CS2020
Data Structures and Algorithms

Welcome!

Problem Sets

Problem Set 2:

- Due: Wednesday, 2pm

Problem Set 3:

- Released today.
- Programming experience.

Upcoming…

Next week: Chinese New Year

- Lecture on Tuesday
- No Friday lecture
- No discussion groups

Two weeks:

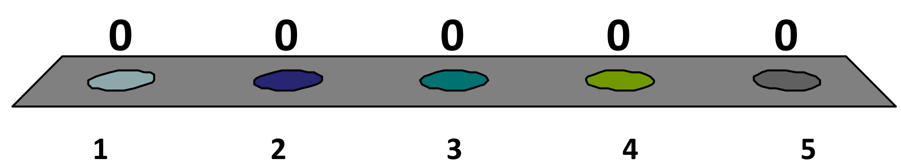
- Quiz 1

Four weeks:

- Practical Programming Quiz

Problem Set 2 was:

1. Very easy
2. A little easy
3. About right
4. A little hard
5. Very hard



0 of 60

Today's Plan

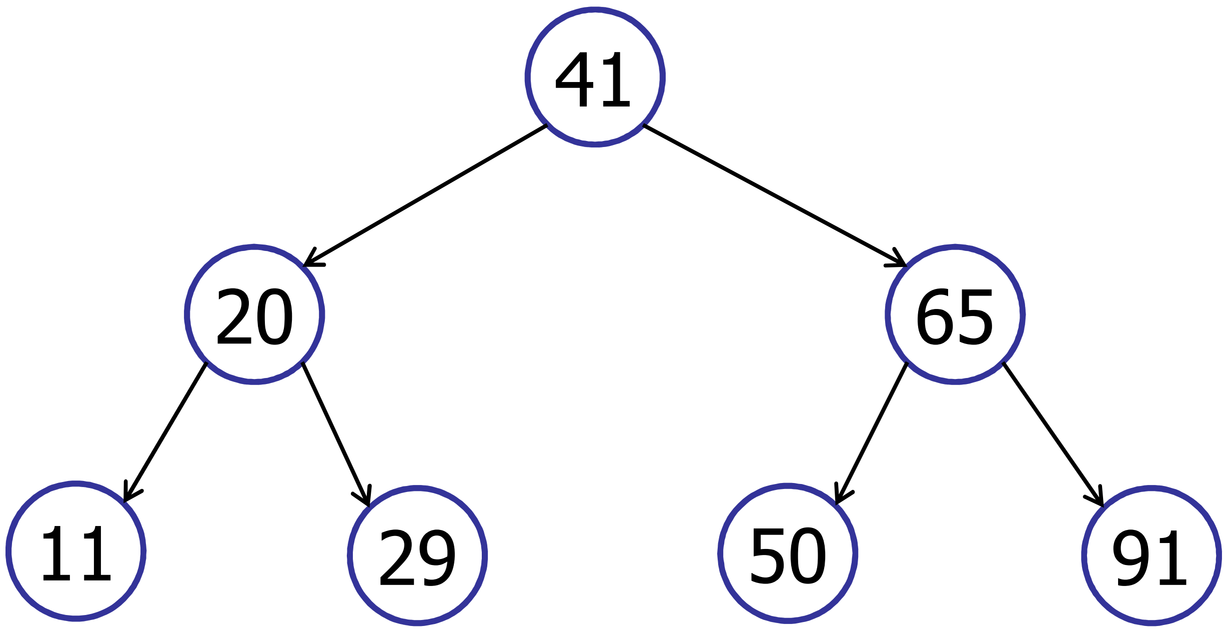
Binary Search Trees

- Review
- delete

On the importance of being balanced

- Height-balanced binary search trees
- AVL trees

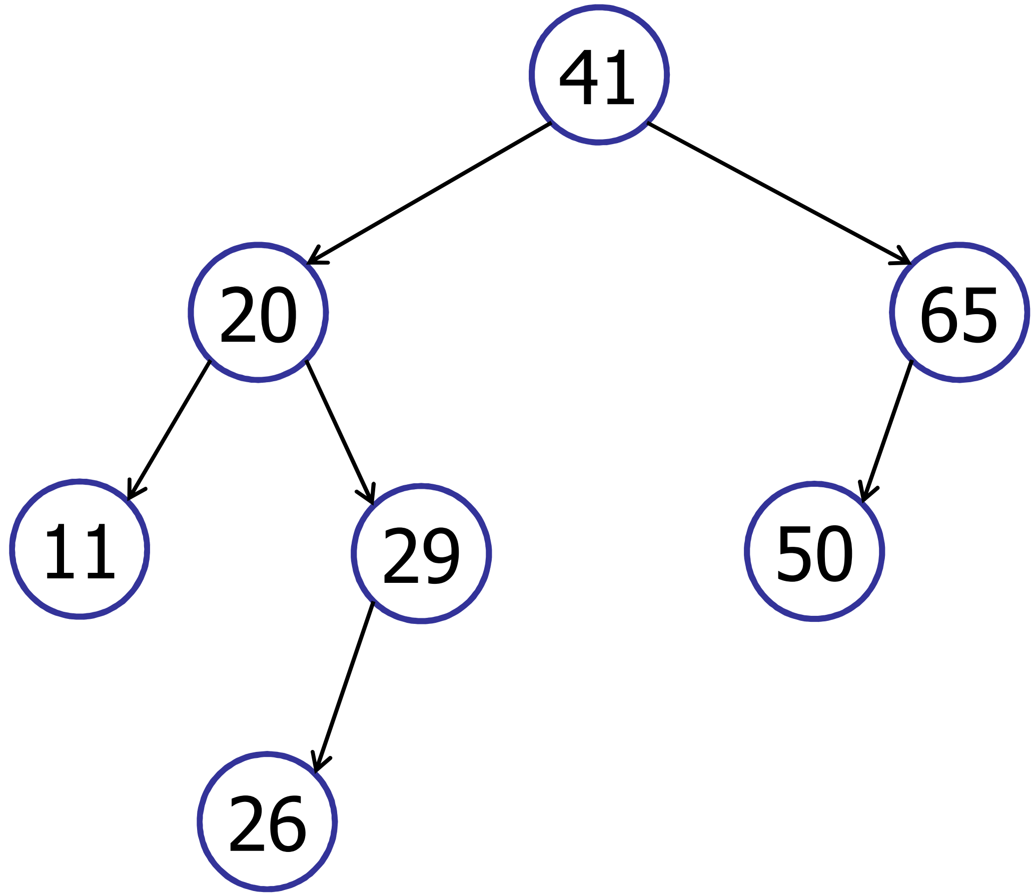
Binary Search Trees

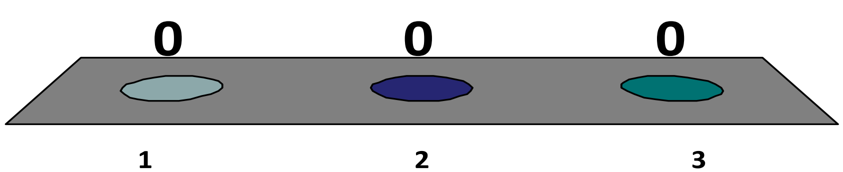


- Two children: v.left, v.right
- Key: v.key
- **BST Property**: all in left sub-tree < key < all in right sub-right

Is this a binary search tree?

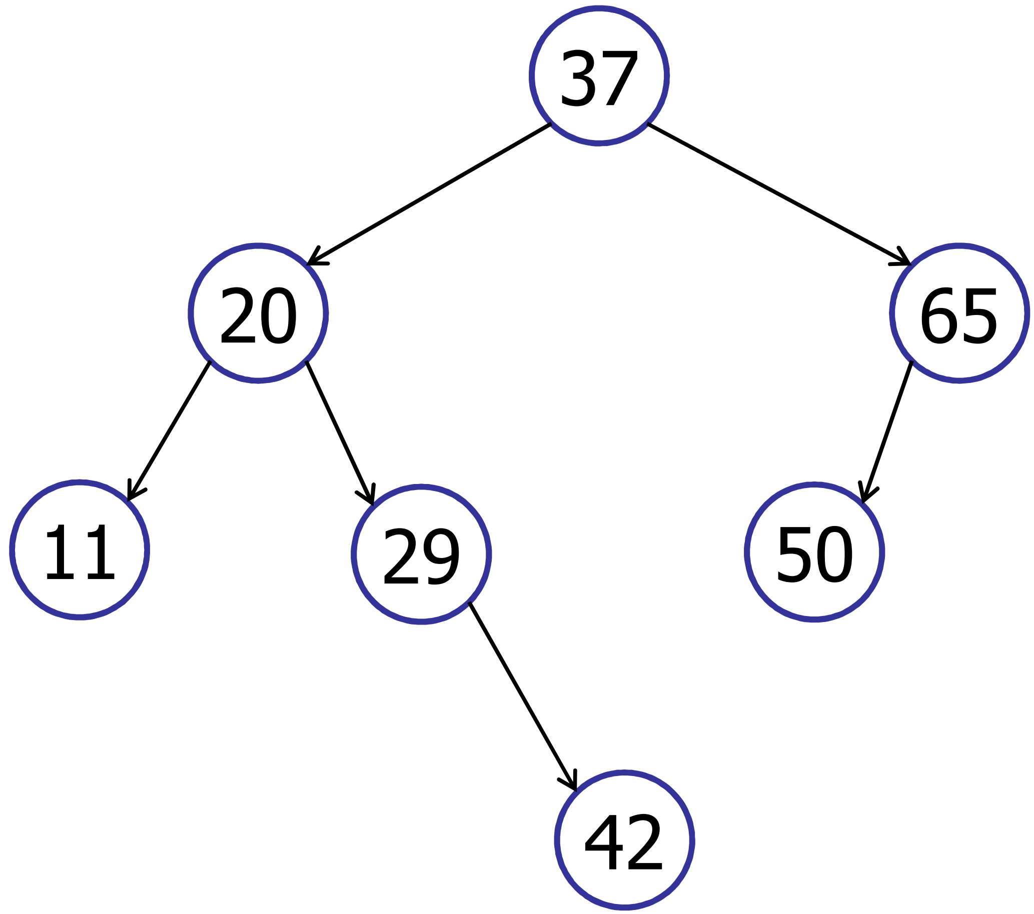
1. Yes
2. No
3. I don’t know.

