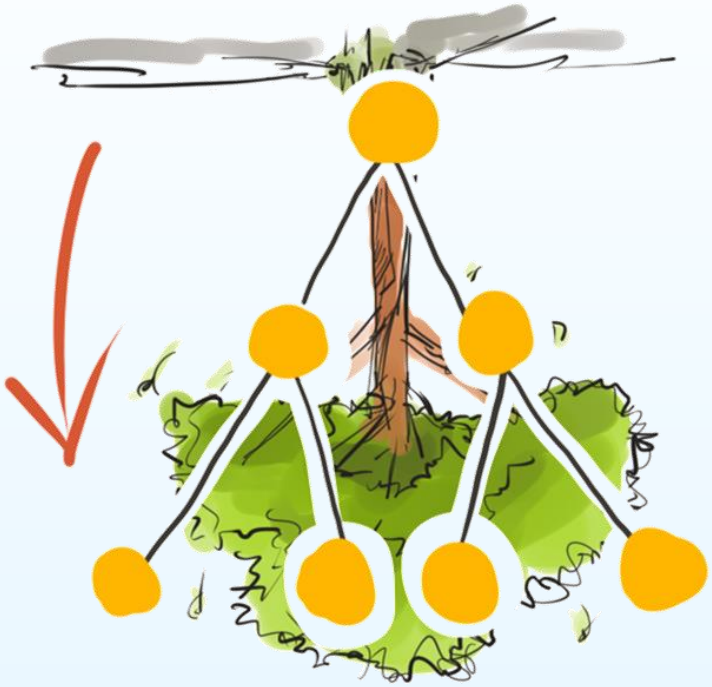


Lecture 9: Search Trees

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.

Department of Computer Engineering Kasetsart University

MIKE LABORATORY

Motivation

Previously, we discussed Abstract Lists (List ADT)

- The objects are explicitly linearly ordered

If we implement an abstract list using an array or a linked list, what are the running time for the following operations?

3	9	5	14	10	6	8	17	15	13	23	12

- Finding a specific value ➡ $O(n)$
- Finding the smallest/largest value ➡ $\Theta(n)$
- Finding the next larger of a given value ➡ $\Theta(n)$

How can we reduce these running times?

Outline

- Binary Search Trees
- Balanced Trees
  - AVL Trees
  - Red-Black Trees
  - Weight-Balanced Trees

Background

Recall that with a binary tree, we can dictate an order on the two children

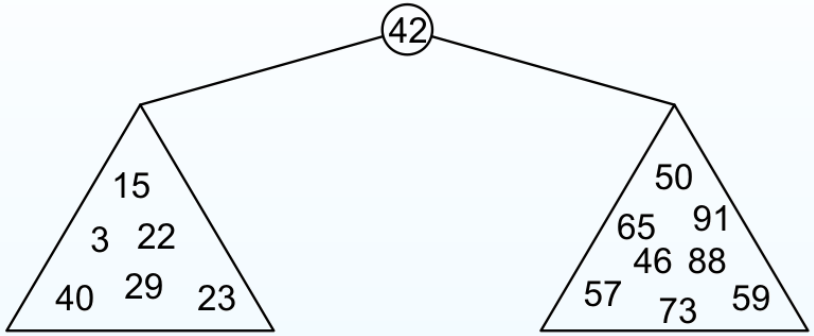
i.e., the left child followed by the right child

We will exploit this order:

- Require all objects in the left subtree to be less than the object stored in the parent node, and
- Require all objects in the right subtree to be greater than the object stored in the parent node

Binary Search Trees

Graphically, we may relationship



Each of the two subtrees will themselves be binary search trees

We can use this structure for searching:

- Examine the current node
- If the object is less than, continue searching in the left subtree
- Otherwise, continue searching in the right subtree

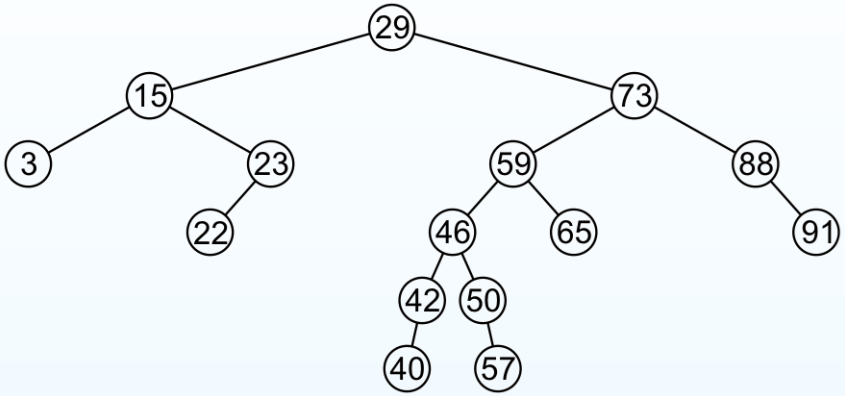
Definition of BST

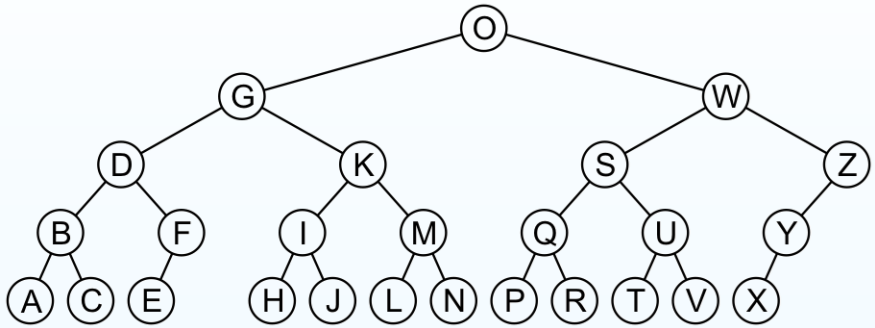
We define a non-empty binary search tree as a binary tree with the following properties:

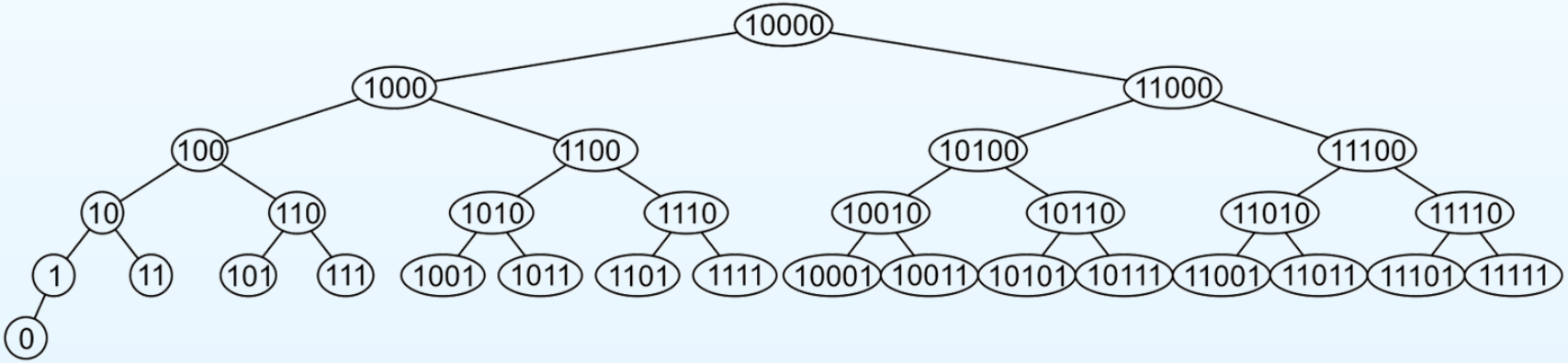
- The left subtree (if any) is a binary search tree and all values are less than the root value, and
- The right subtree (if any) is a binary search tree and all values are greater than the root value

Please note that, for convenience, we will treat data as unique values

Examples

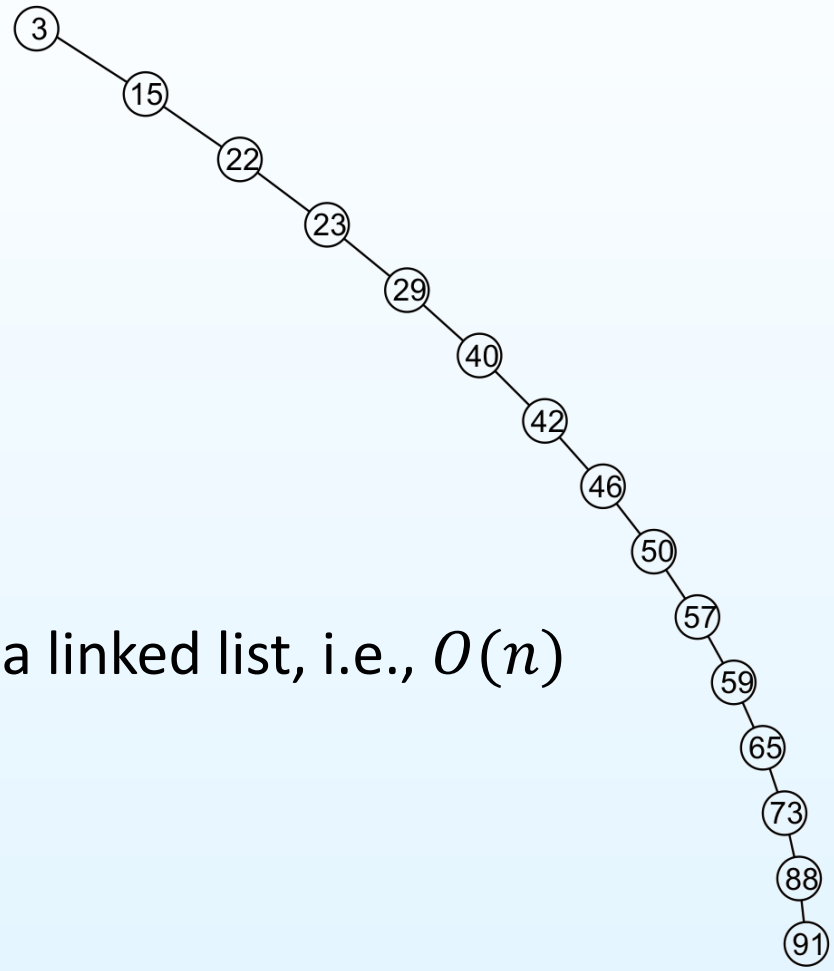

29
15
73
3
23
59
88
22
46
65
91
42
50
40
57


O
G
W
D
K
S
Z
B
F
I
M
Q
U
Y
A
C
E
H
J
L
N
P
R
T
V
X


10000
1000
11000
100
1100
10100
11100
10
110
1010
1110
10010
10110
11010
11110
1
11
101
111
1001
1011
1101
1111
10001
10011
10101
10111
11001
11011
11101
11111
0

Examples

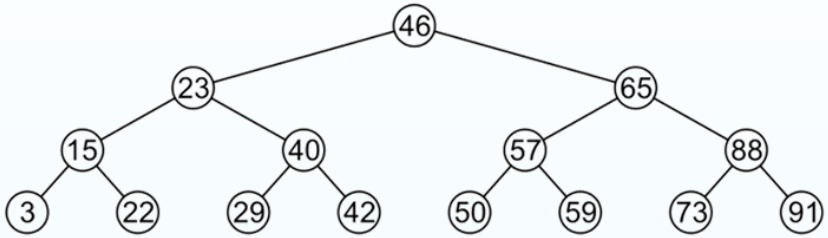
Unfortunately, it is possible to construct degenerate binary search trees

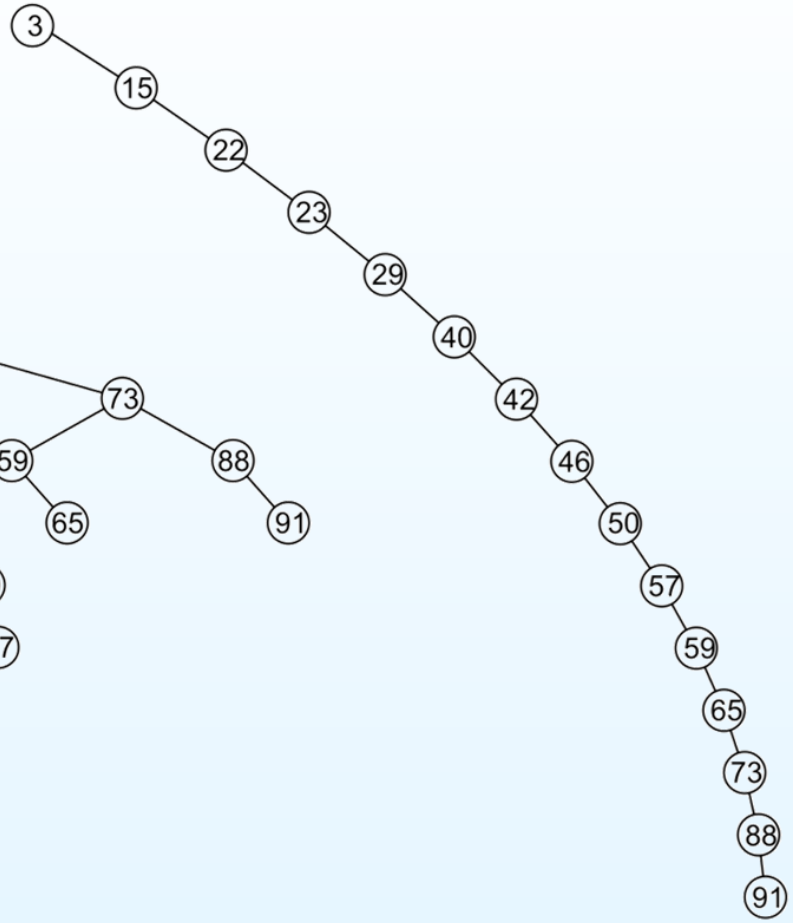


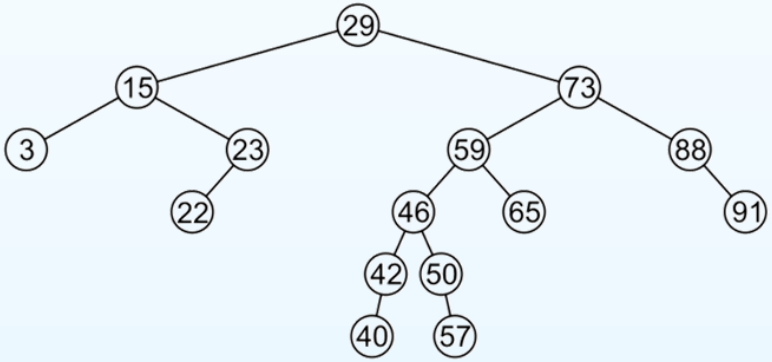
This is equivalent to a linked list, i.e., $O(n)$

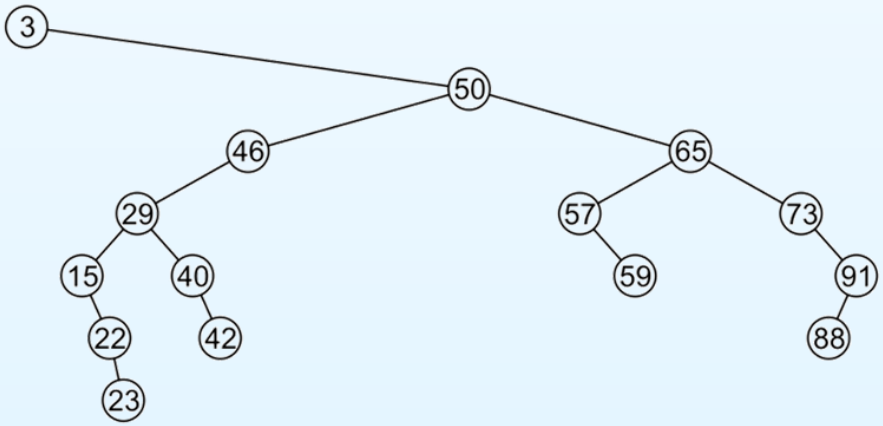
Examples

All these binary search tree store the same data:





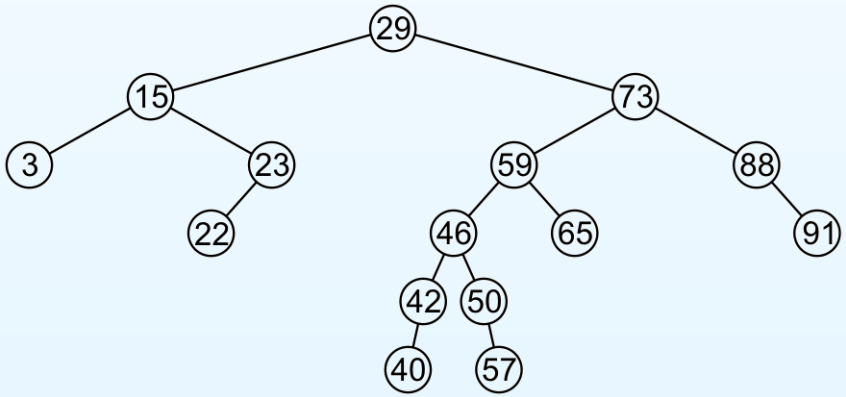




Abstract BST

- The data stored in a binary tree with the properties of the left and right subtrees are less than and greater than their parent, respectively

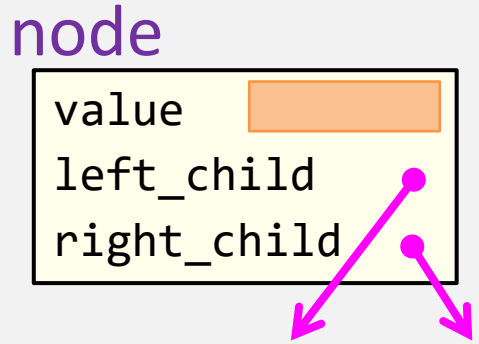
- Operations (some are inherited from the general/binary tree):
  - `insert(tree, node)`
  - `delete(tree, node)` // only that node
  - `find(tree, node)`
  - `find_min(tree)`
  - `find_max(tree)`
  - `size(tree)`
  - `height(tree)`
  - `...`



BST Implementation

Assume that all data are unique integers

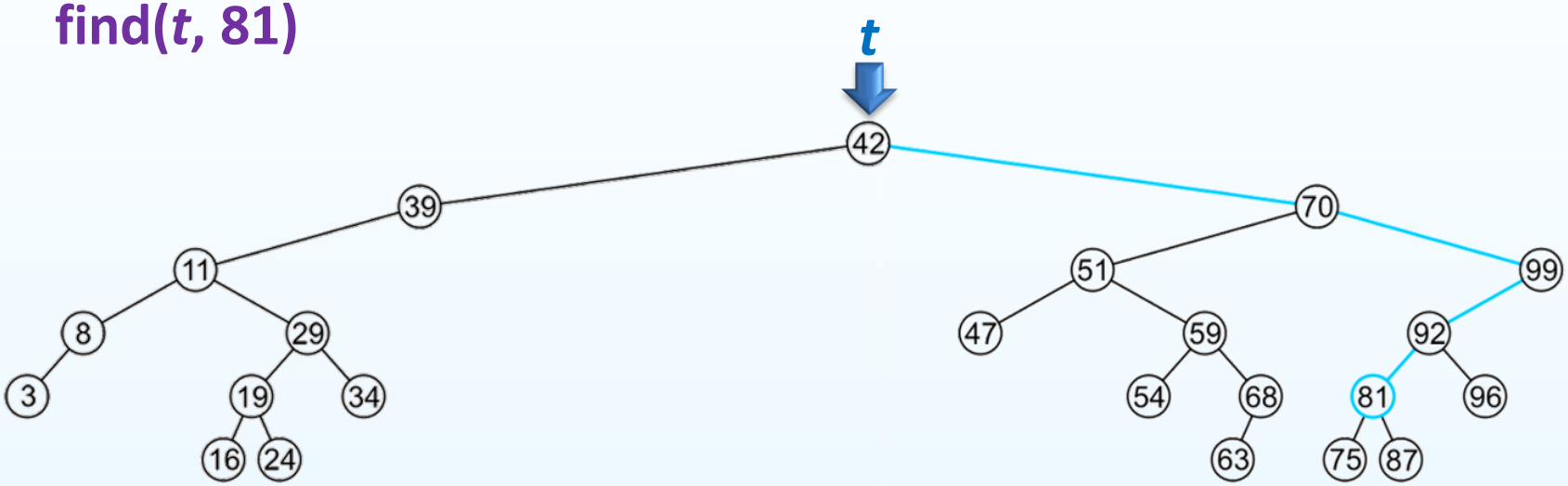
```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:
4:  typedef struct node {
5:    int value;
6:    struct node *left_child;
7:    struct node *right_child;
8:  } node_t;
9:
10: typedef node_t bst_t;
11:
12: int main(void) {
13:   bst_t *t = NULL;
14:   return 0;
15: }
```



The `find()` Operation

Return position of node *v* in BST *t* if found, otherwise `NULL`

find(*t*, 81)



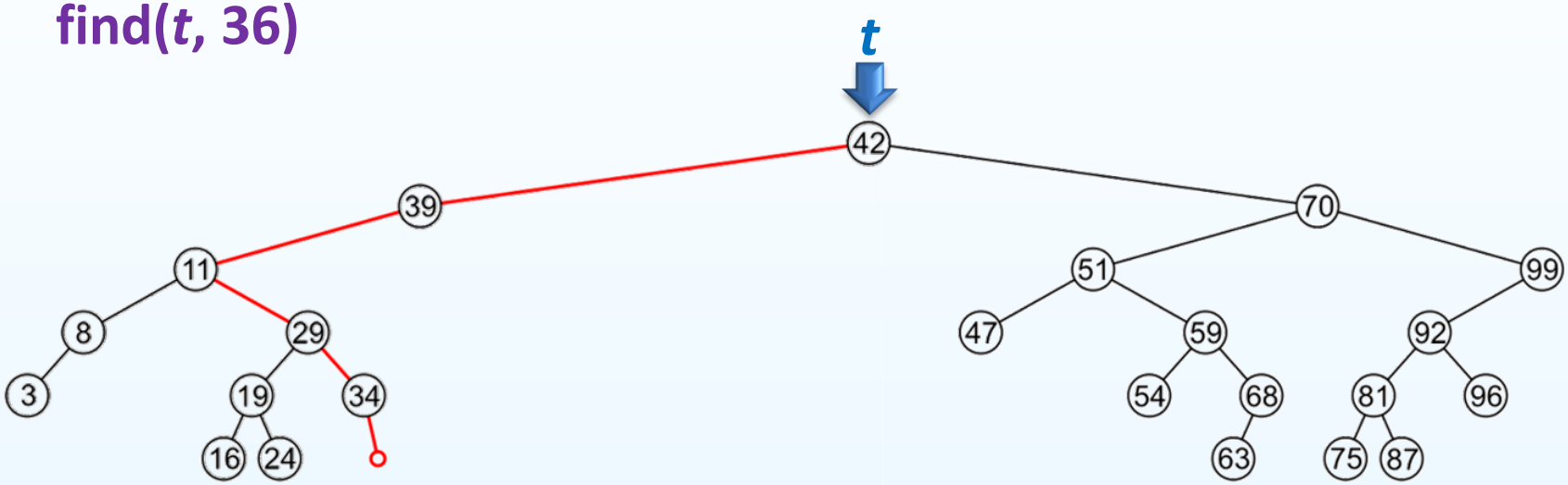
algorithm:

1. If the tree is null, then return
2. Compare the value with root
    2.1 If it is equal, then return
    2.2 If it is less than, then recurse for left
    2.3 Else recurse for right

The `find()` Operation

Return position of node v in BST t if found, otherwise `NULL`

find(t, 36)



algorithm:

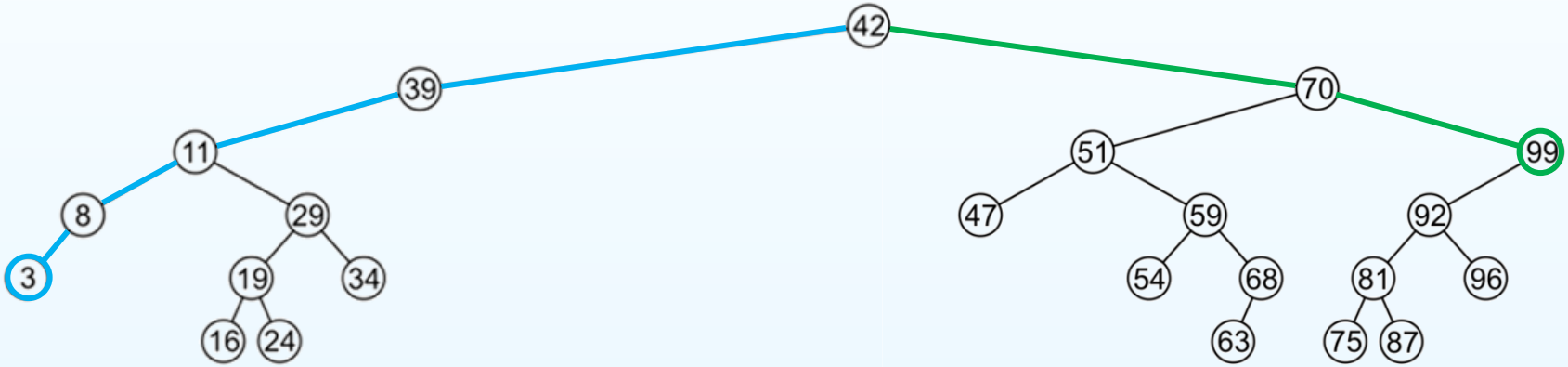
1. If the tree is null, then return
2. Compare the value with root
   - 2.1 If it is equal, then return
   - 2.2 If it is less than, then recurse for left
   - 2.3 Else recurse for right

Running time
$= O(h)$

How about `find_min()` and `find_max()`?

