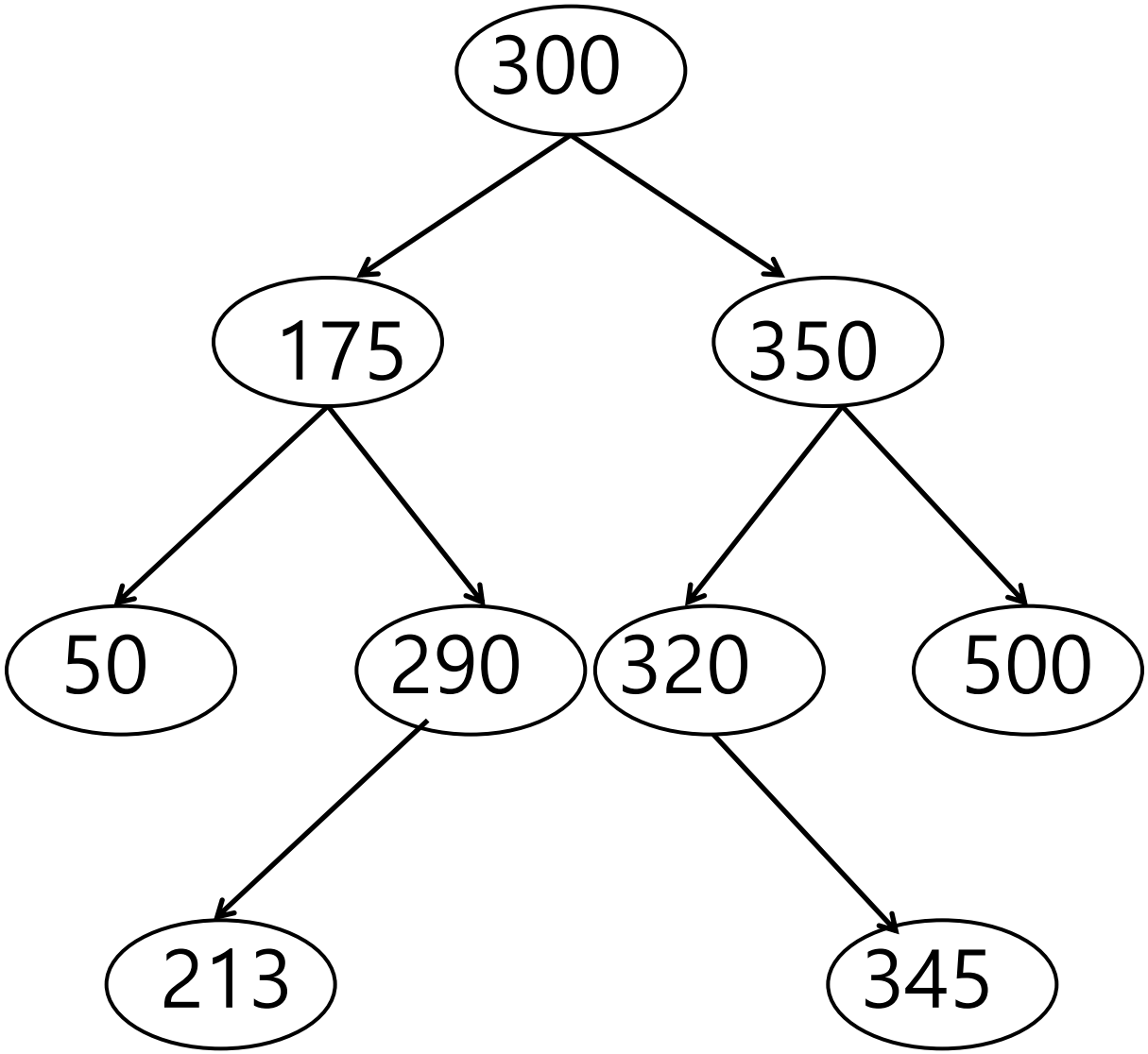

(interior node with two child nodes)

option 2:
promote the smallest node on the right subtree

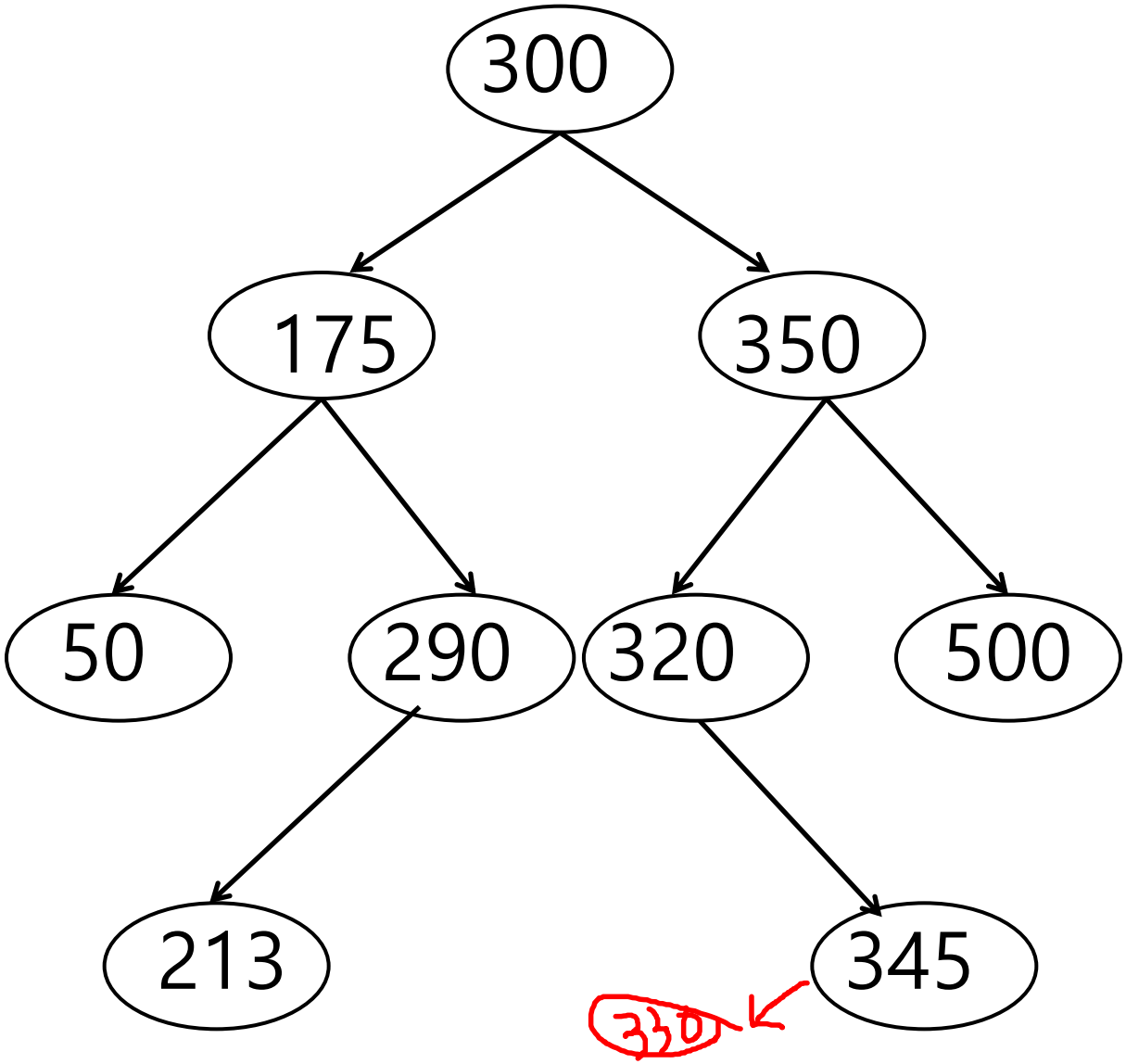
# Example (3/3): Result (2/2)



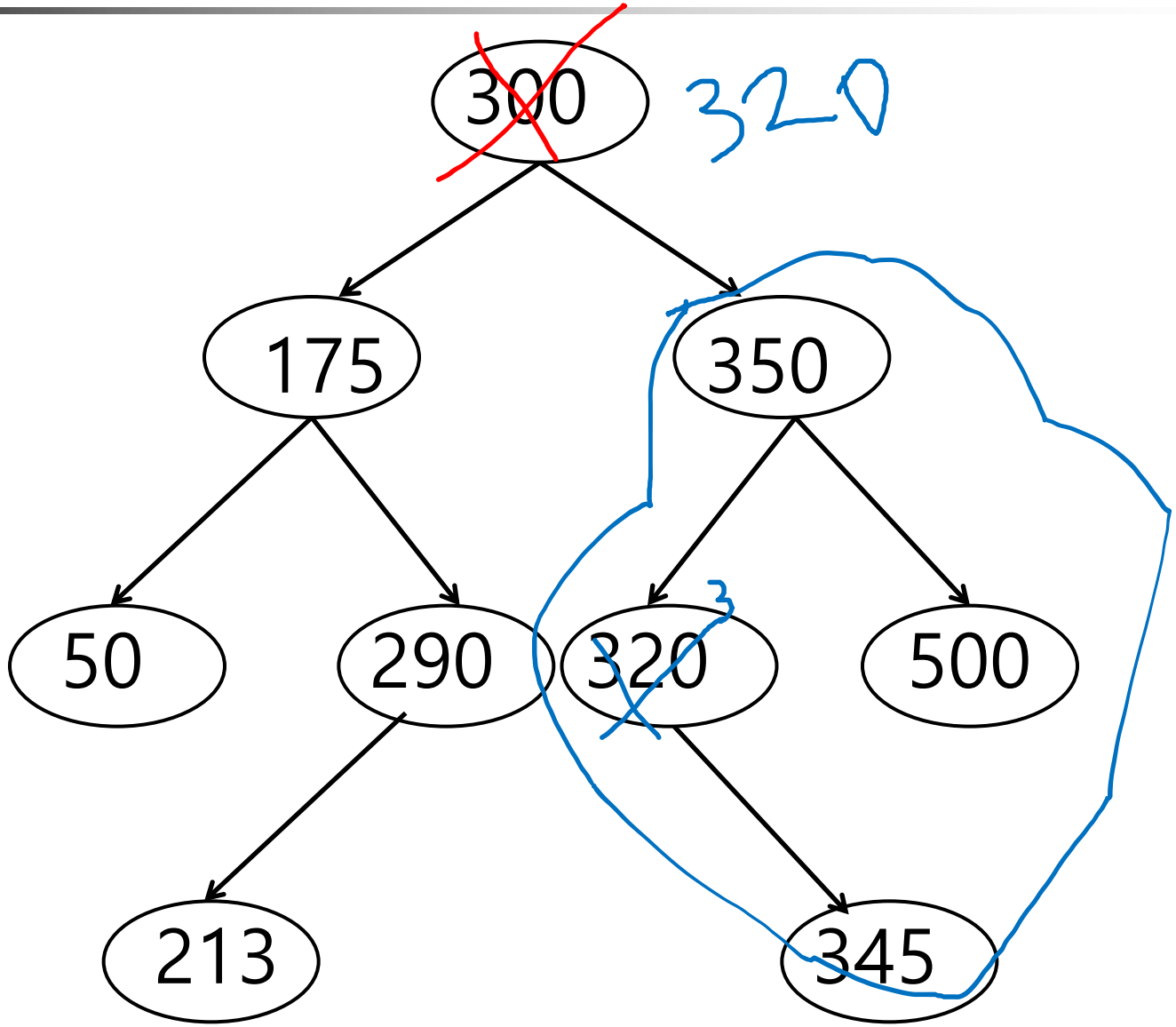
# Exercise: Search for 330 – At Which Node Does the Search Fail?