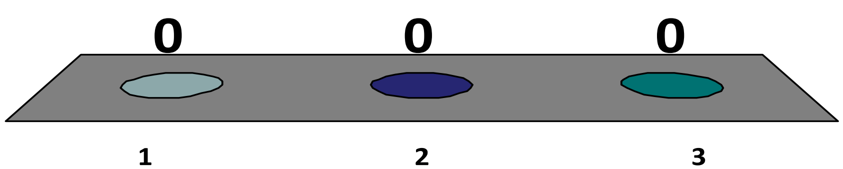




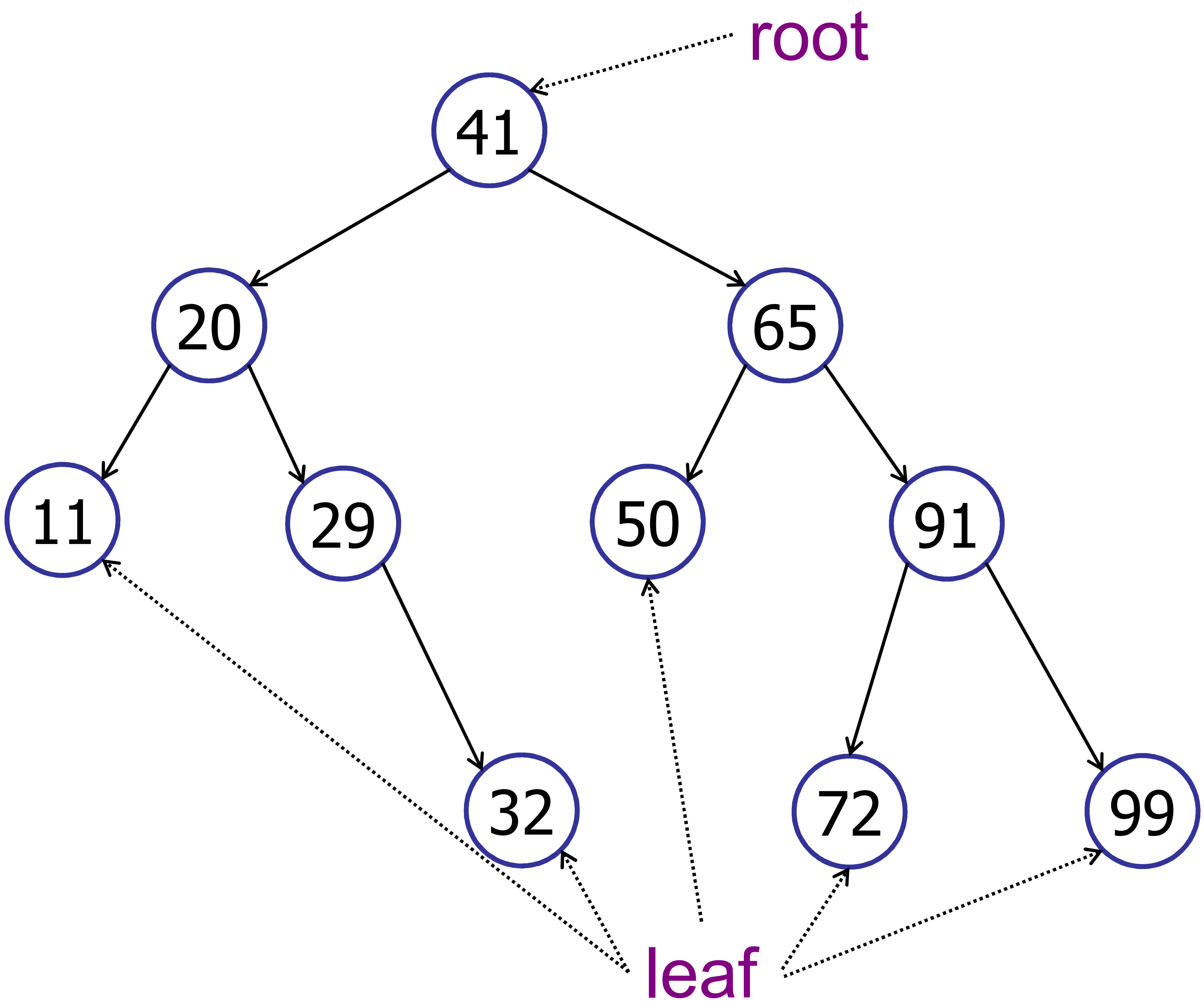
Is this a binary search tree?