Return the smallest/largest value in BST t

find_min(t) **find_max(t)**

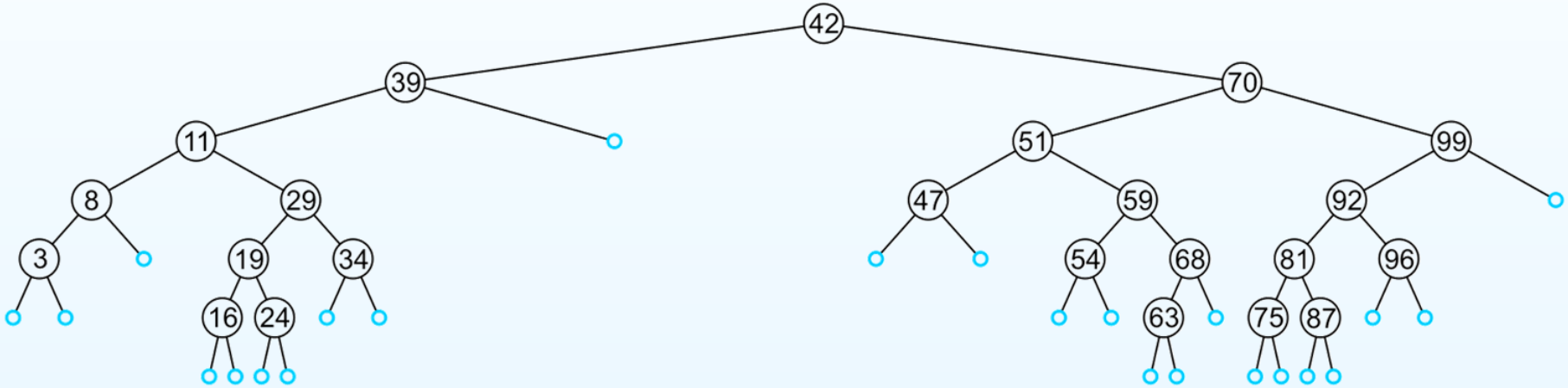


Running time
$= O(h)$

Insertion

An insertion will be performed at a leaf node:

- Any empty node is a possible location for an insertion



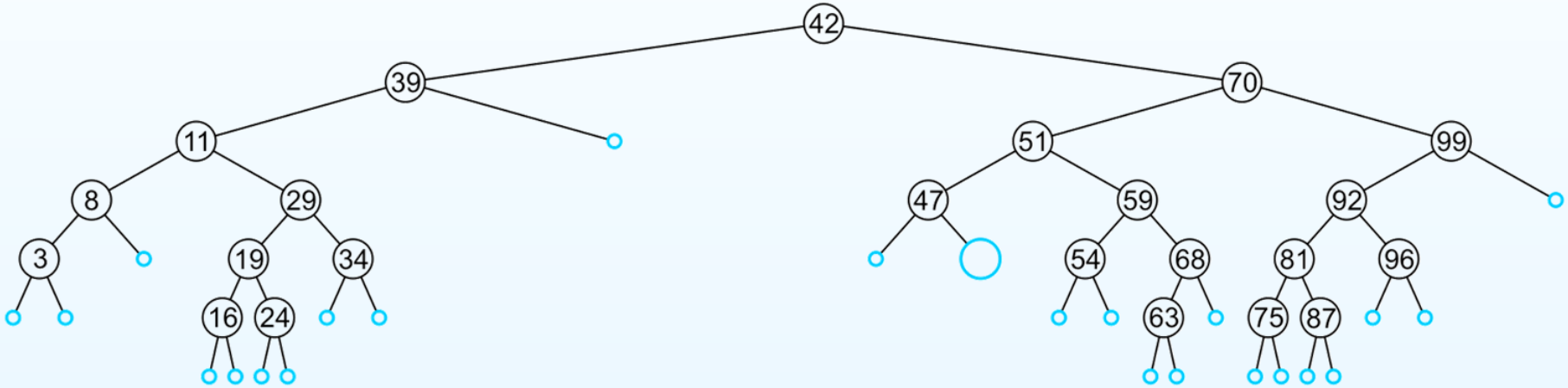
The values which may be inserted at any empty node depend on the surrounding nodes

Insertion

An insertion will be performed at a leaf node:

- Any empty node is a possible location for an insertion

For example, this node my hold **48**, **49**, or **50**



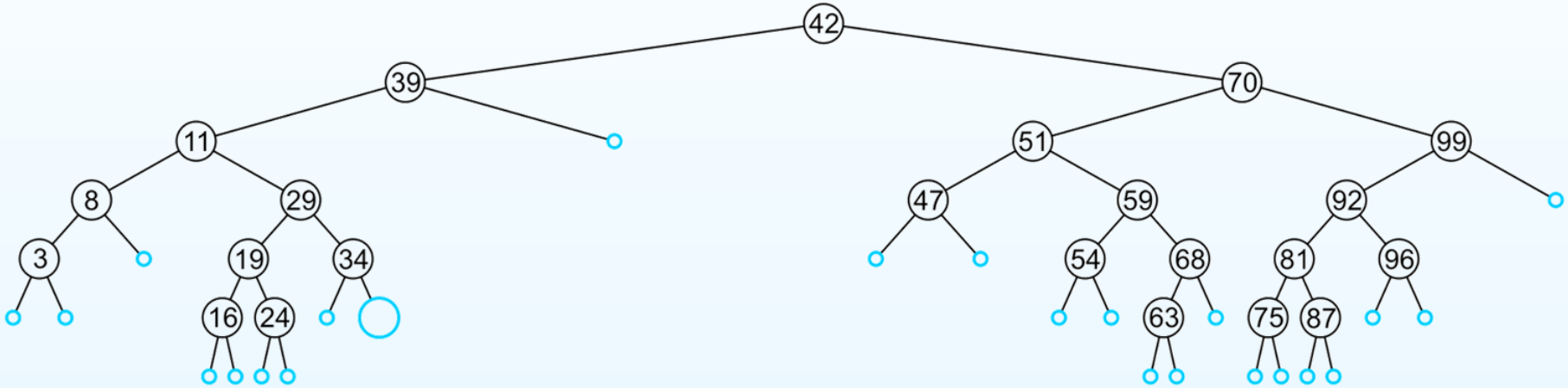
The values which may be inserted at any empty node depend on the surrounding nodes

Insertion

An insertion will be performed at a leaf node:

- Any empty node is a possible location for an insertion

For example, this node my hold **35**, **36**, **37**, or **38**



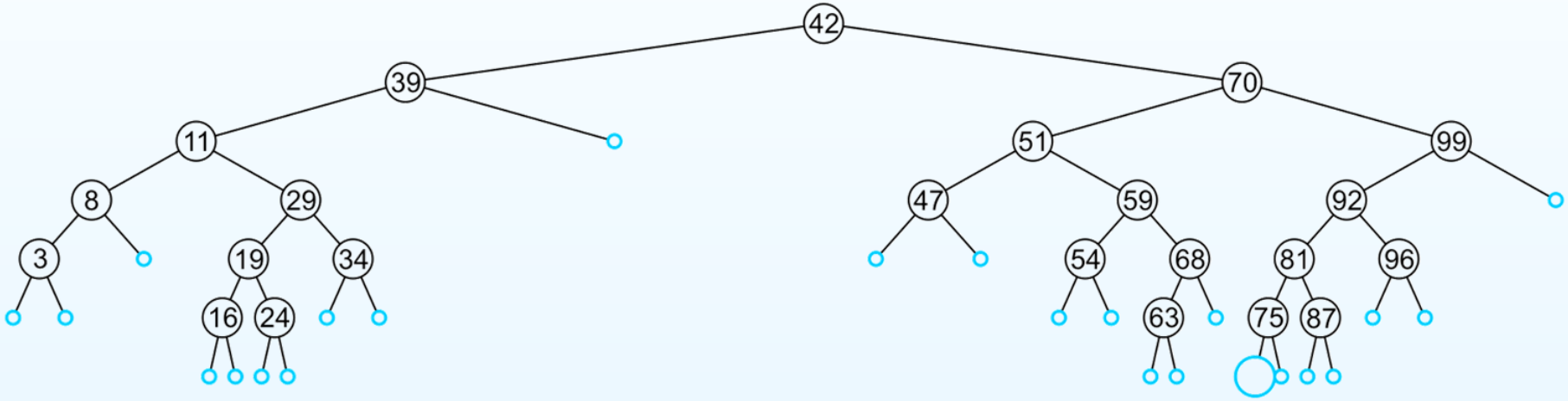
The values which may be inserted at any empty node depend on the surrounding nodes

Insertion

An insertion will be performed at a leaf node:

- Any empty node is a possible location for an insertion

For example, this node my hold **71**, **72**, **73**, or **74**



The values which may be inserted at any empty node depend on the surrounding nodes

Insertion

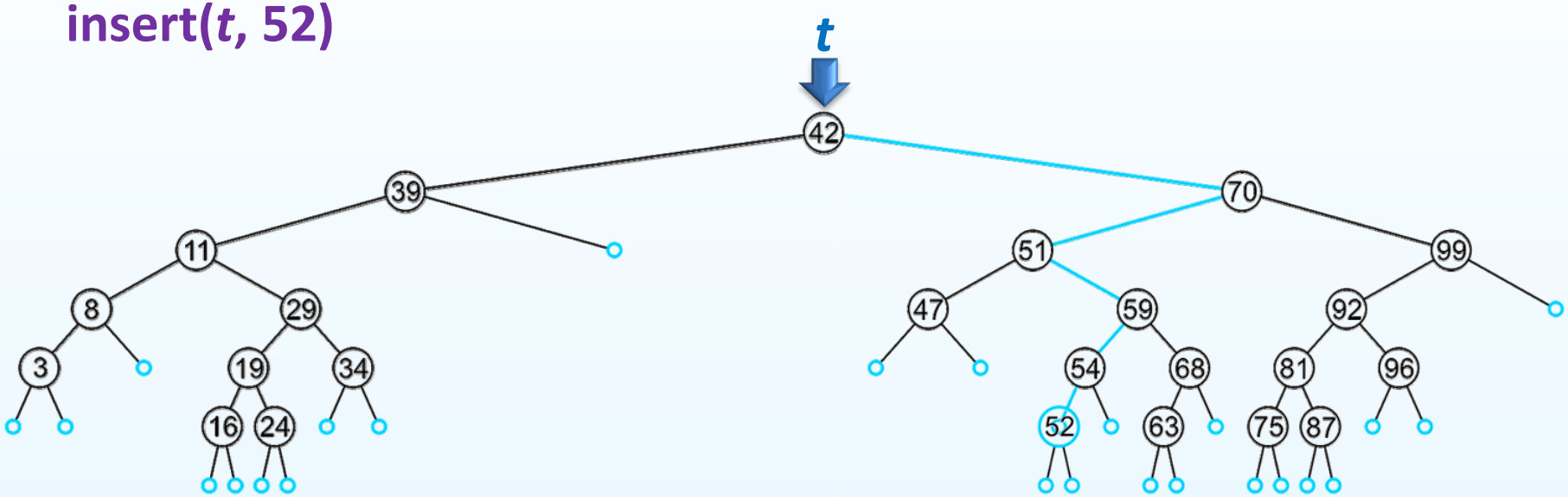
Like the `find()` operation, we will step through the tree

- If we find the object already, we will return the tree (no duplicate)
- Otherwise, we will arrive at an empty node and then insert the object into that location

The `insert()` Operation

Insert a node *v* in BST *t*

insert(*t*, 52)



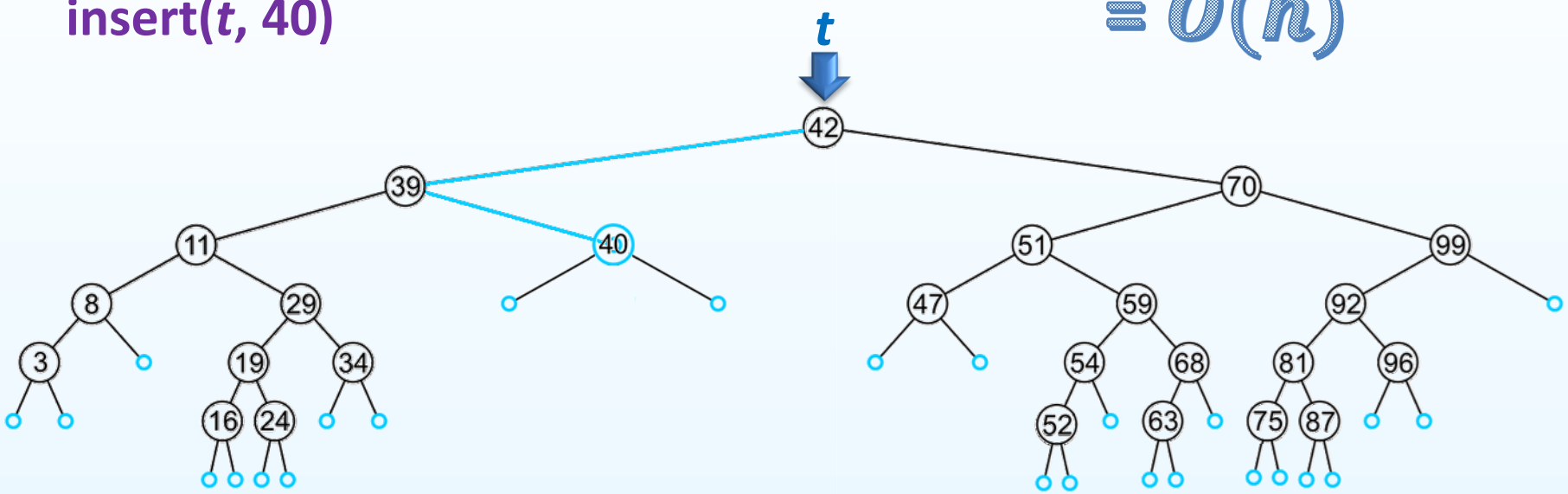
1. Traverse the tree until we found the left subtree of 54 is an empty node
2. A new leaf node is created and assigned to the left subtree of 54

The `insert()` Operation

Insert a node v in BST t

Running time $= O(h)$

insert(t, 40)



1. Traverse the tree until we found the right subtree of 39 is an empty node
2. A new leaf node is created and assigned to the right subtree of 39

Exercise 1 (5 mins.)

- In the given order, insert these objects into an initially empty binary search tree:

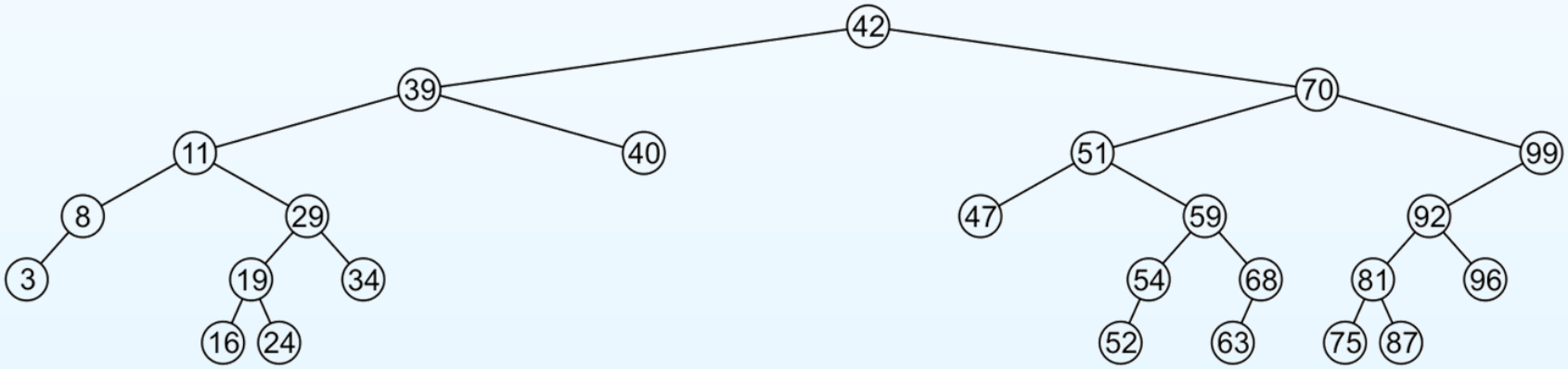
  31 45 36 14 52 42 6 21 73 47 26 37 33 8

- What values could be placed:
  - To the left of 21?
  - To the right of 26?
  - To the left of 47?

- Which values could be inserted to increase the height of the tree?

deletion

- A node being deleted is not always going to be a leaf node
- There are three possible scenarios:
  - The node is a leaf node, or
  - The node has exactly one child, or
  - The node has two children

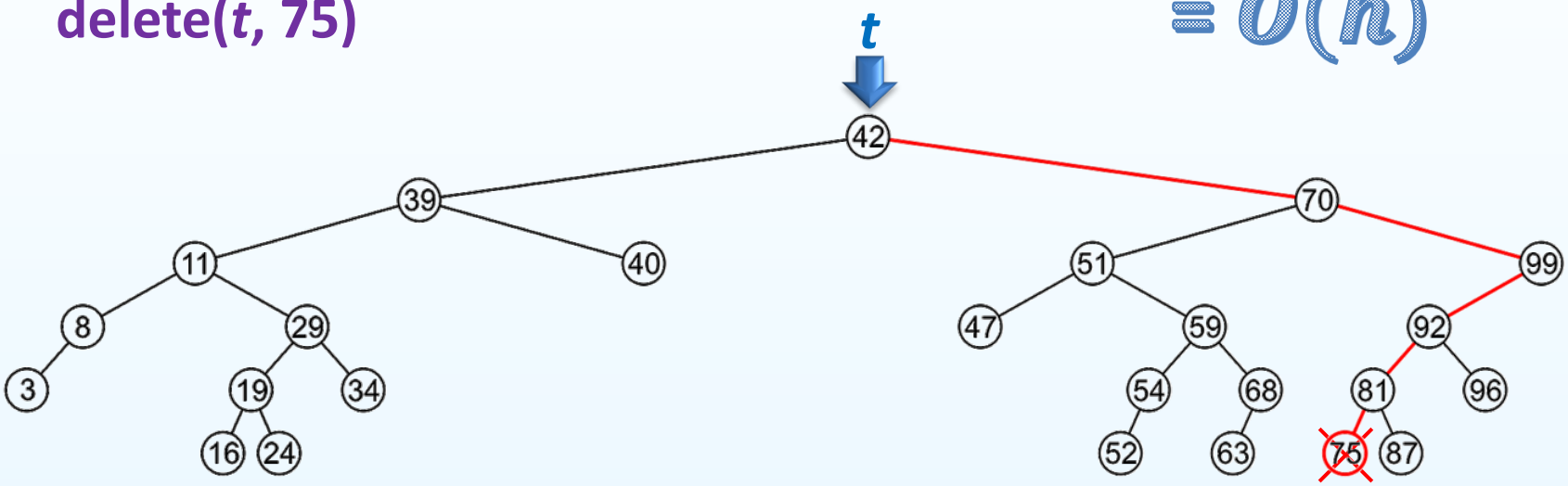


The `delete()` Operation (Case #1)

Remove a node v in BST t

Running time $= O(h)$

delete(t, 75)

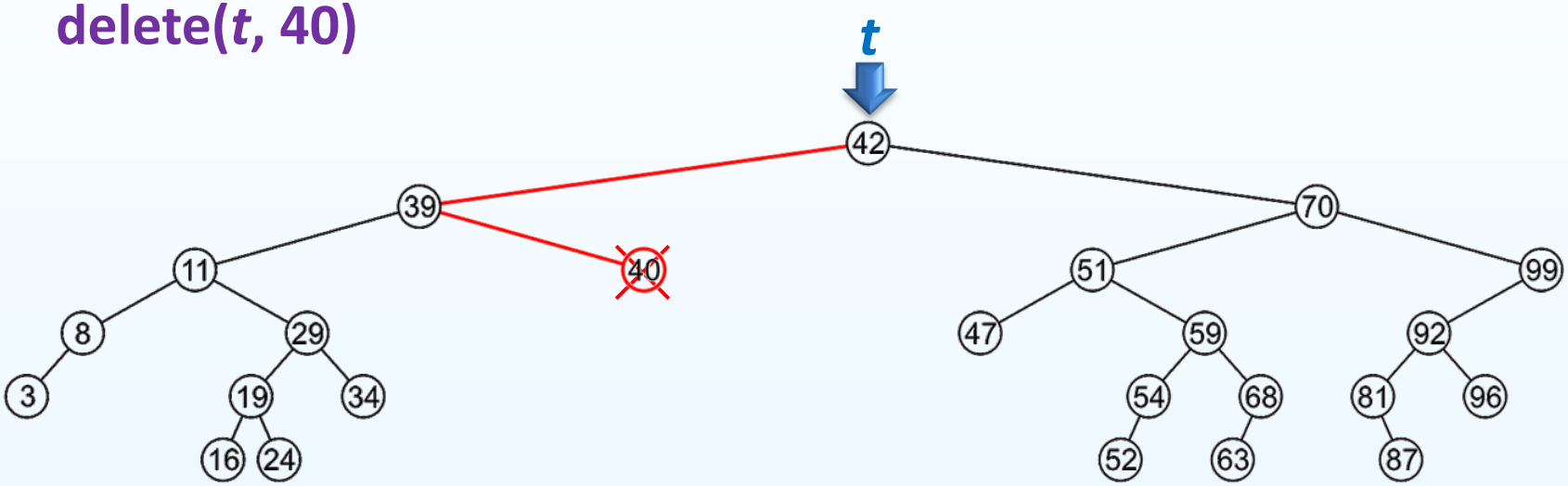


1. Traverse the tree until we found 75
2. A leaf node is simply removed and assigned NULL to the left child of 81

The `delete()` Operation (Case #1)

Remove a node *v* in BST *t*

delete(*t*, 40)

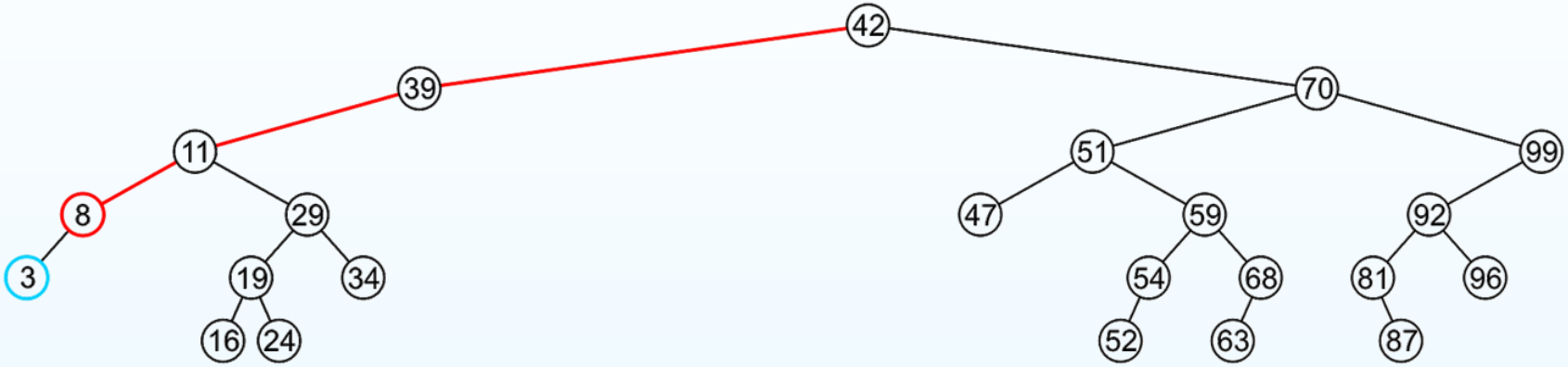


1. Traverse the tree until we found 40
2. A leaf node is simply removed and assigned NULL to the right child of 39

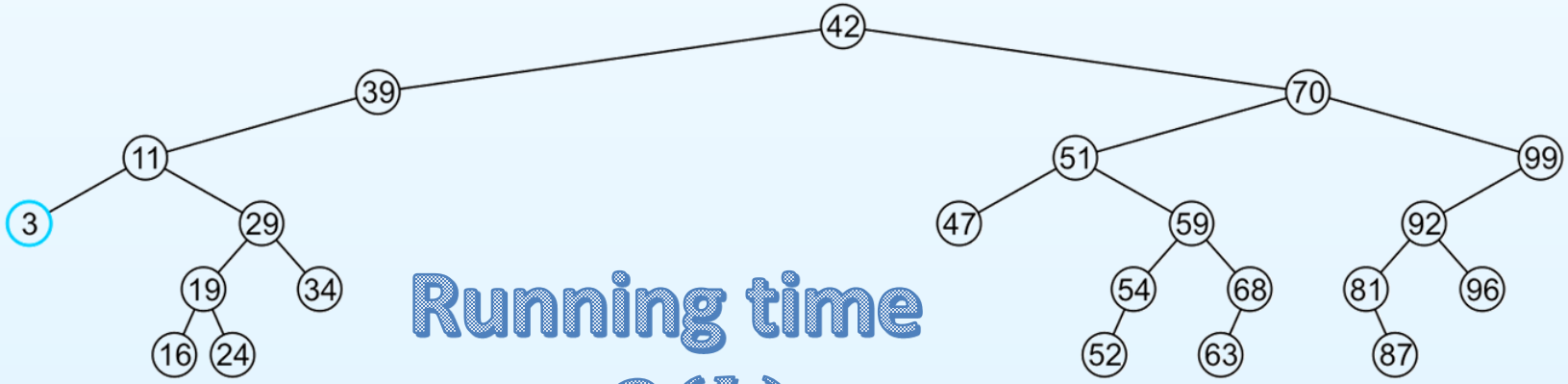
The `delete()` Operation (Case #2)

delete(*t*, 8)