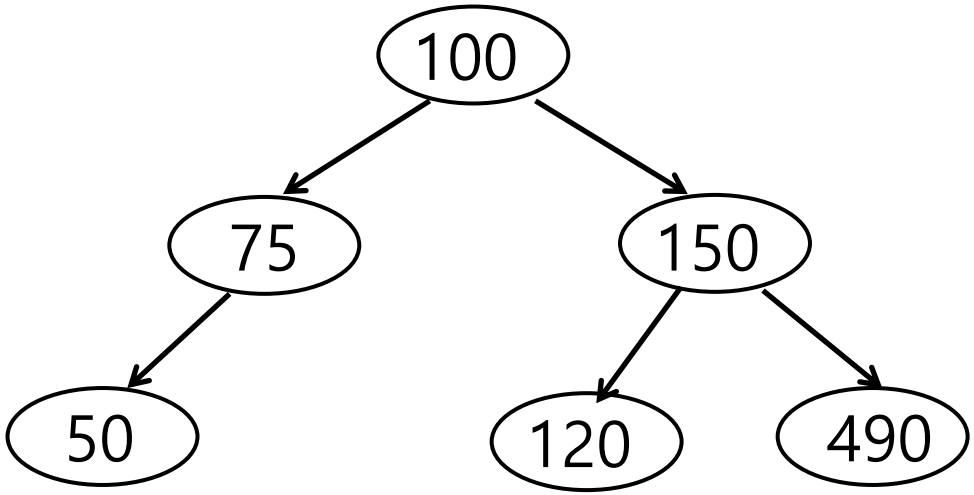
# Exercise: Insert 330



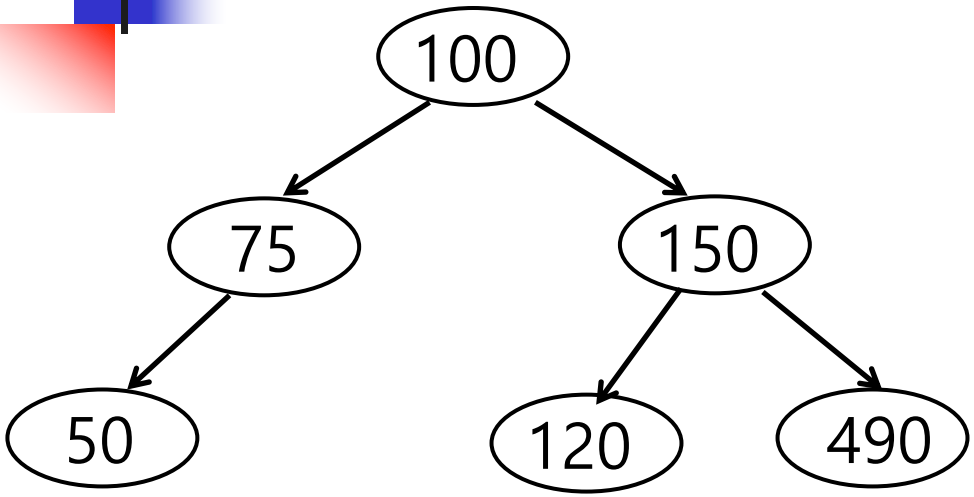
# Exercise: Delete 300



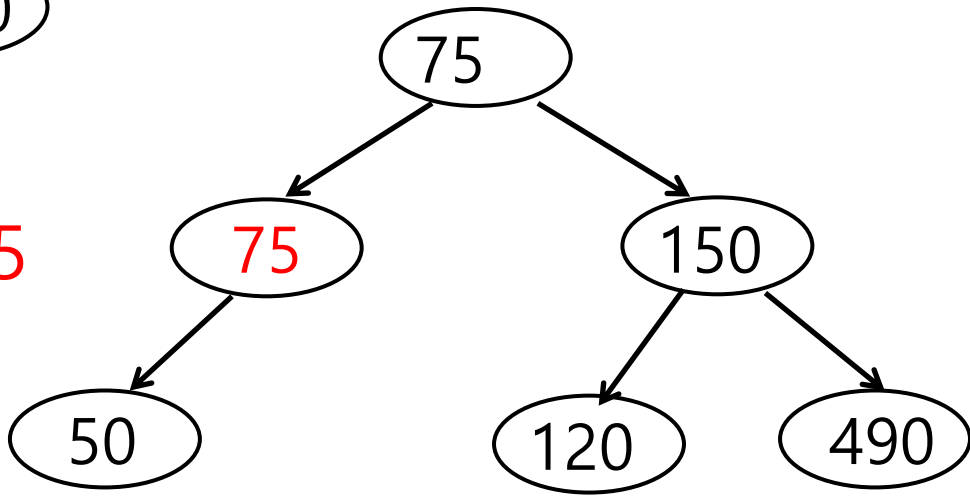
# Exercise: Delete 100



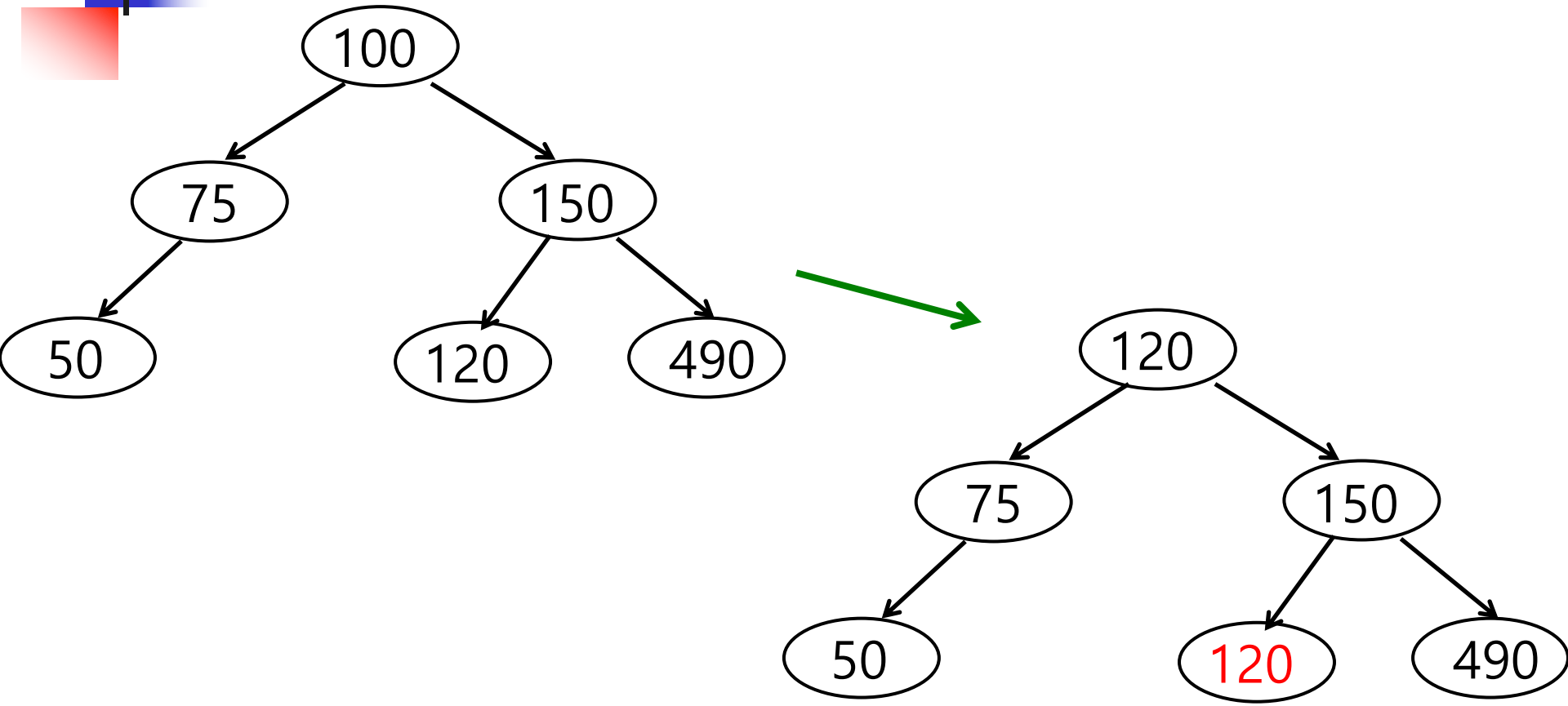
# Solution option 1: replace the key with the largest key on the left subtree




delete 75



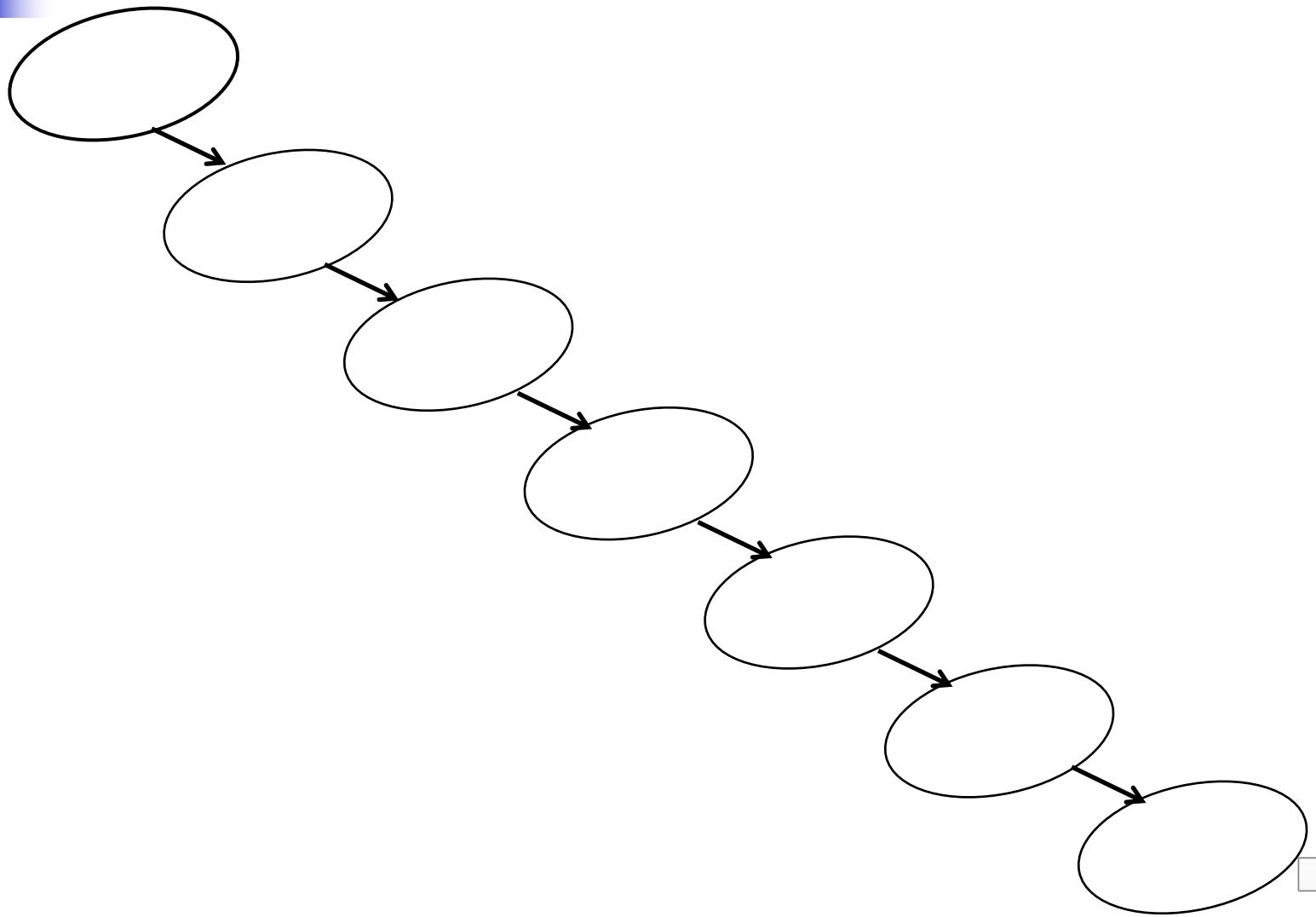
# Solution option 2: replace the key with the smallest key on the right subtree



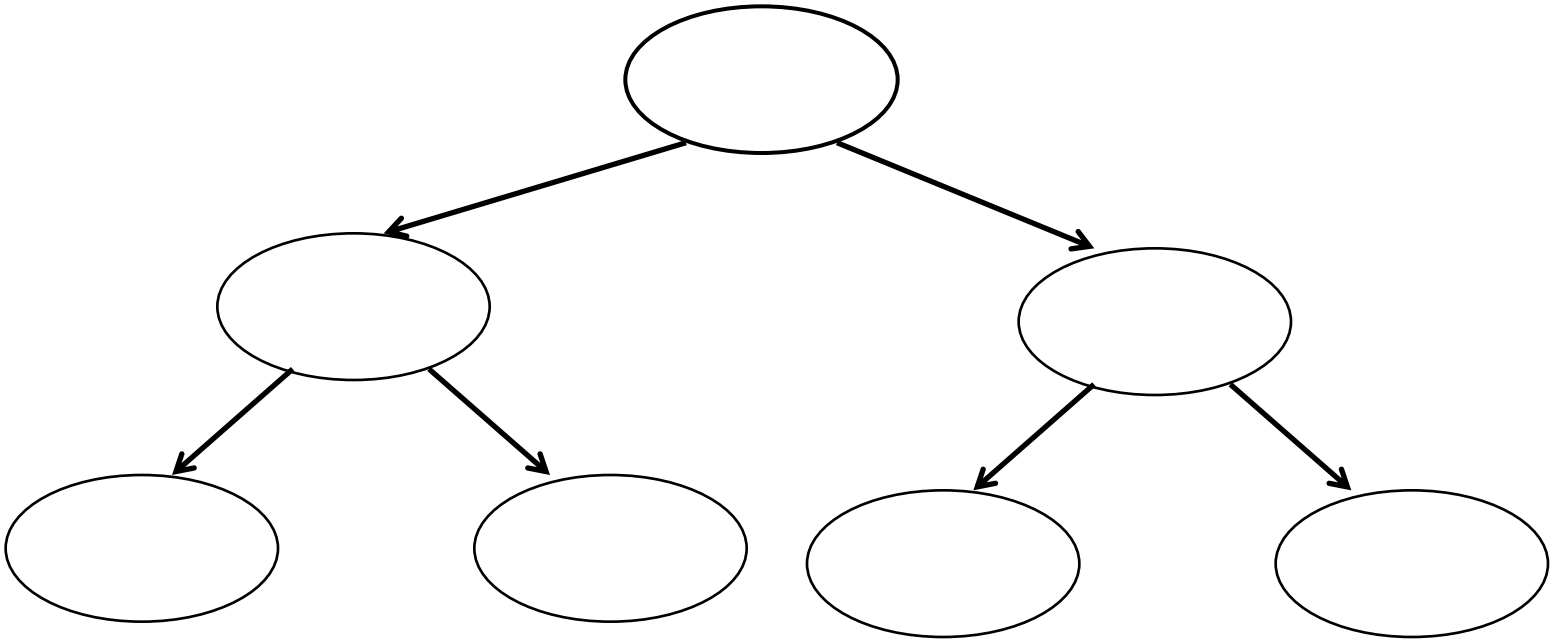
# Performance of Binary Search Trees

- Search Time
- Insertion Time
- Deletion Time

# Degenerate Binary Tree



# A Full or Complete Binary Tree



# Big O (Big Oh, Landau, Asymptotic) Notation (1/3)

- (e.g.) $T(n) = 2n^2 + 4n + 120$
  - As n grows, the $n^2$ term dominates
  - $T(n) = O(n^2)$
- "Big O of $n^2$" = $O(n^2)$
  - "Order of $n^2$" time complexity"
- Think of "Big" n (1,000; 1,000,000;…)

# Big O (Big Oh, Landau, Asymptotic) Notation (2/3)

- O(1) : (constant and small) finding a match right away in a search, deletion of the largest key from a max heap
- $O(\log_2 n)$: (logarithmic) search of a height-balanced binary search tree, binary search of a sorted list
- O(n) : search of a binary search tree
- O(n) : (linear) search of an array, queue, linked list

# Big O (Big Oh, Landau, Asymptotic) Notation (3/3)

- $O(n \log_2 n)$ : (loglinear) heap sort
- $O(n^2)$ : (quadratic) insertion sort

- $O(n!)$ : (factorial) traveling salesman problem
- $2^{O(n)}$ : (exponential) general traveling salesman problem

# Source of the Performance Problem of Binary Search Trees

- Height of the Binary Search Tree
  - The average case: $\log_2 n = h$, where n is the total number of nodes
  - The worst case: n
- Need to Maintain the Search Tree Height-Balanced.
  - AVL tree, red-black tree, AA tree,…
  - Search, insertion, deletion each takes $O(\log_2 n)$
  - Overhead for insertion and deletion

# A Height-Balanced Tree

# Height-Balanced Tree

- AVL Tree, (Red-Black Tree)
- (2-3 Tree, T-Tree, B-Tree, B+ Tree)

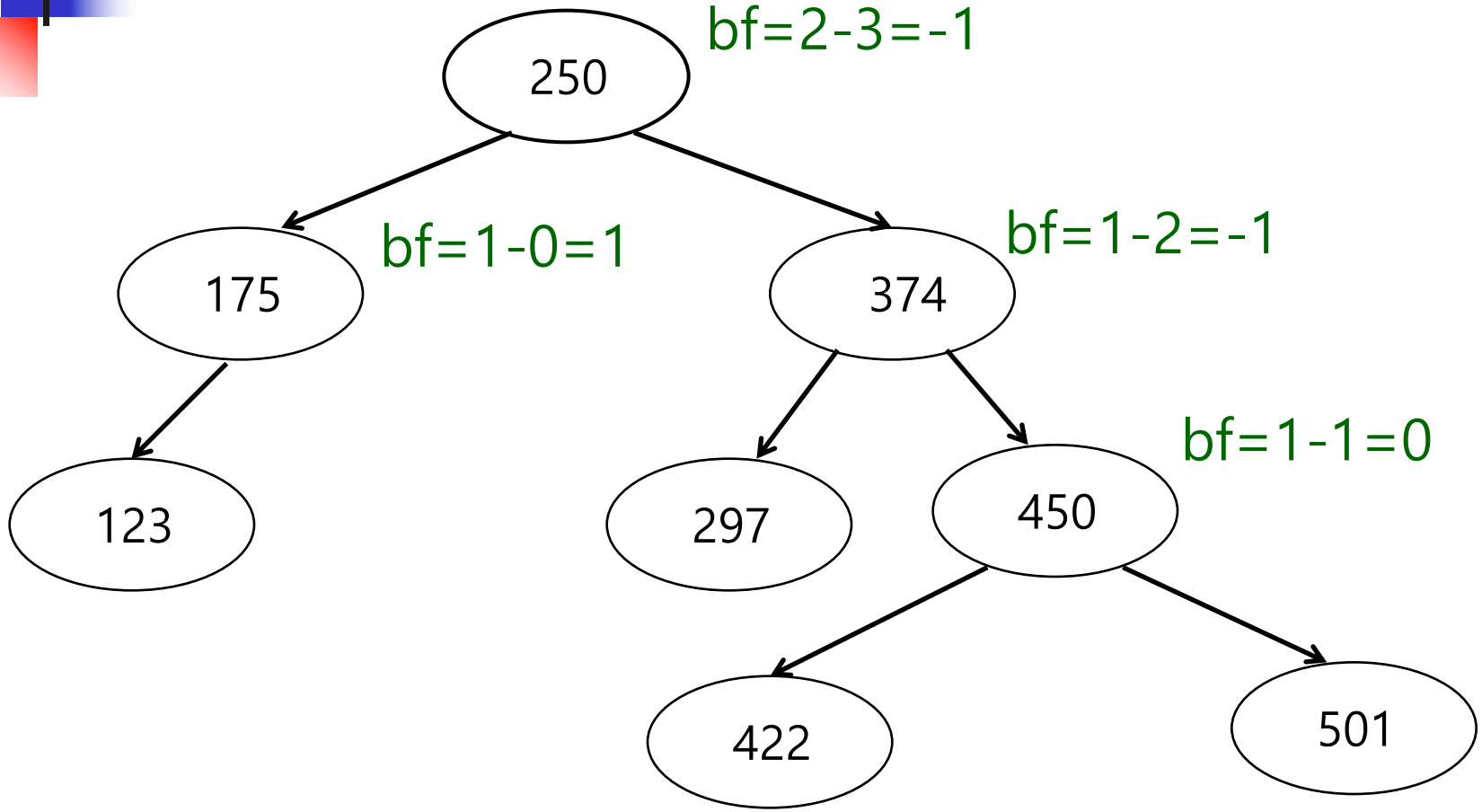
# Reading

- http://sky.fit.qut.edu.au/~maire/avl/System/AVLTree.html
- http://cis.stvincent.edu/html/tutorials/swd/avltrees/avltrees.html

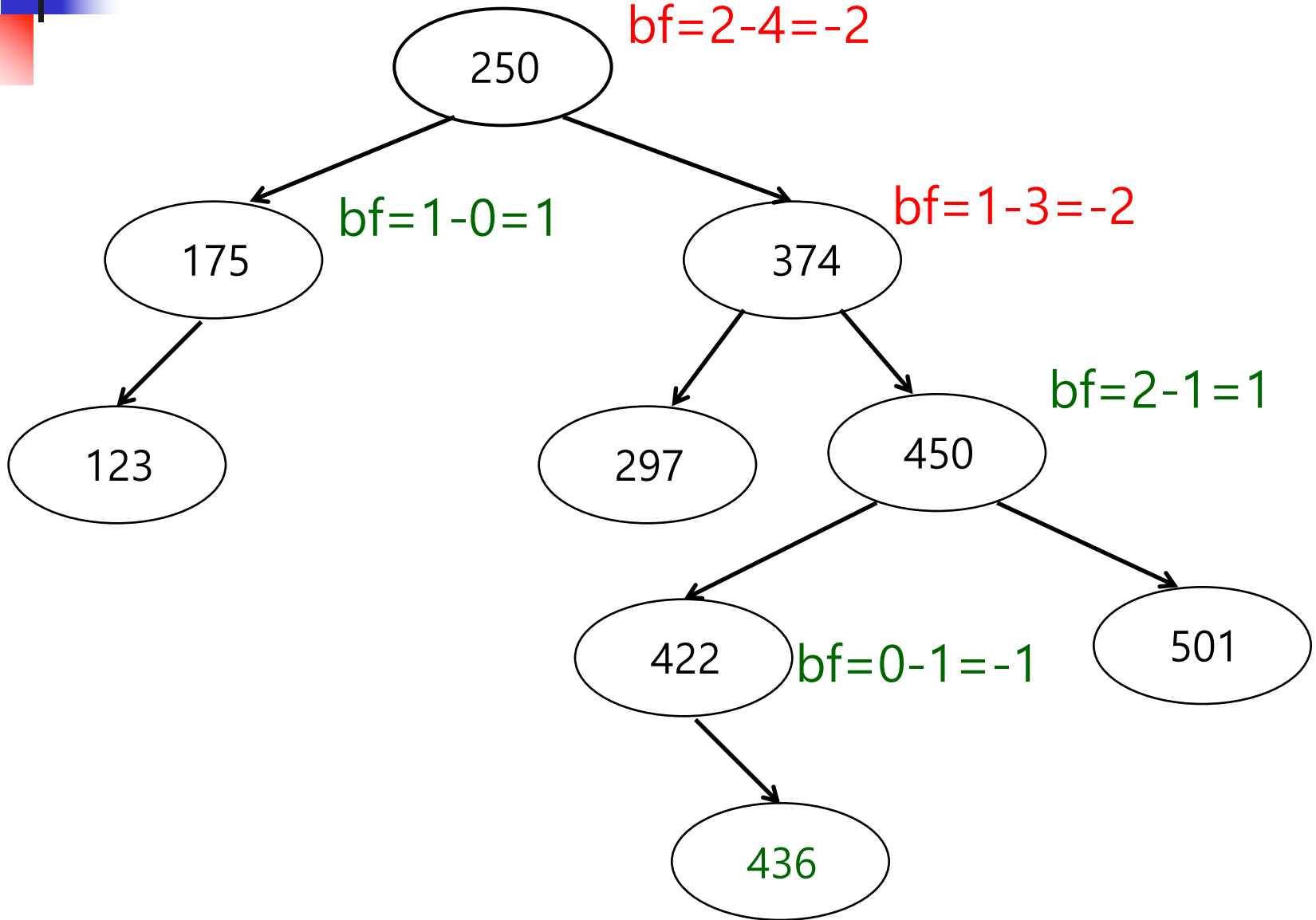
# AVL Tree

- Invented by G.M. Adelson-Velskii and E.M. Landis (1962)
- A Height-Balanced Binary Search Tree
  - "not perfectly balanced"
- Balance Factor of Each Node Must Be 0, +1, or -1.
  - (Balance Factor = $h^L - h^R$, where $h^L$ is the height of left subtree and $h^R$ is the height of the right subtree.)