1. Yes
2. No
3. I don’t know.

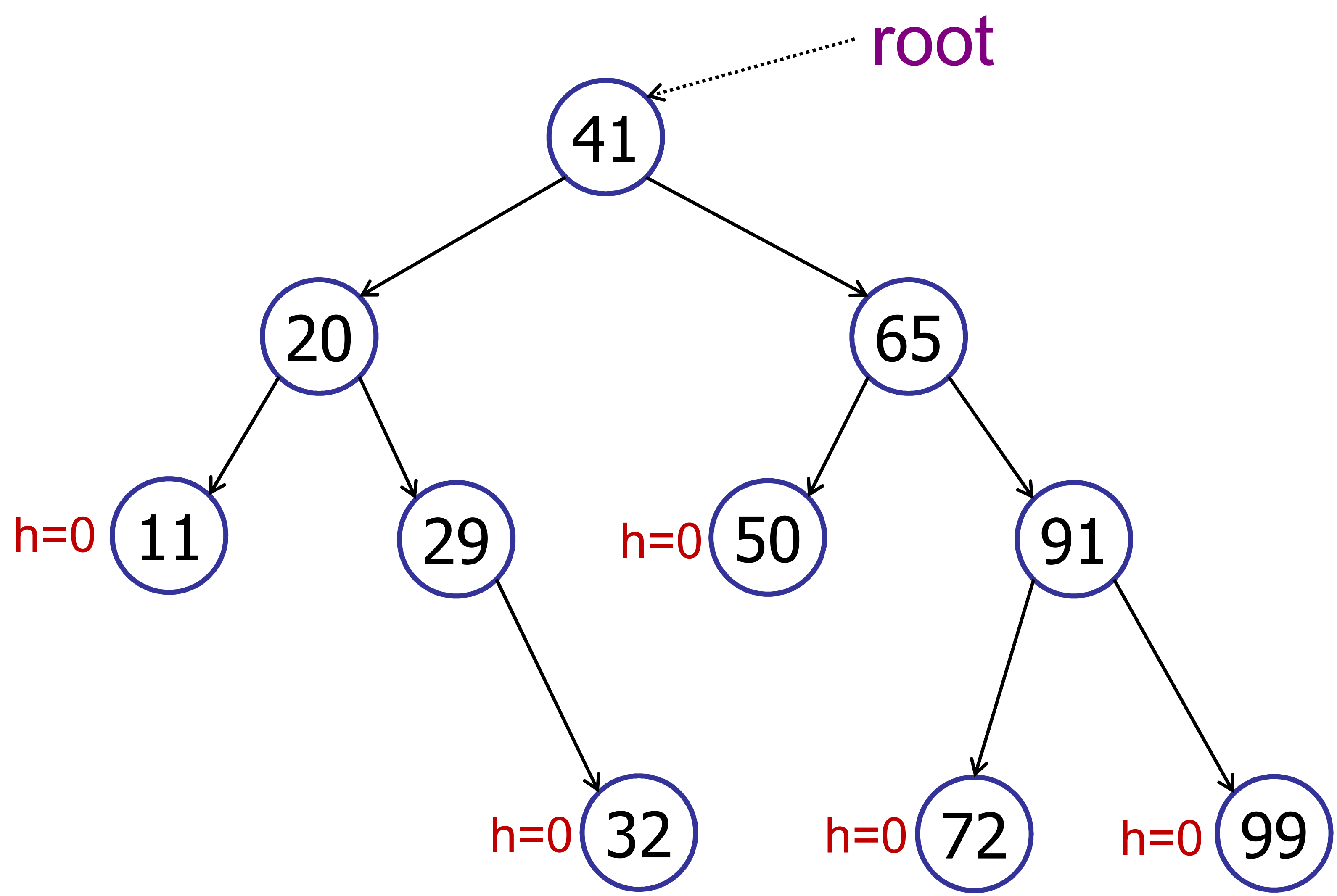




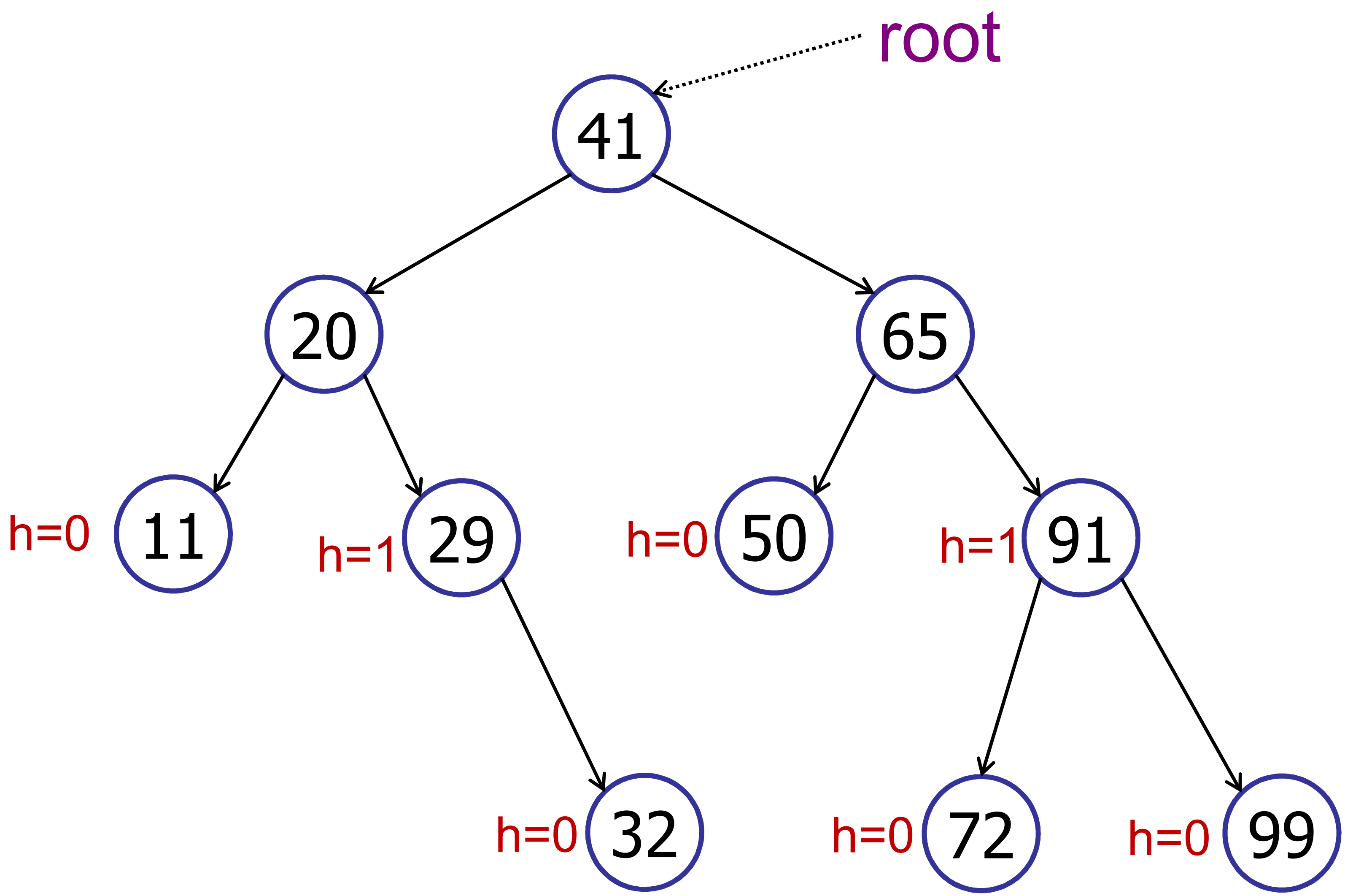
Binary Search Trees



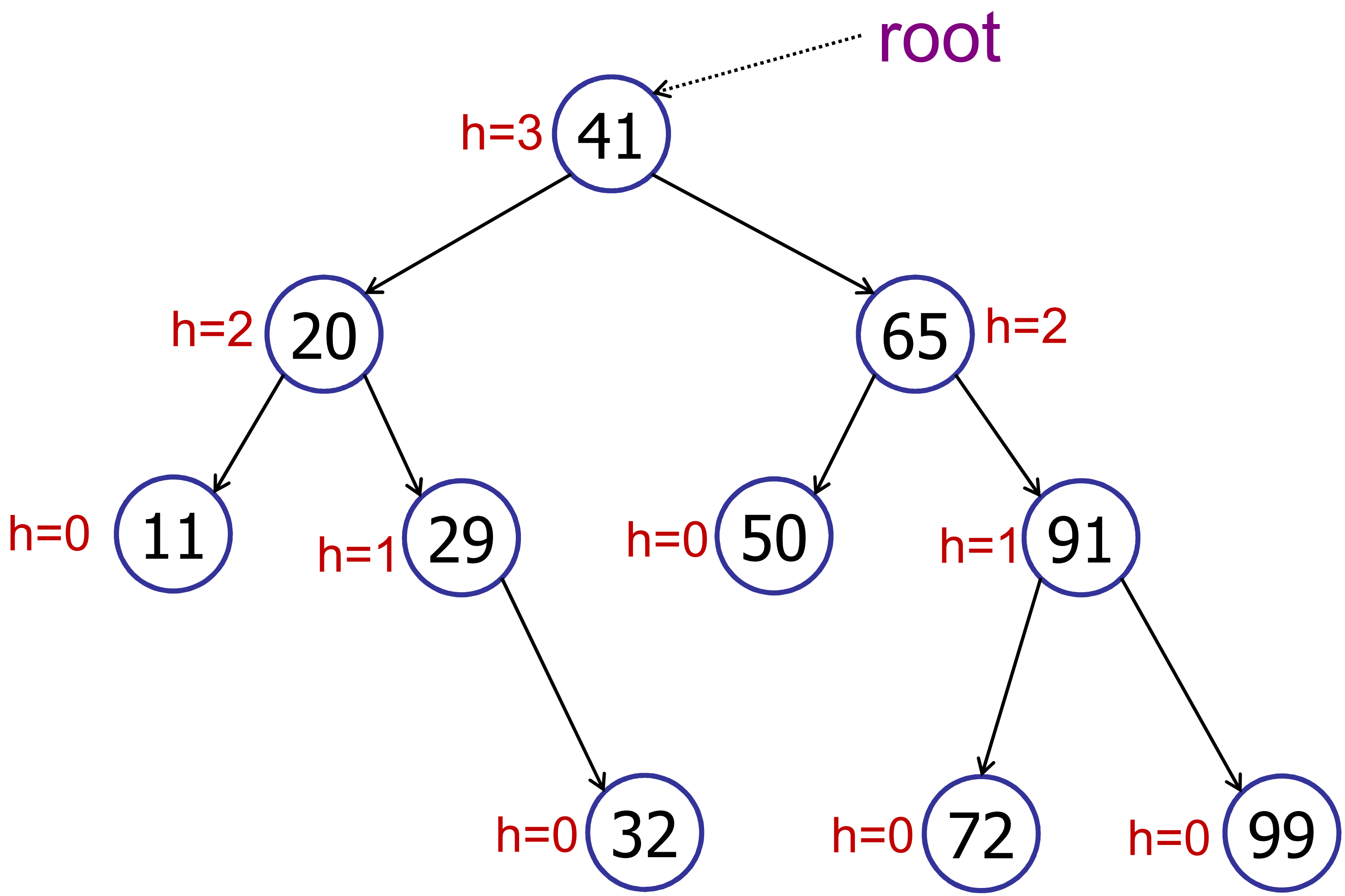
Binary Search Trees



Binary Search Trees



Binary Search Trees



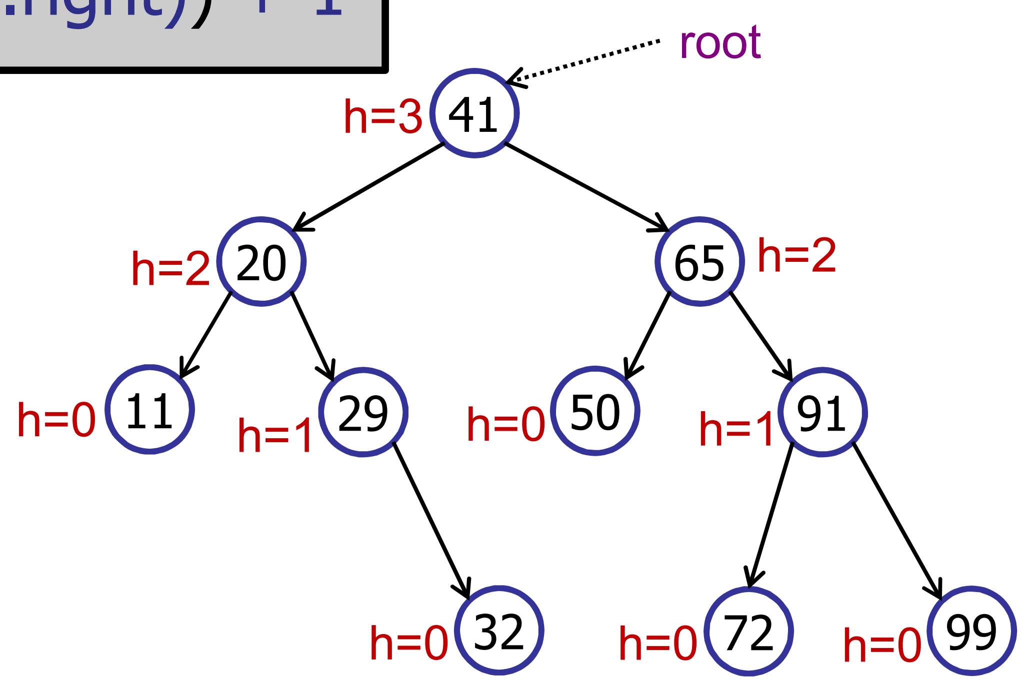
Binary Search Trees

Height:

Number of edges on longest path from root to leaf.

h(v) = 0 (if v is a leaf)

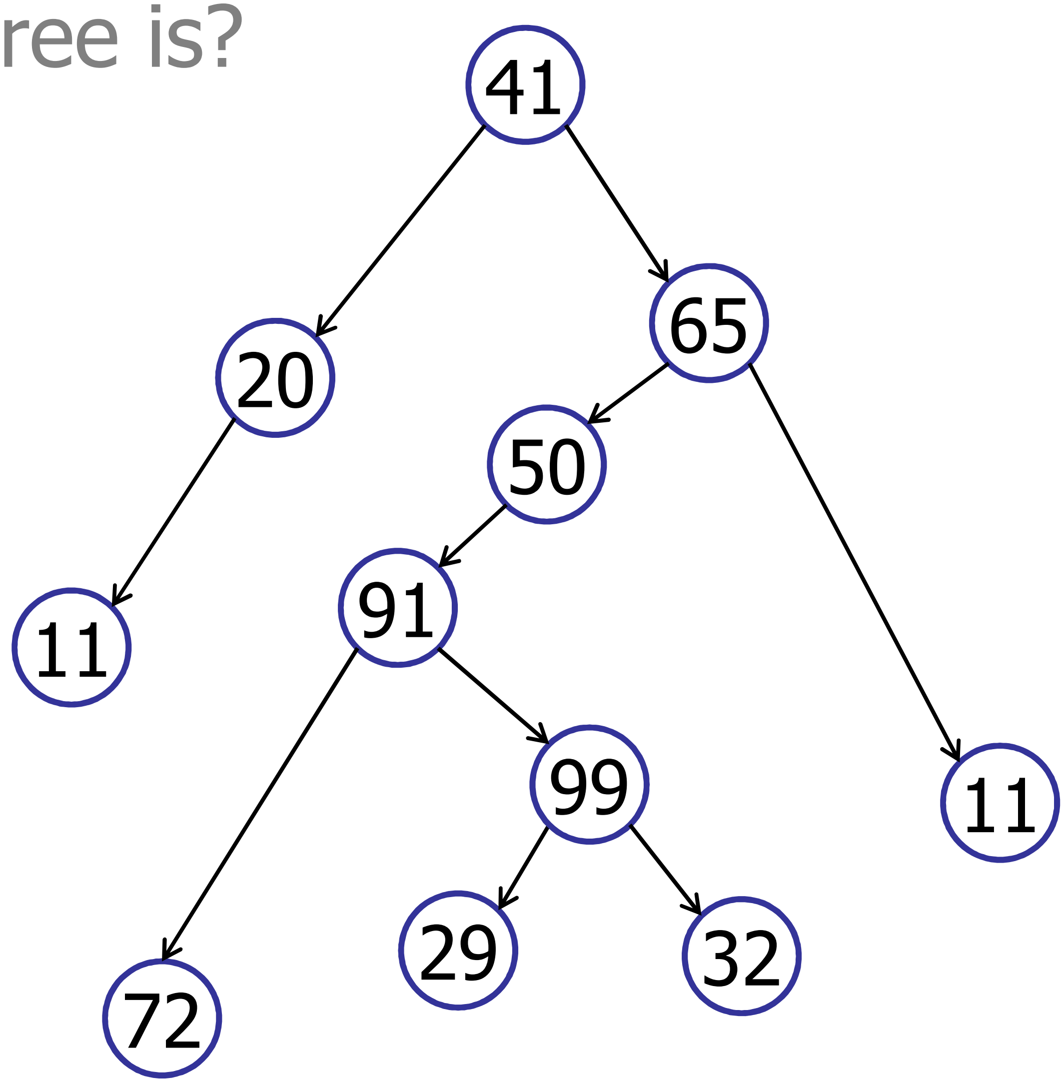
h(v) = max(h(v.left), h(v.right)) + 1

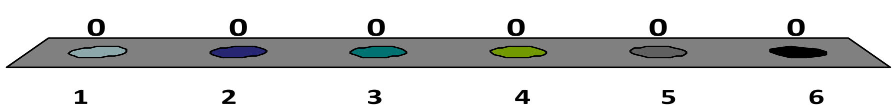


(For simplicity: h(null) = -1)

The height of this tree is?