1. Traverse the tree until we found 8, so it has one left child



2. Let the left child of 11 point to 3, and then delete 8

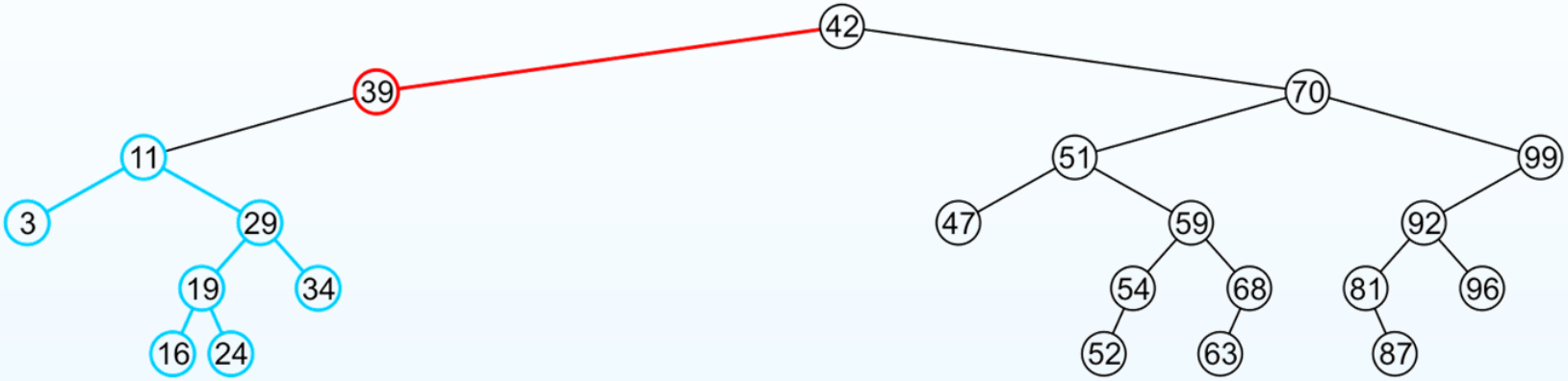


Running time = $O(h)$

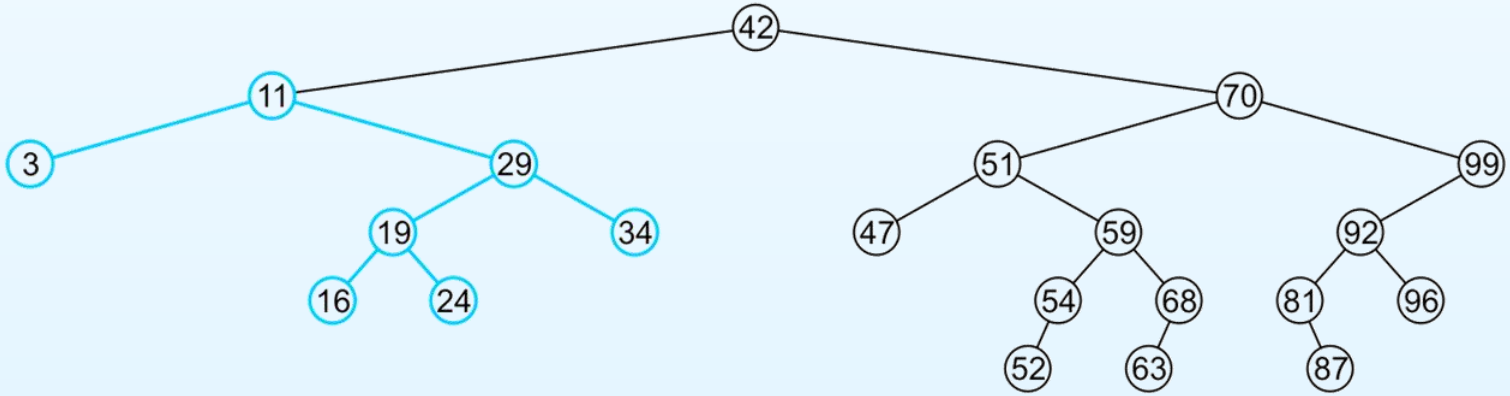
The `delete()` Operation (Case #2)

delete(*t*, 39)

1. Traverse the tree until we found 39, so it has one left child



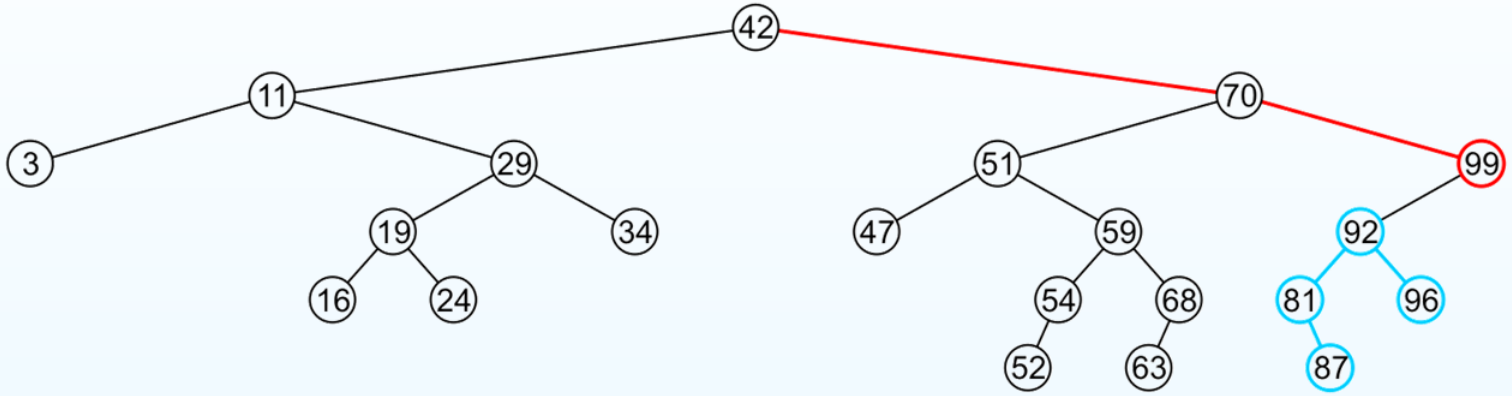
2. Let the left child of 42 point to 11, and then delete 39



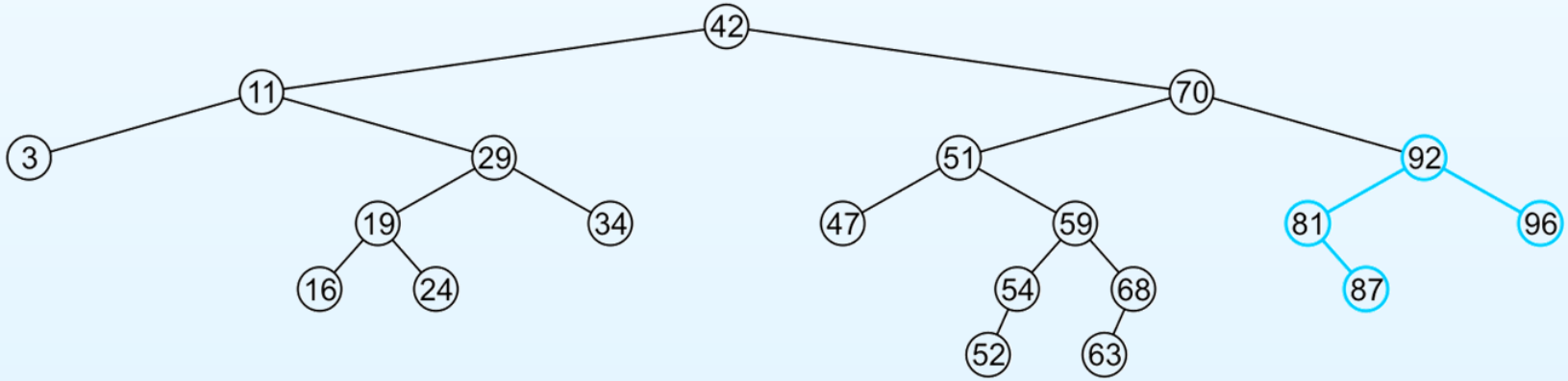
The `delete()` Operation (Case #2)

delete(*t*, 99)

1. Traverse the tree until we found 99, so it has one left child



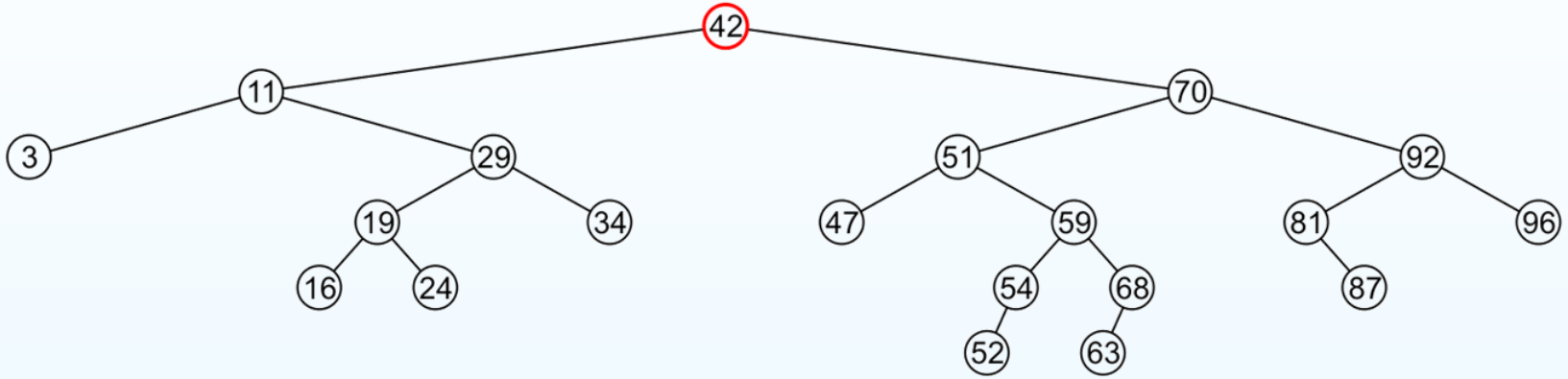
2. Let the right child of 70 point to 92, and then delete 99



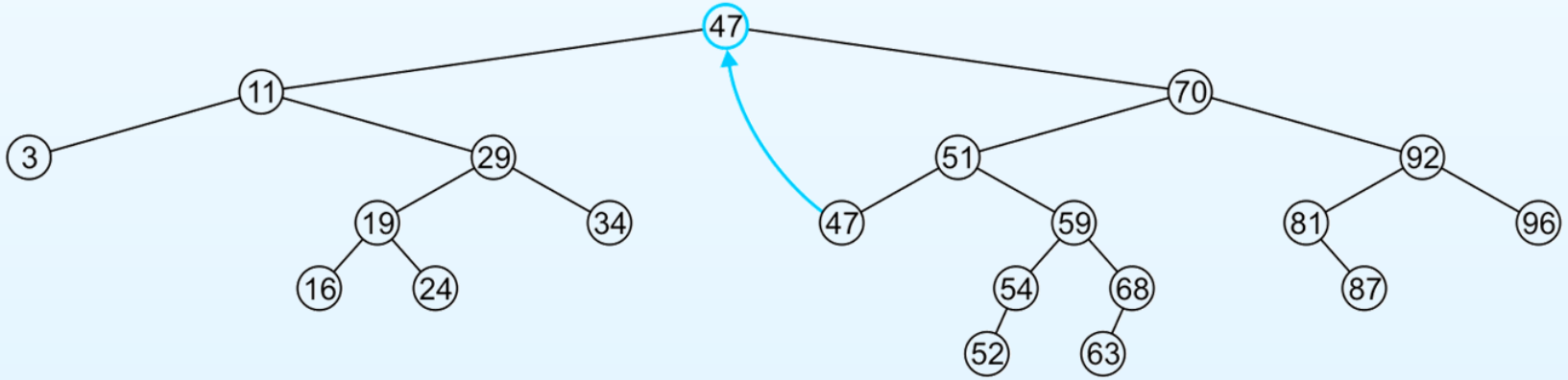
The `delete()` Operation (Case #3)

delete(*t*, 42)

1. Traverse the tree until we found 42, so it has two children



2. Replace 42 with the **minimum** object (say, 47) in the right subtree

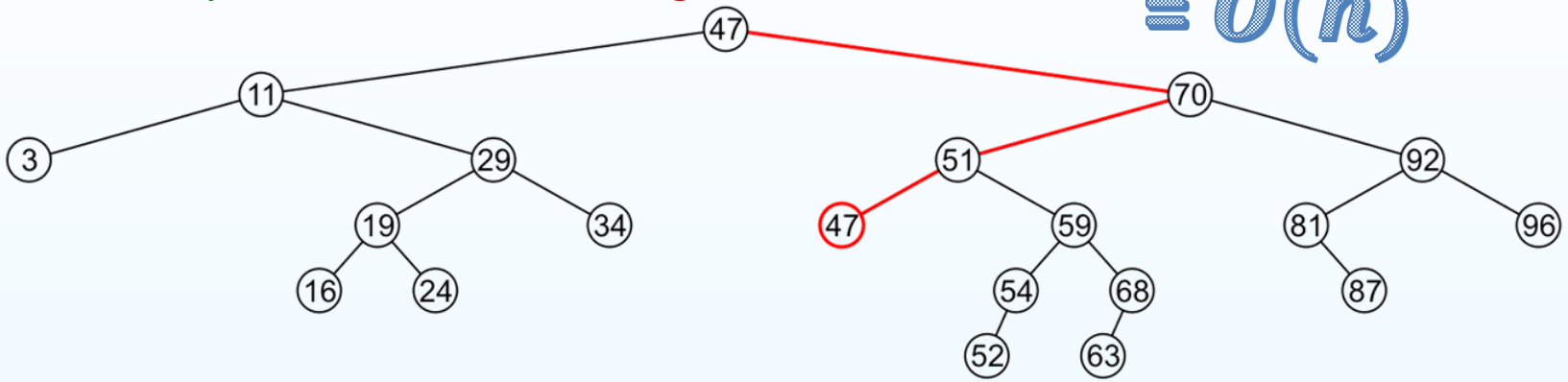


The `delete()` Operation (Case #3)

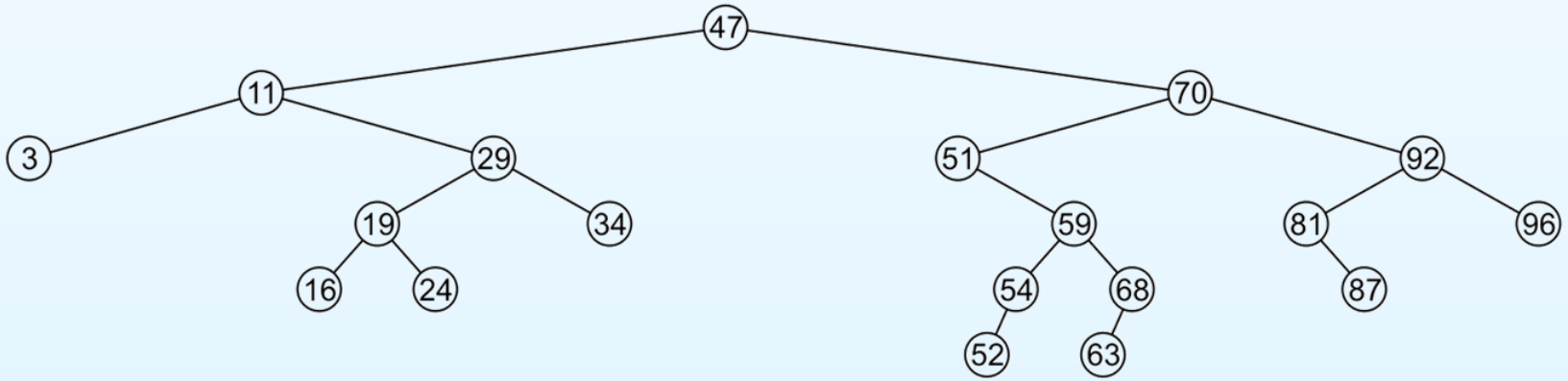
delete(*t*, 42)

Running time = $O(h)$

3. Recursively delete 47 from the **right subtree**



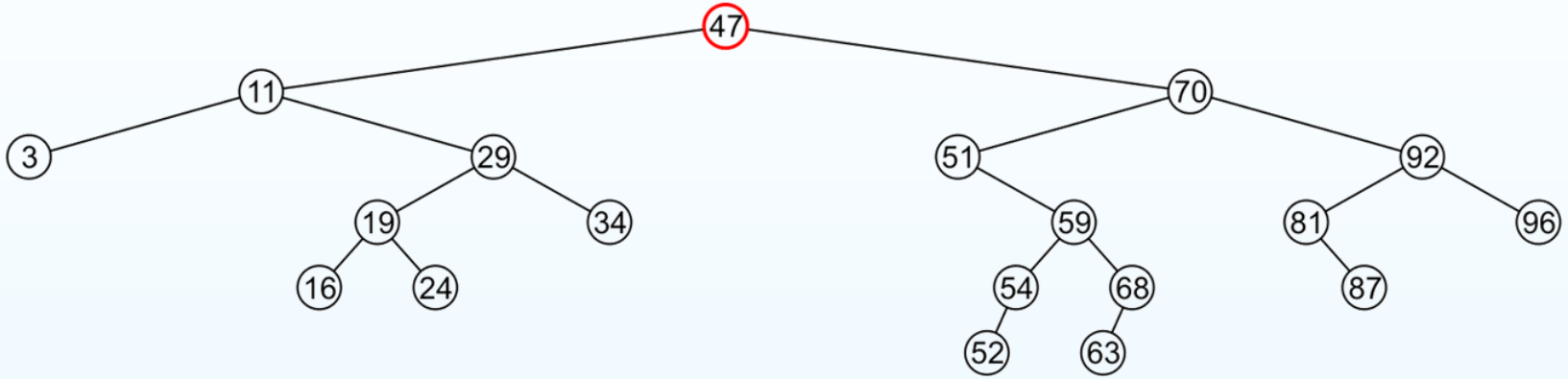
4. We found that it is a **leaf** and simply deleted, then set NULL to left child of 51



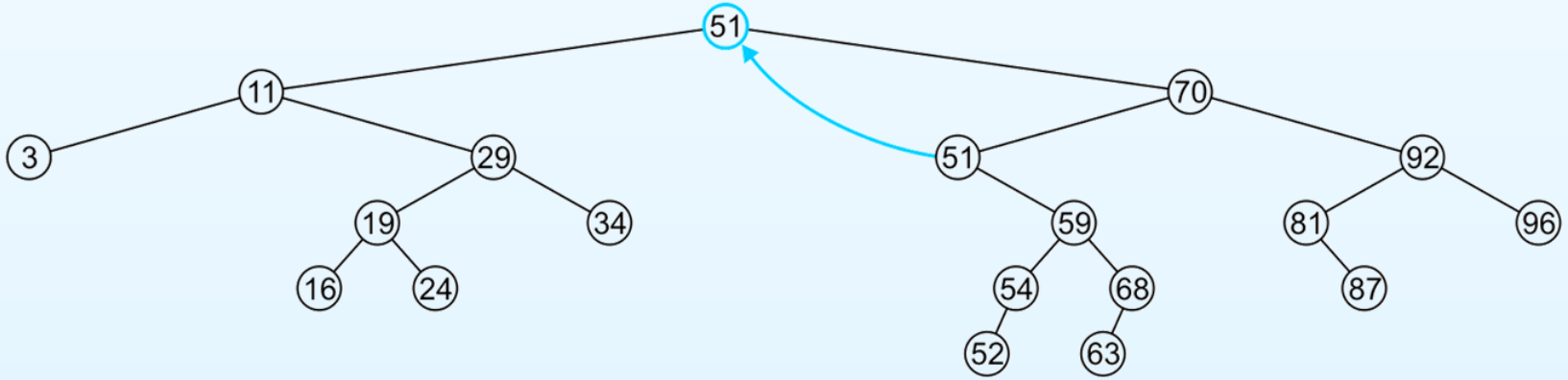
The `delete()` Operation (Case #3)

delete(*t*, 47)

1. Traverse the tree until we found 47, so it has two children



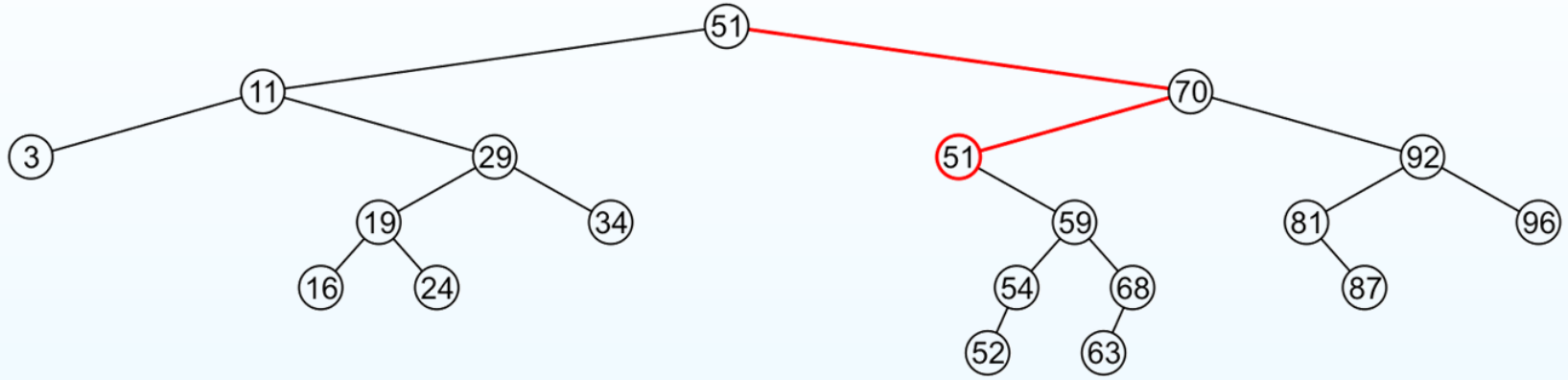
2. Replace 47 with the **minimum** object in the right subtree



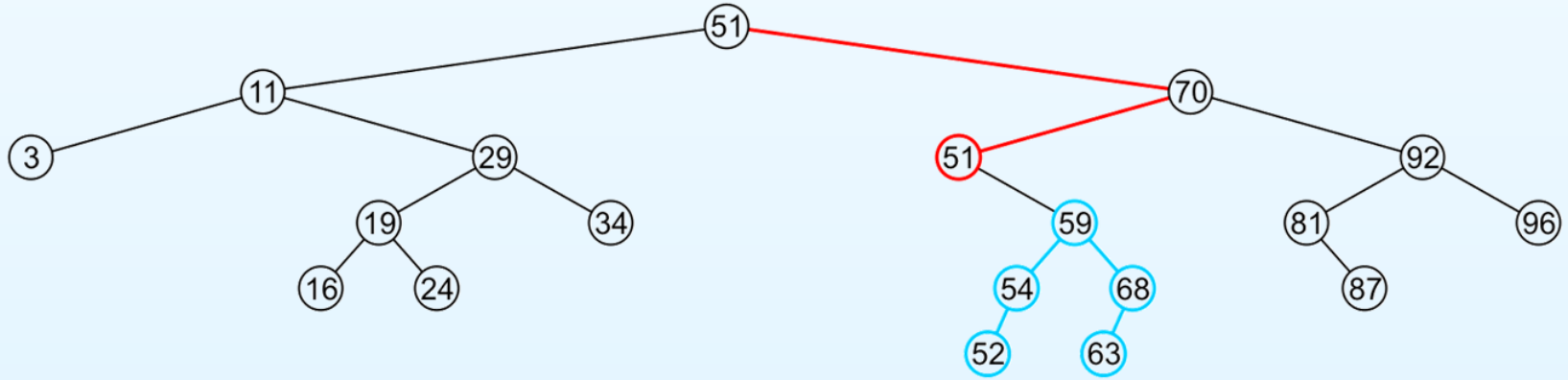
The `delete()` Operation (Case #3)

delete(*t*, 47)