# An AVL Tree



# Not an AVL Tree



# AVL Tree Rotations

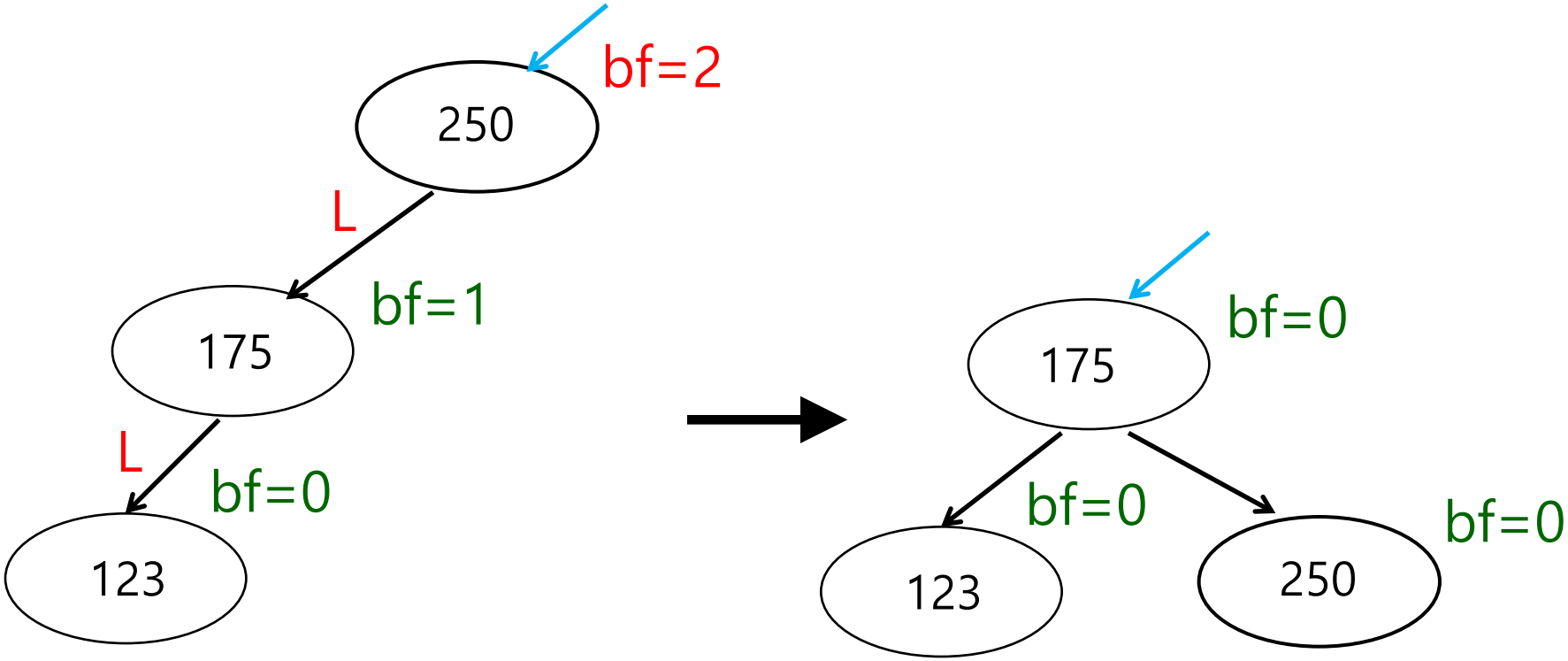
- If the Balance Factor of a Node Becomes 2 or -2, Upon Insertion or Deletion of a Node
  - The tree must be re-balanced by performing “tree rotations” around the node whose balance factor becomes 2 or -2.
- Two Objectives
  - Maintain AVL tree height balanced.
  - Maintain AVL tree as a binary search tree.

# AVL Tree Rotations

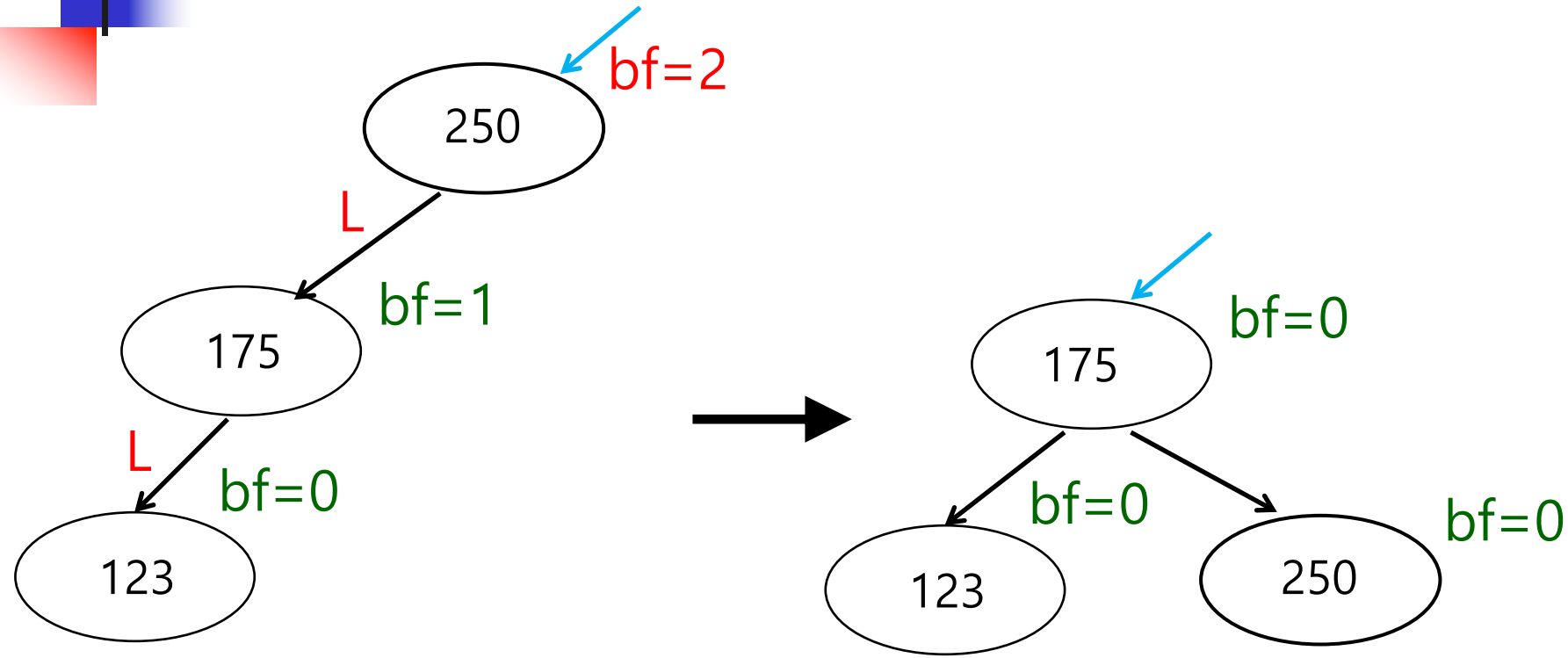
- 4 Types of Rotations
  - form a chain of 3 nodes down from the node for a single rotation (right or left), or
  - form a chain of 3 or 4 nodes down from the node for a double rotation (right and left, or left and right)

# AVL Tree Rotations (1/4)

LL: Single Rotation (Right)



# Implementing the LL Rotation

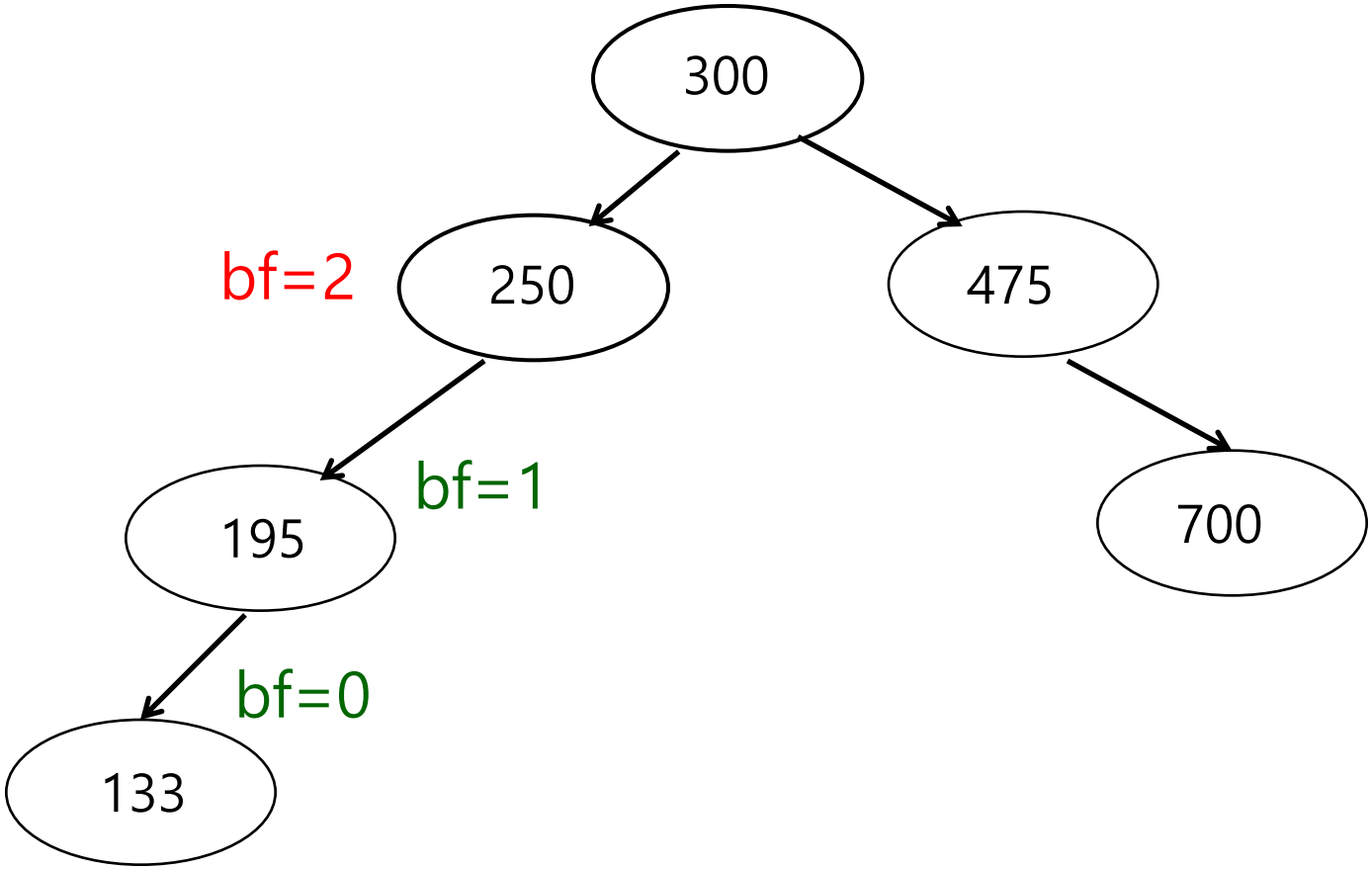



change node 175's right child pointer from NULL to node 250

change the left child pointer of the parent of node 250
from node 250 to node 175

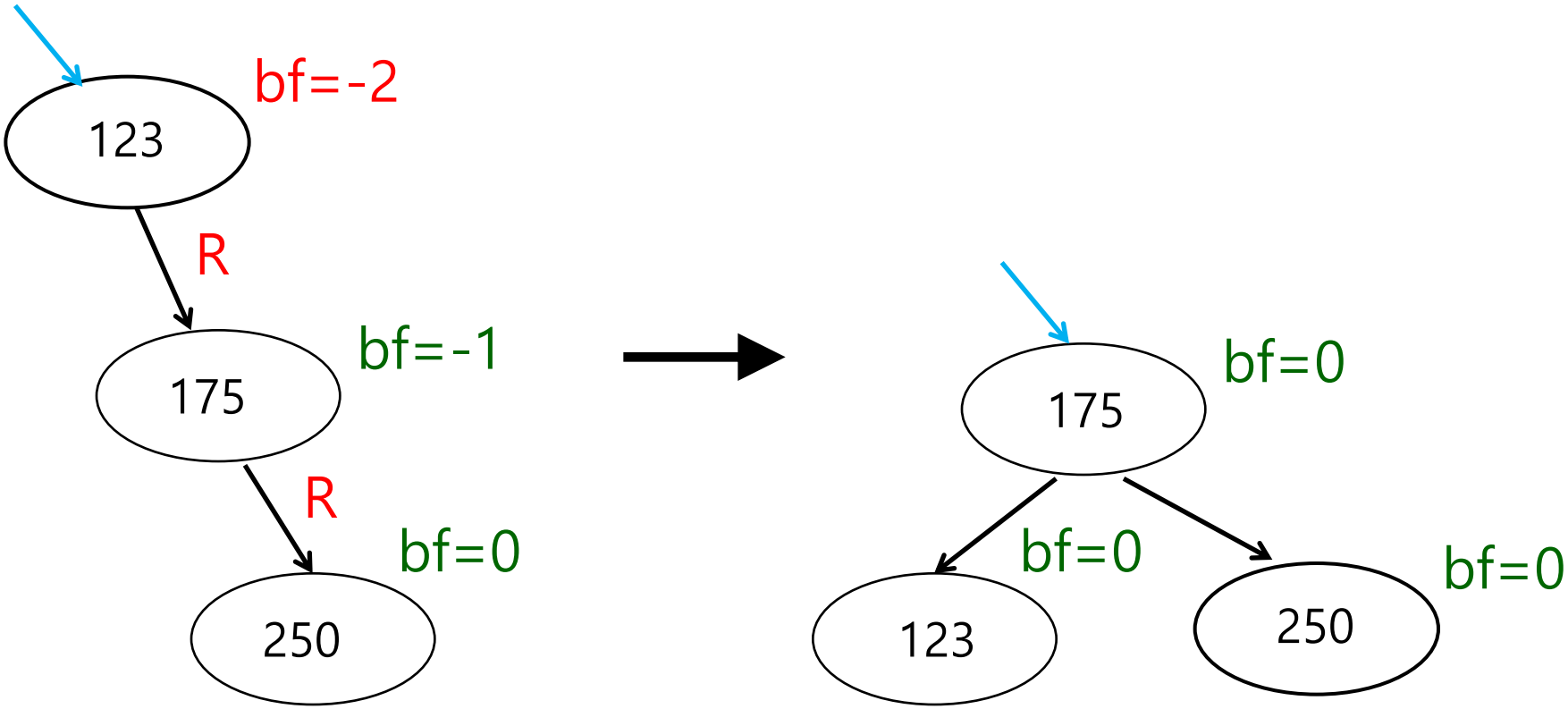
change the balance factor in the affected nodes