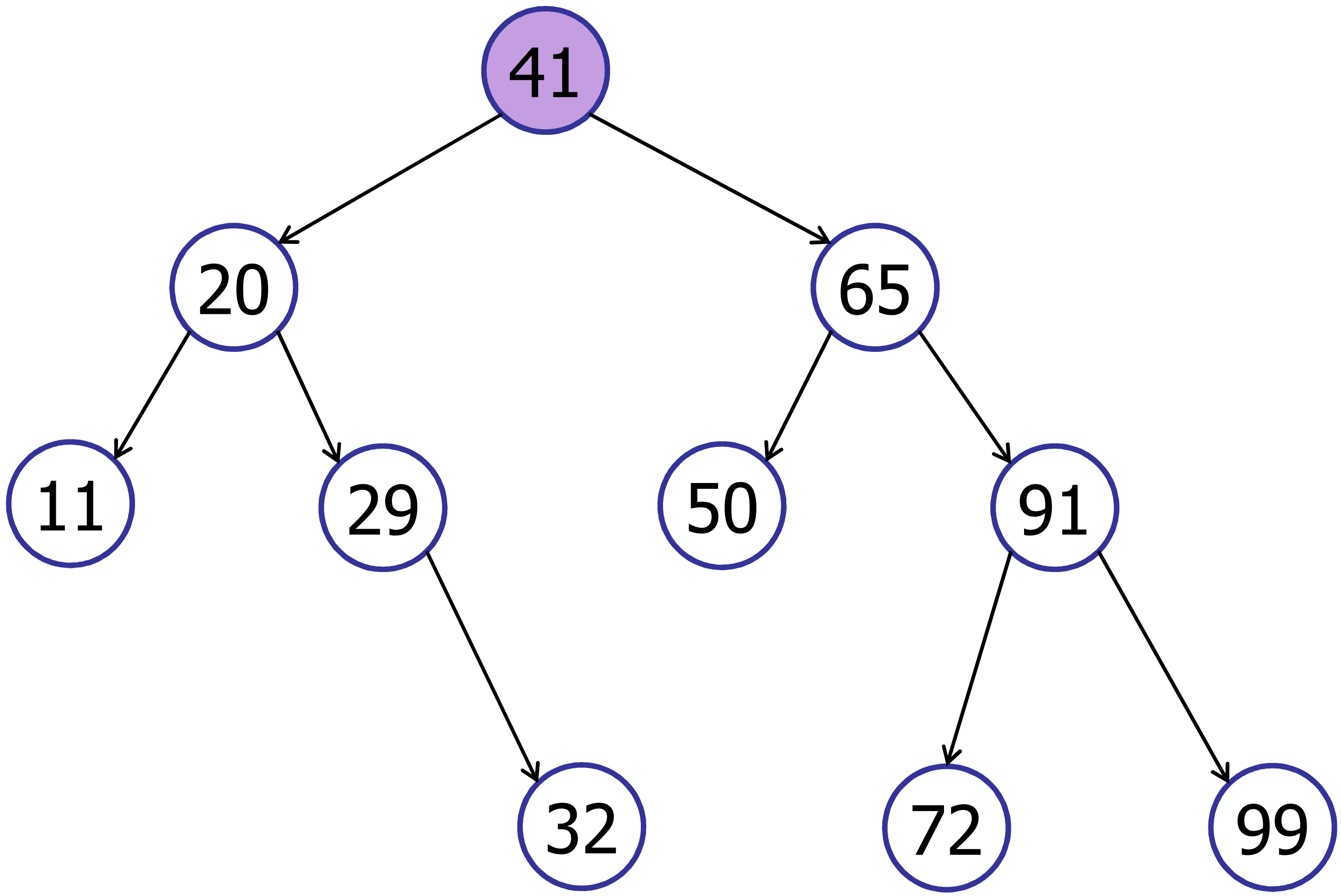
1. 2
2. 4
3. 5
4. 6
5. 7
6. 42





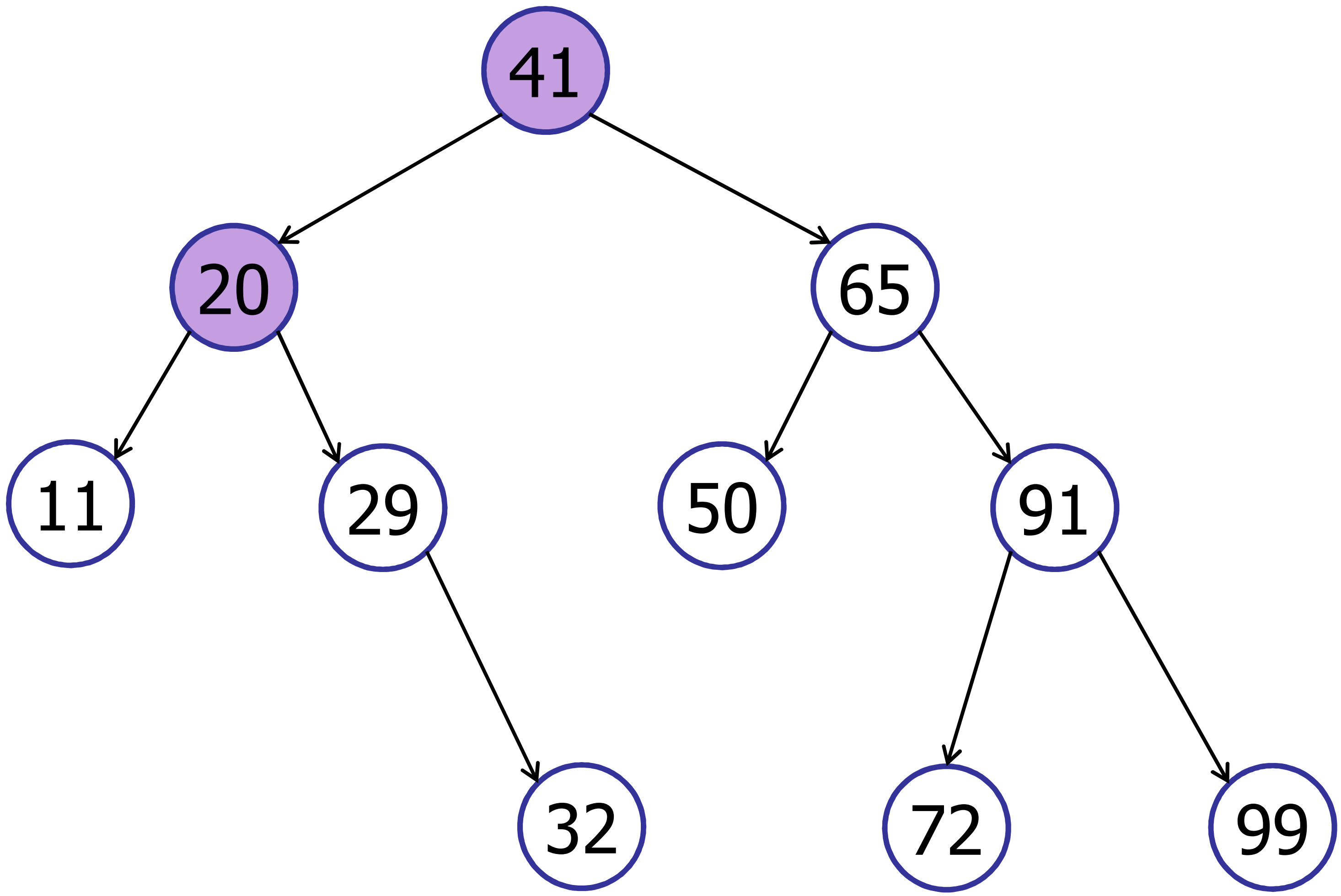
Binary Search Trees (review)

insert(27)



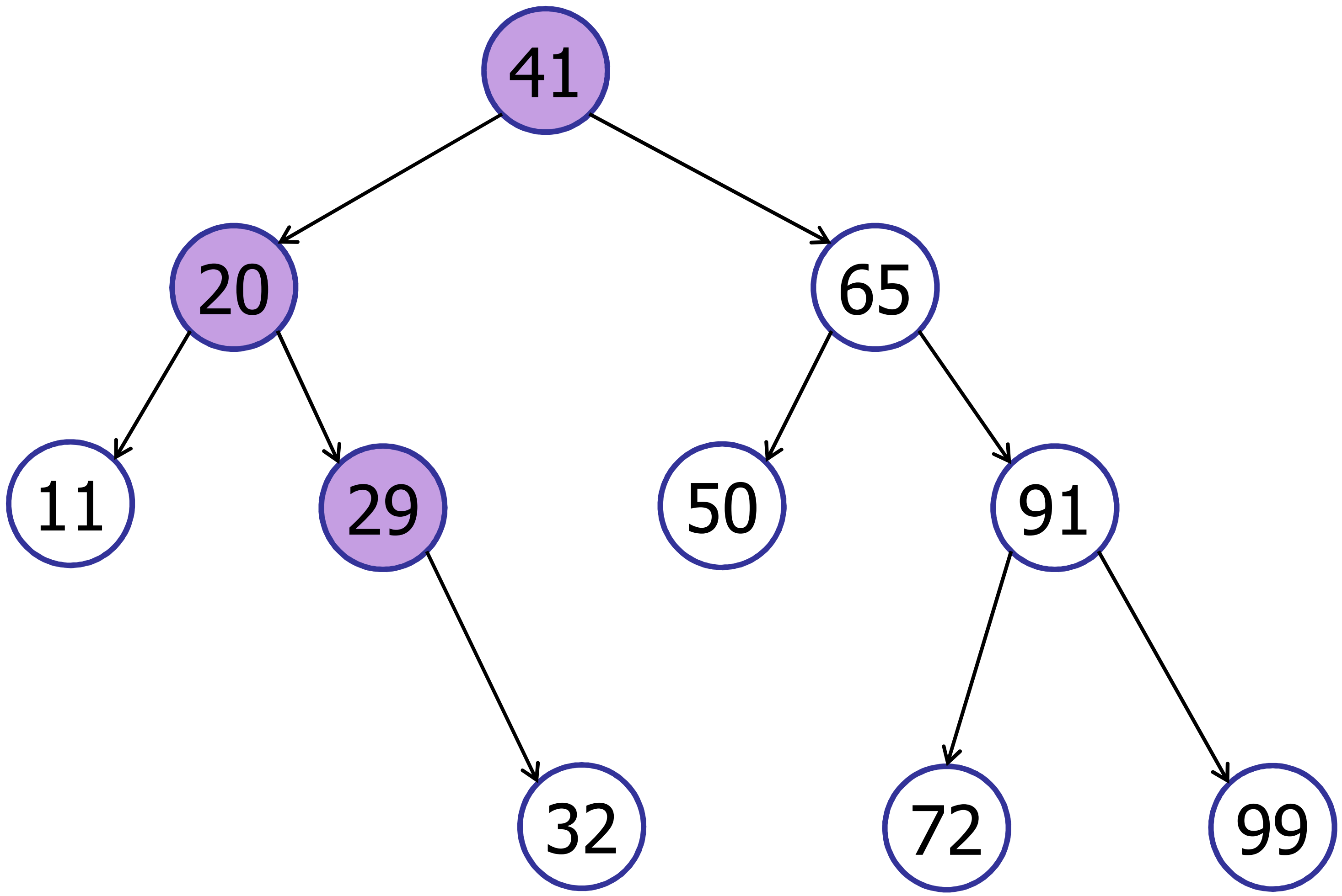
Binary Search Trees (review)

insert(27)



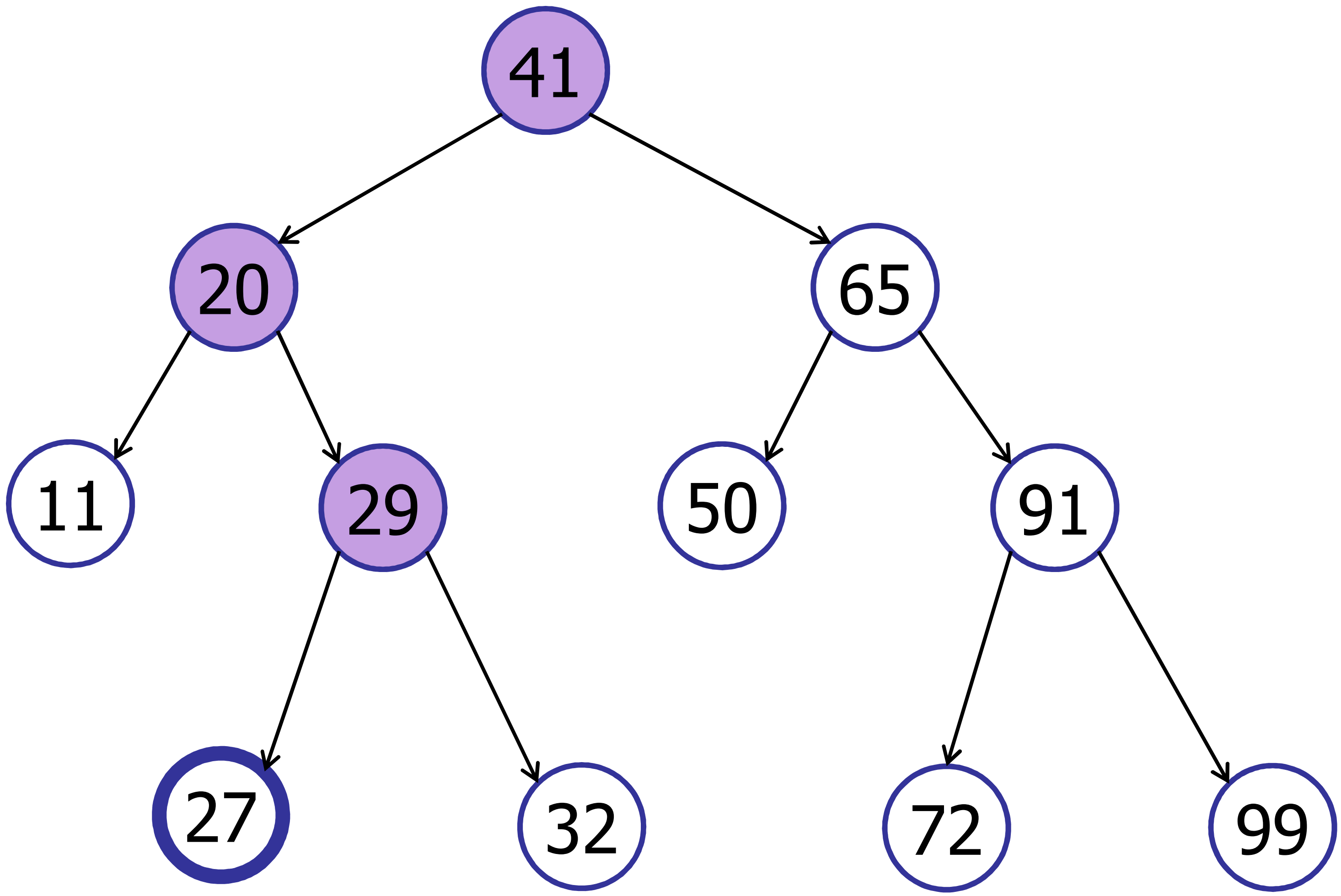
Binary Search Trees (review)

insert(27)