3. Recursively delete 51 from the **right subtree**



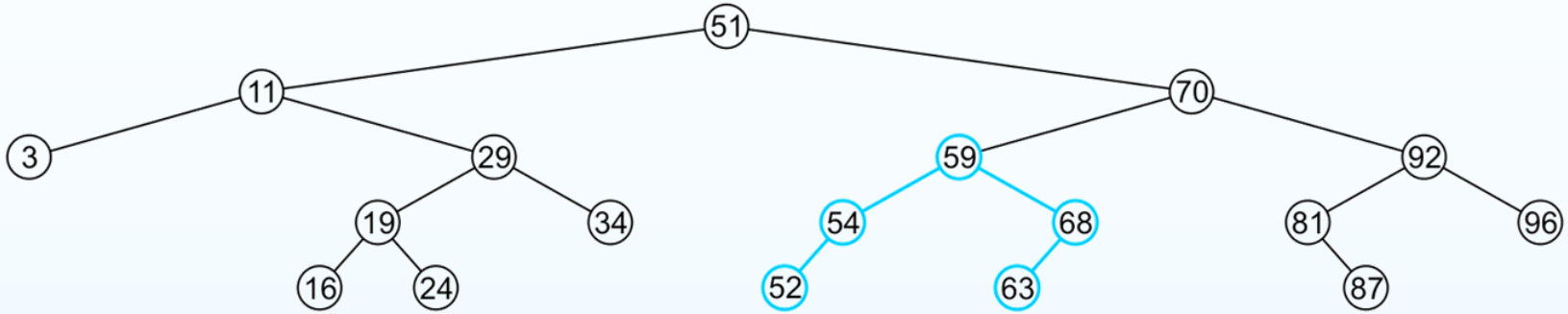
4. We found that it has one right child



The `delete()` Operation (Case #3)

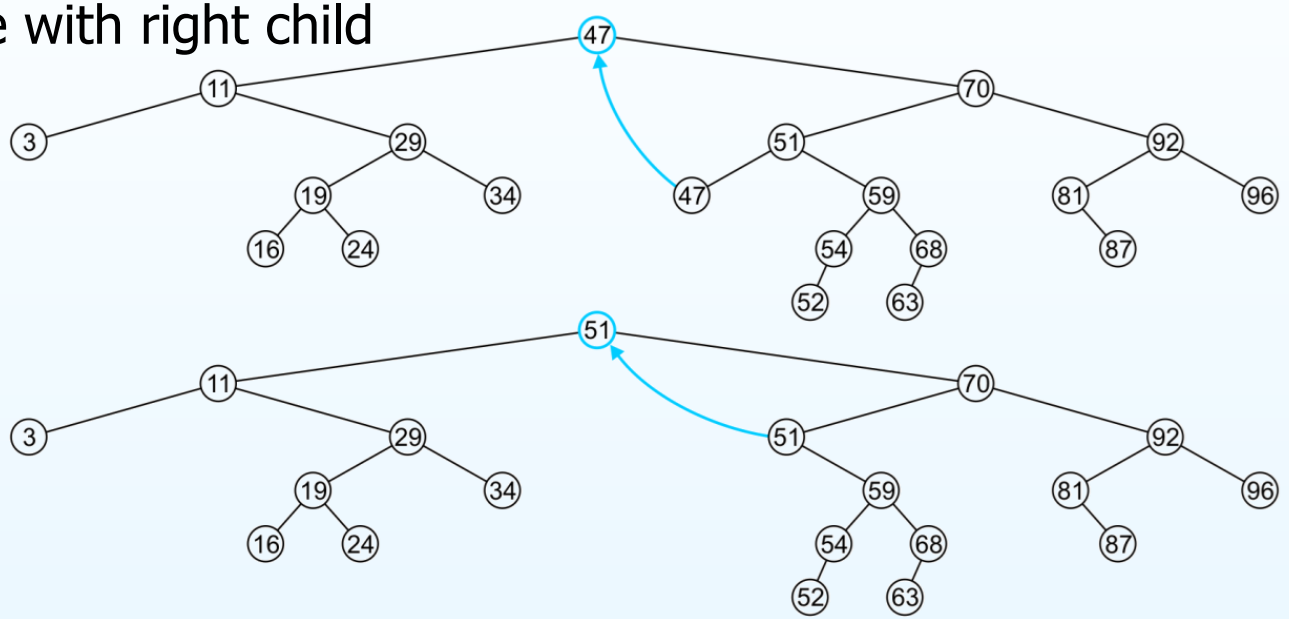
delete(*t*, 47)

5. Let the left child of 70 point to 59, and then delete that old 51



Deletion

- In two examples of removing a full node, we promoted:
  - A node with no children
  - A node with right child



- Is it possible, in removing a full node, to promote a node with two children?
  - No. If that node had a left subtree, that subtree would contain a smaller value

Exercise 2 (5 mins.)

- In the binary search tree generated previously:

  31 45 36 14 52 42 6 21 73 47 26 37 33 8

- Update the tree if we perform the following operations:
  - `delete(t,47)`
  - `delete(t,21)`
  - `delete(t,45)`
  - `delete(t,31)`
  - `delete(t,36)`

Outline

- Binary Search Trees

- Balanced Trees
  - AVL Trees
  - Red-Black Trees
  - Weight-Balanced Trees

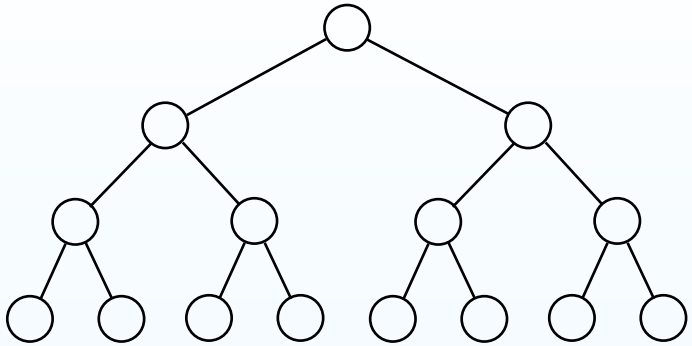
Background

- The running times depend on the height of the trees
- As stated in the previous lecture and the BSTs today:
  - The best-case height is $\Theta(\log n)$
  - The worst-case height is $O(n)$
- The average height of a randomly generated binary search tree is actually $\Theta(\log n)$
  - However, following random insertions and deletions, the average height tends to increase to $\Theta(\sqrt{n})$

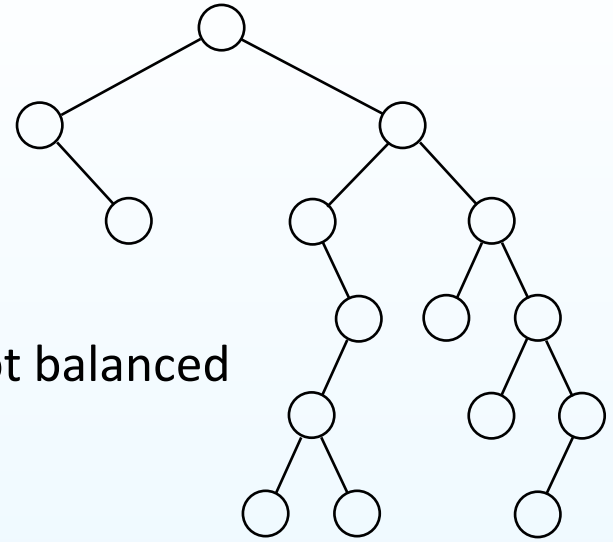
Requirement for Balance

- We want to ensure that the running times never fall into $\omega(\log n)$
- Requirement:
  - We must maintain a height which is $\Theta(\log n)$
- To do this, we will define an idea of balance

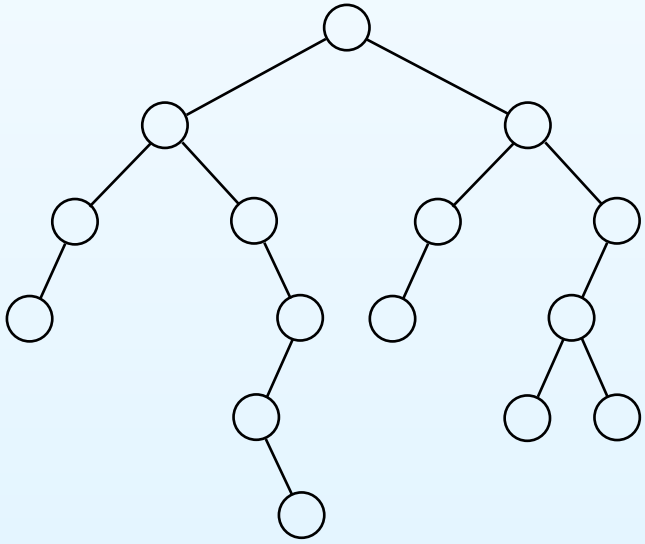
Are the following trees balanced?



Perfect binary trees are definitely balanced



Not balanced

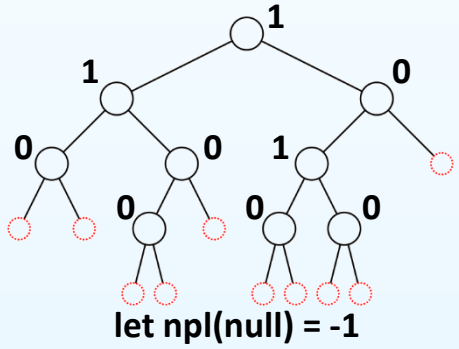


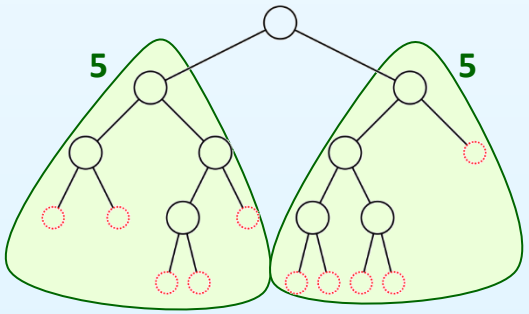
The root seems balanced, but not for some others

Definition for Balance

- We must develop a quantitative definition for balance which can be applied

- The balance may be defined by:
  - Height balancing: comparing the heights of the two subtrees
    - e.g., AVL trees
  - Null-path-length balancing: comparing the null-path-length of each the two subtrees (length to the closest null subtree/empty node)
    - e.g., red-black trees
  - Weight balancing: comparing the number of null subtrees in each of the two subtrees
    - e.g., weight-balanced trees





Outline

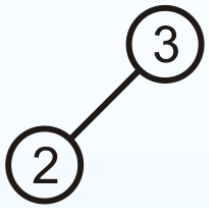
- Binary Search Trees

- Balanced Trees
  - AVL Trees
  - Red-Black Trees
  - Weight-Balanced Trees

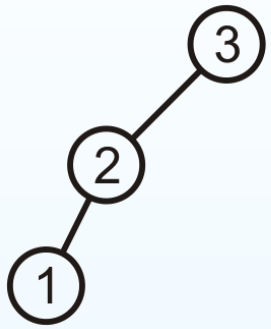
Prototypical Example #1

We will demonstrate how we can correct for imbalances

1. Start with this tree

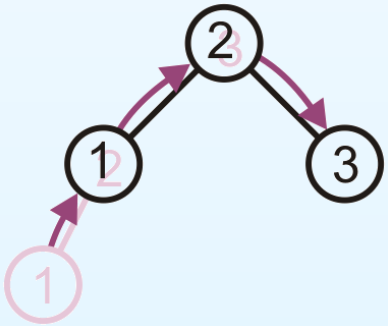


2. Insert 1

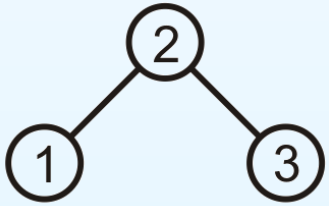


How can we fix this?

3. Promote 2 to root, demote 3 to be 2's right child, and 1 remains 2's left child



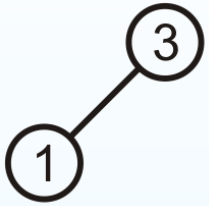
4. The result is a perfect



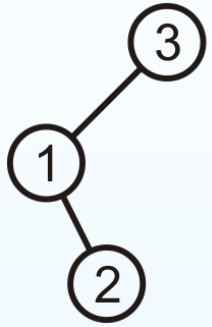
Prototypical Example #2

We will demonstrate how we can correct for imbalances

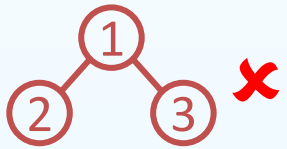
1. Start with this tree



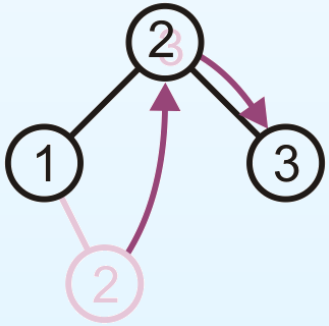
2. Insert 2



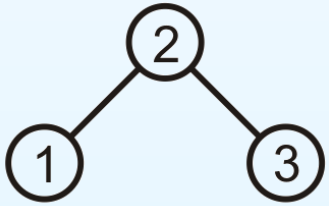
How can we fix this?



3. Promote 2 to root, assign 1 to be 2's left child and 3 to be 2's right child



4. The result is, again, a perfect



All the steps are the basis for the AVL trees

AVL Trees

- We will focus on the first balanced strategy: AVL Trees
  - Named after Adelson-Velskii and Landis

- Balance is defined by comparing the height of the two subtrees

- Recall:
  - An empty tree has height `-1`
  - A tree with a single node has height `0`

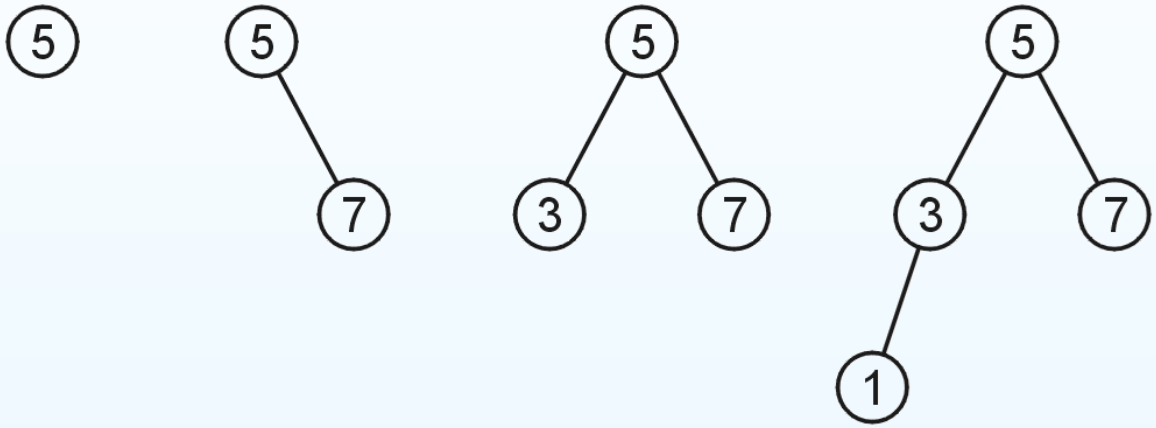
Definition

A binary search tree is said to be an AVL-balanced tree if

- The difference in the heights between the left and right subtrees is at most 1, and
- Both subtrees are themselves AVL trees

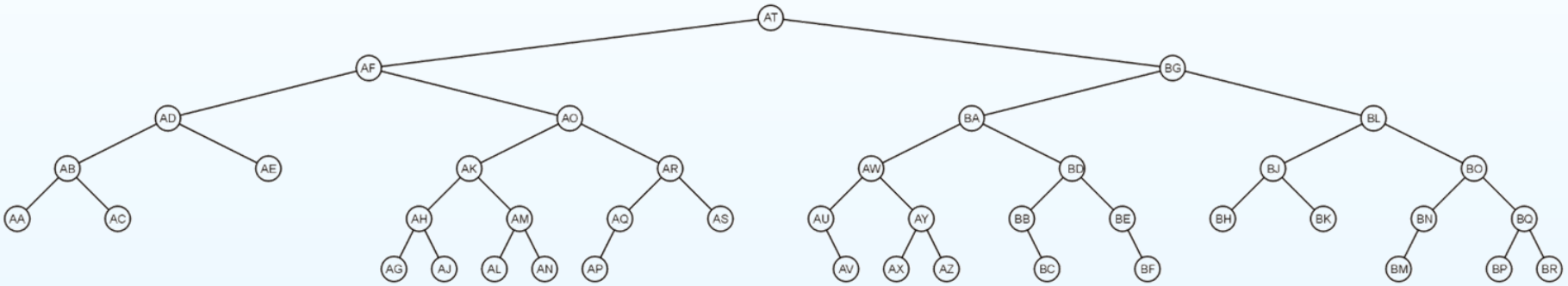
Examples of AVL Trees

AVL trees with 1, 2, 3, and 4 nodes:



Examples of AVL Trees

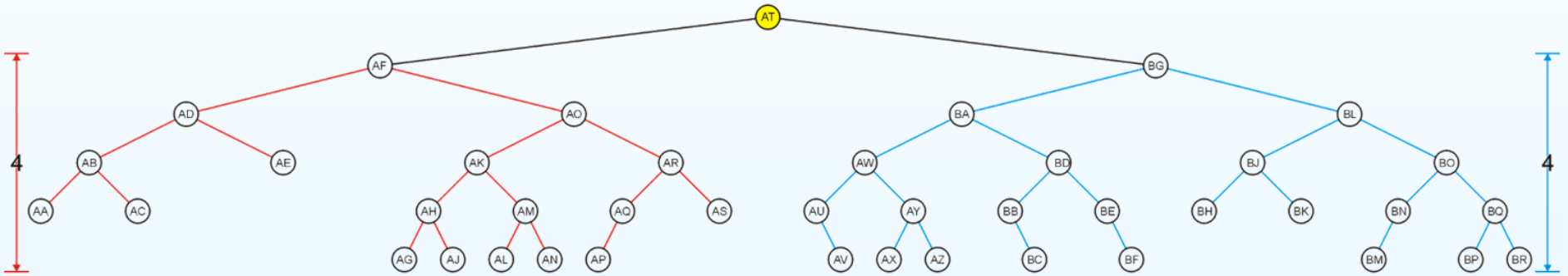
Here is a larger AVL tree with 42 nodes:



Examples of AVL Trees

Here is a larger AVL tree with 42 nodes:

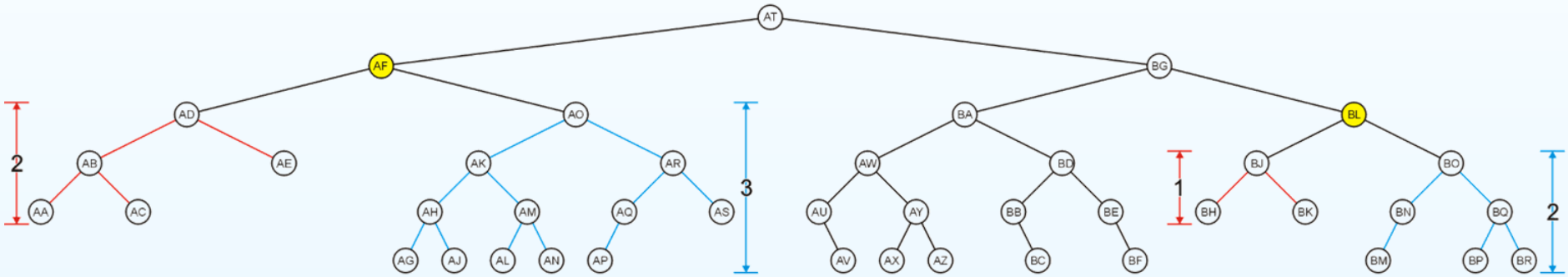
- The root node is AVL-balanced since both subtrees are of height 4



Examples of AVL Trees

Here is a larger AVL tree with 42 nodes:

- The root node is AVL-balanced since both subtrees are of height 4
- All other nodes (e.g., AF and BL) are AVL-balanced since their subtrees differ in height by at most one



Height of an AVL Tree

By the definition of complete trees, any complete BST is an AVL tree

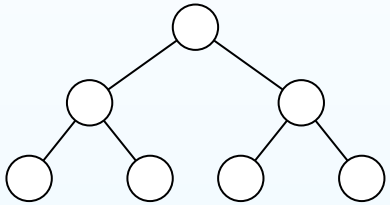
Thus, an upper bound on the number of nodes in an AVL tree of height h is a perfect BST with $2^{h+1} - 1$ nodes

What is a lower bound?

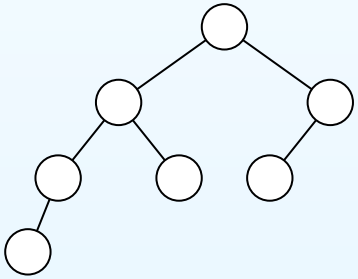
- i.e., the fewest number of nodes in an AVL tree of height h
- In other words, the worst-case AVL tree of height h

Height of an AVL Tree

Ex1: Suppose we have 7 nodes. What are the best- and worst-case of height h for an AVL tree?



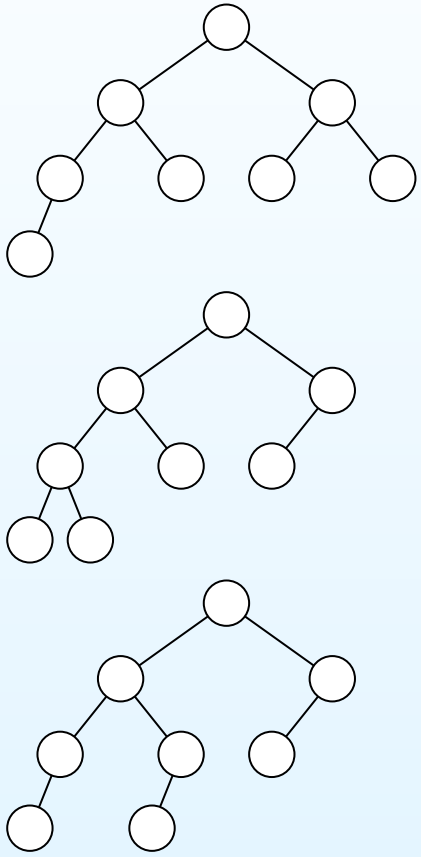
Best-case: $h = 2$



Worst-case: $h = 3$

Height of an AVL Tree

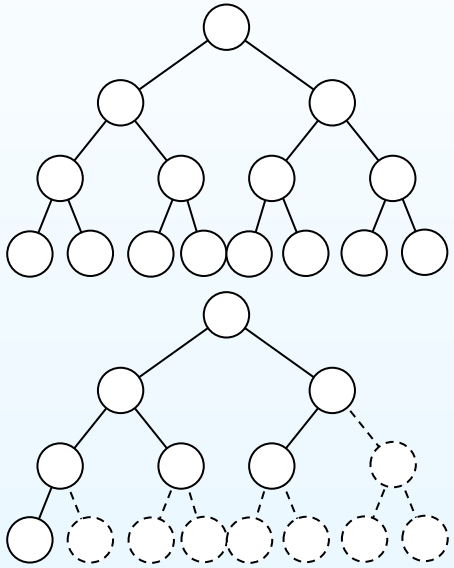
Ex2: Suppose we have 8 nodes. What are the best- and worst-case of height h for an AVL tree?



Best- and worst-case: $h = 3$

Height of an AVL Tree

Ex3: Suppose we need an AVL tree of height $h = 3$. What is the maximum and minimum number of nodes?