# Exercise: Re-Balance the AVL Tree

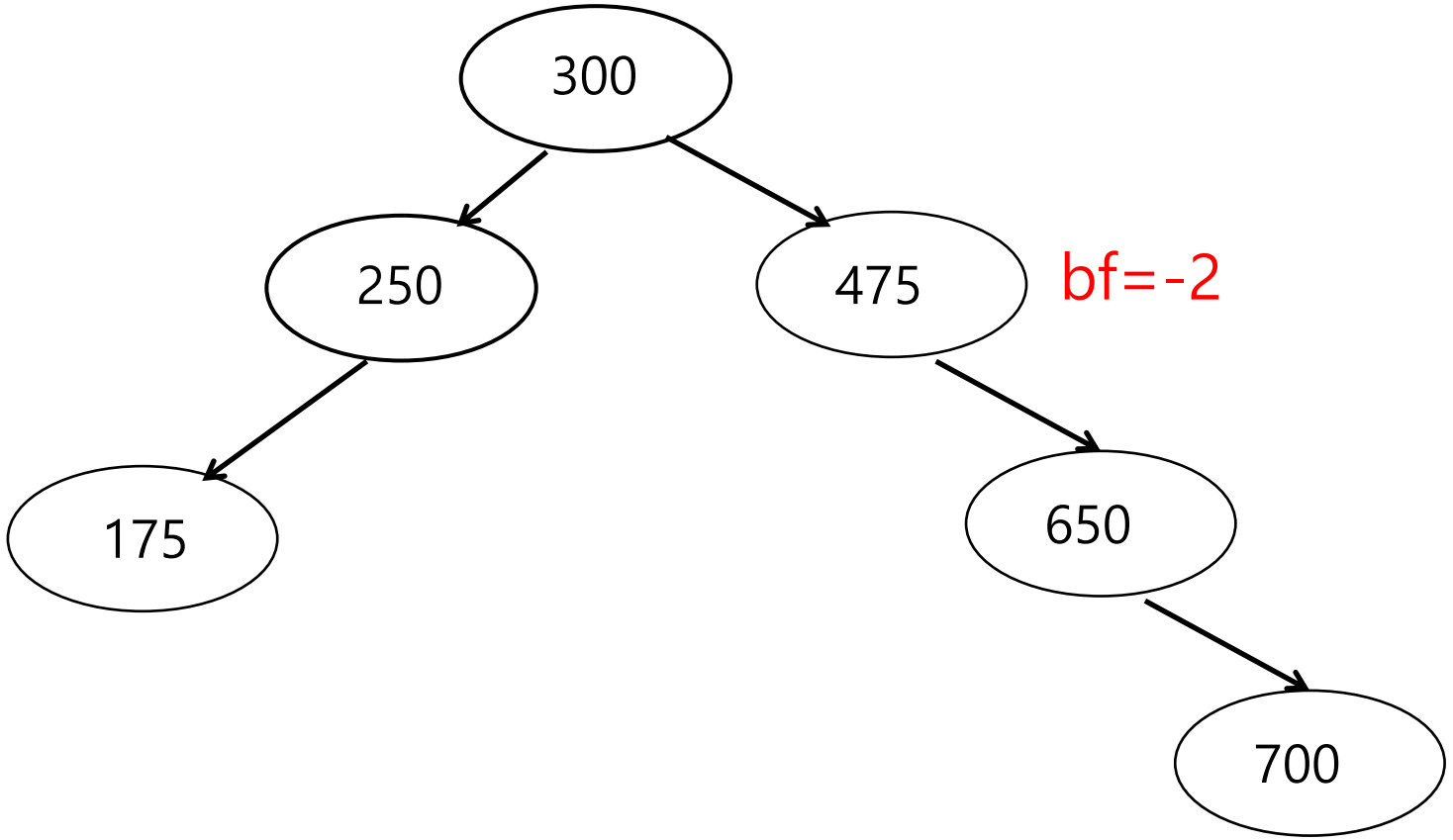


# AVL Tree Rotations (2/4)

RR: Single Rotation (Left)

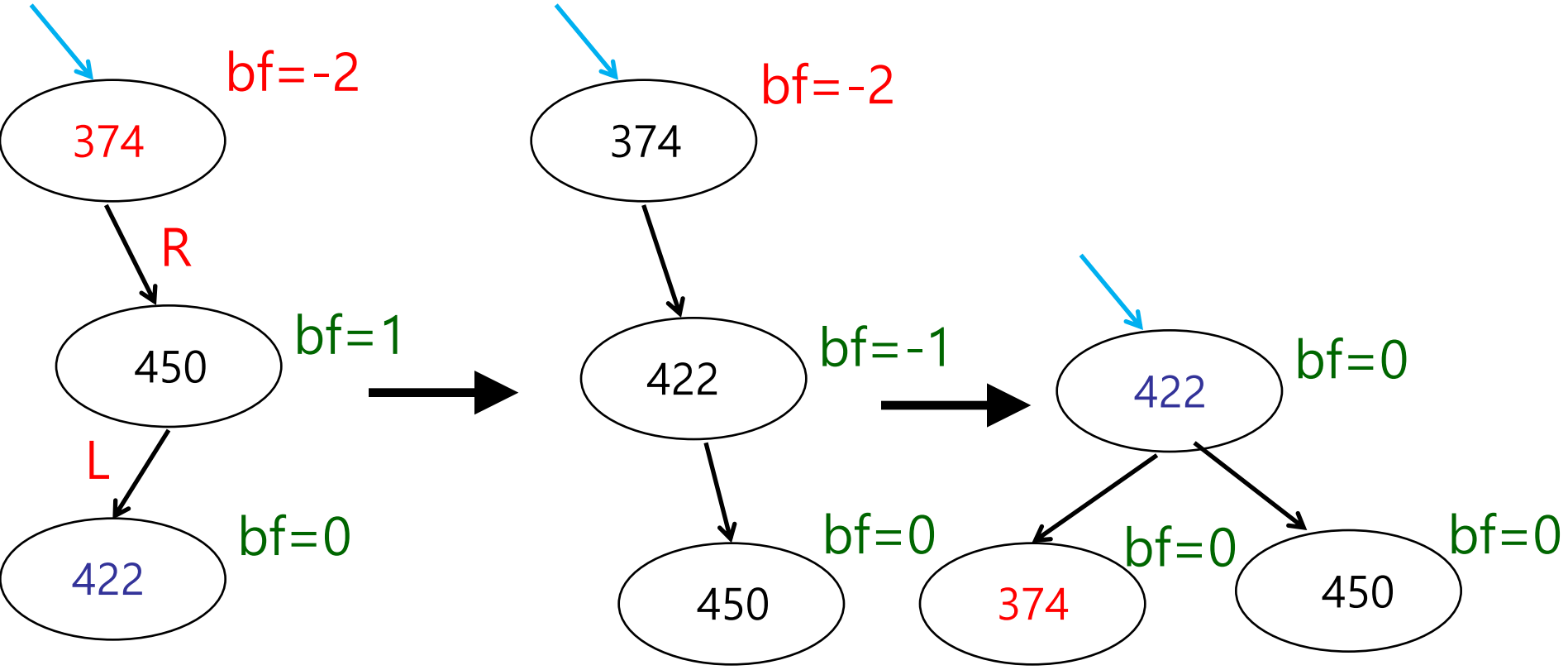


# Exercise: Re-Balance the AVL Tree

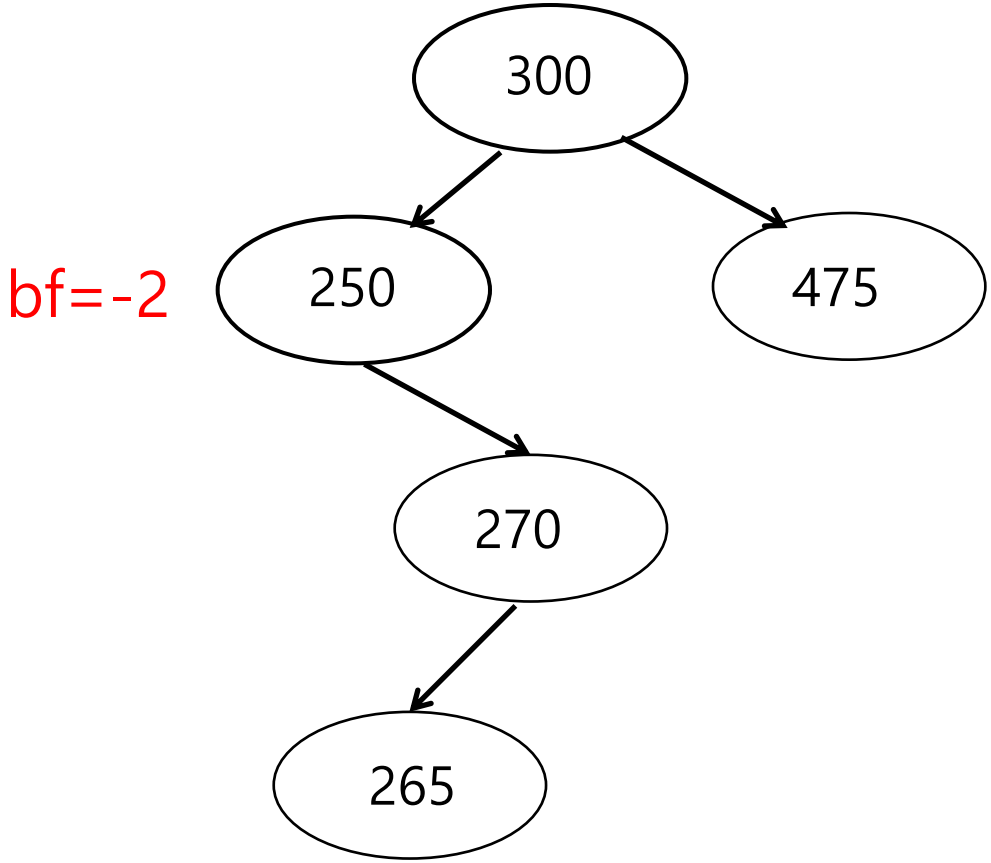


# AVL Tree Rotations (3/4)

RL: Double Rotation (Right and Left)

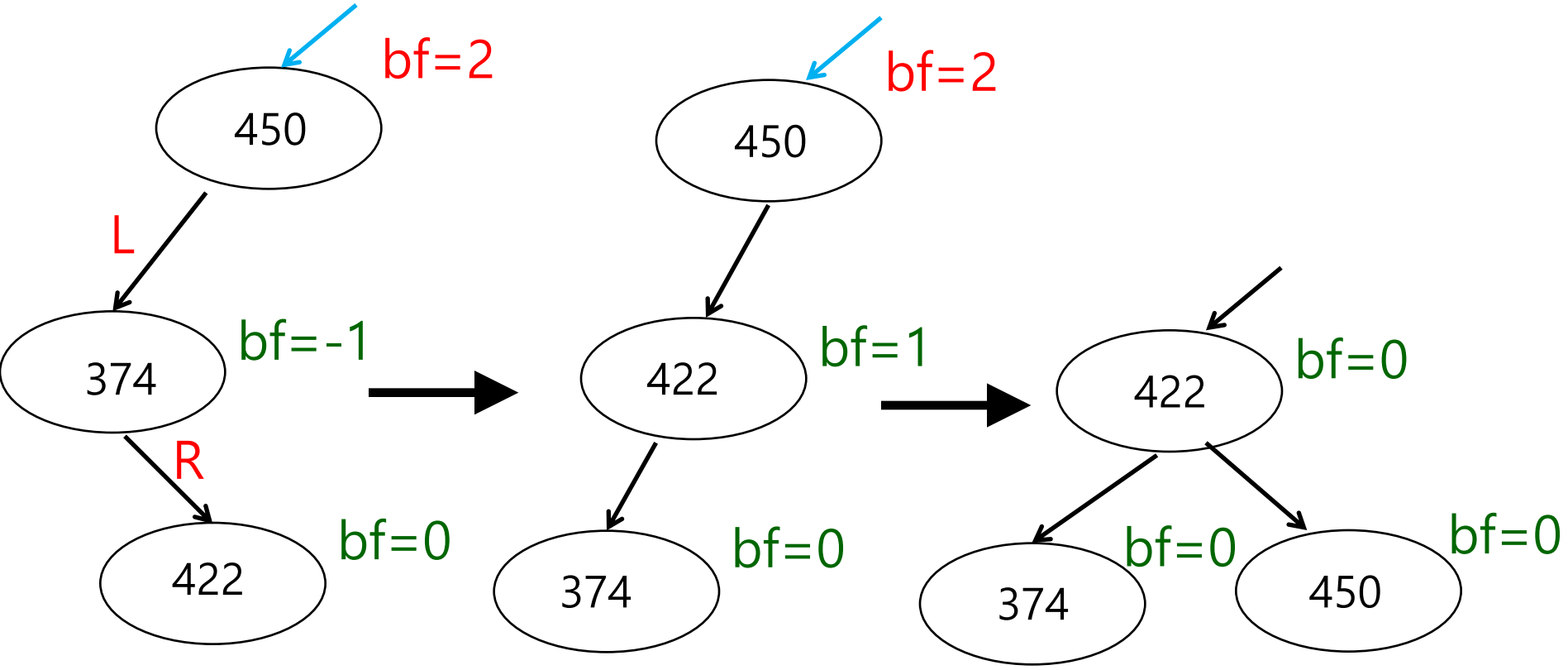


# Exercise: Re-Balance the AVL Tree

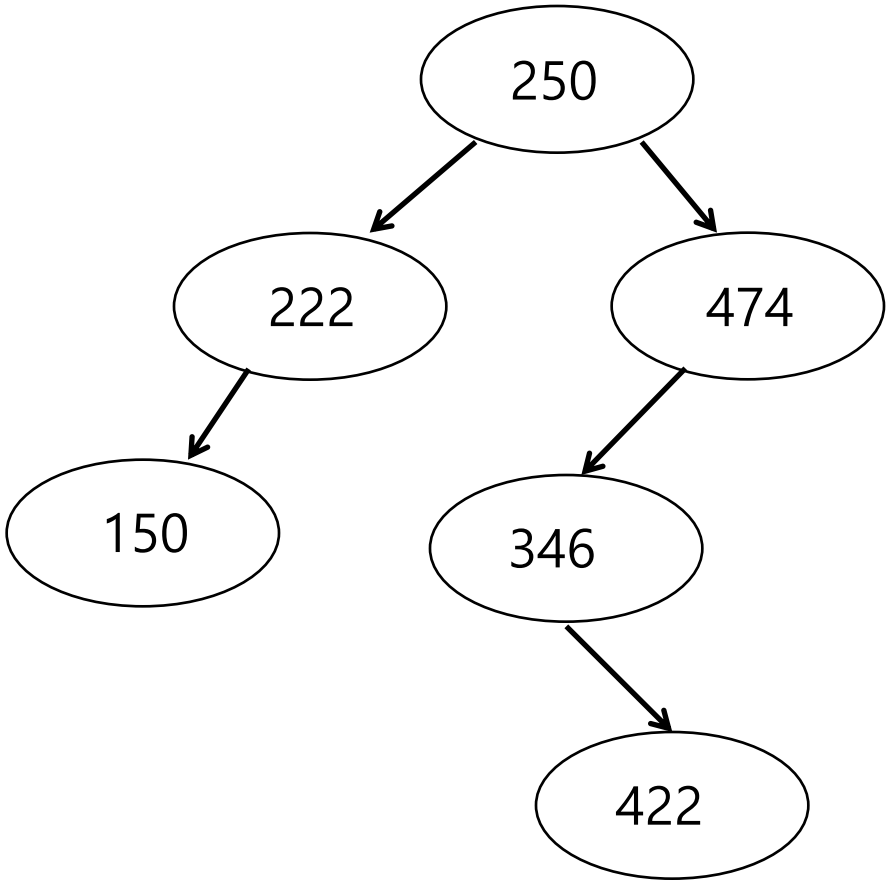


# AVL Tree Rotations (4/4)

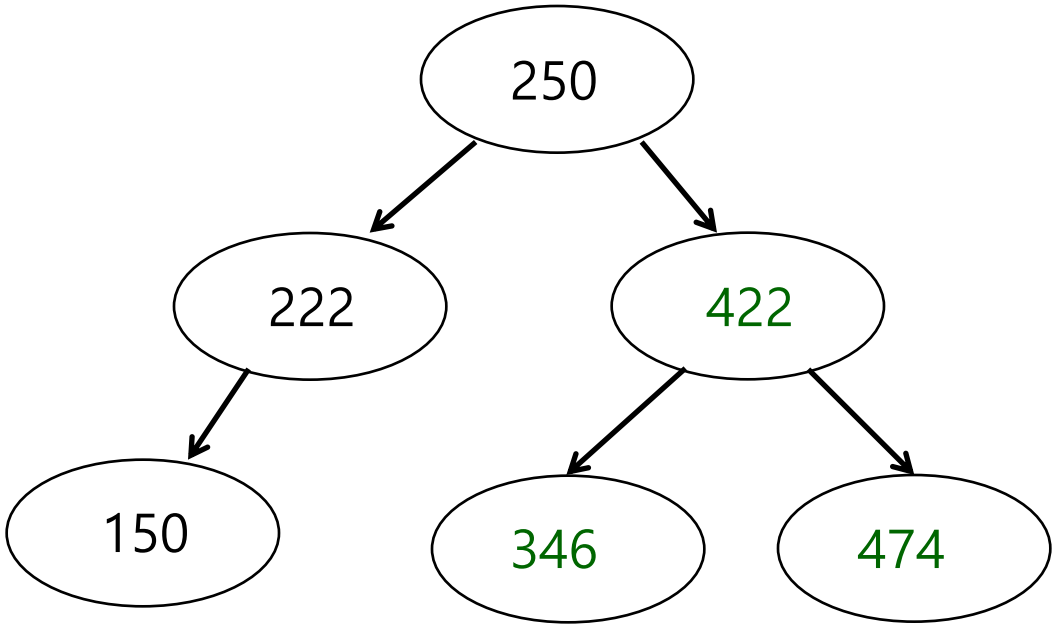
LR: Double Rotation (Left and Right)