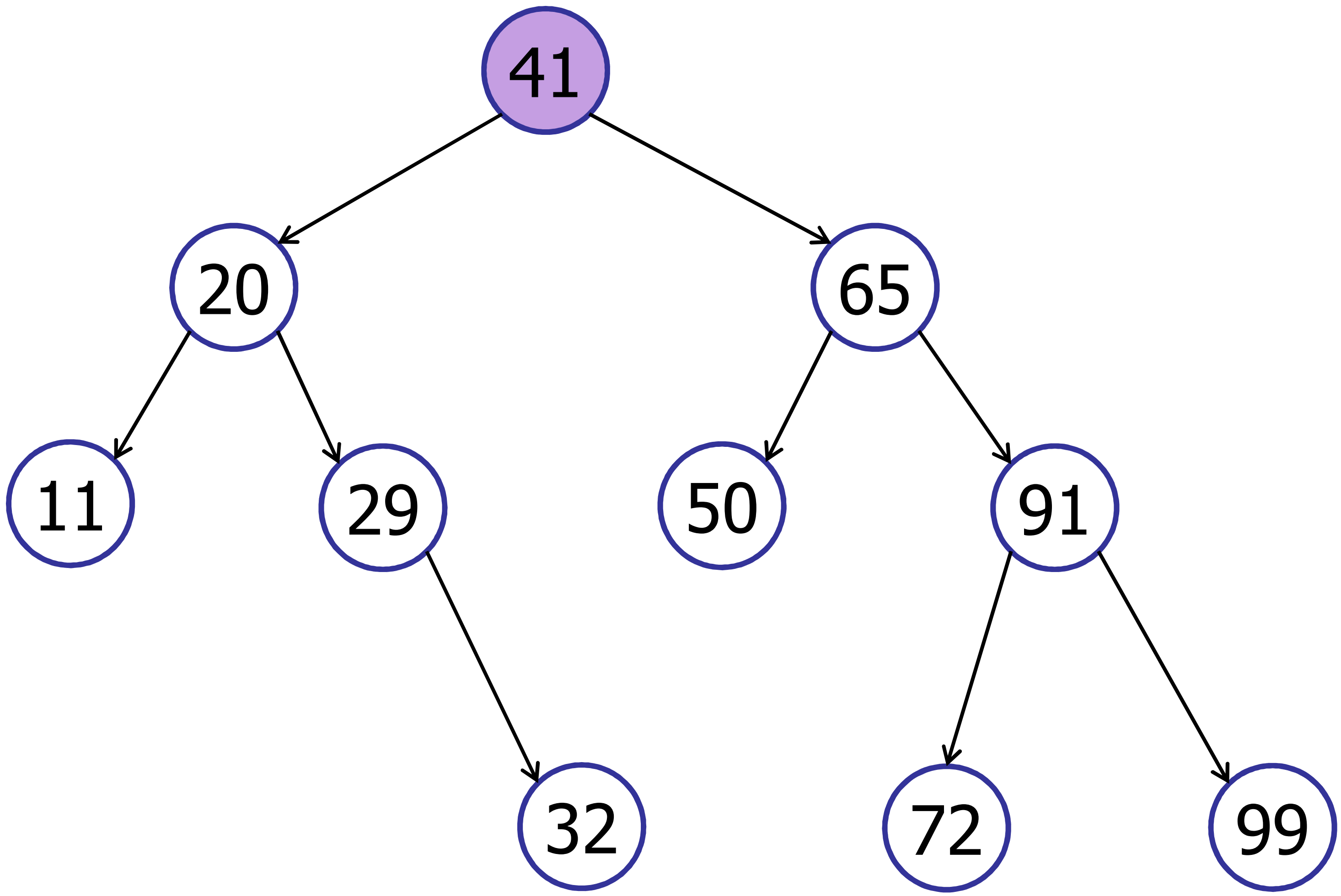
Binary Search Trees (review)

insert(27)



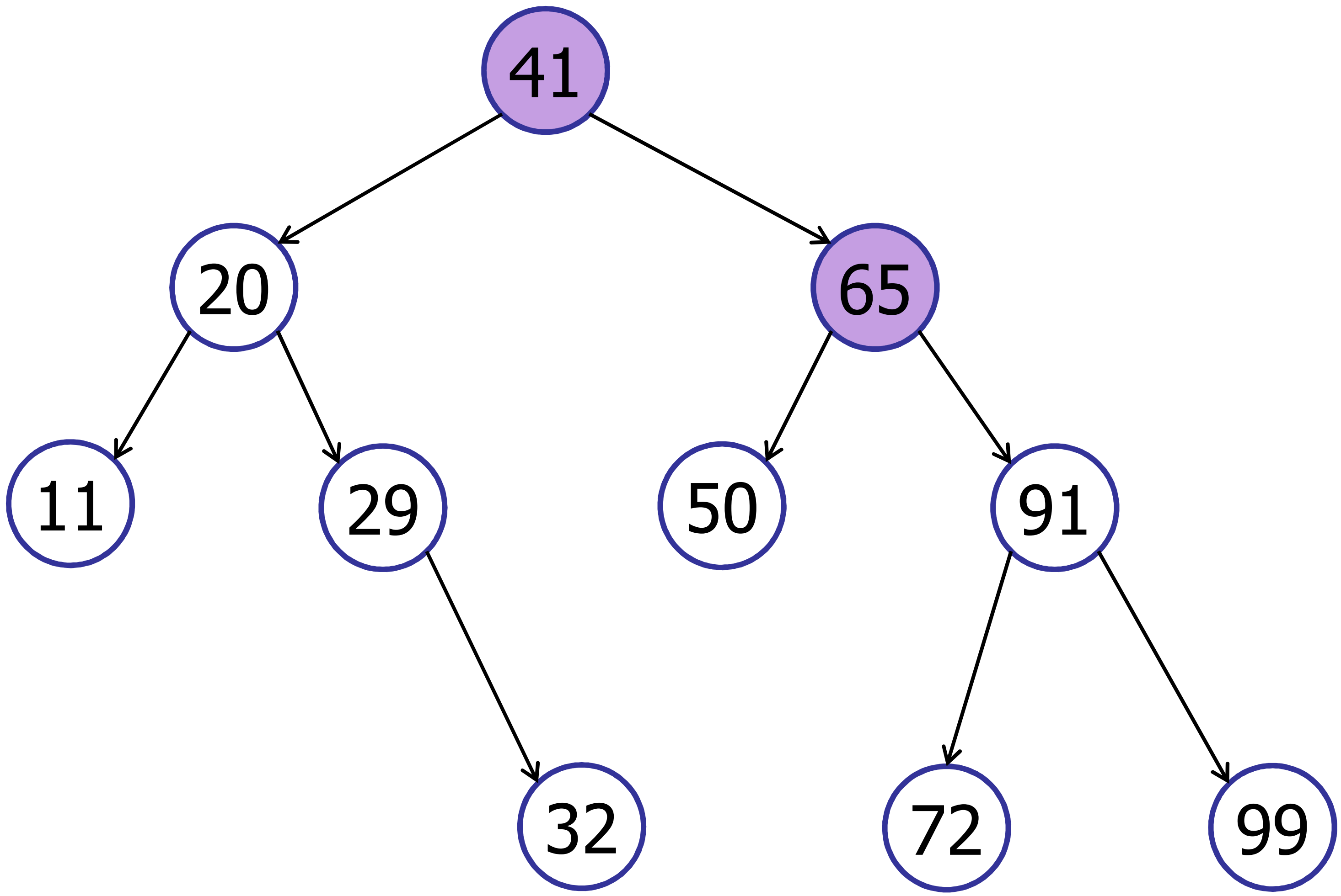
Binary Search Trees (review)

search(72)



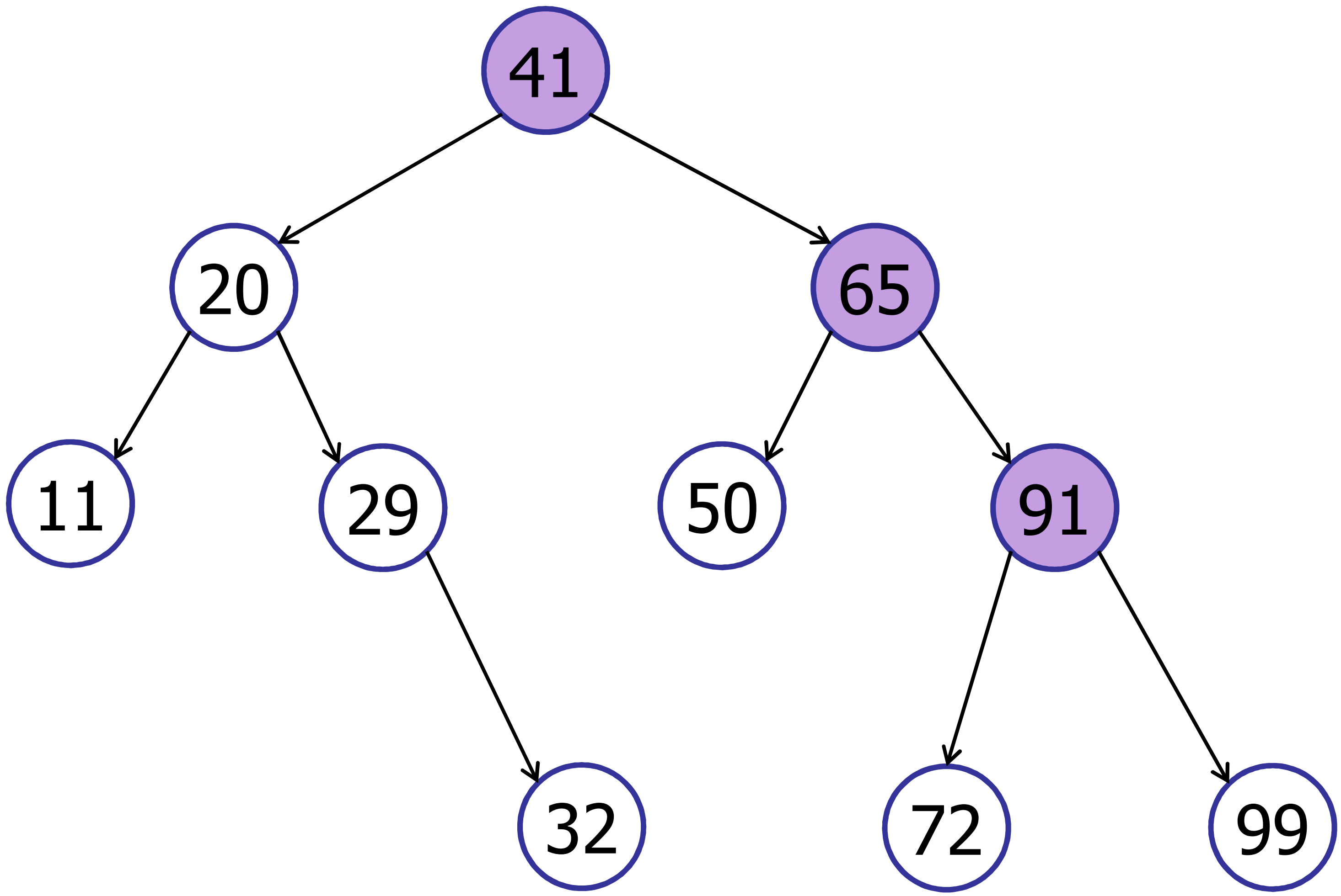
Binary Search Trees (review)

search(72)