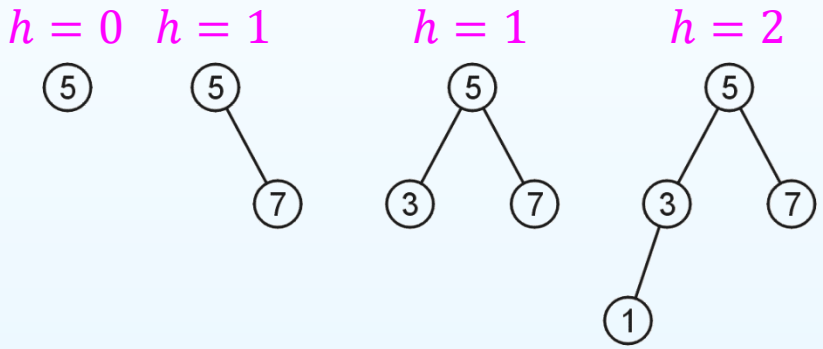
Max: $n = 15$
(best-case of height $h = 3$):

Min: $n = 7$
(worst-case of height $h = 3$)

Height of an AVL Tree

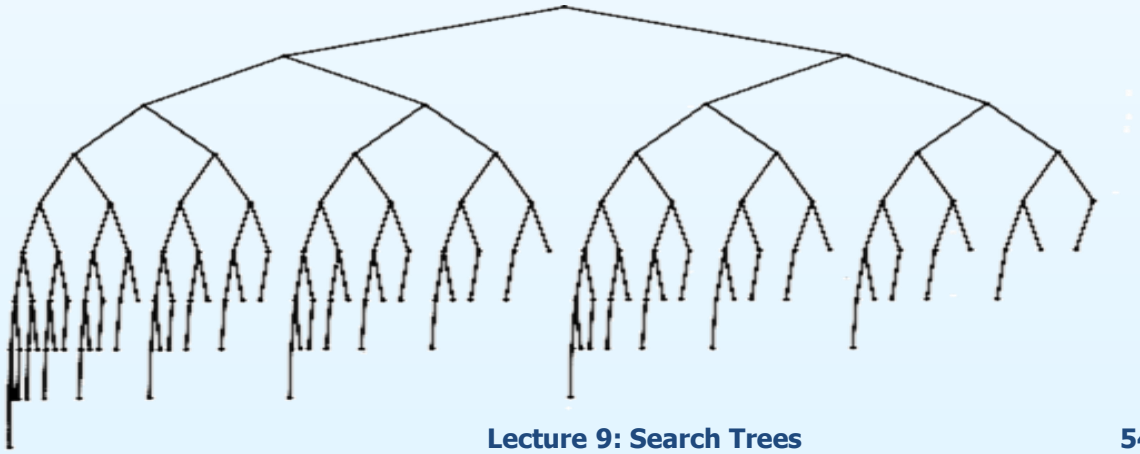
Let $F(h)$ be the fewest number of nodes in an AVL Tree of height h

Consider the following AVL trees:



$$F(0) = 1$$
$$F(1) = 2$$
$$F(2) = 4$$
$$\vdots$$

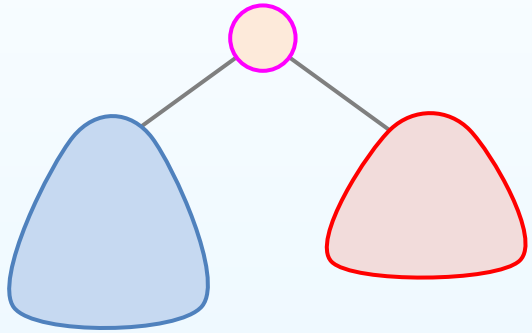
Can we find $F(h)$?



Height of an AVL Tree

The worst-case AVL tree of height h would have

- A worst-case AVL tree of height $h-1$ on one side, and
- A worst-case AVL tree of height $h-2$ on the other side, and
- The root node



So that we get $F(h) = F(h-1) + F(h-2) + 1$

Height of an AVL Tree

We have a recurrence relation:

$$F(h) = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ F(h-1) + F(h-2) + 1 & h > 1 \end{cases}$$

Note:

$$\begin{aligned}
F(0) = 1 &\rightarrow F(0) + 1 = 2 \\
F(1) = 2 &\rightarrow F(1) + 1 = 3 \\
F(2) = 4 &\rightarrow F(2) + 1 = 5 \\
F(3) = 7 &\rightarrow F(3) + 1 = 8 \\
F(4) = 12 &\rightarrow F(4) + 1 = 13 \\
F(5) = 20 &\rightarrow F(5) + 1 = 21 \\
F(6) = 33 &\rightarrow F(6) + 1 = 34 \\
\vdots &\rightarrow F(h) + 1 = (F(h-1) + 1) + (F(h-2) + 1)
\end{aligned}$$

$$F(h) + 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^{h+3}$$

$F(h) + 1$ is a Fibonacci number

Height of an AVL Tree

- This is approximately

$$F(h) \approx 1.8944\phi^h - 1$$

  where $\phi \approx 1.6180$ is the golden ratio
  - That is, the fewest #nodes $F(h) = \Omega(\phi^h)$

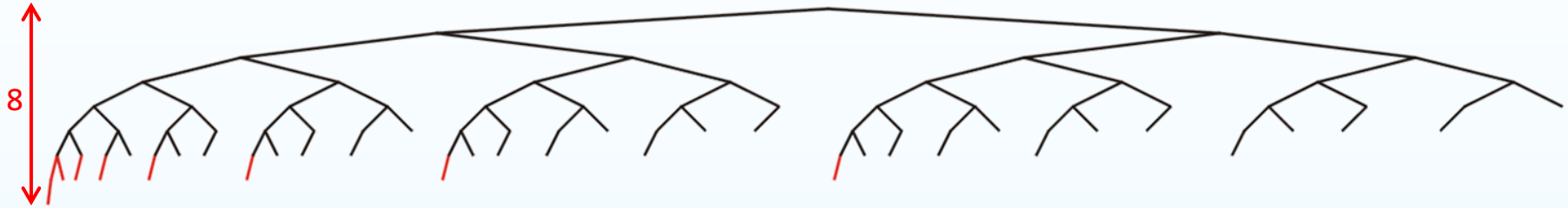
- Thus, we may find the maximum height h for a given n:

$$h \approx \log_\phi \left(\frac{n+1}{1.8944}\right) \approx 1.4404 \lg(n+1) - 1.3277$$

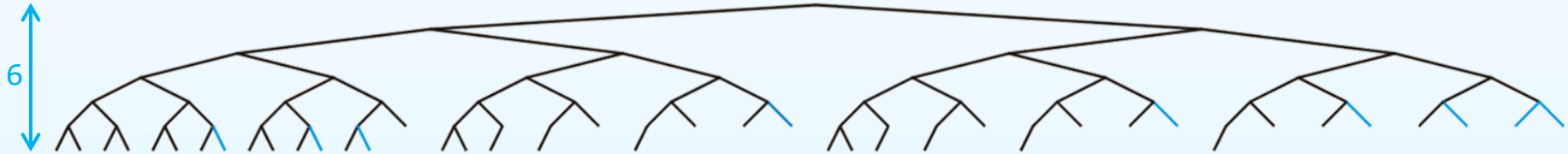
- Recall that a complete AVL tree provides the minimum height $h = \lfloor \lg n \rfloor$

Examples: Height of an AVL Tree

- If $n = 88$,
  the worst-case has $h \approx 1.4404 \lg(88 + 1) - 1.3277 = 8$



the best-case has $h = \lfloor \lg 88 \rfloor = 6$



- So that the worst- and best-case scenarios differ in height by only 2

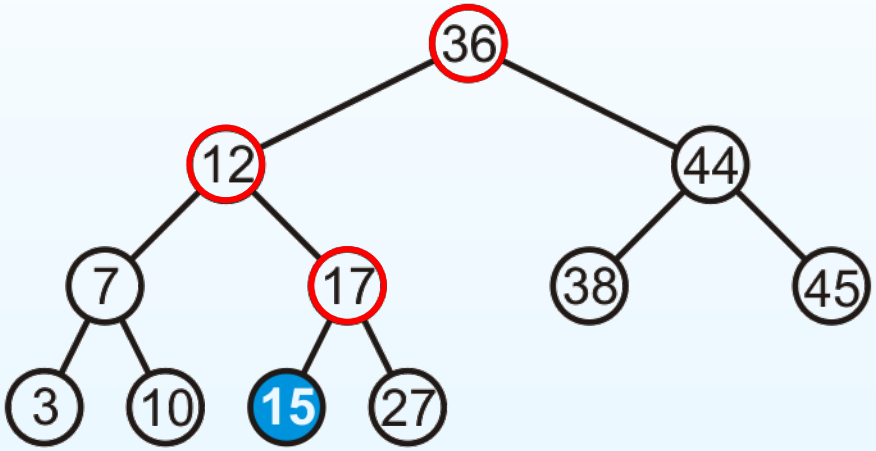
Examples: Height of an AVL Tree

- If $n = 10^6$, the bounds on h are
  - The maximum height: $1.4404 \lg(10^6 + 1) - 1.3277 \approx 27.38$
  - The minimum height: $\lfloor \lg 10^6 \rfloor = 19$

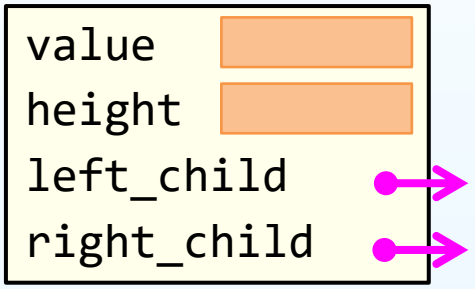
Maintaining Balance

Consider this AVL tree:

- Inserting 15 into the tree
  - Check whether nodes along the path (i.e., 17, 12, 36) is balanced
  - All other nodes are not affected



node




```
height(t) = max(height(t->left_child),height(t->right_child))+1
```

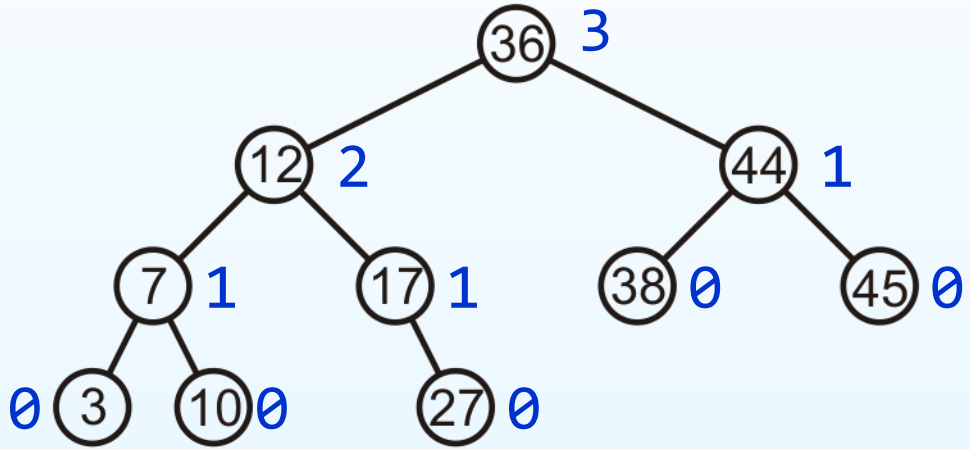
➡ Calling the function recursively will get $\Theta(n)$

➡ How can we get the height in $\Theta(1)$?

Insertion with Maintaining Balance

For an AVL-balanced tree, an insertion causes unbalanced if

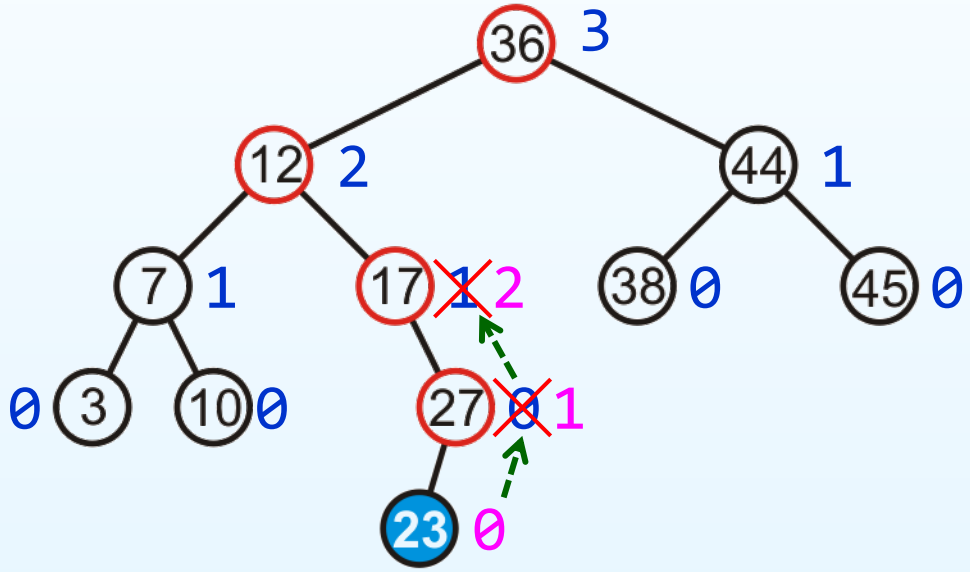
- The heights of the subtrees must differ by 1, and
- The insertion must increase the height of the deeper subtree by 1



Insertion with Maintaining Balance

Suppose we insert 23 into the initial AVL tree

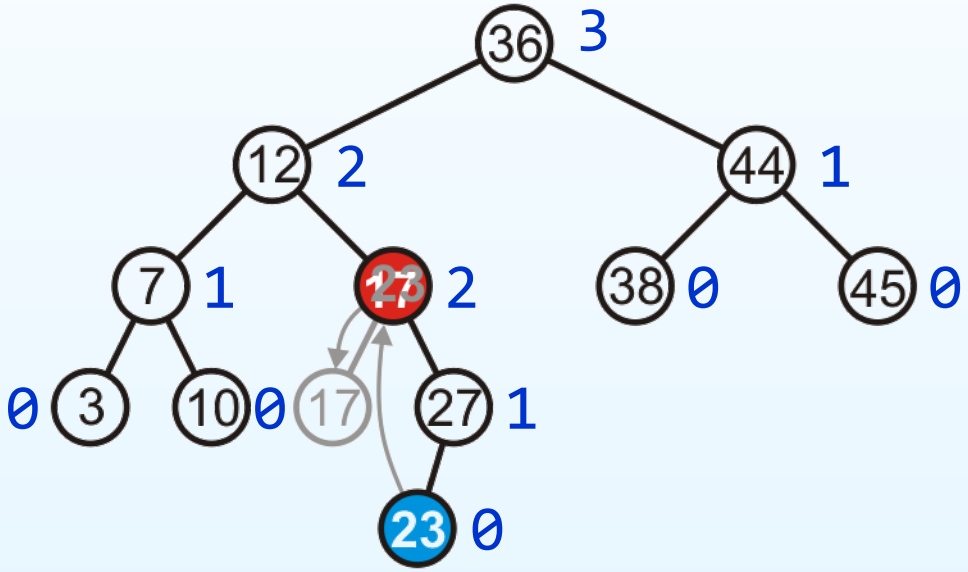
- The heights of each of the subtrees from root will be increase by 1
  - The process have been done after backtracking



Insertion with Maintaining Balance

Suppose we insert 23 into the initial AVL tree

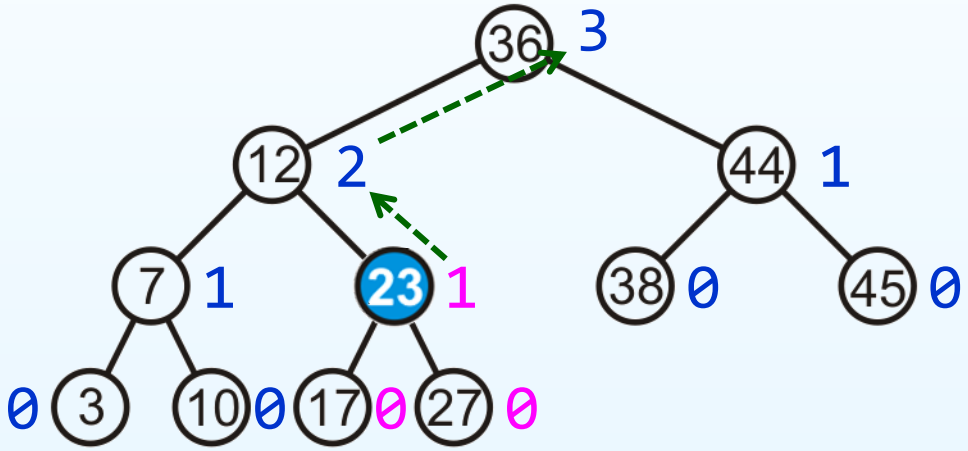
- We found that 17 is unbalanced
  - Promote 23 to where 17 is, and make 17 the left child of 23
  - Together with adjusting their heights



Insertion with Maintaining Balance

Suppose we insert 23 into the initial AVL tree

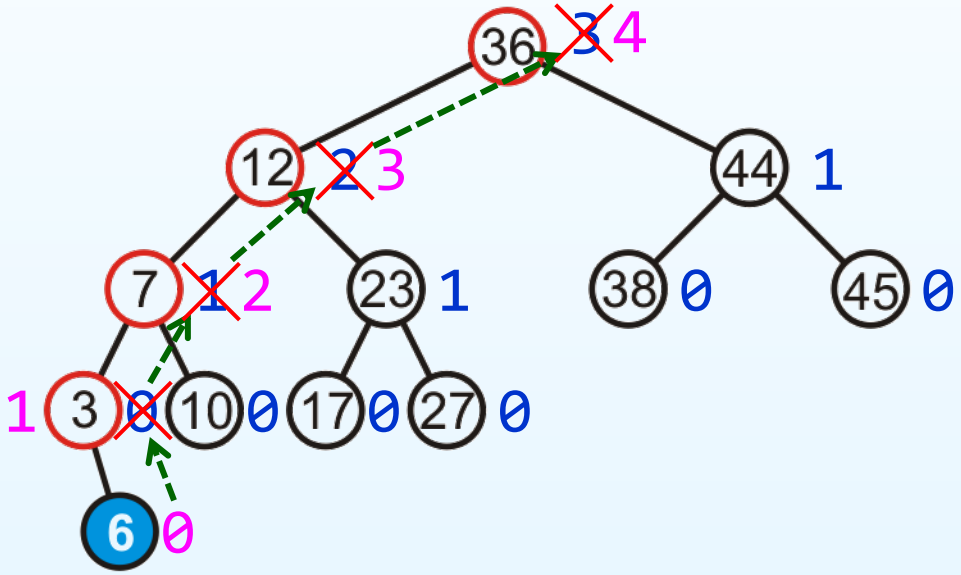
- So that we will get that balanced subtree
- Continue backtracking until the root
  - No more unbalance again, why?



Insertion with Maintaining Balance

For another example, suppose we insert 6

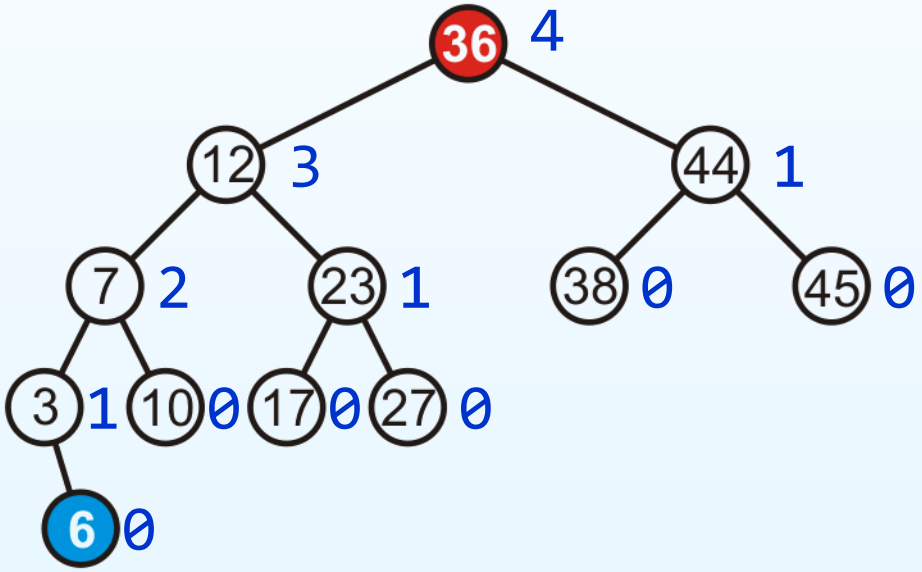
- After insertion, the process will backtrack to increase the height



Insertion with Maintaining Balance

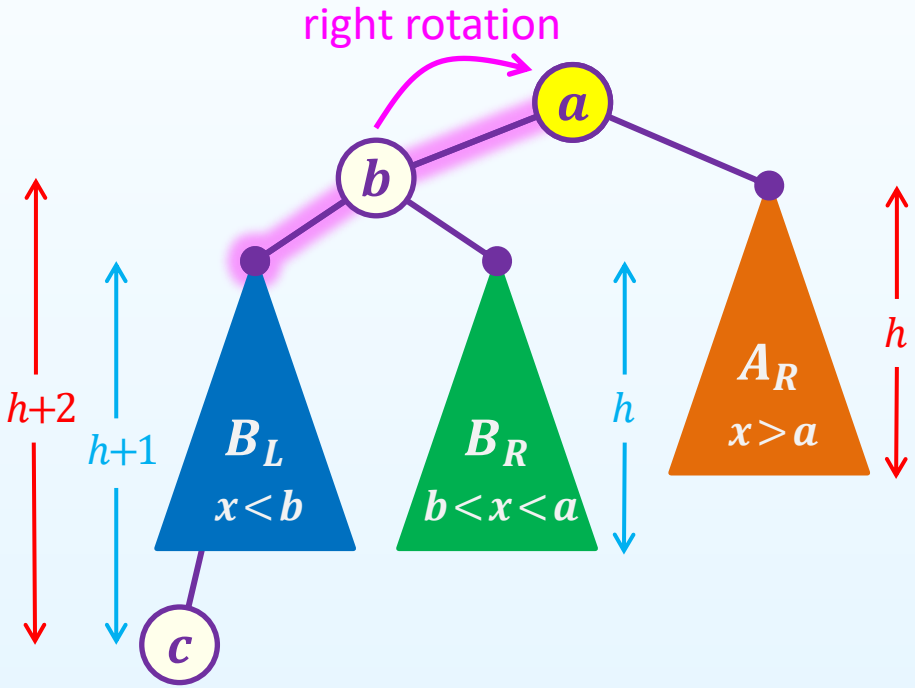
For another example, suppose we insert 6

- We found that 36 is unbalanced
  - To fix this, we will look at a general case …

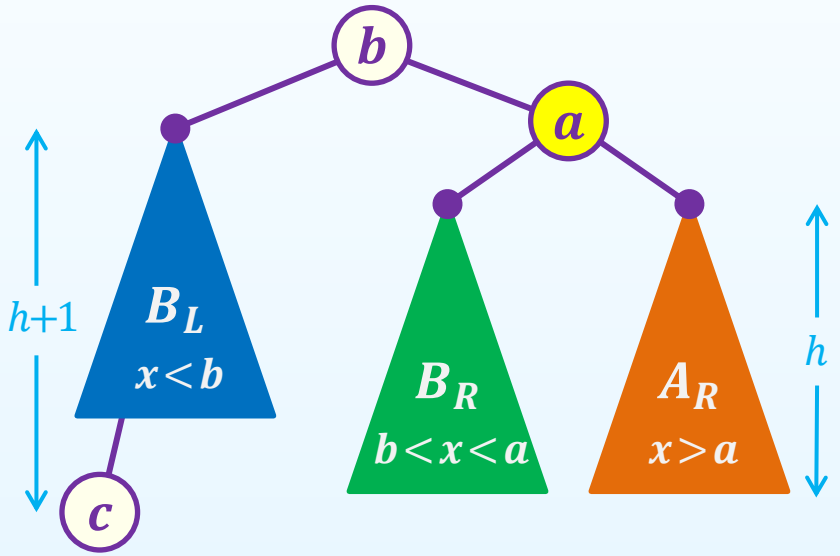


Case 1: Right Rotation

1. c is inserted to the **left** of b, then tree is unbalanced at a



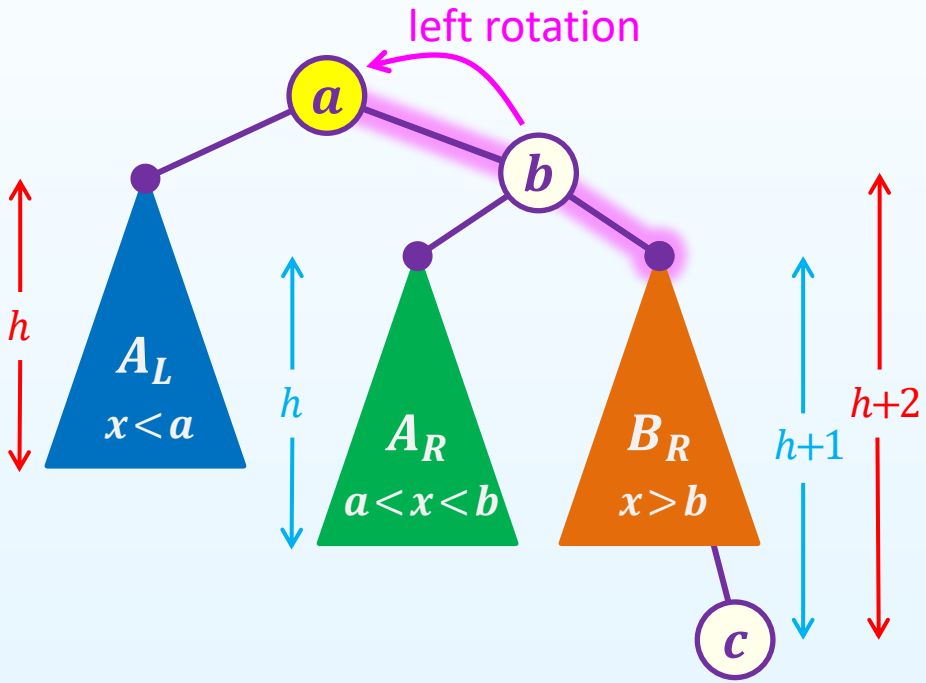
2. a's left holds b's right subtree, and b's right holds a