# Exercise: Rebalance the AVL Tree

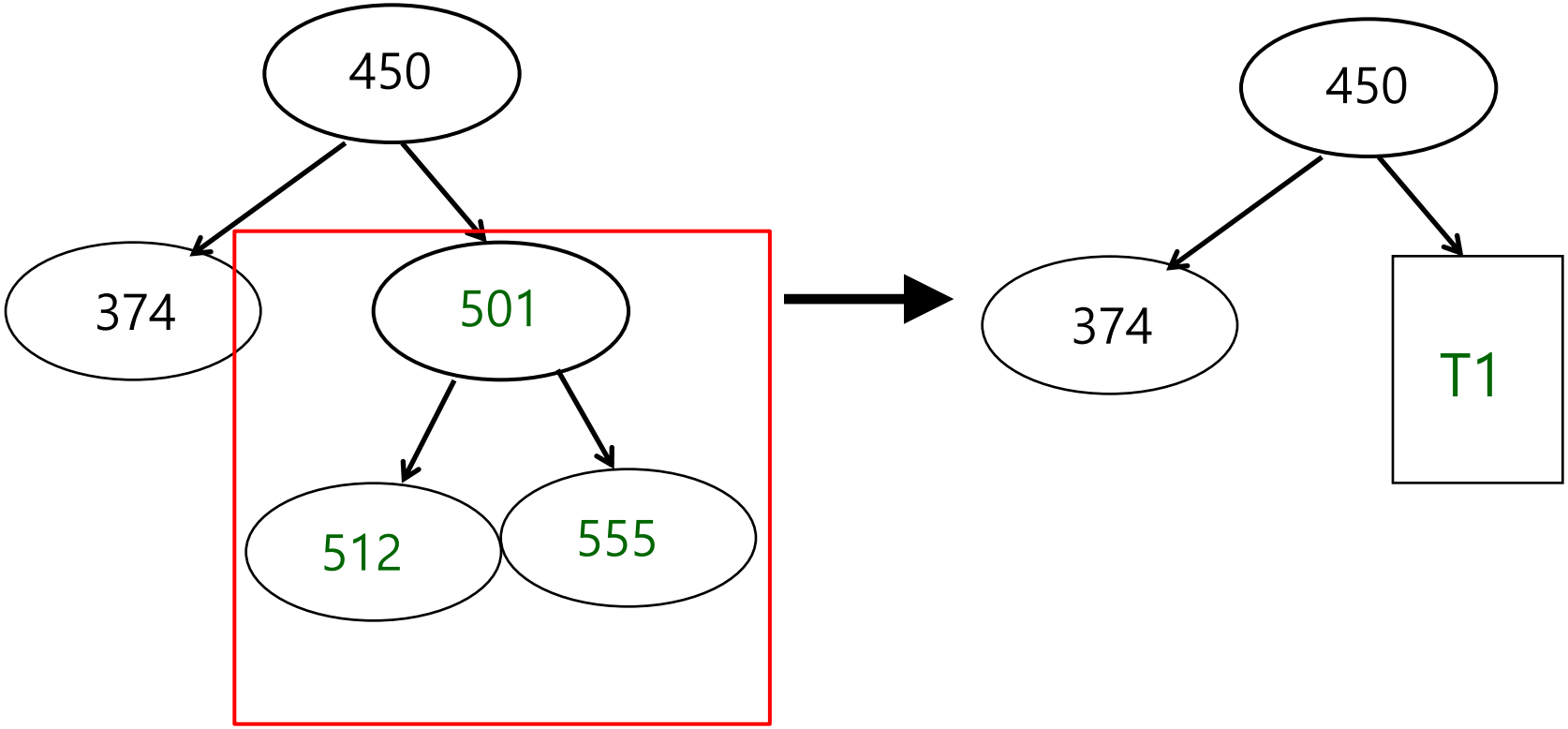


# Exercise: Result



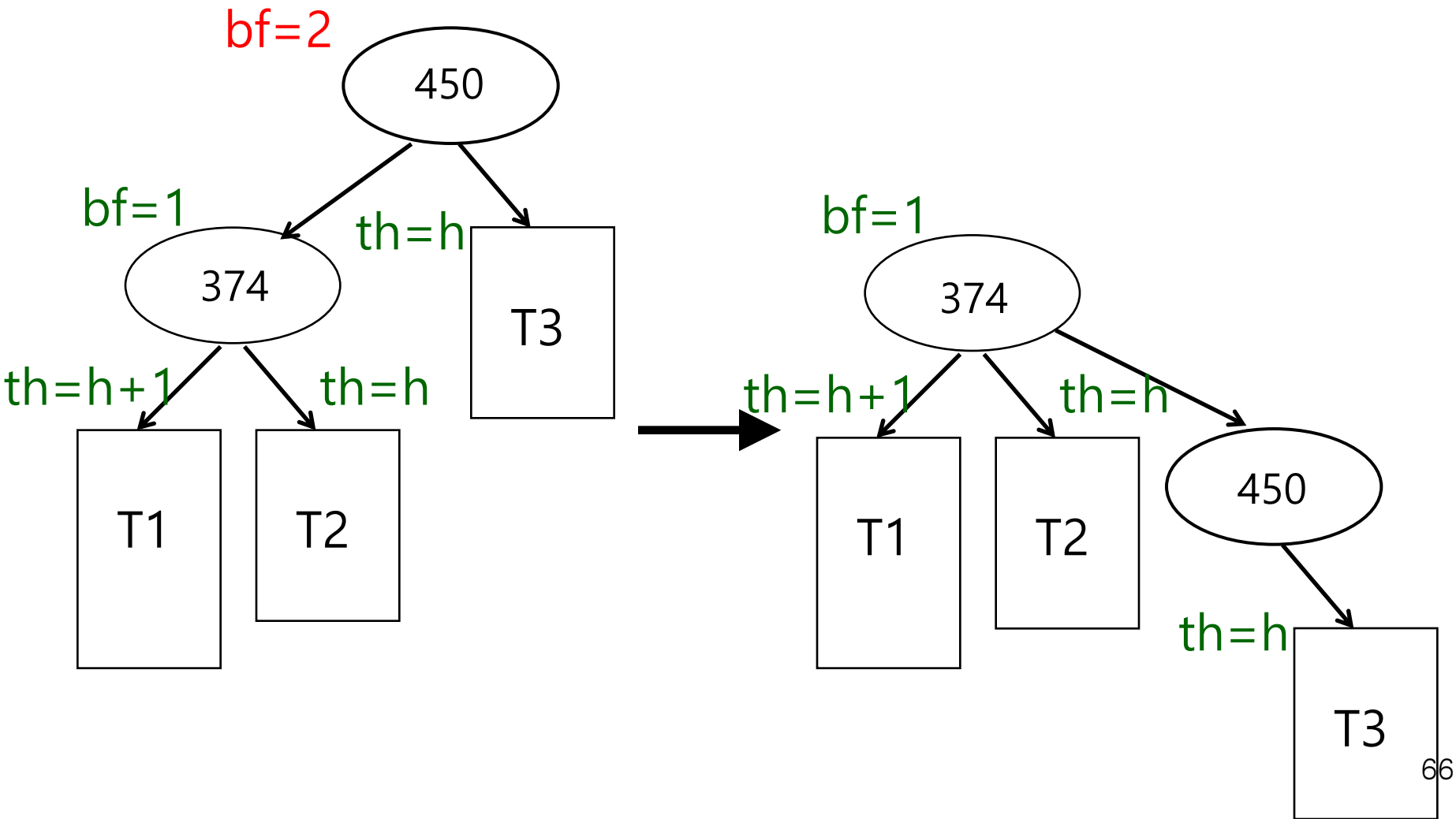
# Rotation of an Entire Subtree

# <Notation>



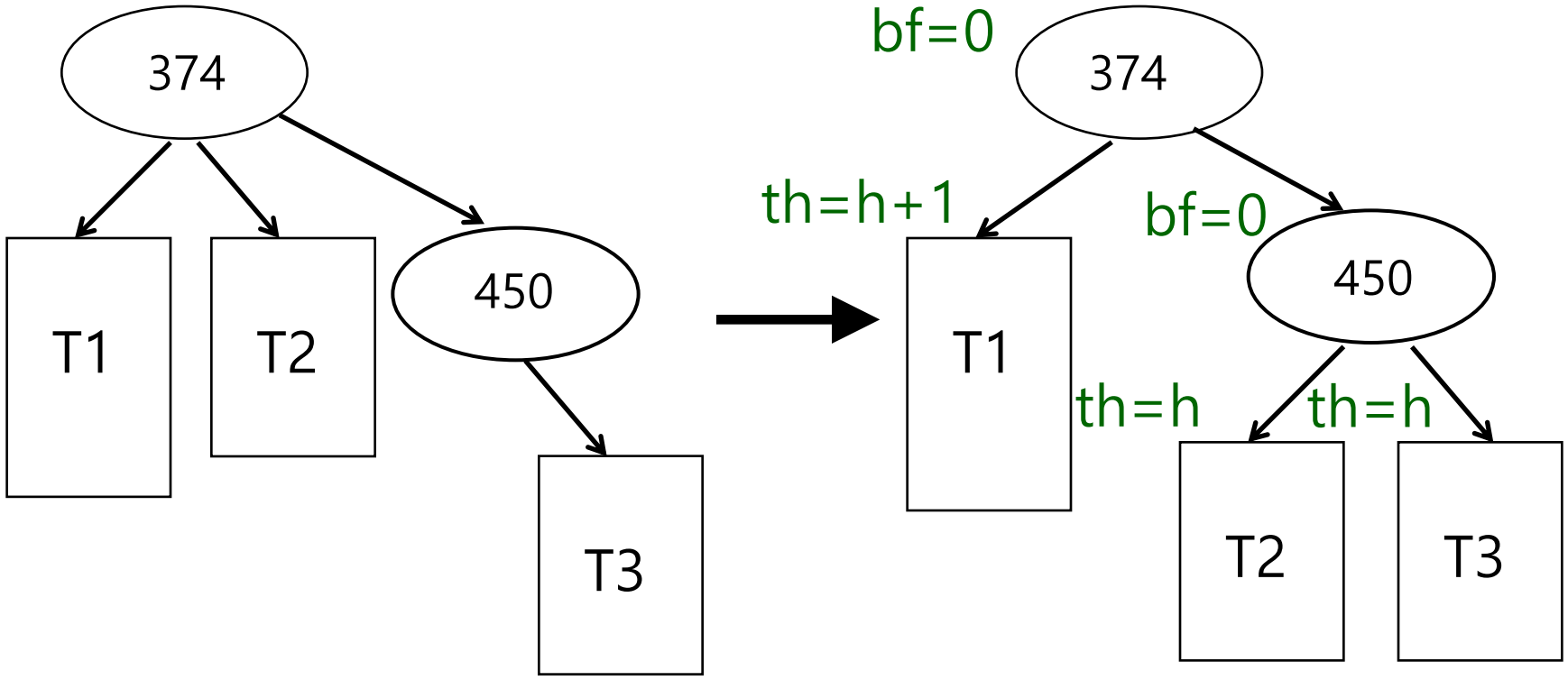
# AVL Tree Rotations: General (1/4)

LL: Single Rotation (Right)



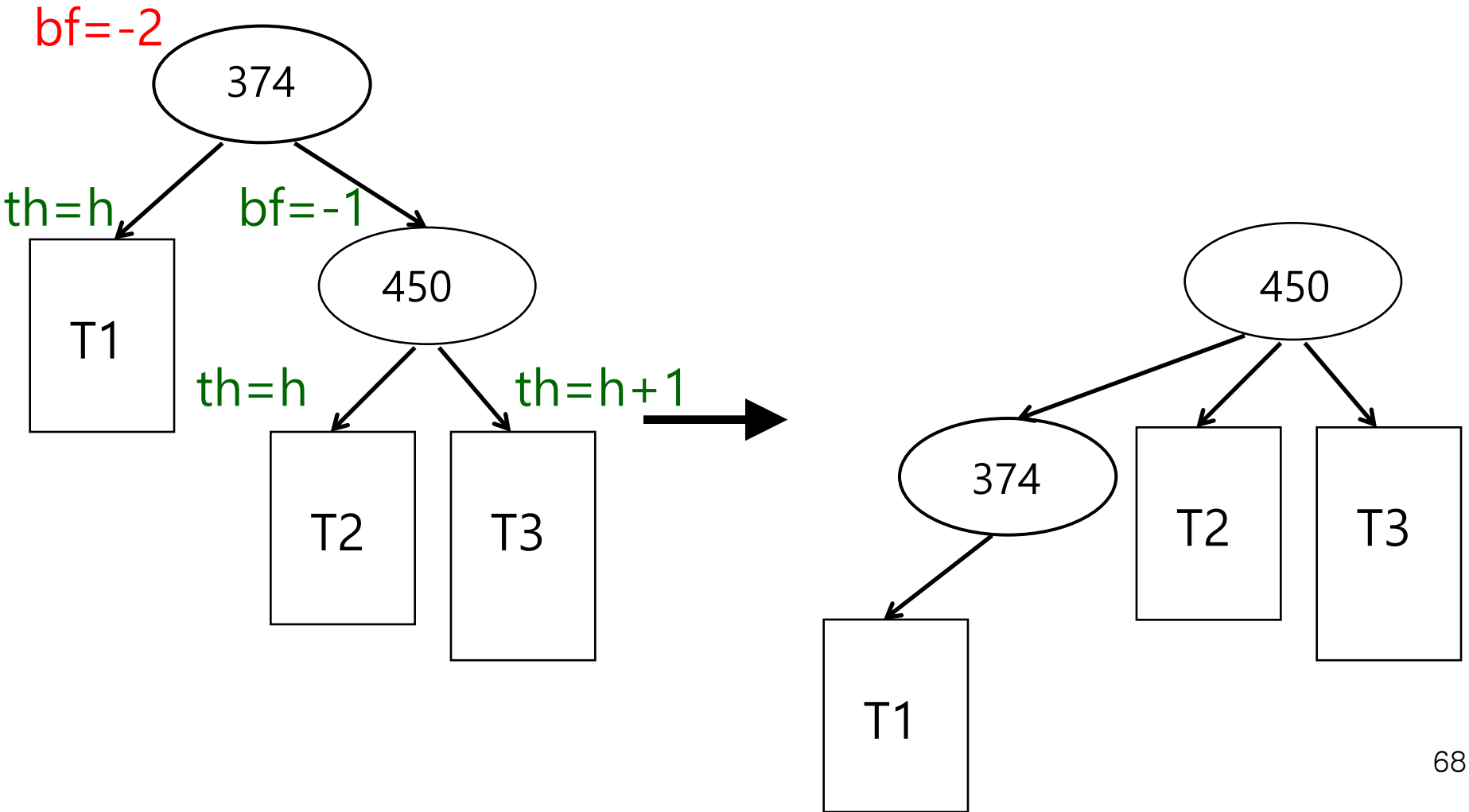
# AVL Tree Rotations: General (1/4) cont'd

LL: Single Rotation (Right)