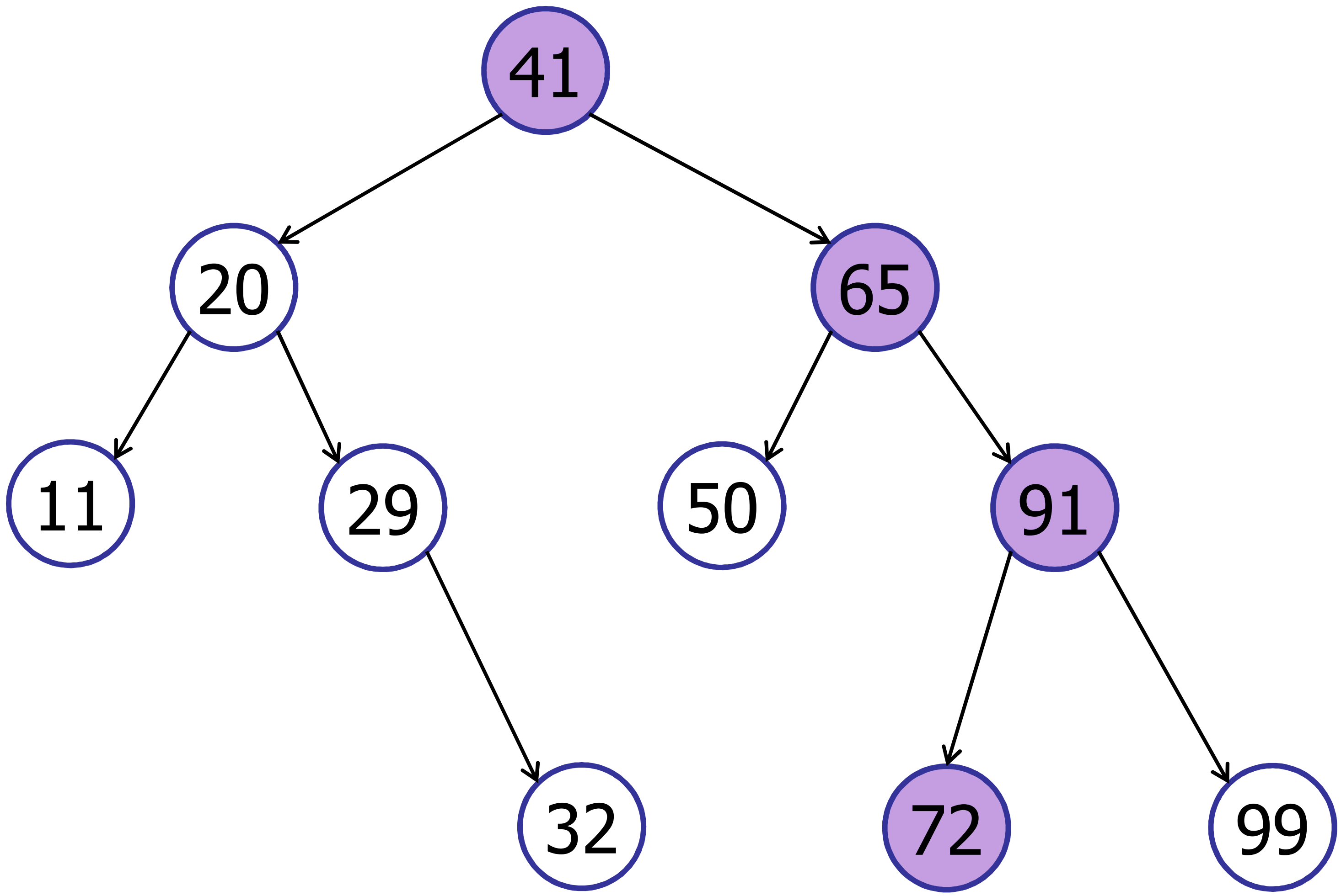
Binary Search Trees (review)

search(72)



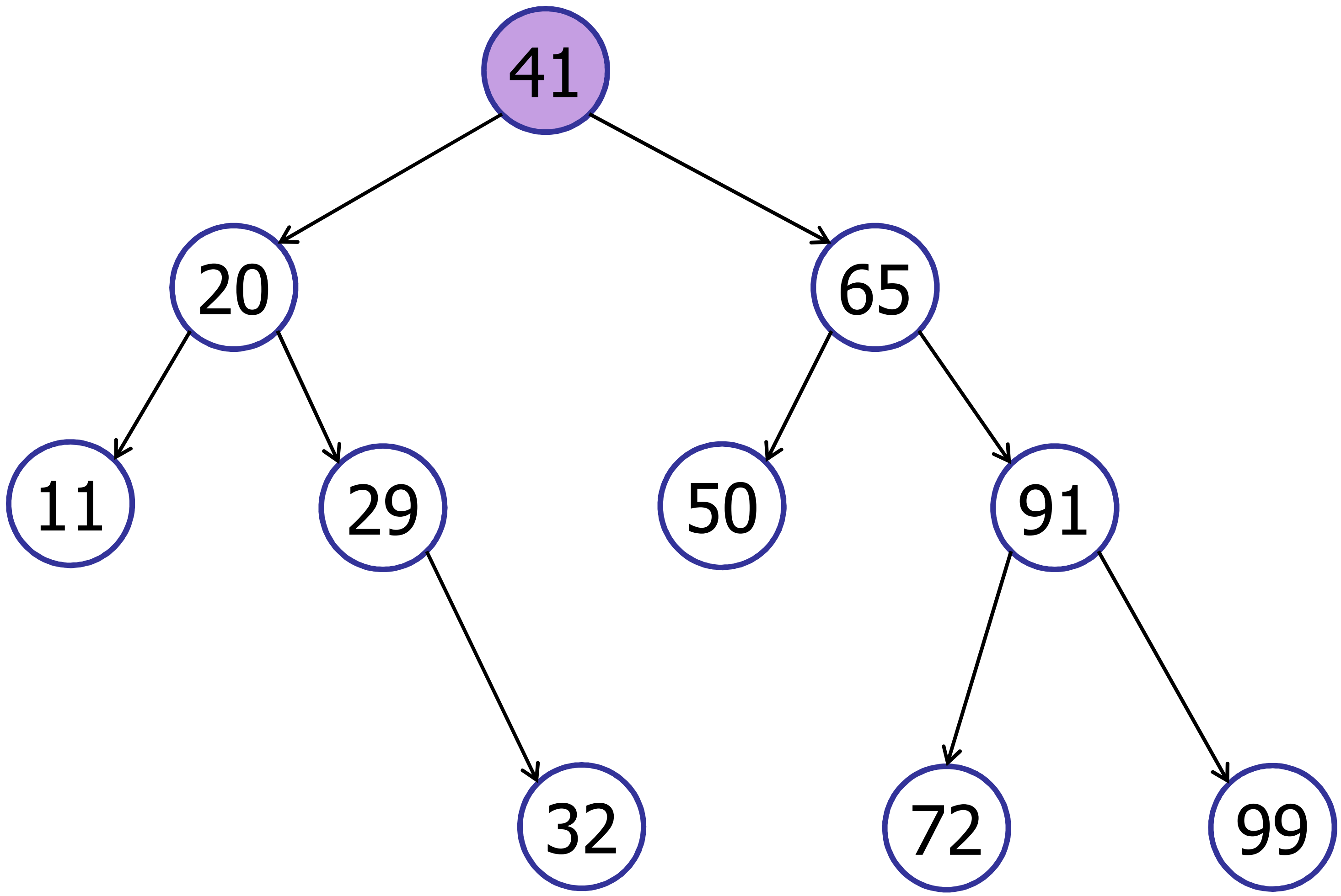
Binary Search Trees (review)

search(72)