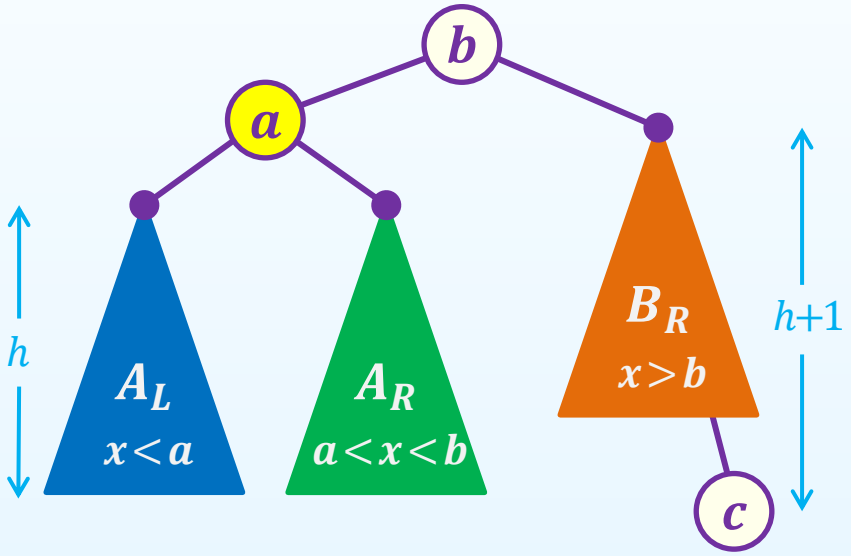
3. Update heights of a and b

Case 2: Left Rotation

1. c is inserted to the **right** of b, then tree is unbalanced at a



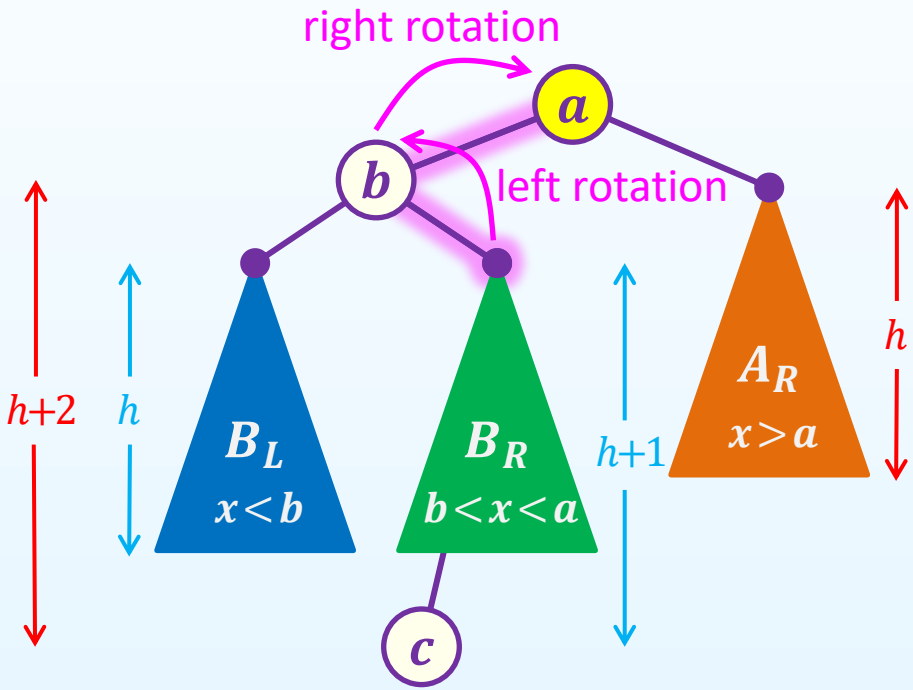
2. a's right holds b's left subtree, and b's left holds a



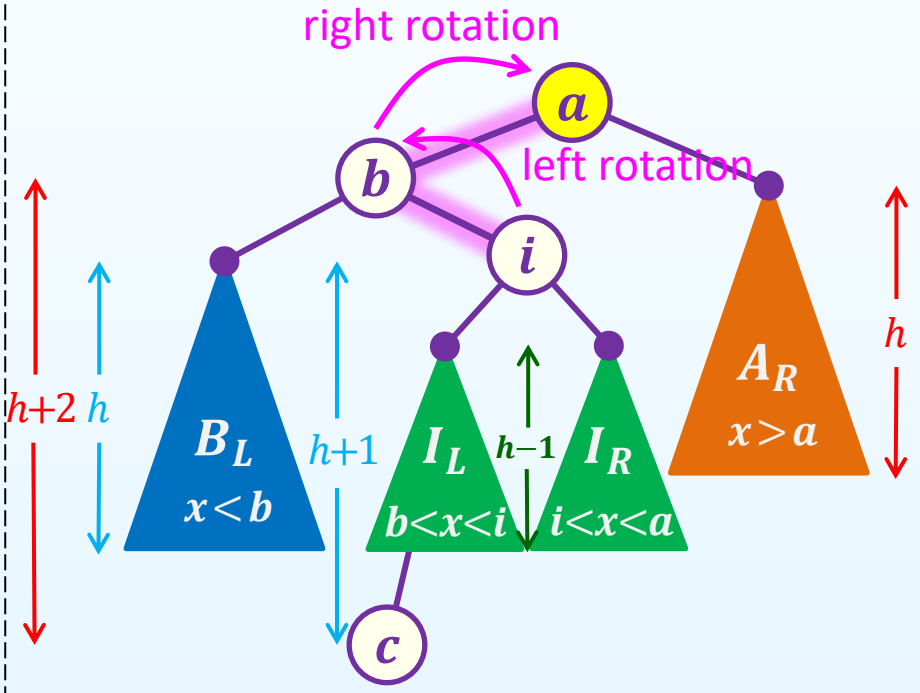
3. Update heights of a and b

Case 3: Left-Right Rotation

1. c is inserted to the **right** of b, then tree is unbalanced at a

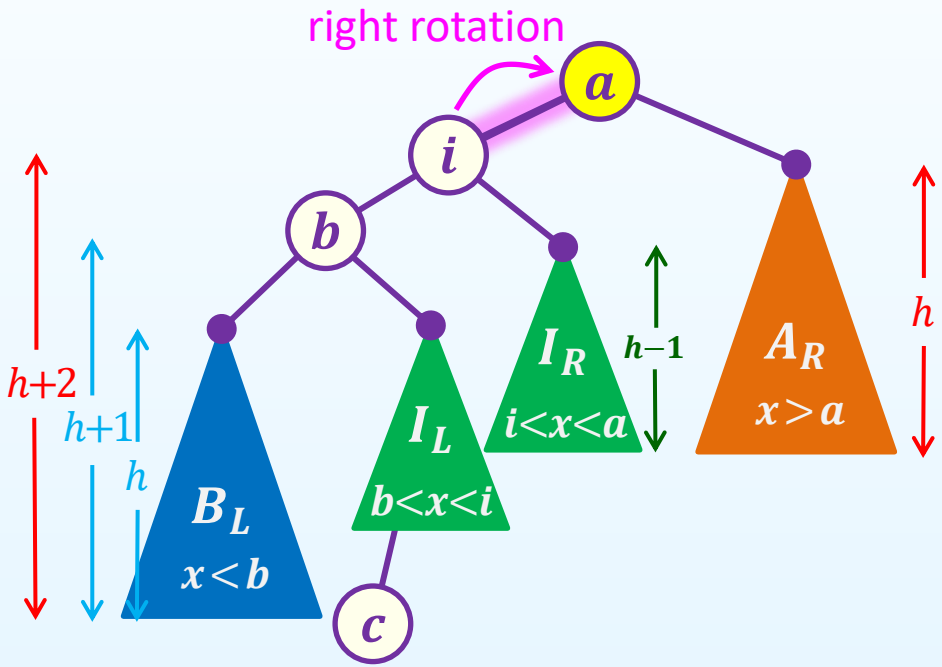


Suppose the root of B_R is i



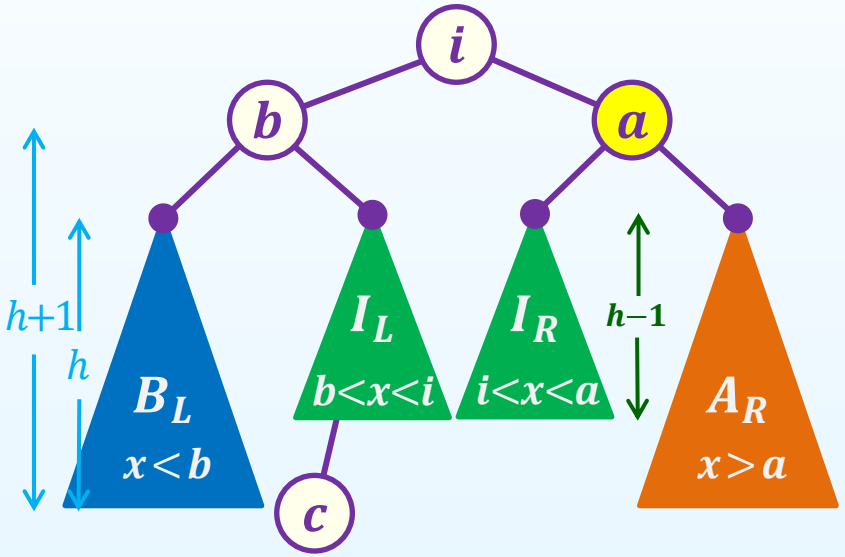
Case 3: Left-Right Rotation

2.1 b's right holds i's left subtree, and i's left holds b



2.2 Update heights of b and i

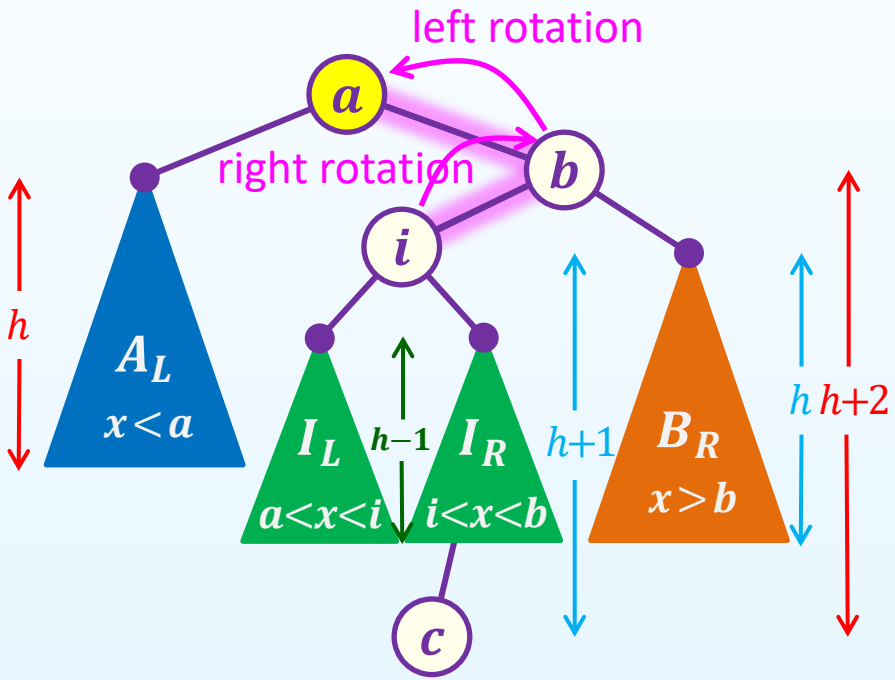
2.3 a's left holds i's right subtree, and i's right holds a



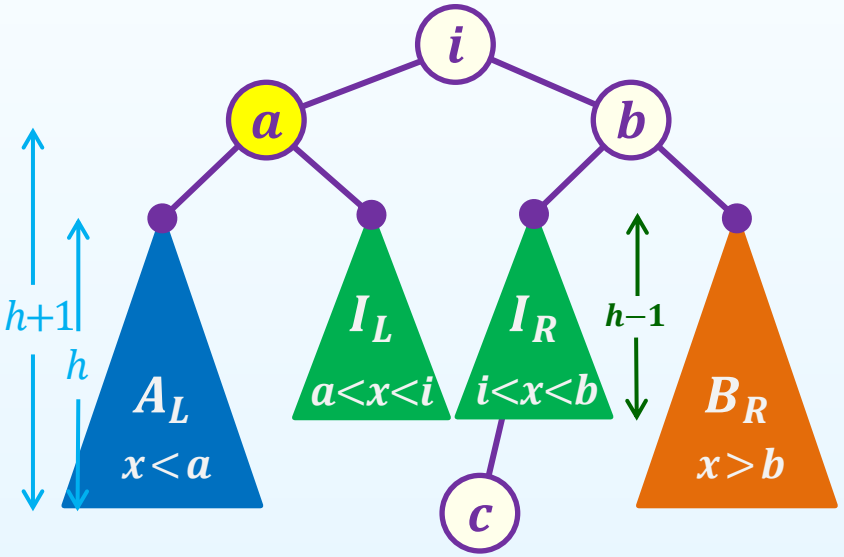
2.4 Update heights of a and i

Case 4: Right-left Rotation

1. c is inserted to the **left** of b, then tree is unbalanced at a

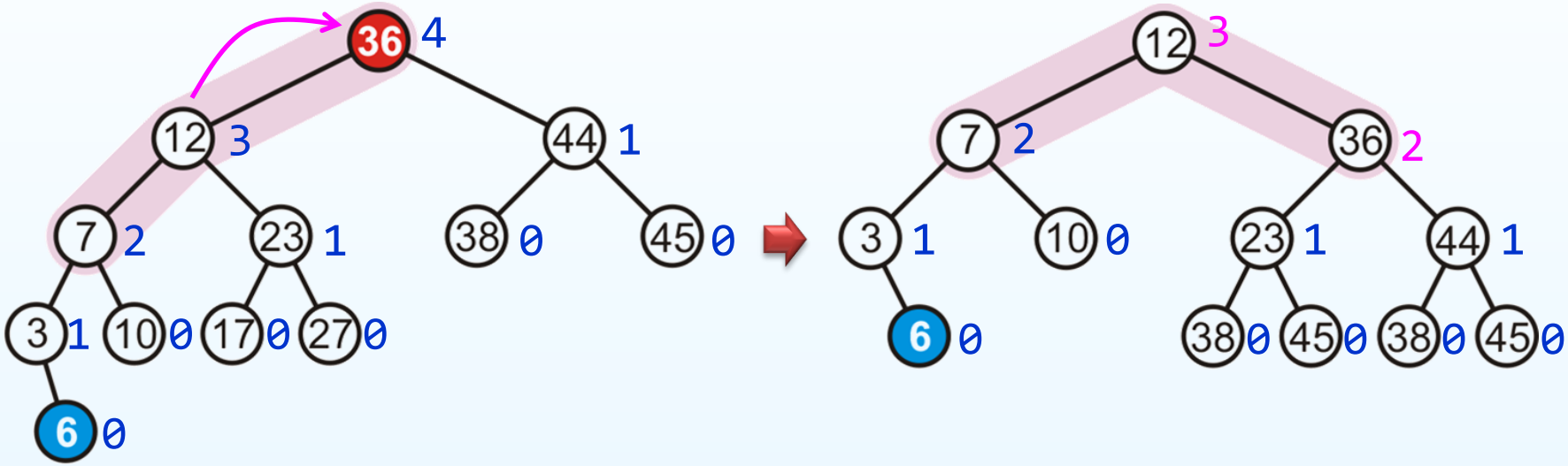


2. Perform the single right- the single left-rotation functions



Back to the Last Example

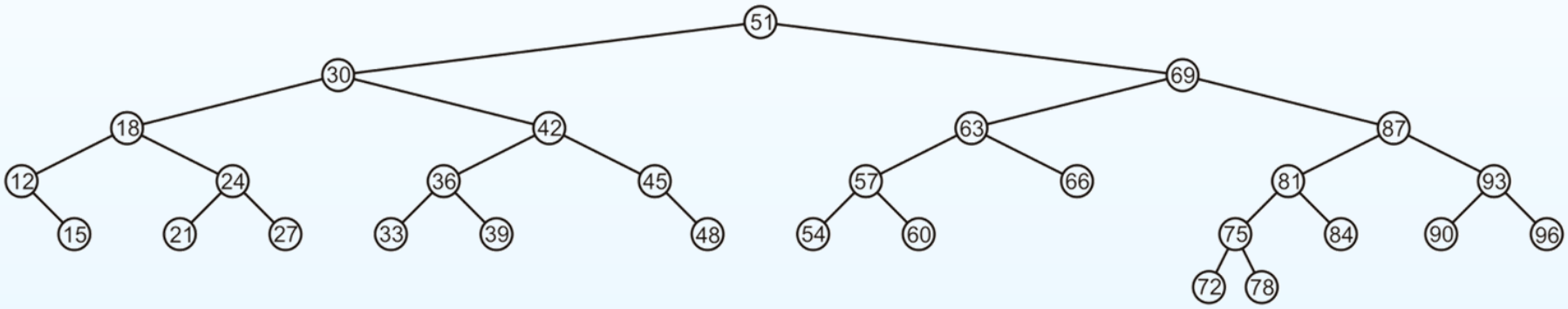
Recall that we insert 6 and then get unbalanced at 36



case 1: right rotation

Insertion

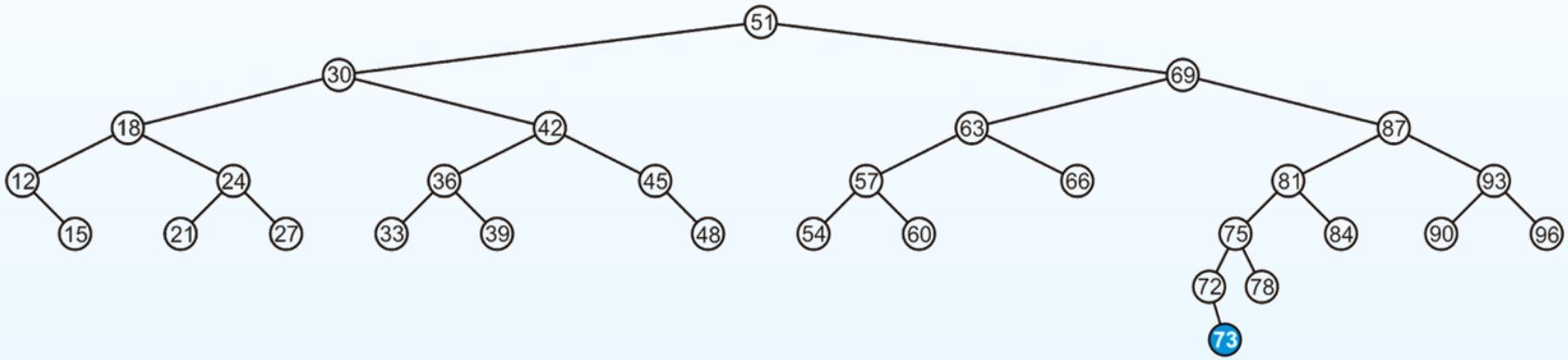
Consider this AVL tree:



Insertion

Consider this AVL tree:

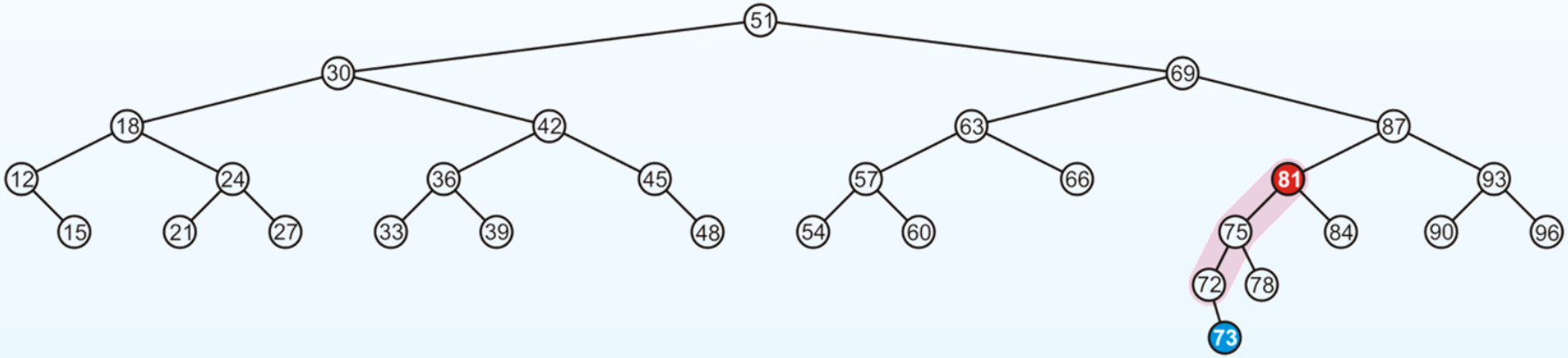
- Insert 73



Insertion

Consider this AVL tree:

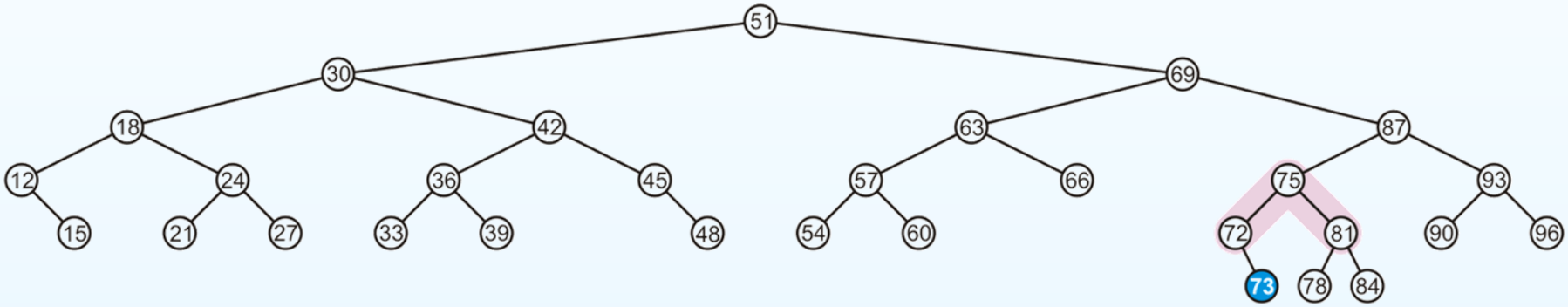
- Insert 73
  - The node 81 is unbalanced



Insertion

Consider this AVL tree:

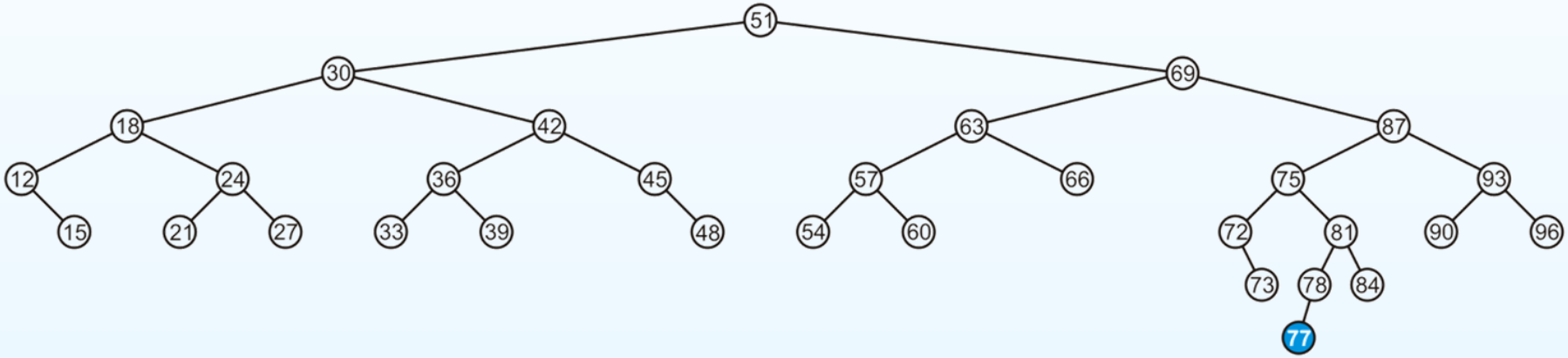
- Insert 73
  - The node 81 is unbalanced ➡ Perform the right rotation