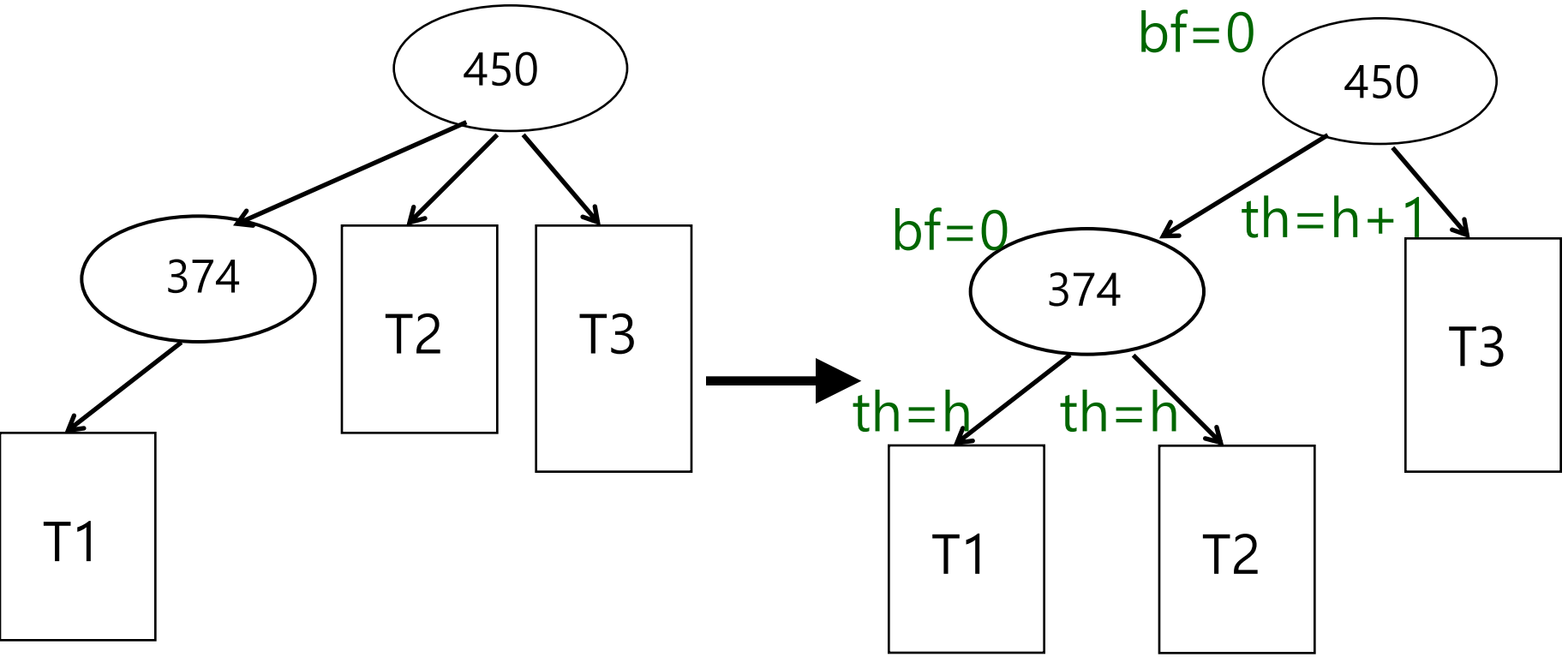
# AVL Tree Rotations: General (2/4)

RR: Single Rotation (Left)



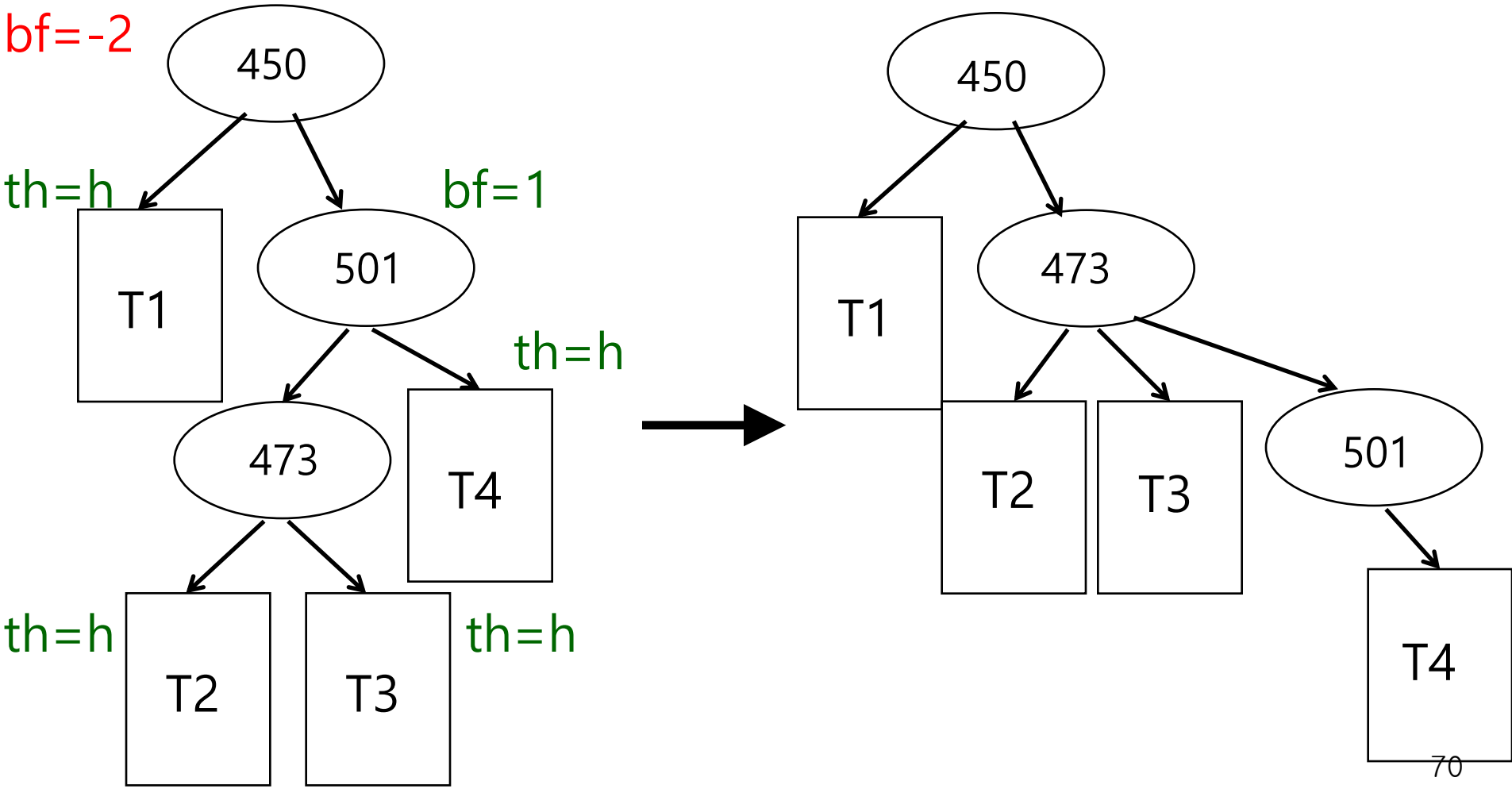
# AVL Tree Rotations: General (2/4) cont'd

RR: Single Rotation (Left)



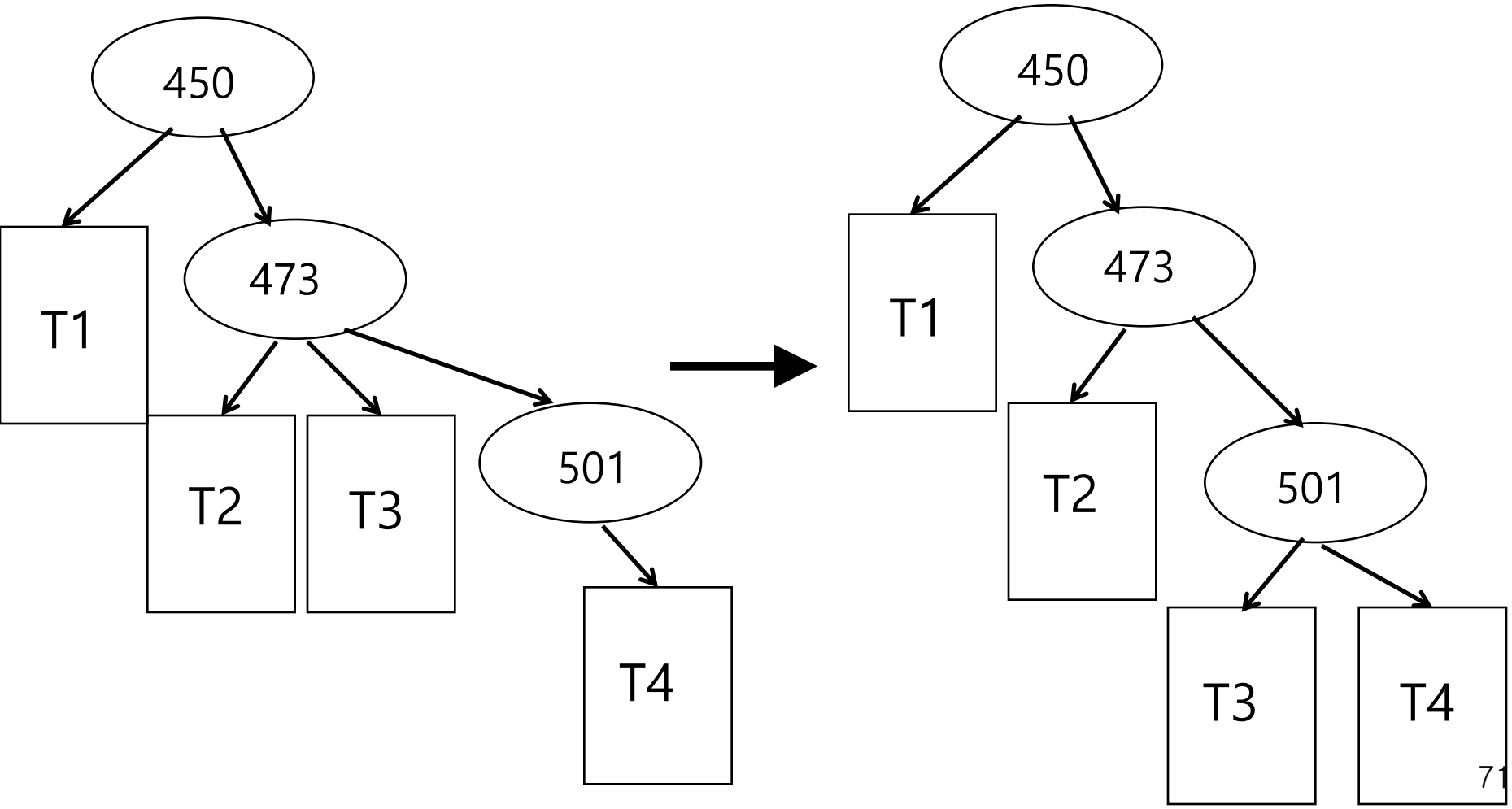
# AVL Tree Rotations: General (3/4)

RL: Double Rotation (Right and Left)



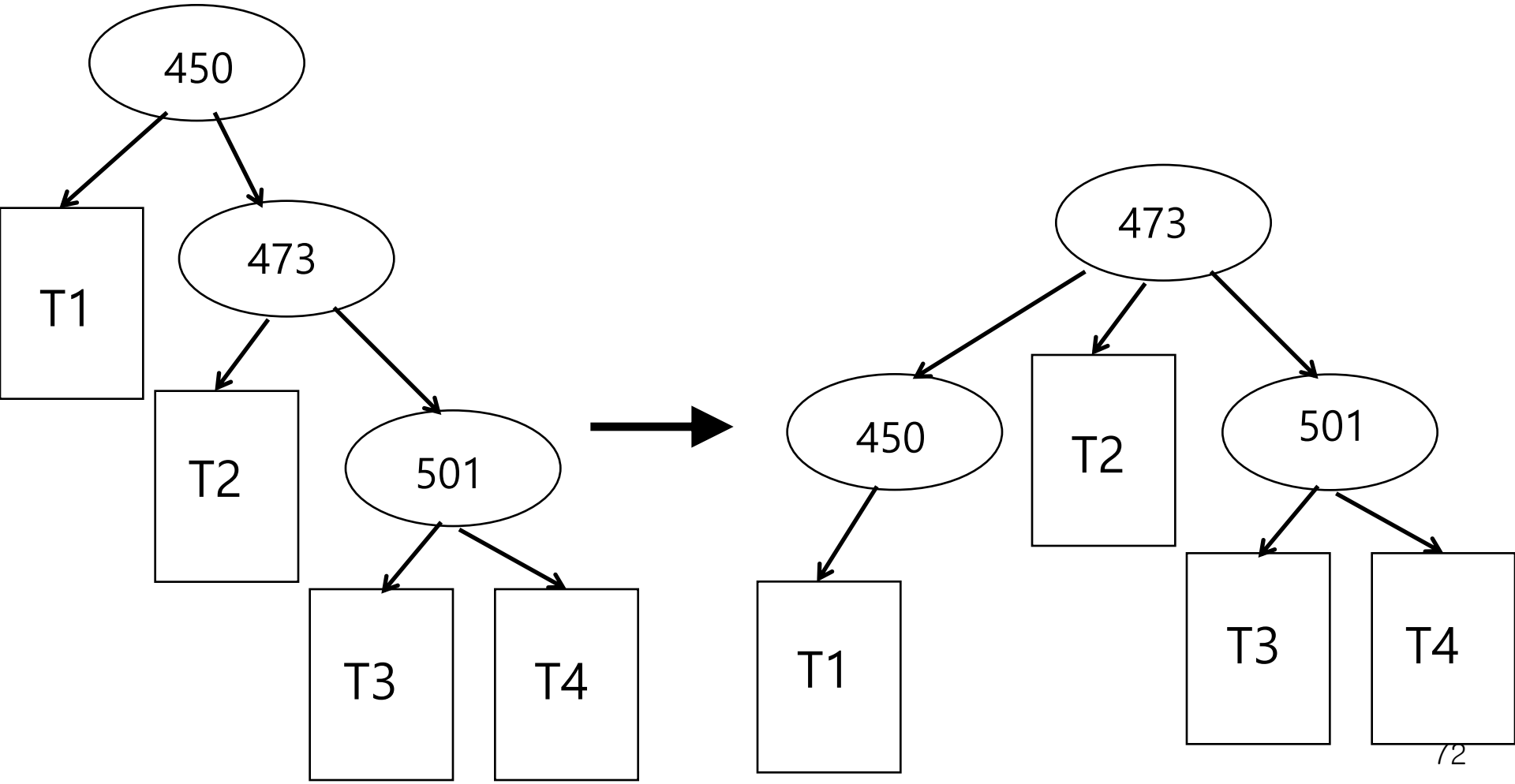
# AVL Tree Rotations: General (3/4) cont'd

RL: Double Rotation (Right and Left)



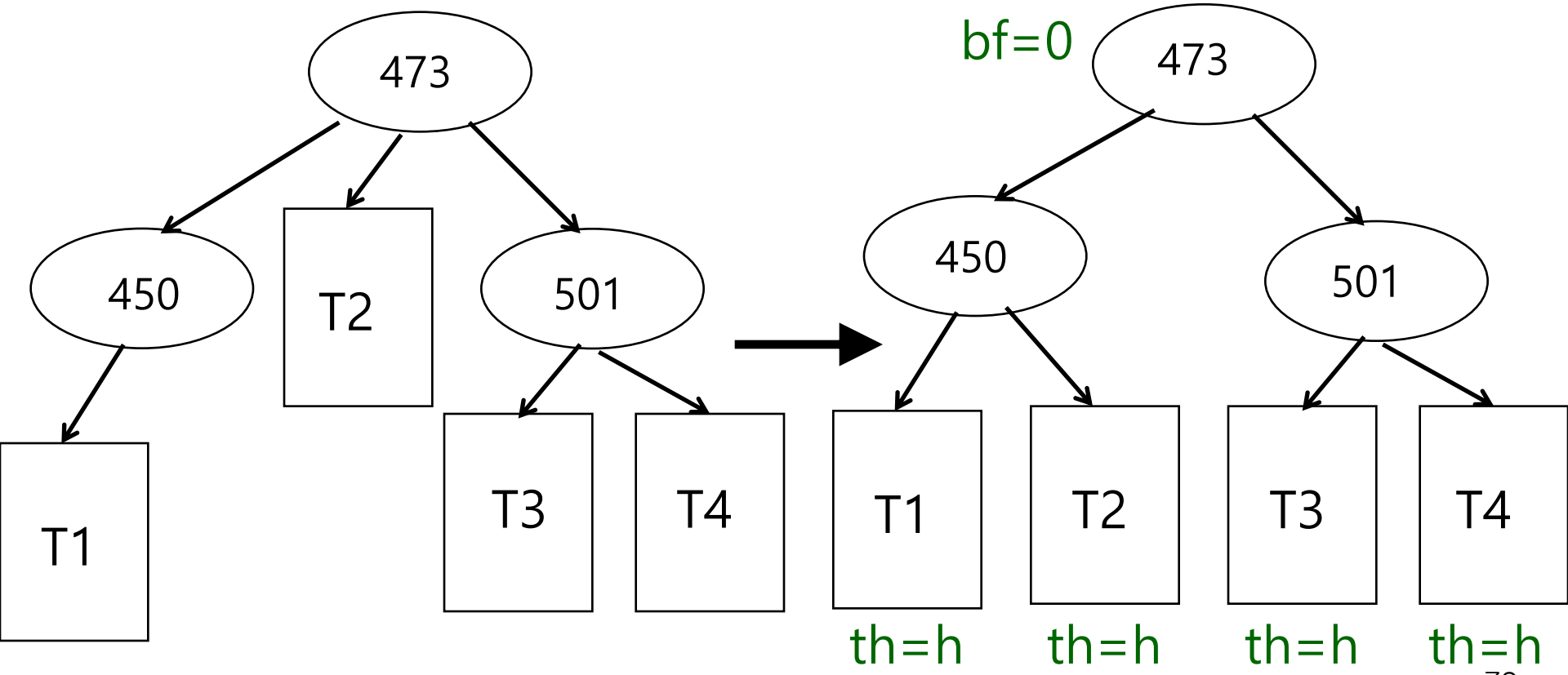
# AVL Tree Rotations: General (3/4) cont'd

RL: Double Rotation (Right and Left)