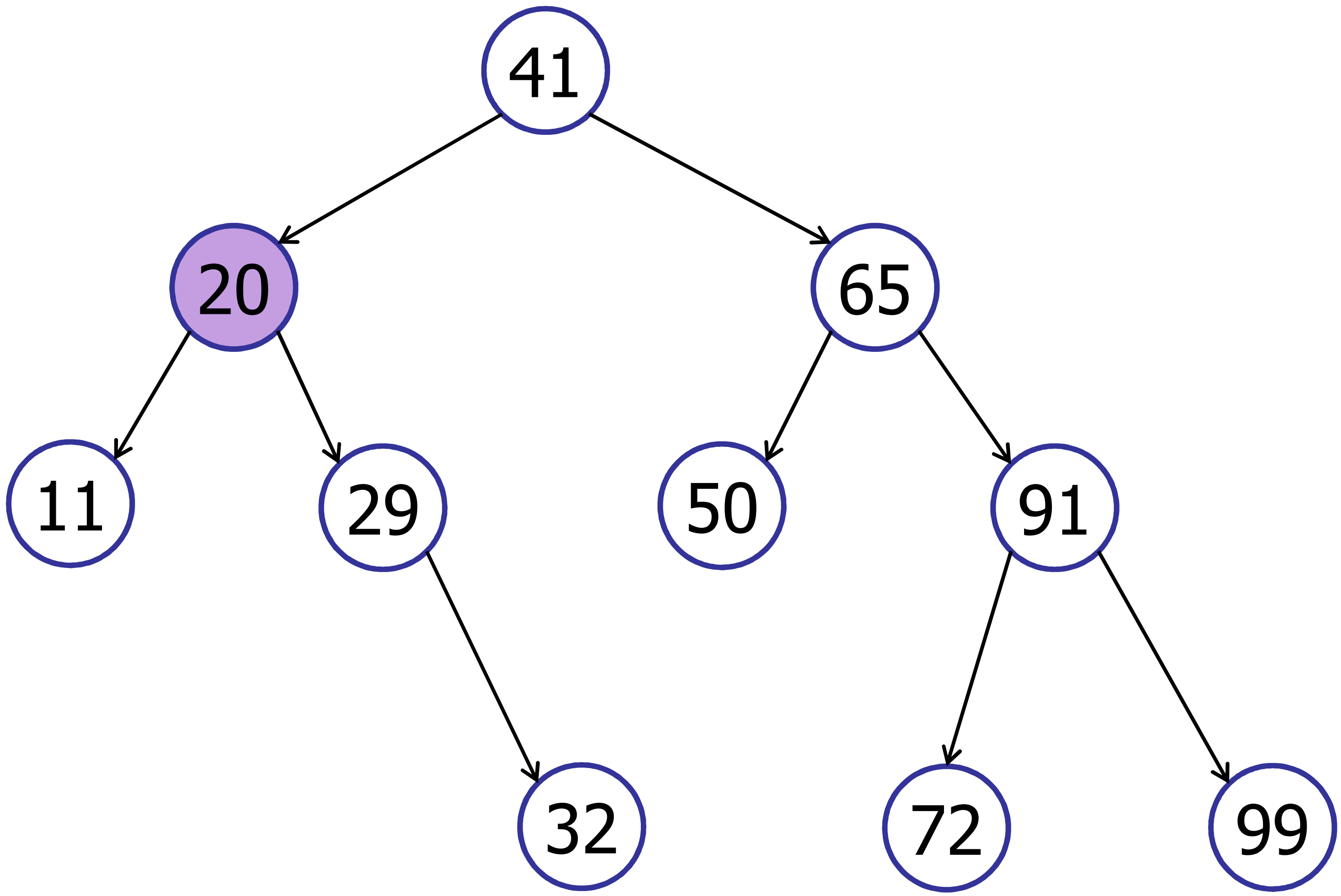
Binary Search Trees

in-order-traversal



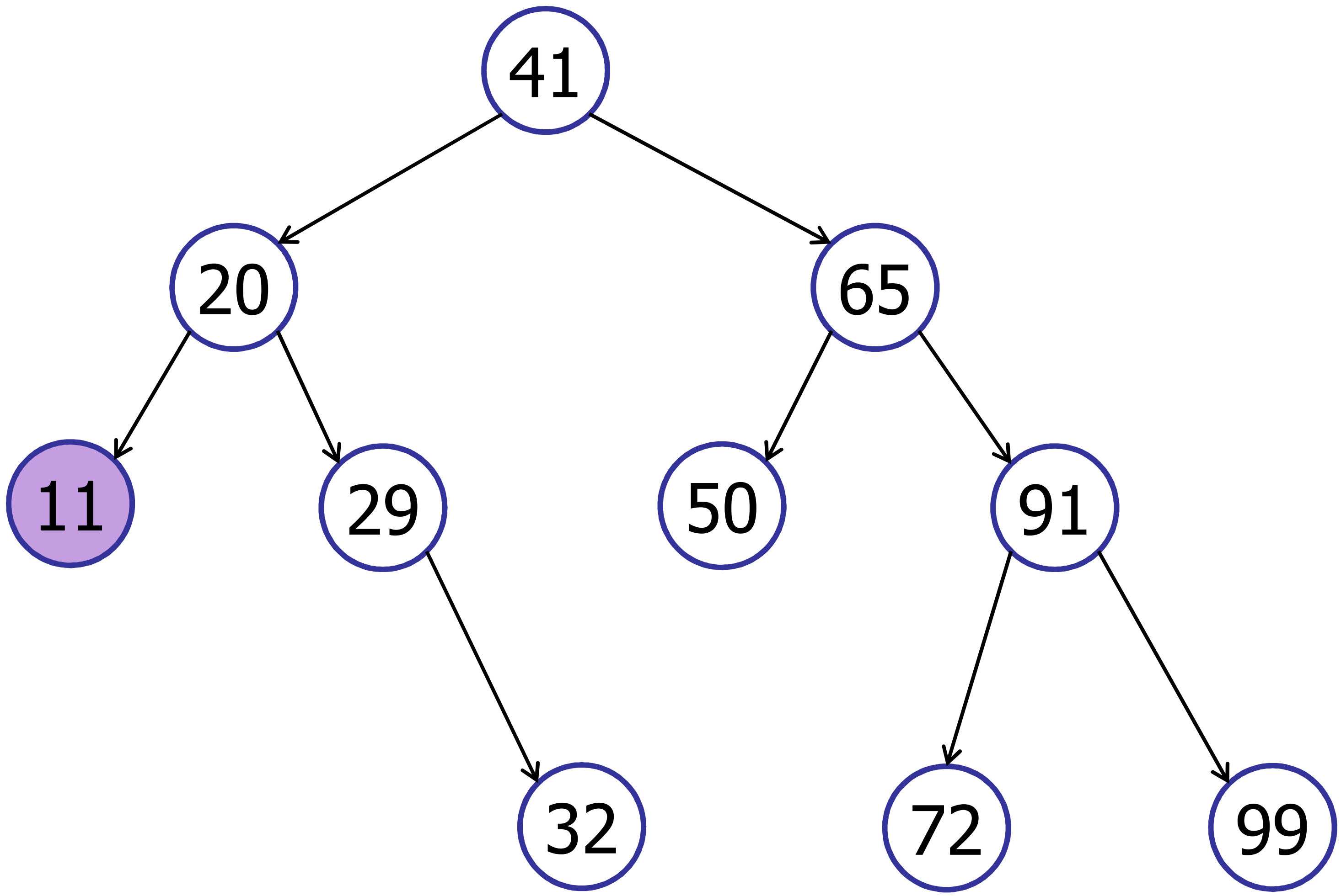
Binary Search Trees

in-order-traversal



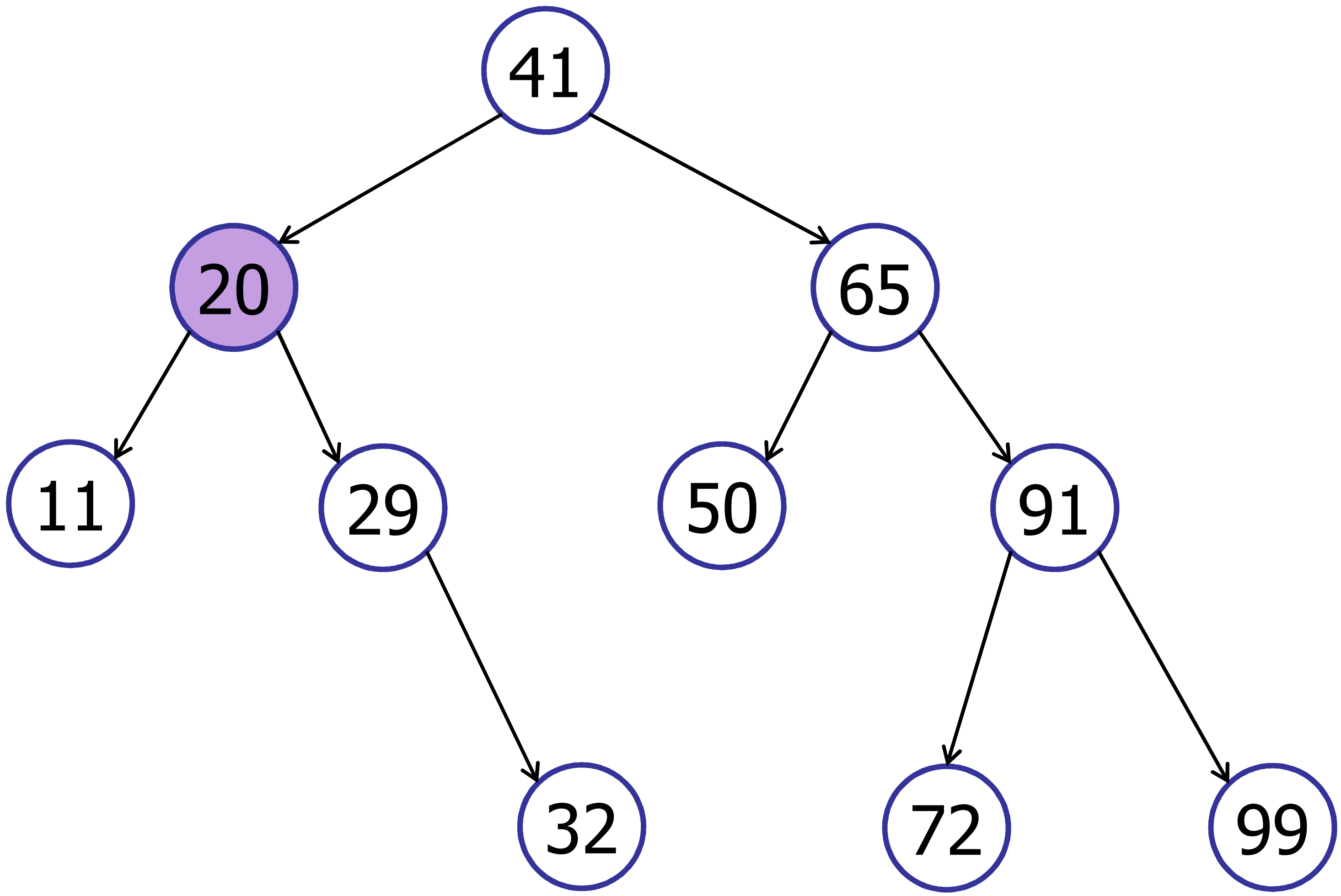
Binary Search Trees

in-order-traversal



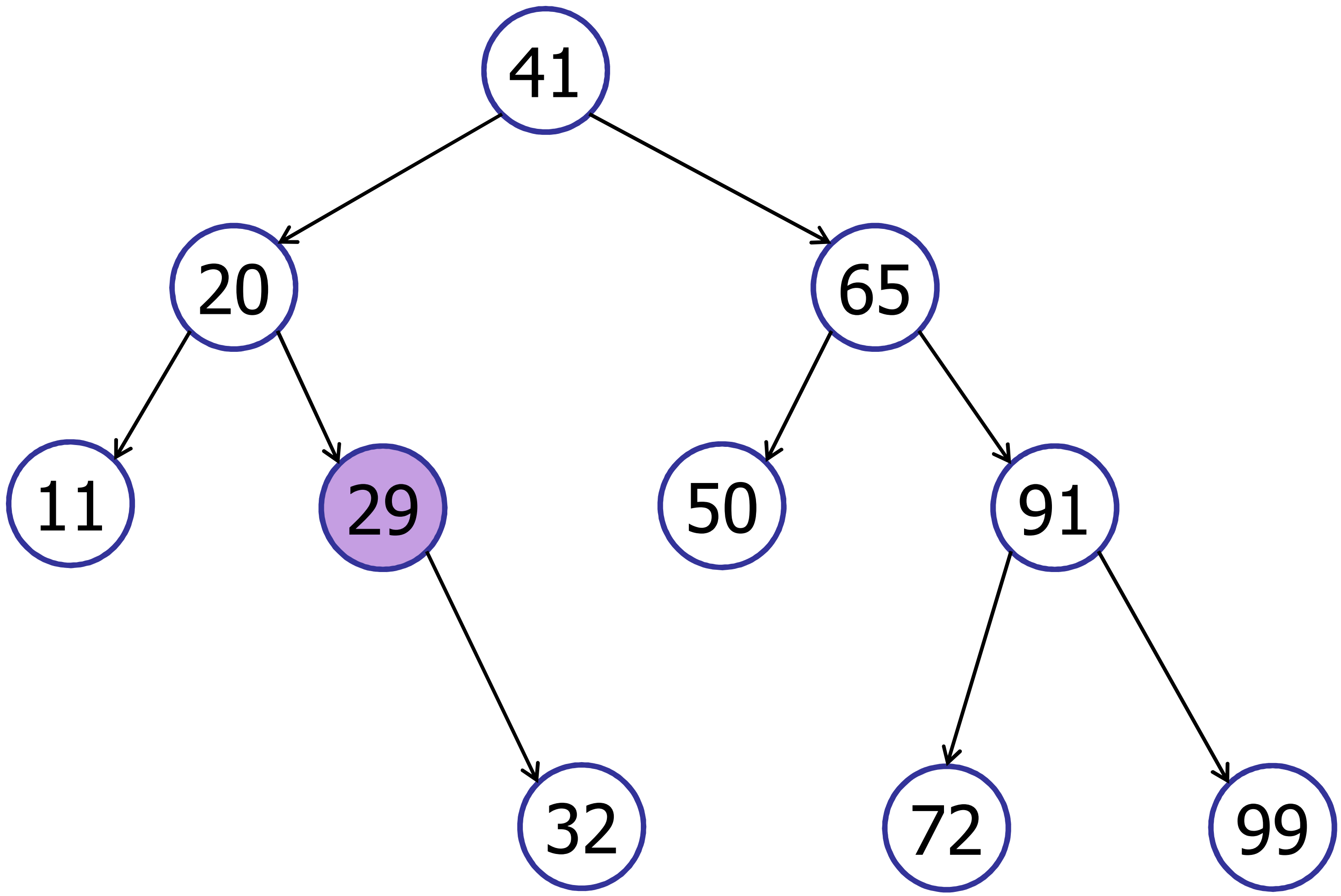