Insertion

Consider this AVL tree:

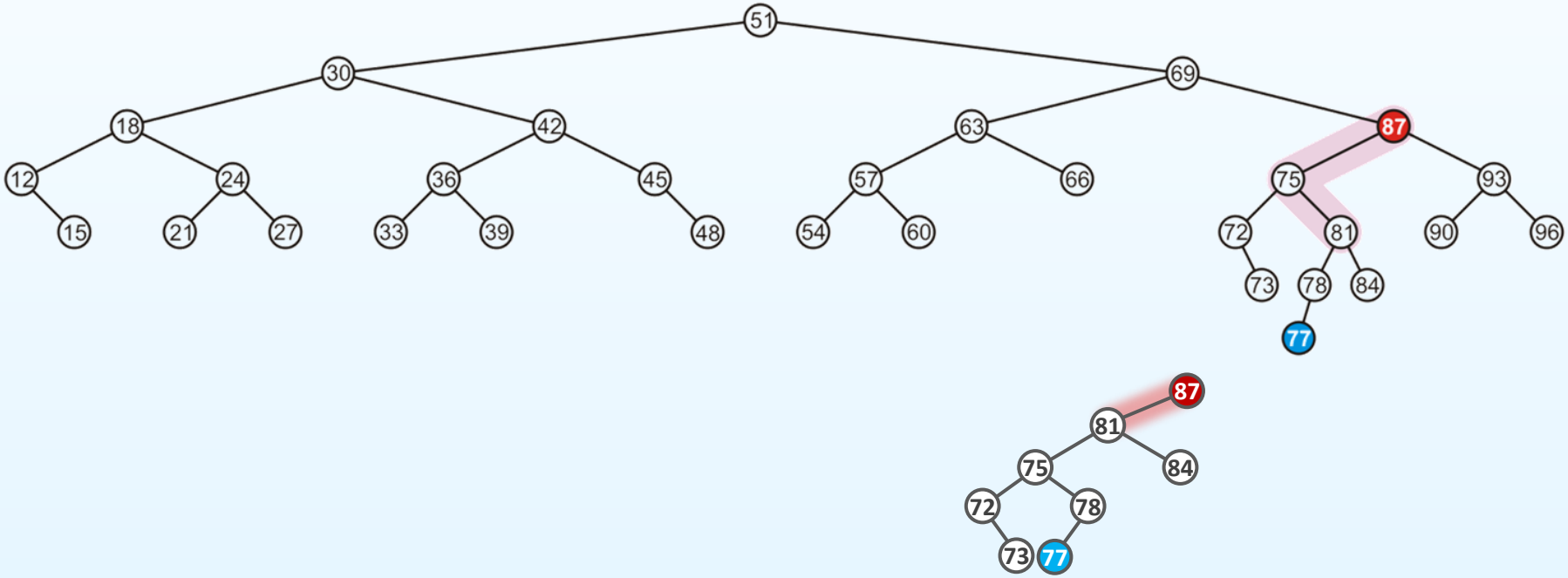
- Insert 77



Insertion

Consider this AVL tree:

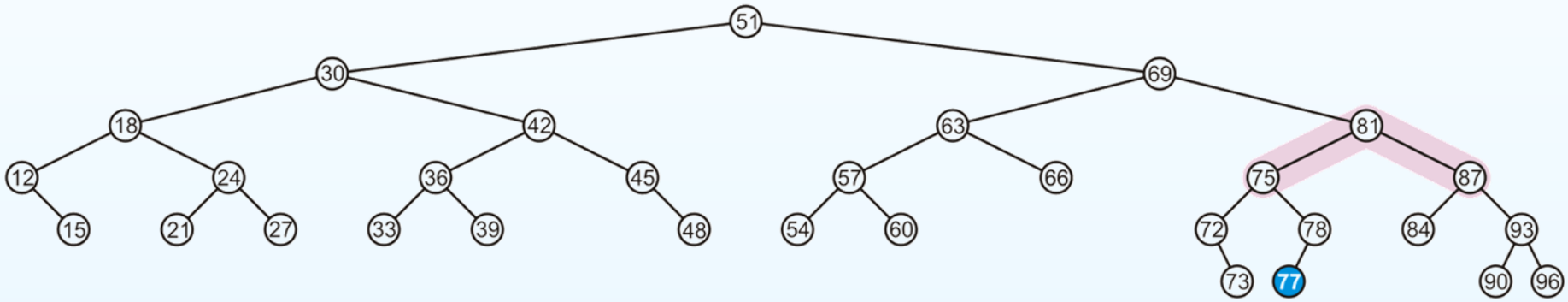
- Insert 77
  - The node 87 is unbalanced



Insertion

Consider this AVL tree:

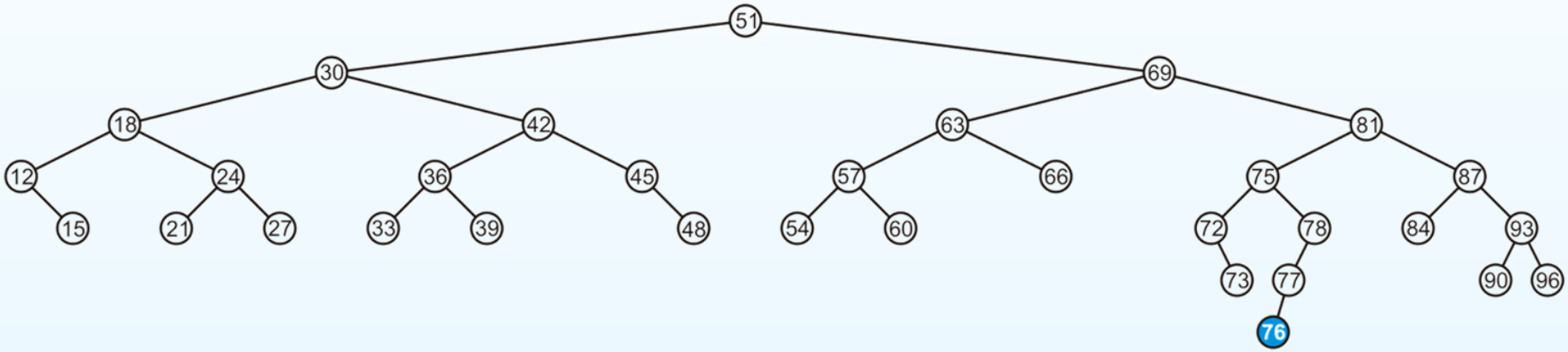
- Insert 77
  - The node 87 is unbalanced ➡ Perform the left-right rotation



Insertion

Consider this AVL tree:

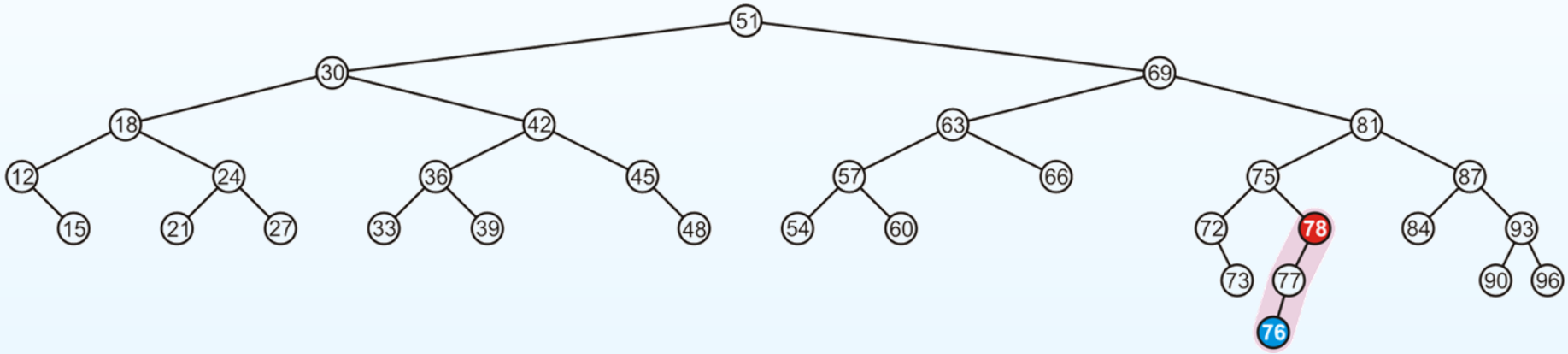
- Insert 76



Insertion

Consider this AVL tree:

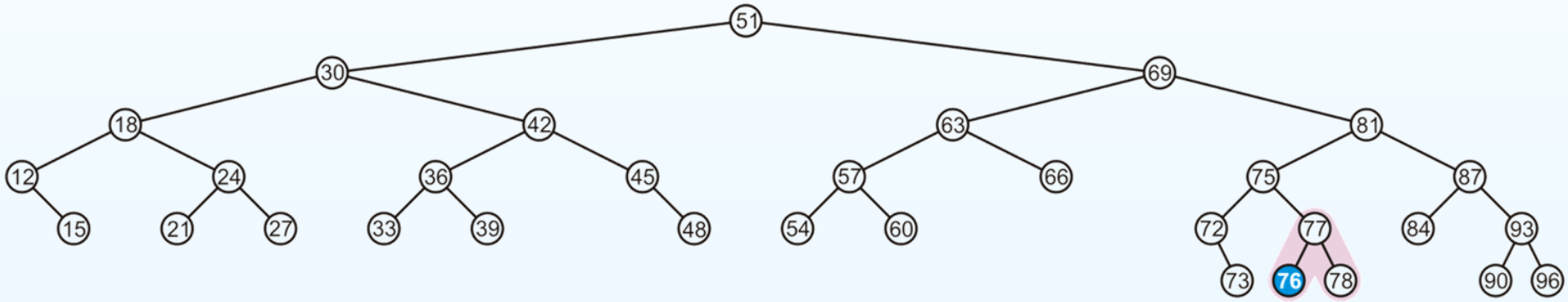
- Insert 76
  - The node 78 is unbalanced



Insertion

Consider this AVL tree:

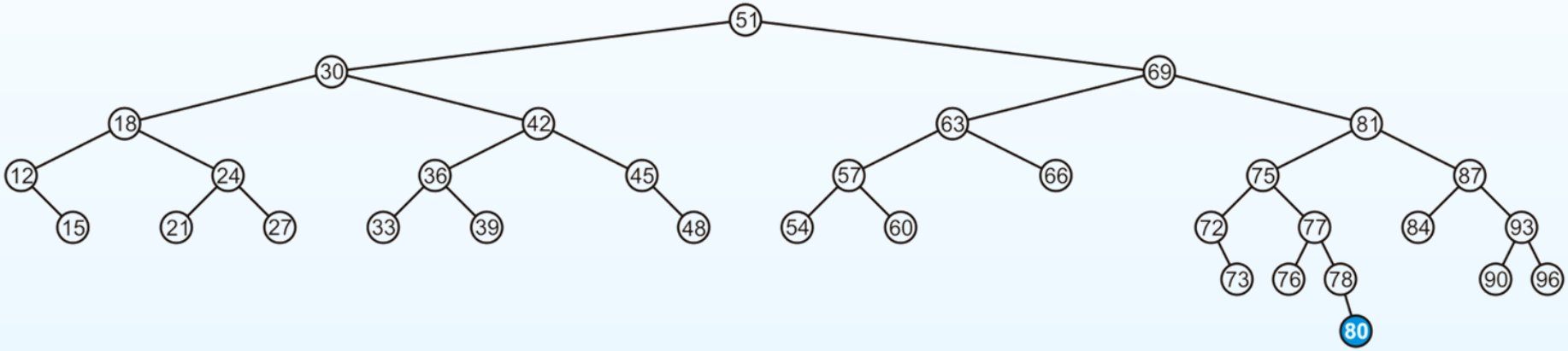
- Insert 76
  - The node 78 is unbalanced ➡ Perform the right rotation



Insertion

Consider this AVL tree:

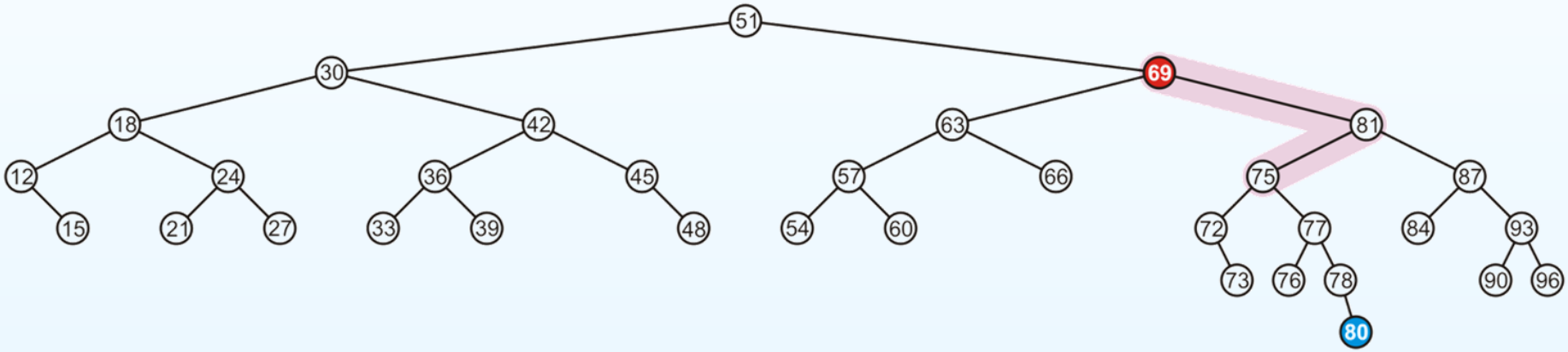
- Insert `80`

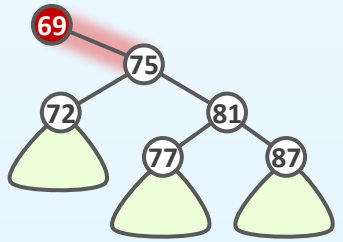


Insertion

Consider this AVL tree:

- Insert 80
  - The node 69 is unbalanced

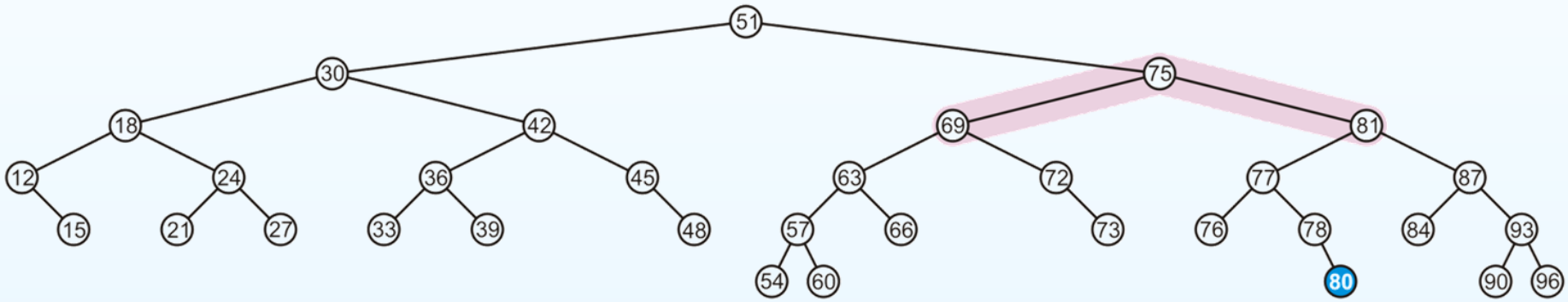




Insertion

Consider this AVL tree:

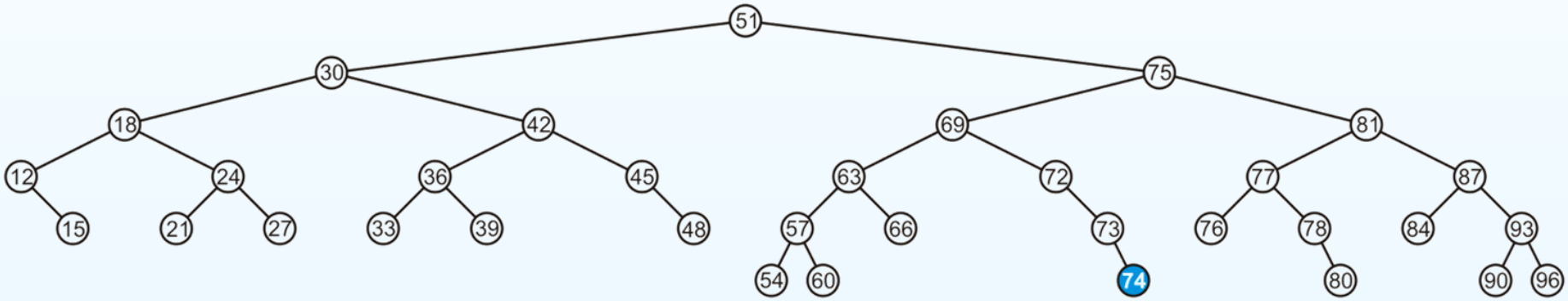
- Insert 80
  - The node 69 is unbalanced ➡ Perform the right-left rotation



Insertion

Consider this AVL tree:

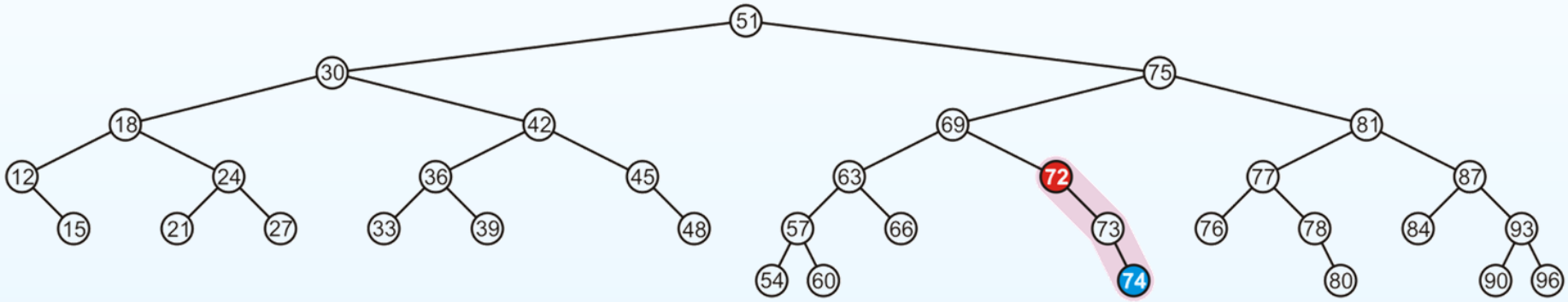
- Insert 74



Insertion

Consider this AVL tree:

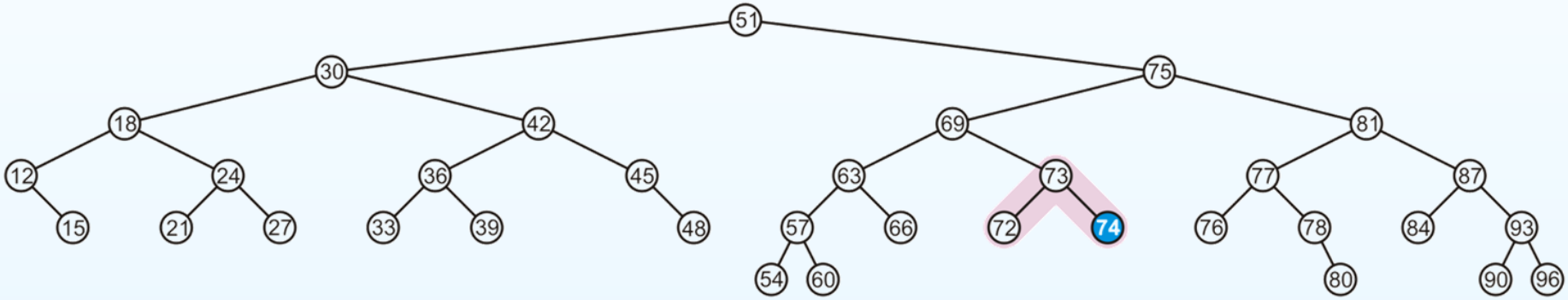
- Insert 74
  - The node 72 is unbalanced



Insertion

Consider this AVL tree:

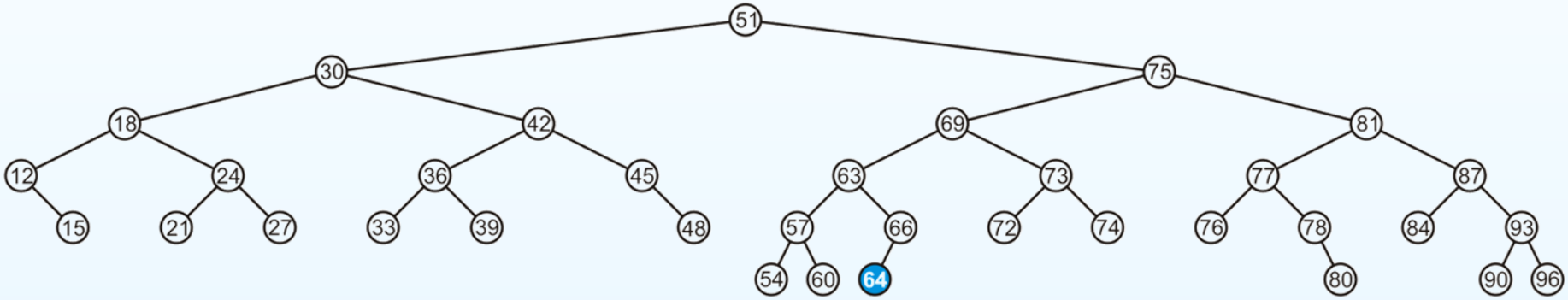
- Insert 74
  - The node 72 is unbalanced ➡ Perform the left rotation



Insertion

Consider this AVL tree:

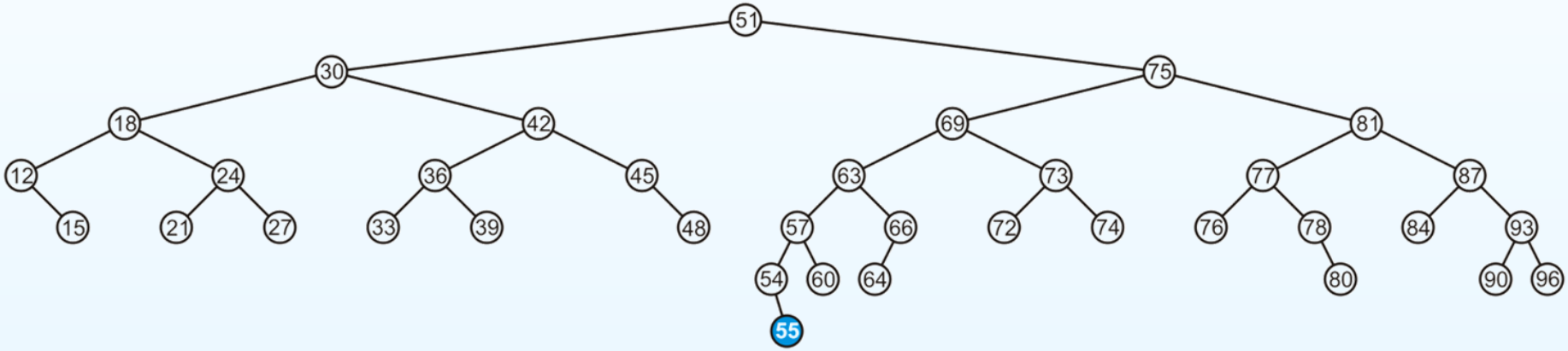
- Insert 64
  - There is no unbalanced



Insertion

Consider this AVL tree:

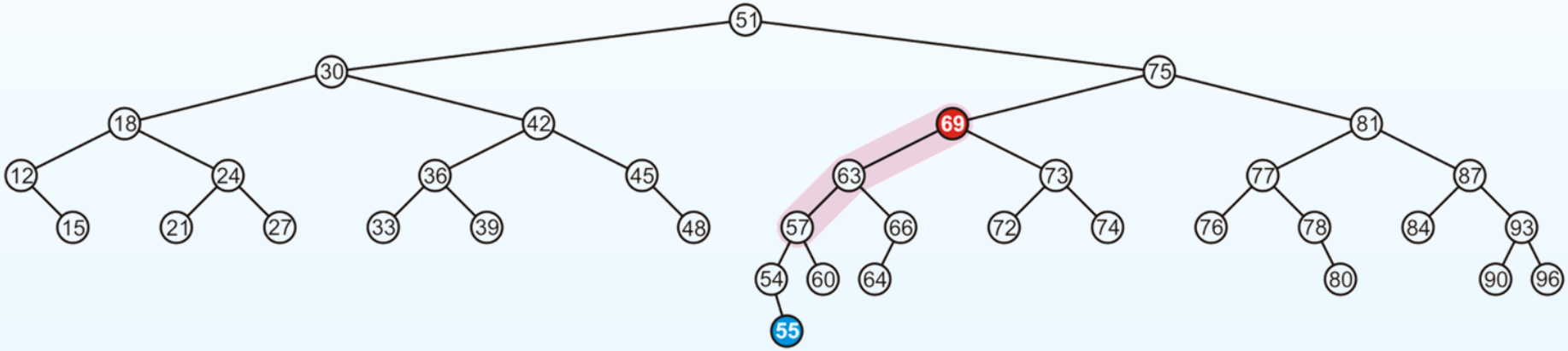
- Insert 55



Insertion

Consider this AVL tree:

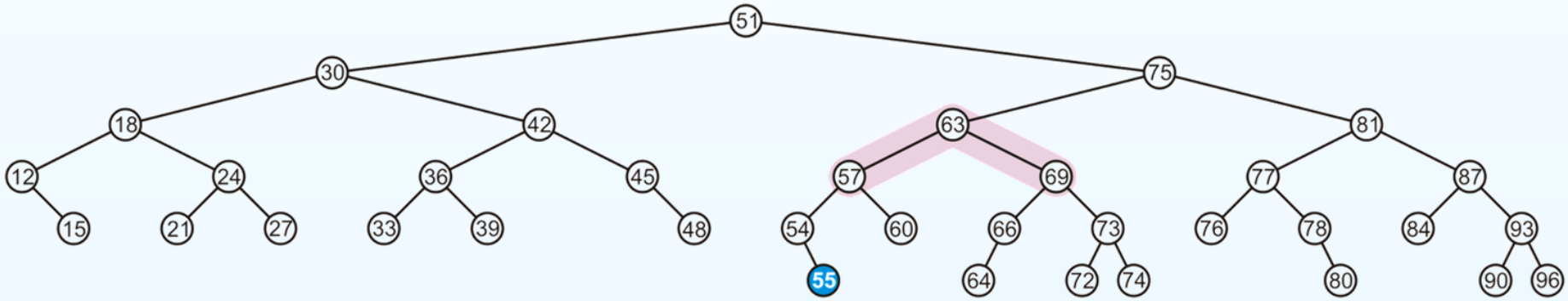
- Insert 55
  - The node 69 is unbalanced



Insertion

Consider this AVL tree:

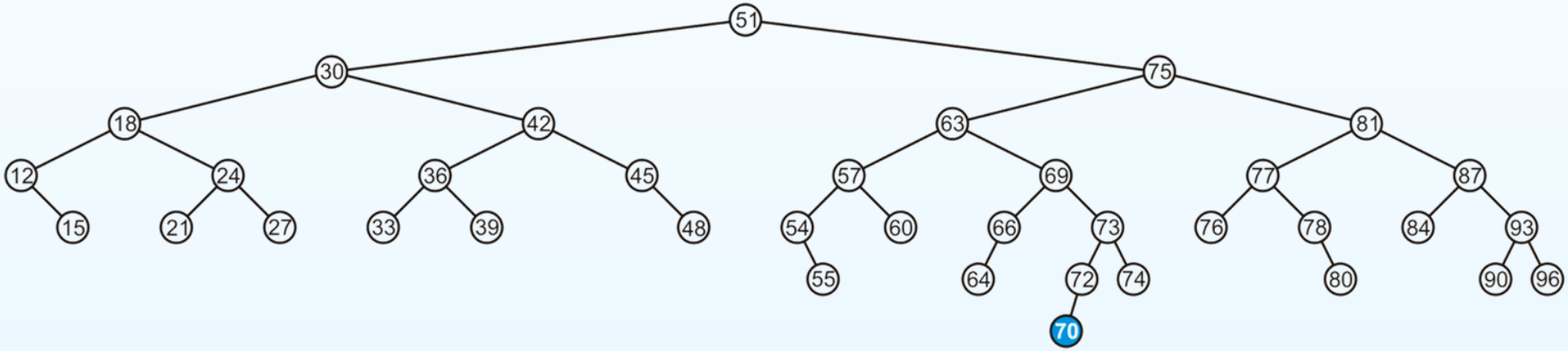
- Insert 55
  - The node 69 is unbalanced ➡ Perform the right rotation



Insertion

Consider this AVL tree:

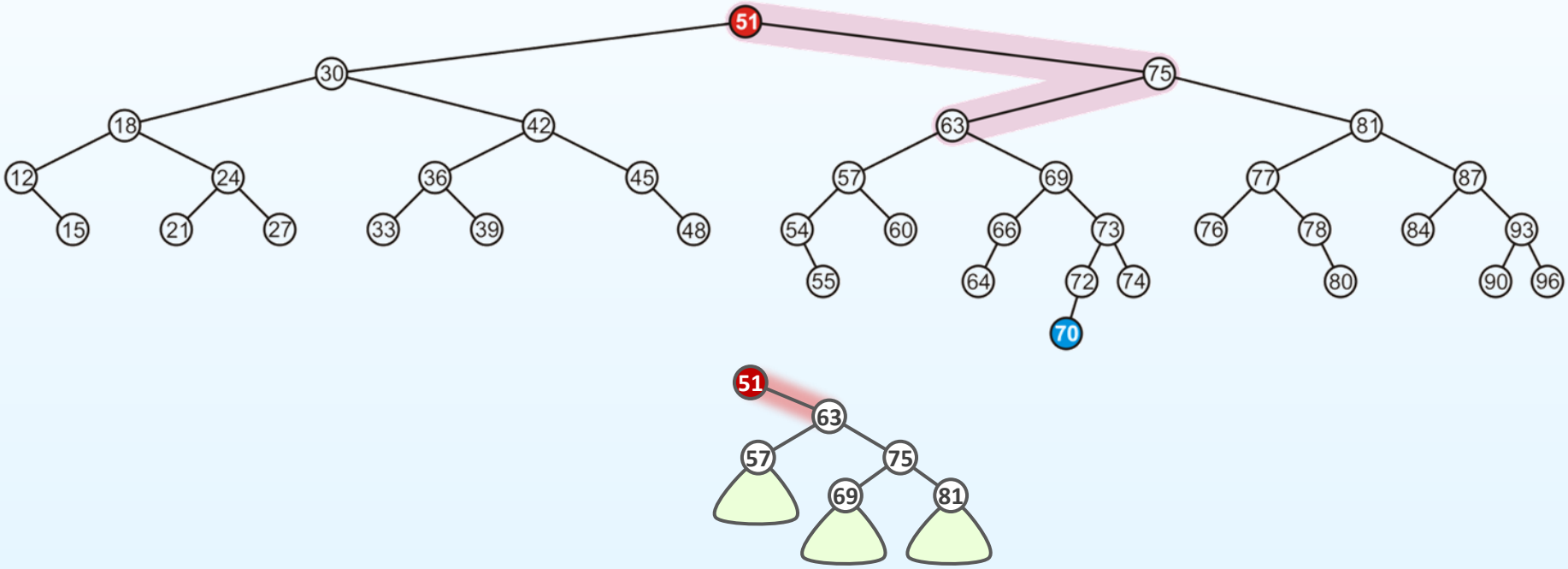
- Insert `70`



Insertion

Consider this AVL tree:

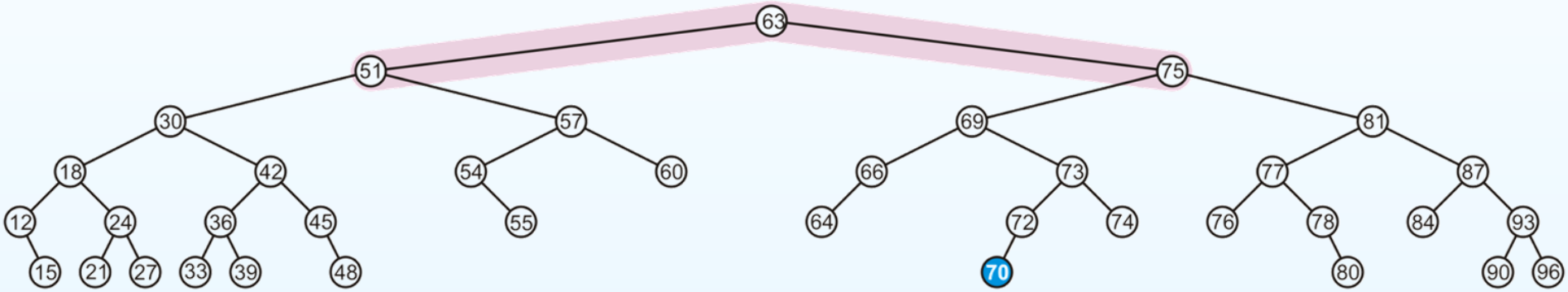
- Insert `70`
  - The node `51` is unbalanced



Insertion

Consider this AVL tree:

- Insert `70`
  - The node `51` is unbalanced ➡ Perform the right-left rotation



Deletion

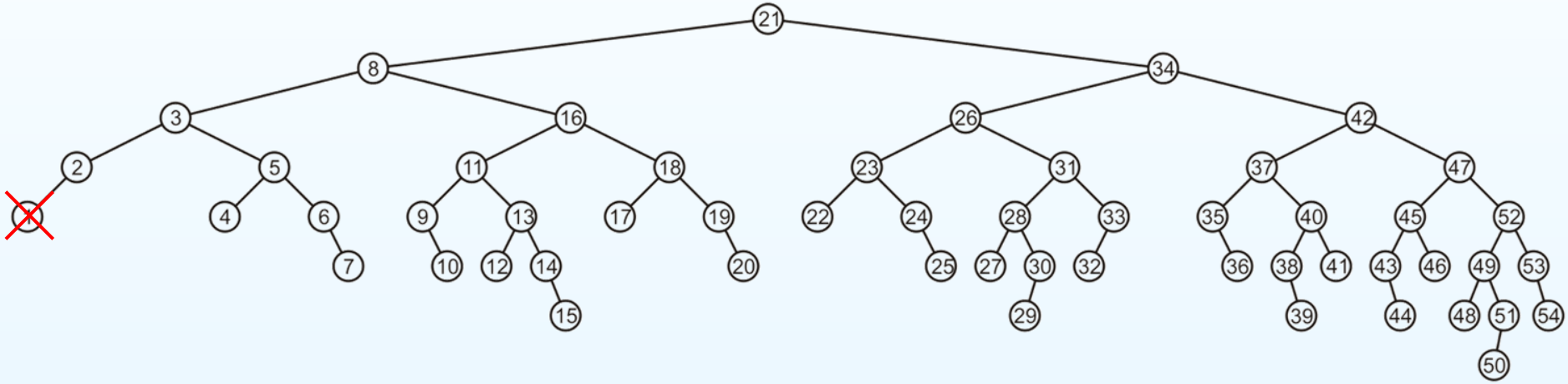
Removing a node from an AVL tree may cause more than one unbalance

- Like insertion, deletion must check after it has been successfully called on a child to see if it caused an unbalance
- Unfortunately, it may cause $O(h)$ unbalances that must be corrected
  - Insertions will only cause one unbalance

Deletion

Consider the following AVL tree:

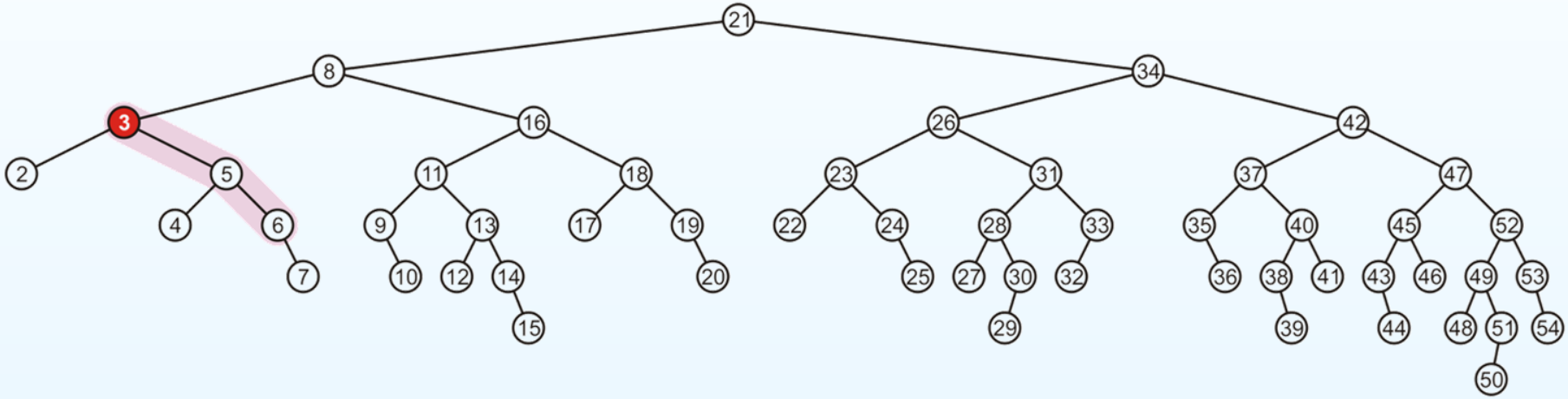
- Suppose we would like to delete 1



Deletion

Consider this AVL tree:

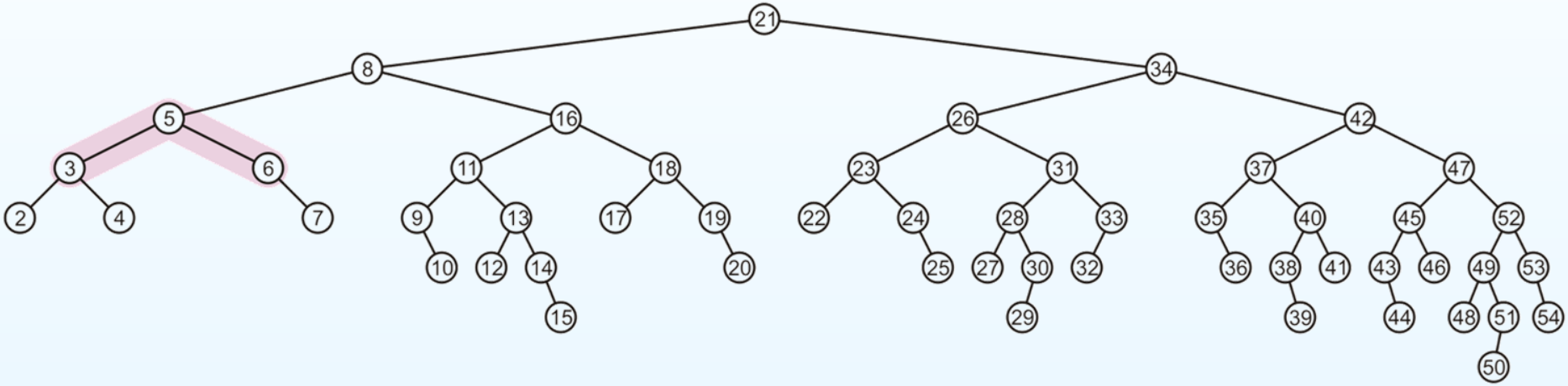
- Suppose we would like to delete 1
  - After backtracking, we found that the node 3 is unbalanced



Deletion

Consider this AVL tree:

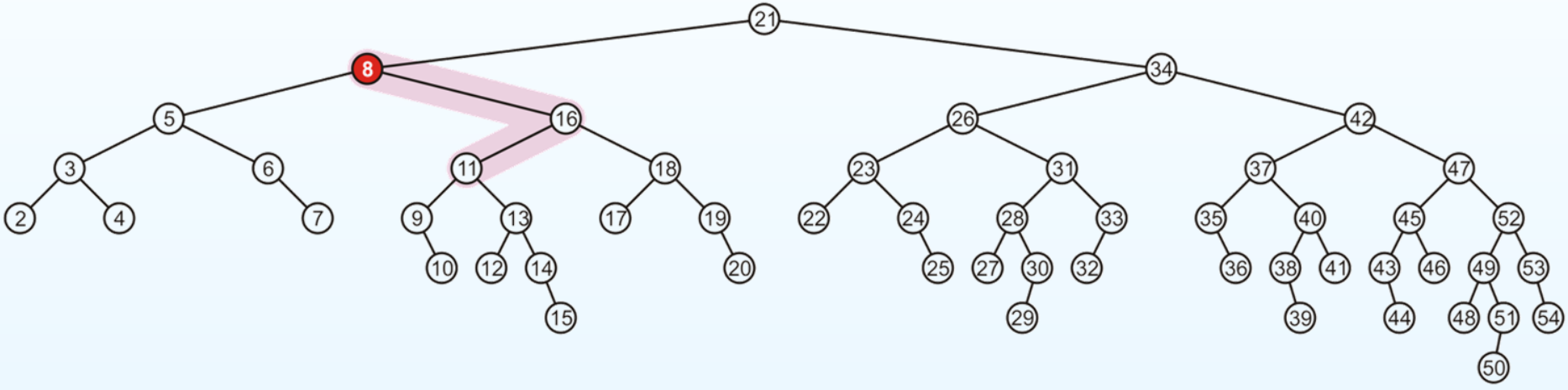
- Suppose we would like to delete 1
  - The node 3 is unbalanced ➡ Perform the left rotation

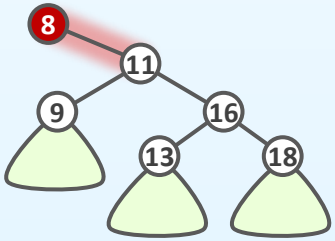


Deletion

Consider this AVL tree:

- Suppose we would like to delete 1
  - The node 8 is still unbalanced

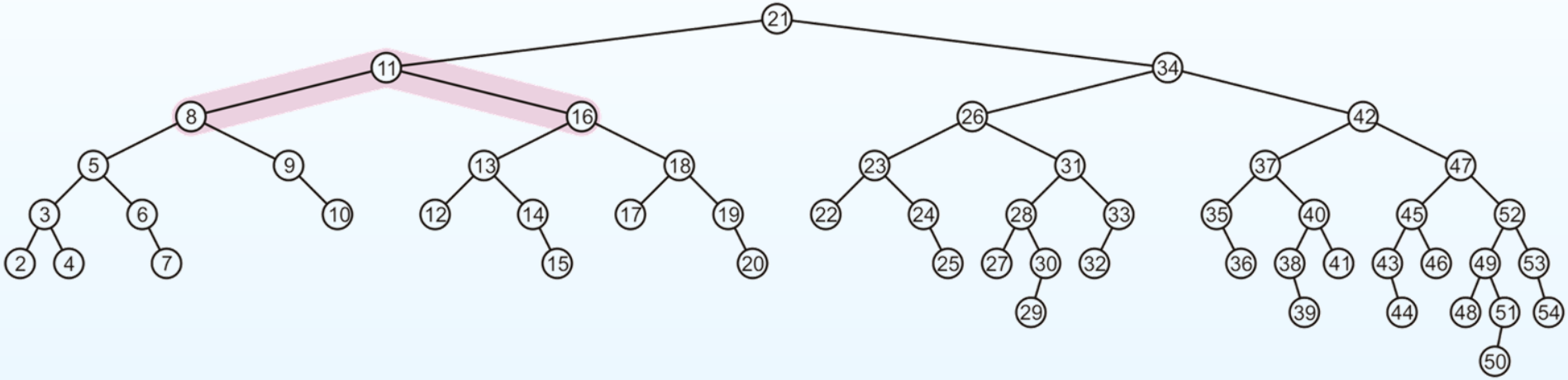




Deletion

Consider this AVL tree:

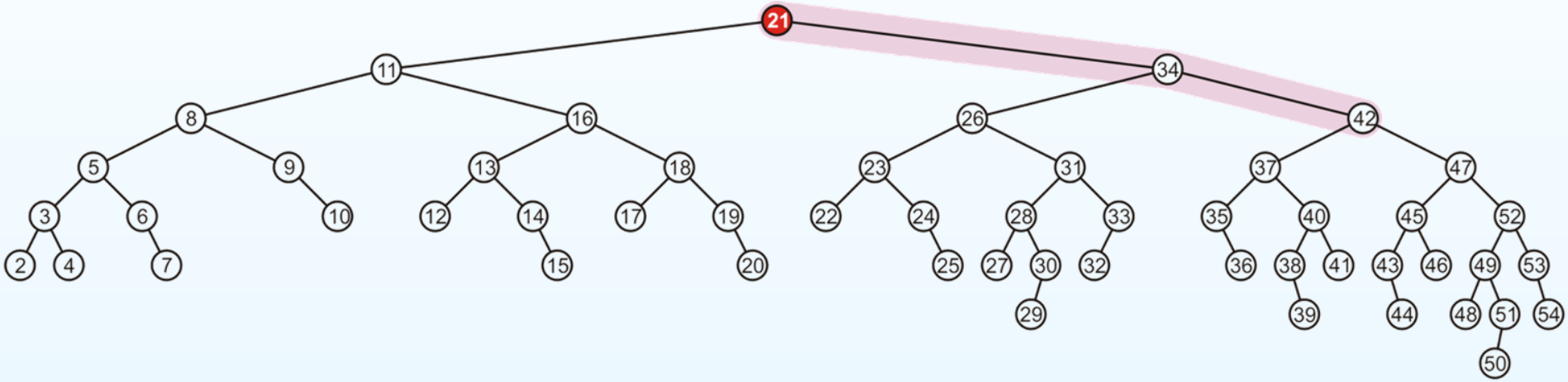
- Suppose we would like to delete 1
  - The node 8 is unbalanced ➡ Perform the right-left rotation



Deletion

Consider this AVL tree:

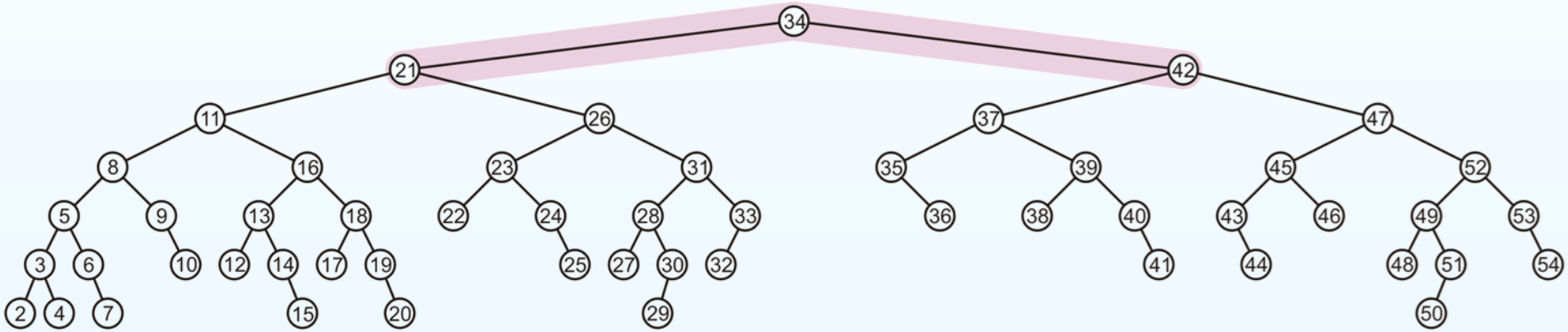
- Suppose we would like to delete 1
  - The node 21 is still unbalanced



Deletion

Consider this AVL tree:

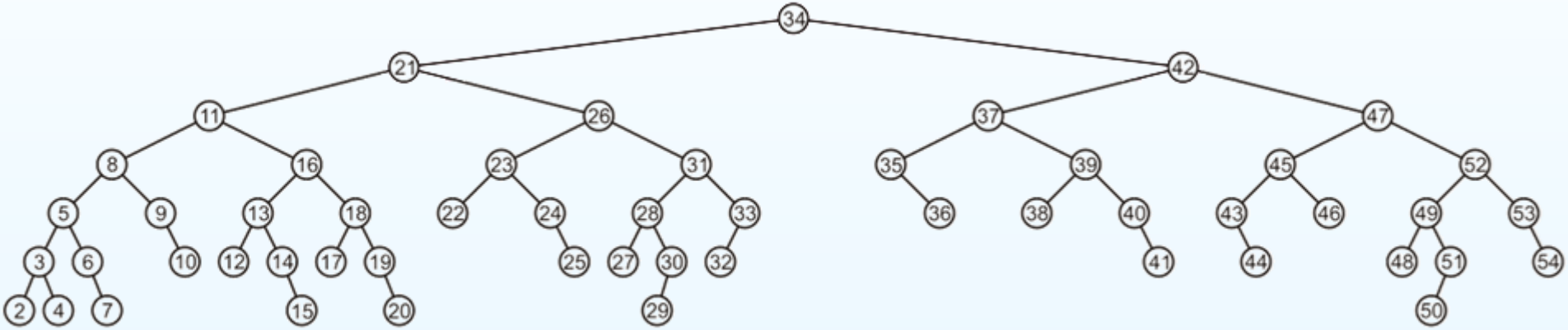
- Suppose we would like to delete 1
  - The node 21 is unbalanced ➡ Perform the left rotation



Exercise 3 (5 mins.)

Consider this AVL tree:

- Suppose we would like to delete 34



Any Question?