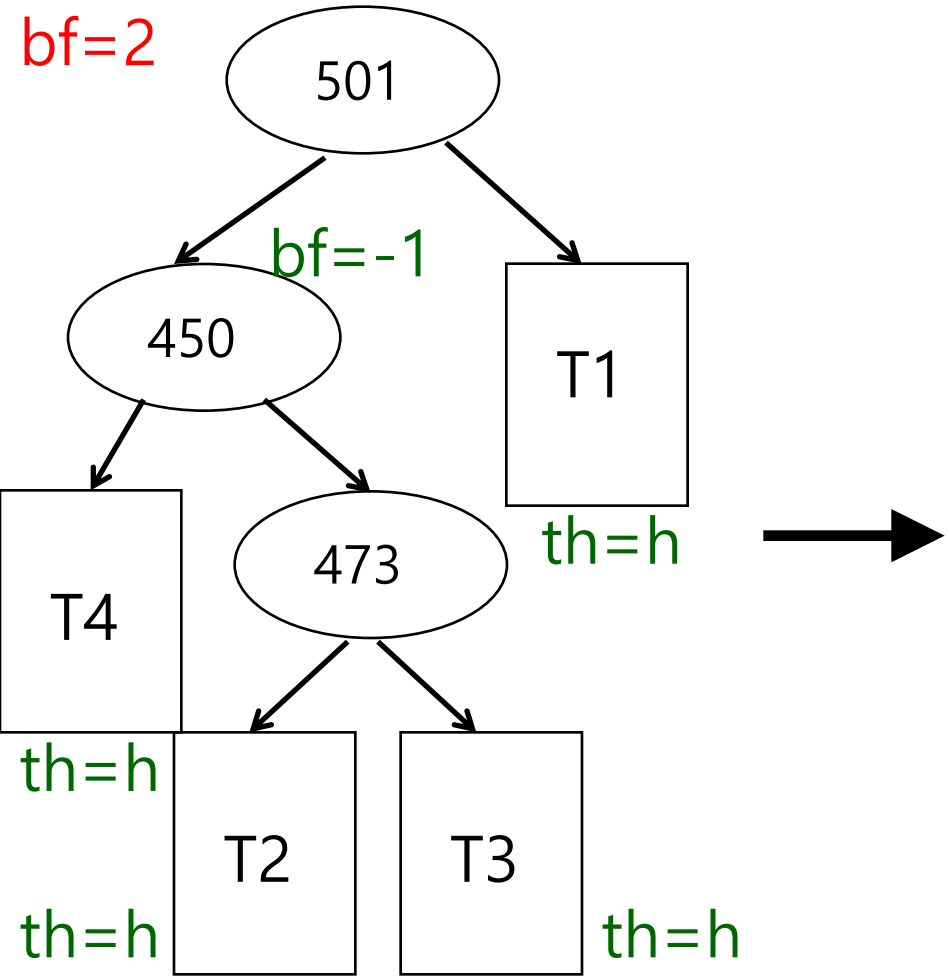
# AVL Tree Rotations: General (3/4) cont'd

RL: Double Rotation (Right and Left)



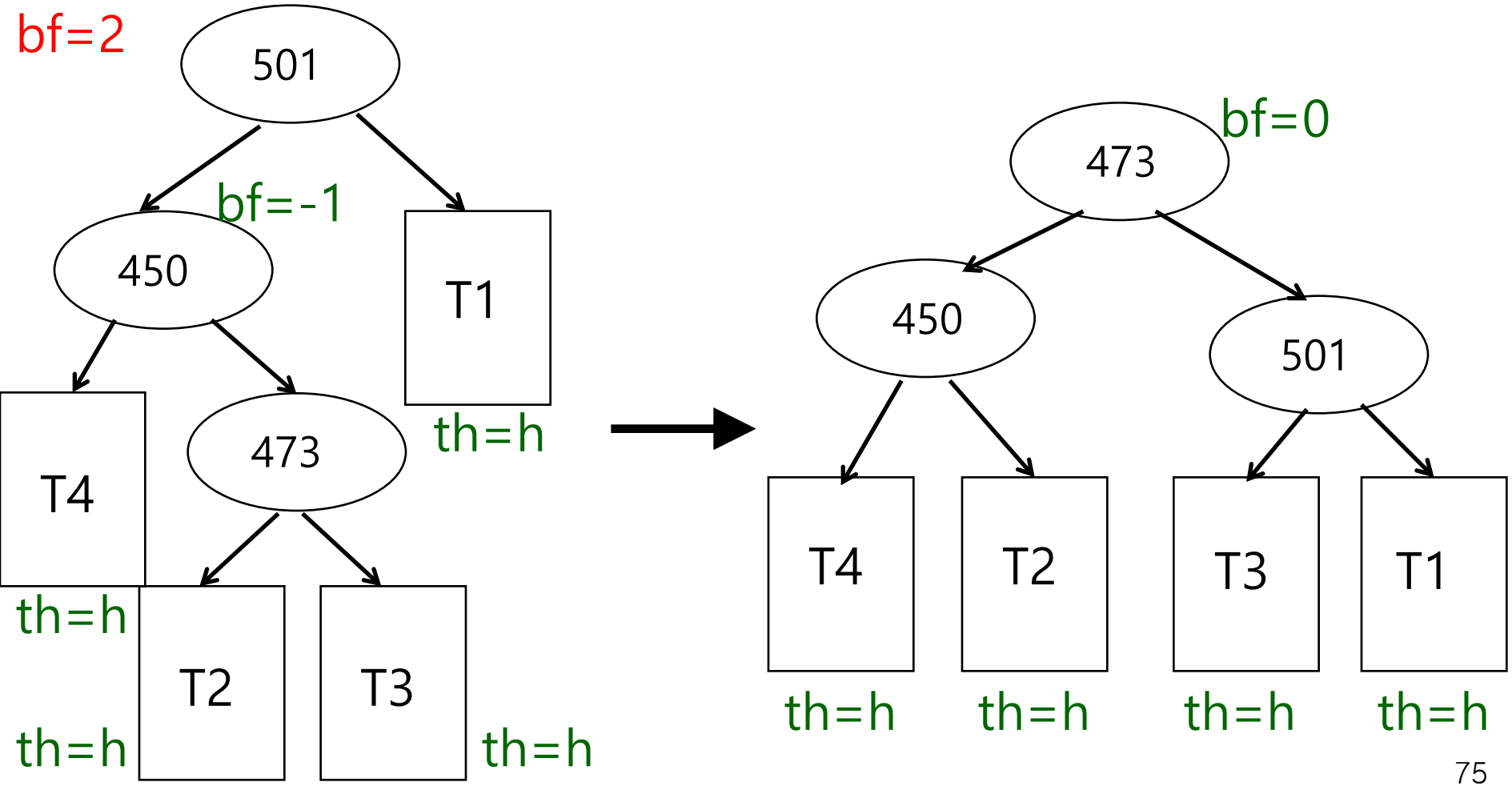
# AVL Tree Rotations: General (4/4)

LR: Double Rotation (Right and Left)

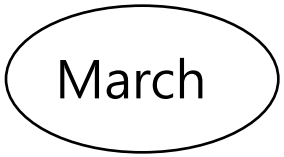


# AVL Tree Rotations: General (4/4)

LR: Double Rotation (Right and Left)

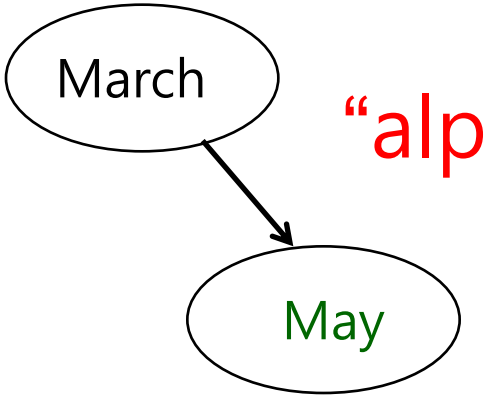


# Example: Build an AVL Tree (1/12)




insert May

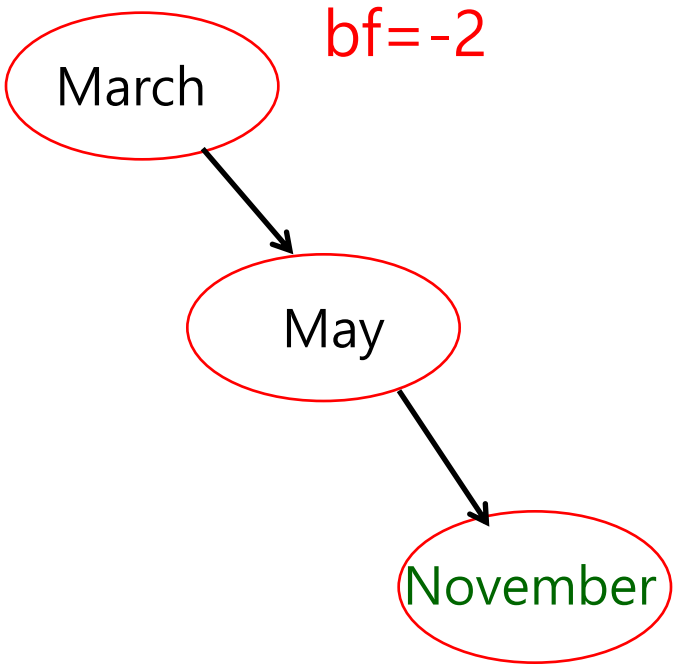
# Example: Build an AVL Tree (2/12)



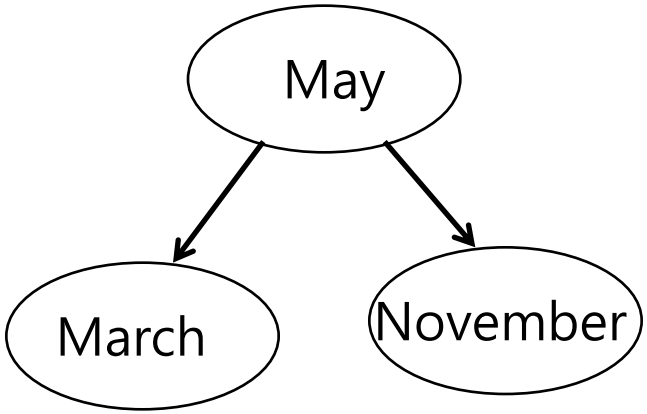

"alphabetical order"

insert November

# Exercise: Build an AVL Tree (3/12)



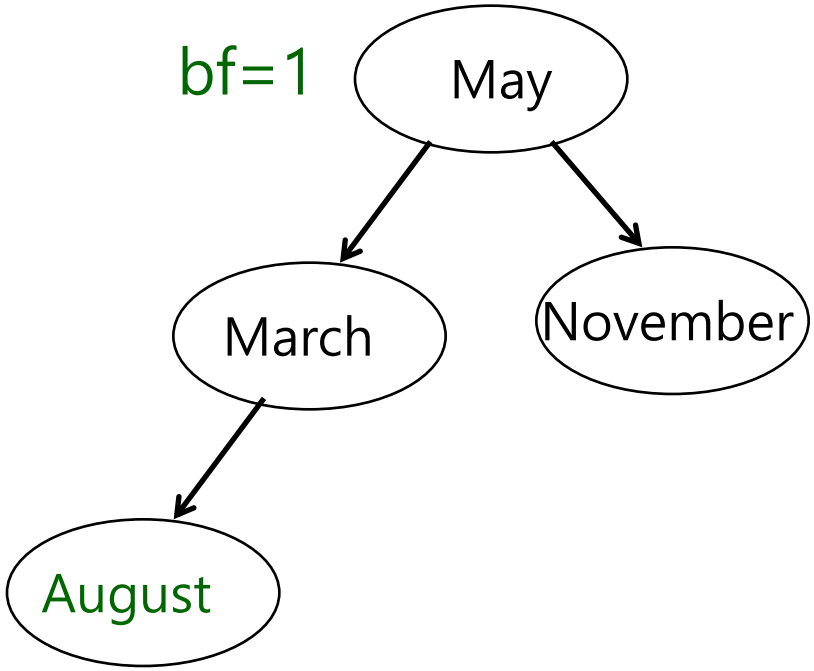
# Exercise: Build an AVL Tree (4/12)




RR rotation

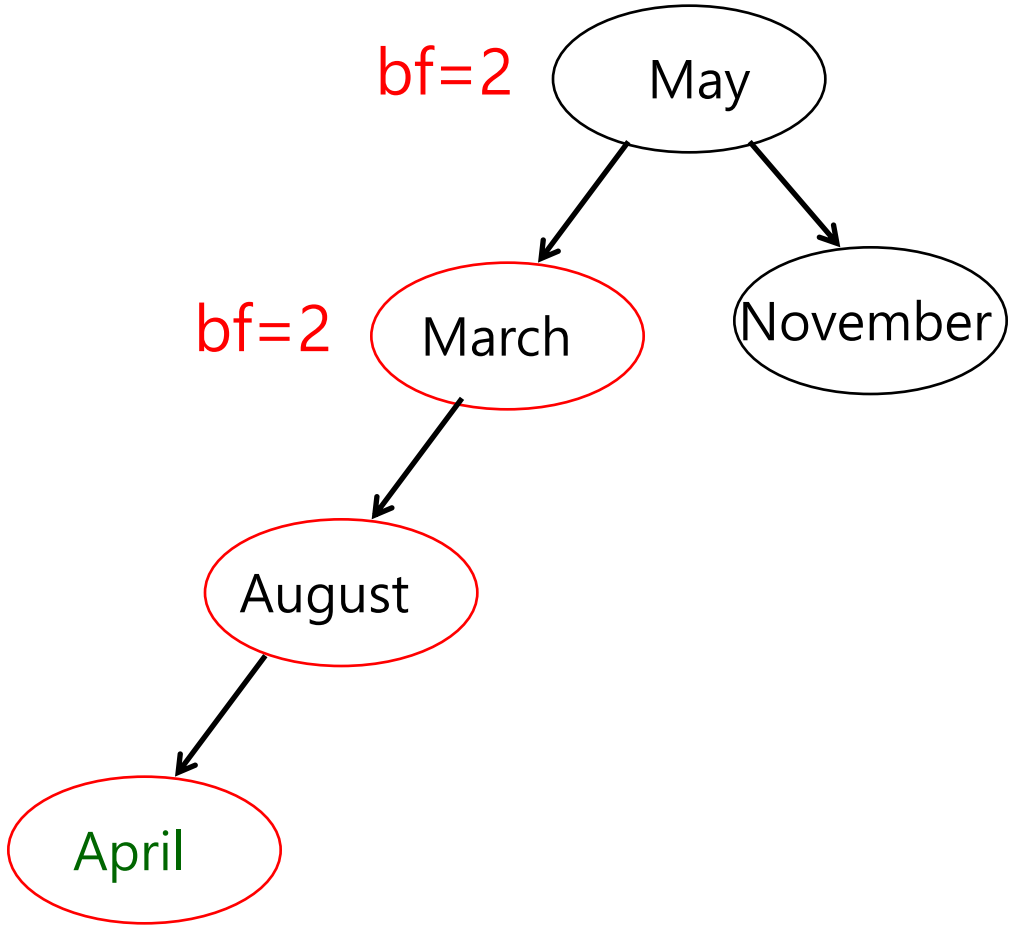
insert August

# Example: Build an AVL Tree (5/12)

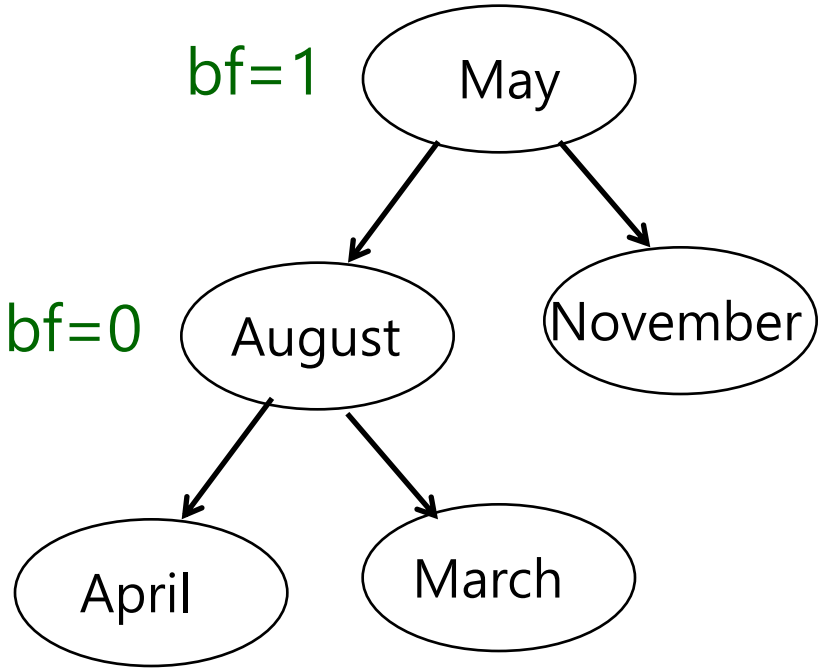



insert April

# Exercise: Build an AVL Tree (6/12)



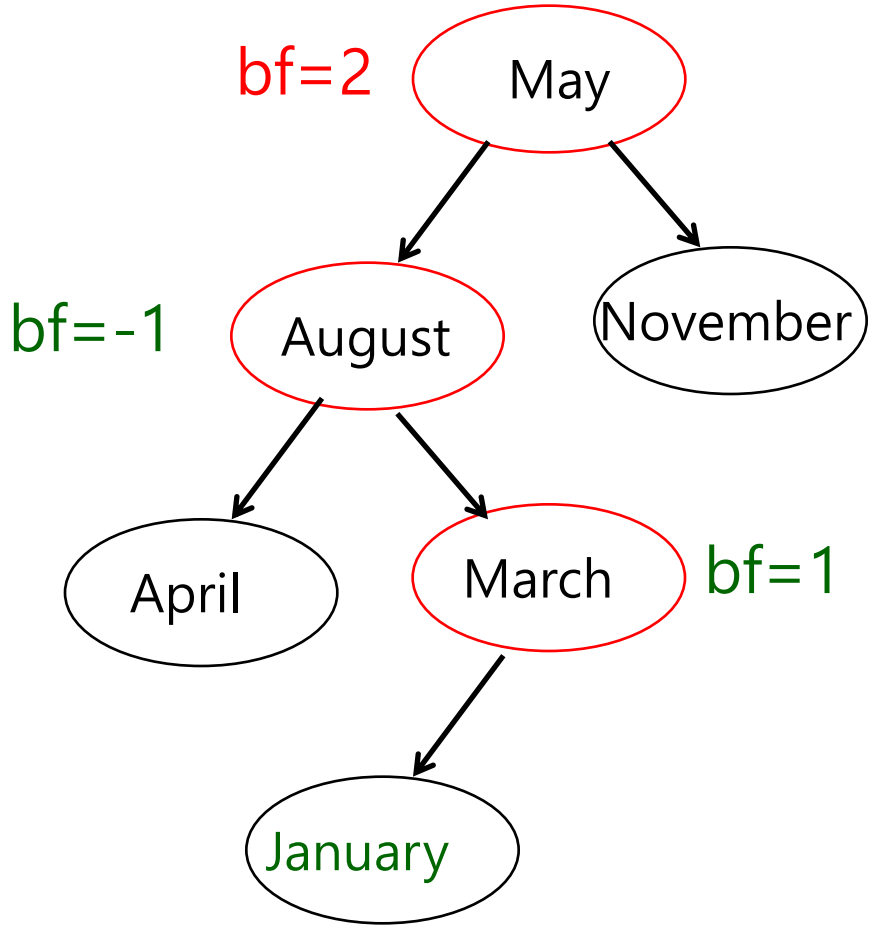
# Example: Build an AVL Tree (7/12)



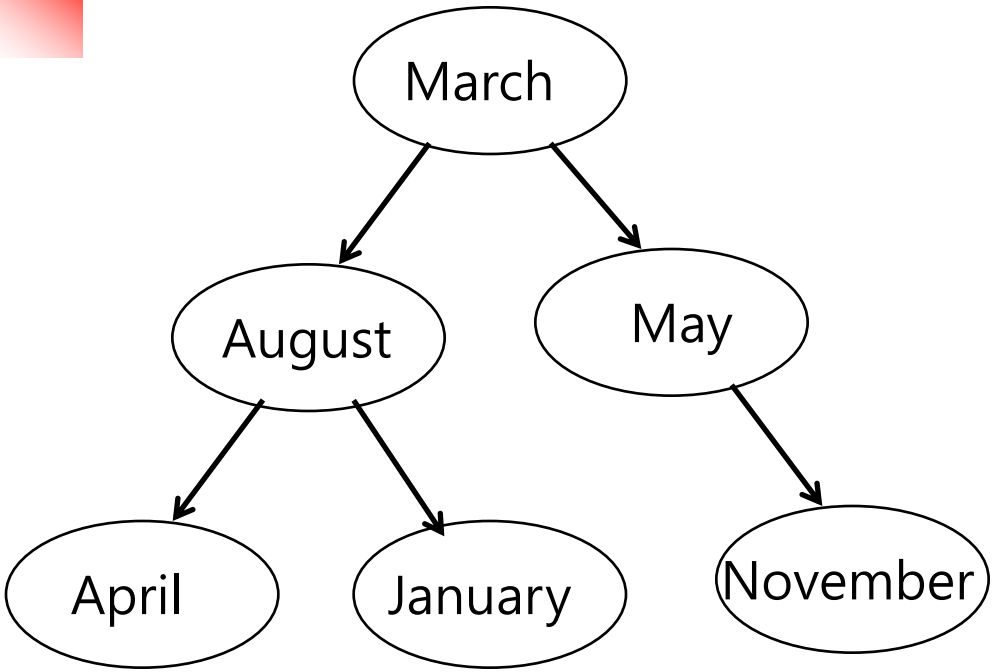

insert January

LL rotation

# Exercise: Build an AVL Tree (8/12)



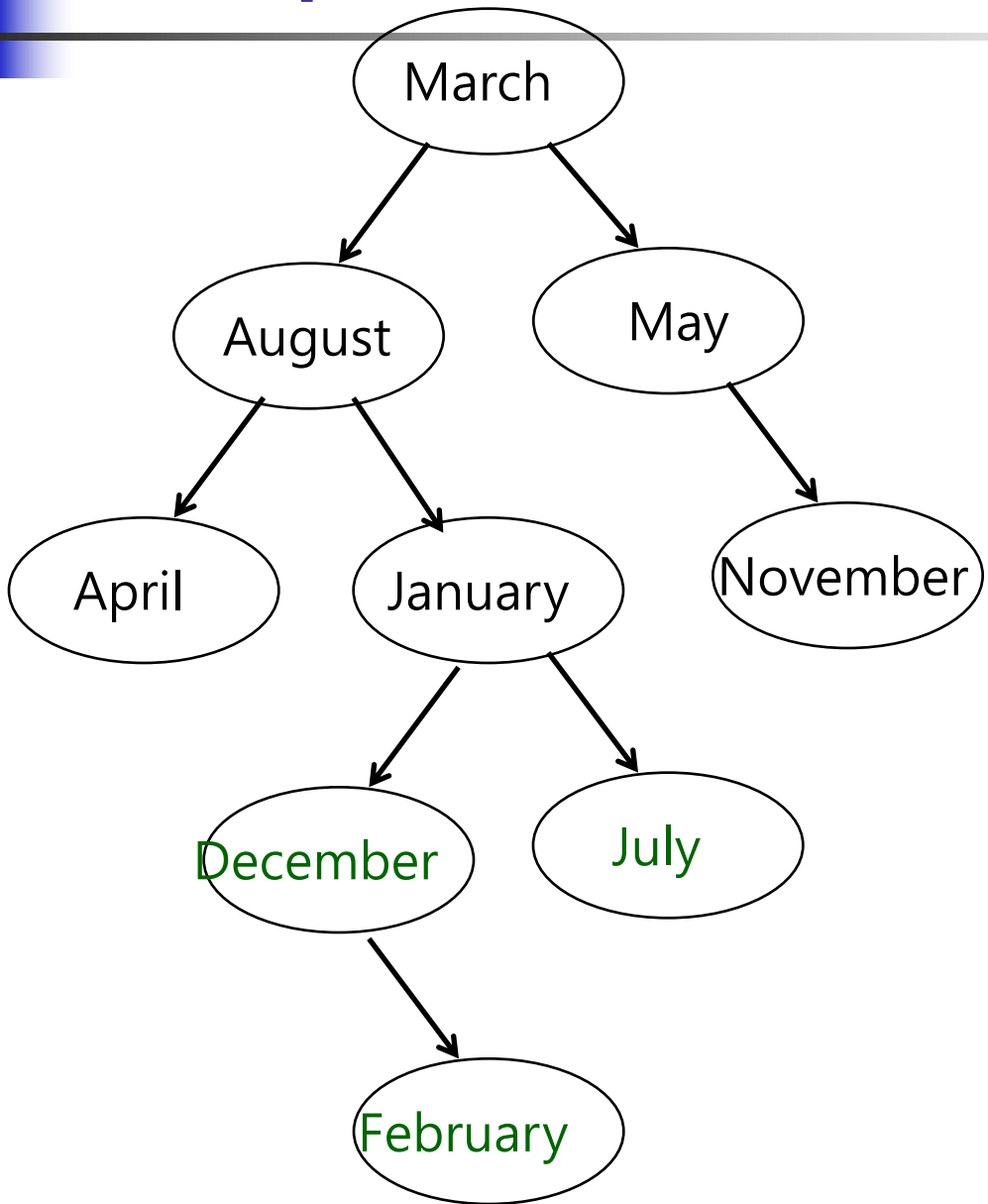
# Example: Build an AVL Tree (9/12)



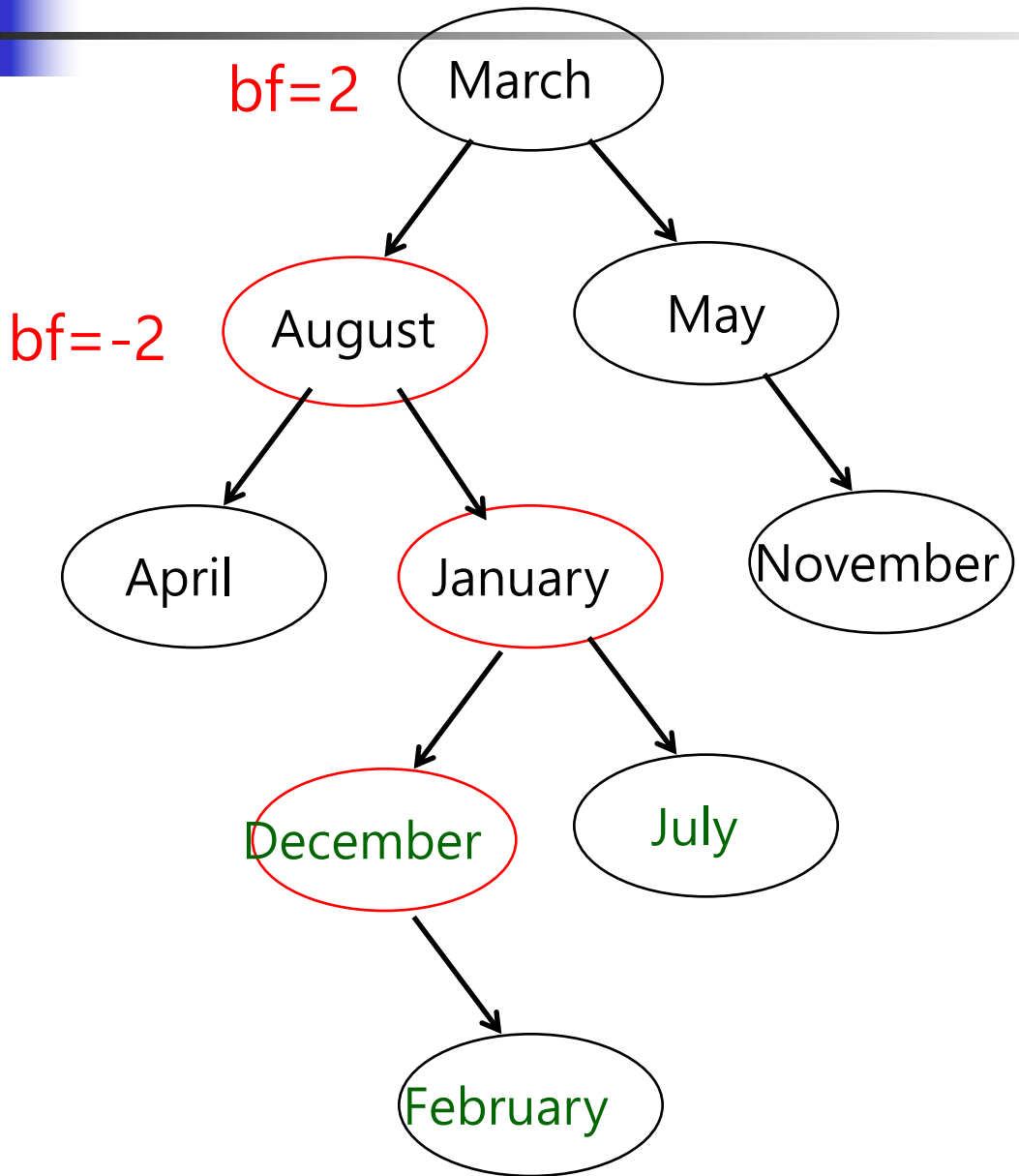

insert December
insert July
insert February

LR rotation

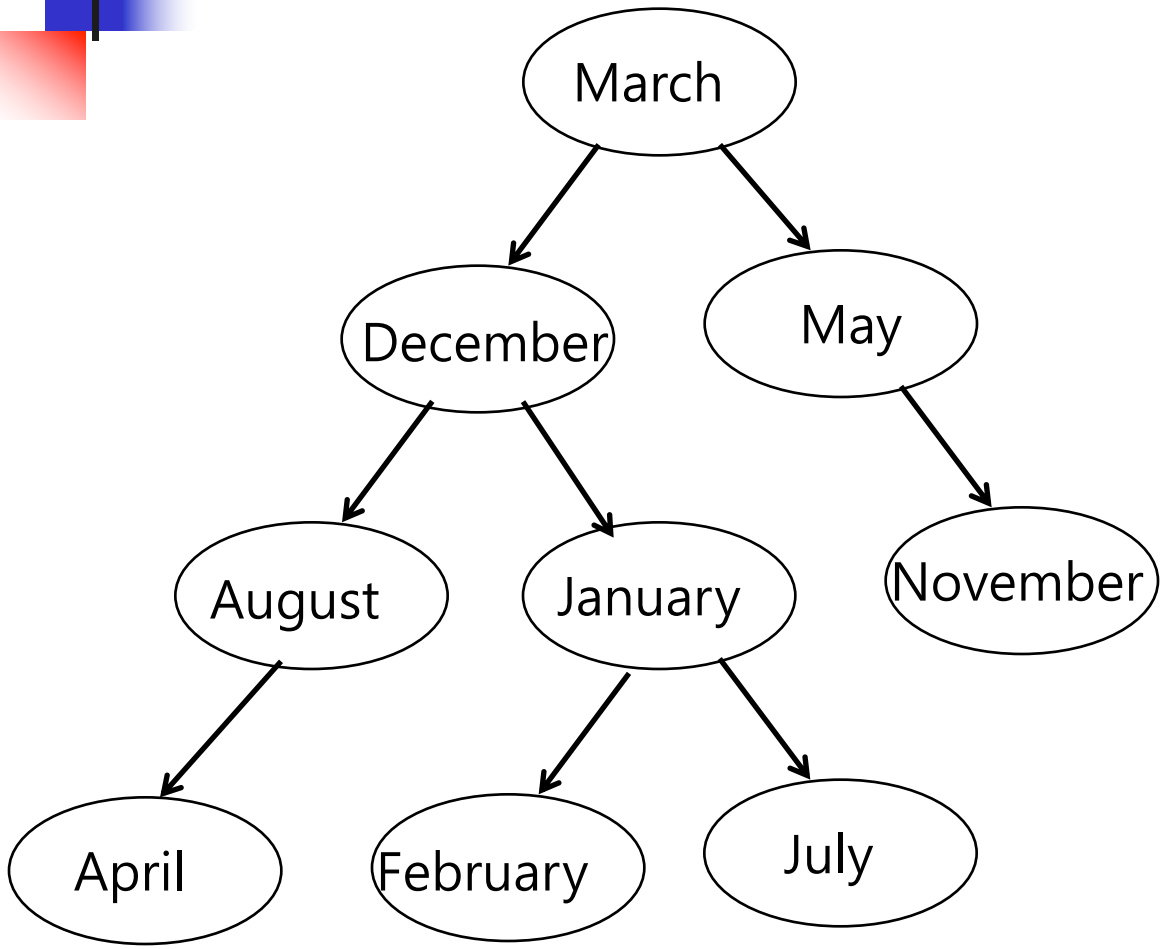
# Example: Build an AVL Tree (10/12)



# Exercise: Build an AVL Tree (11/12)



# Exercise: Build an AVL Tree (12/12)




RL rotation

# WHW 2-1: AVL Programs

- https://www.thecrazyprogrammer.com/2014/03/c-program-for-avl-tree-implementation.html
- (30 points) Hand trace the Above program in doing the following sequence of 5 inserts:
  - (for the purpose of easier conceptualization, assume the node key is a string, and node address is an integer, as shown below)
  - fish (address: 100), dog (addr: 120), cat (addr: 130), mouse (addr: 140), lion (addr: 150)
  - "hand tracing a program" means following the program line by line (as if you are the computer), plugging in the values for all the variables, and the function calls and returns.

# Note on Hand Tracing of Programs

- Hand tracing programs is extremely important.
- Code inspection, code review, and debugging are required work of software engineers.
- Code inspection
  - Hand tracing your own program or others' programs.
- Code review
  - Hand tracing others' programs
- Debugging
  - Hand tracing to find the source of errors discovered.

# WHW 2-2: (10 points)
# AVL Tree Insert and Delete

- Insert
  - Cat Dog Bat Fish Chicken Cow Tiger Eagle Lion Snake Bird Owl Mouse
- Delete
  - Cow Snake Owl Cat Mouse Eagle Bird

# End of Lecture