Binary Search Trees

in-order-traversal



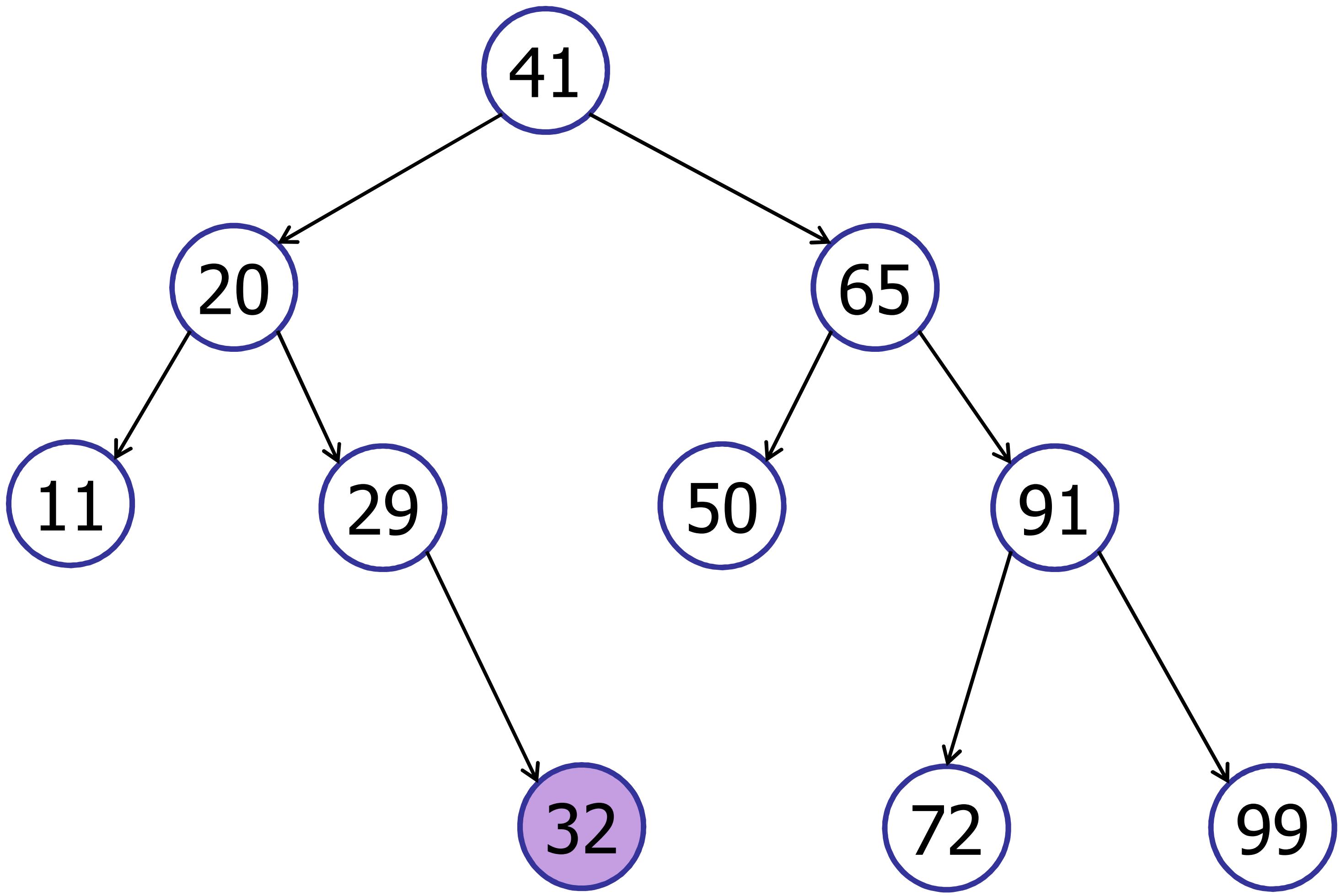
Binary Search Trees

in-order-traversal



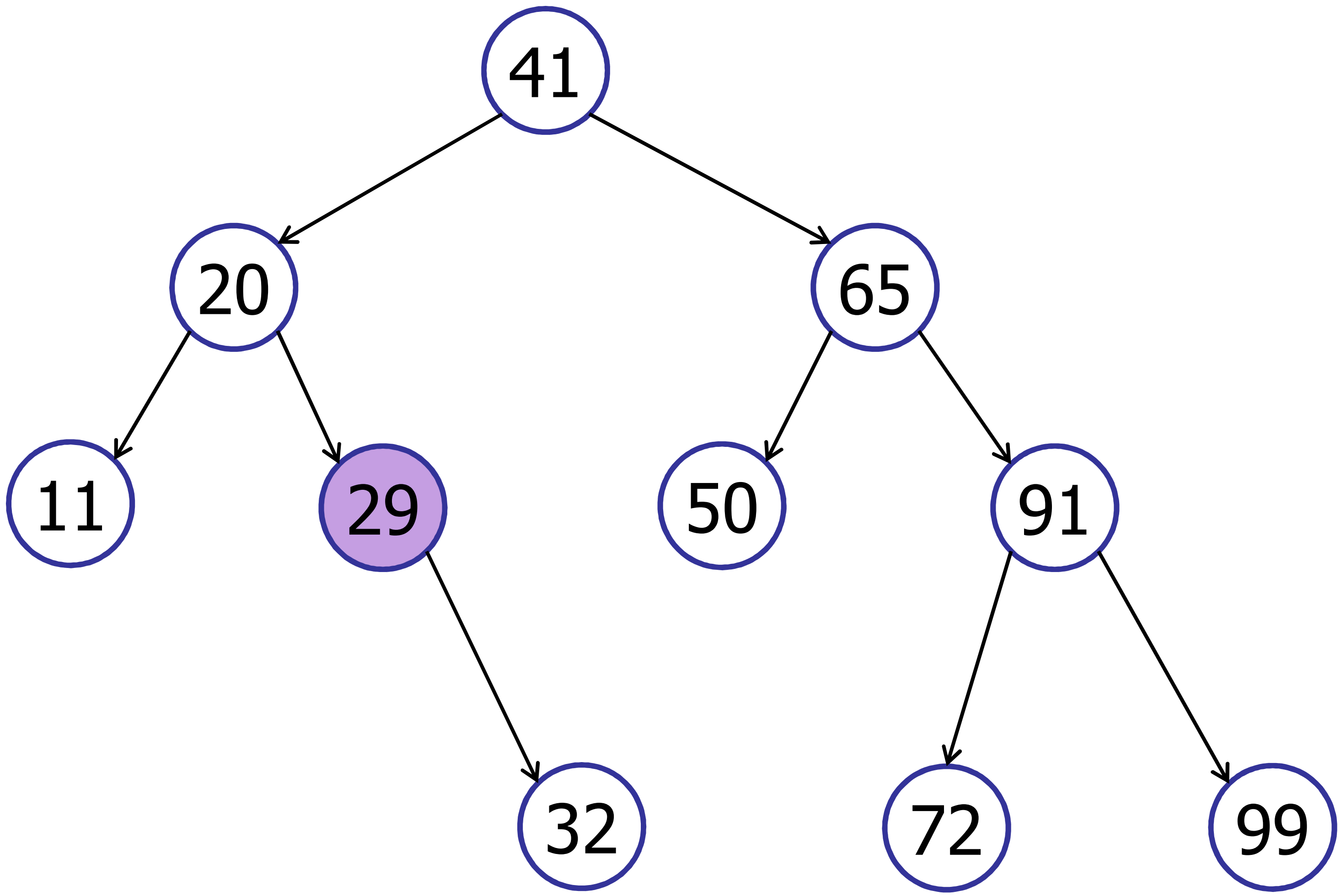
Binary Search Trees

in-order-traversal



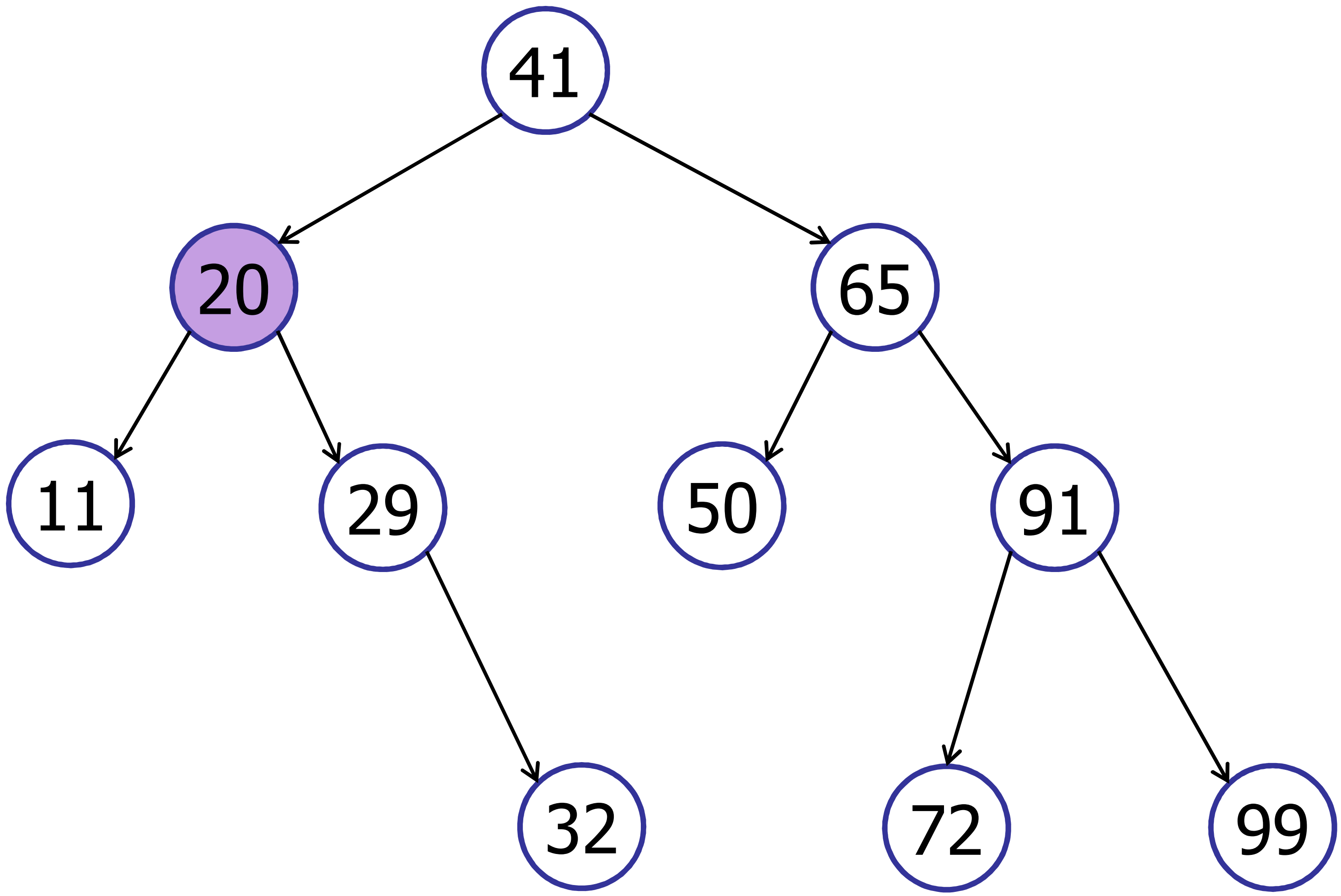
Binary Search Trees

in-order-traversal



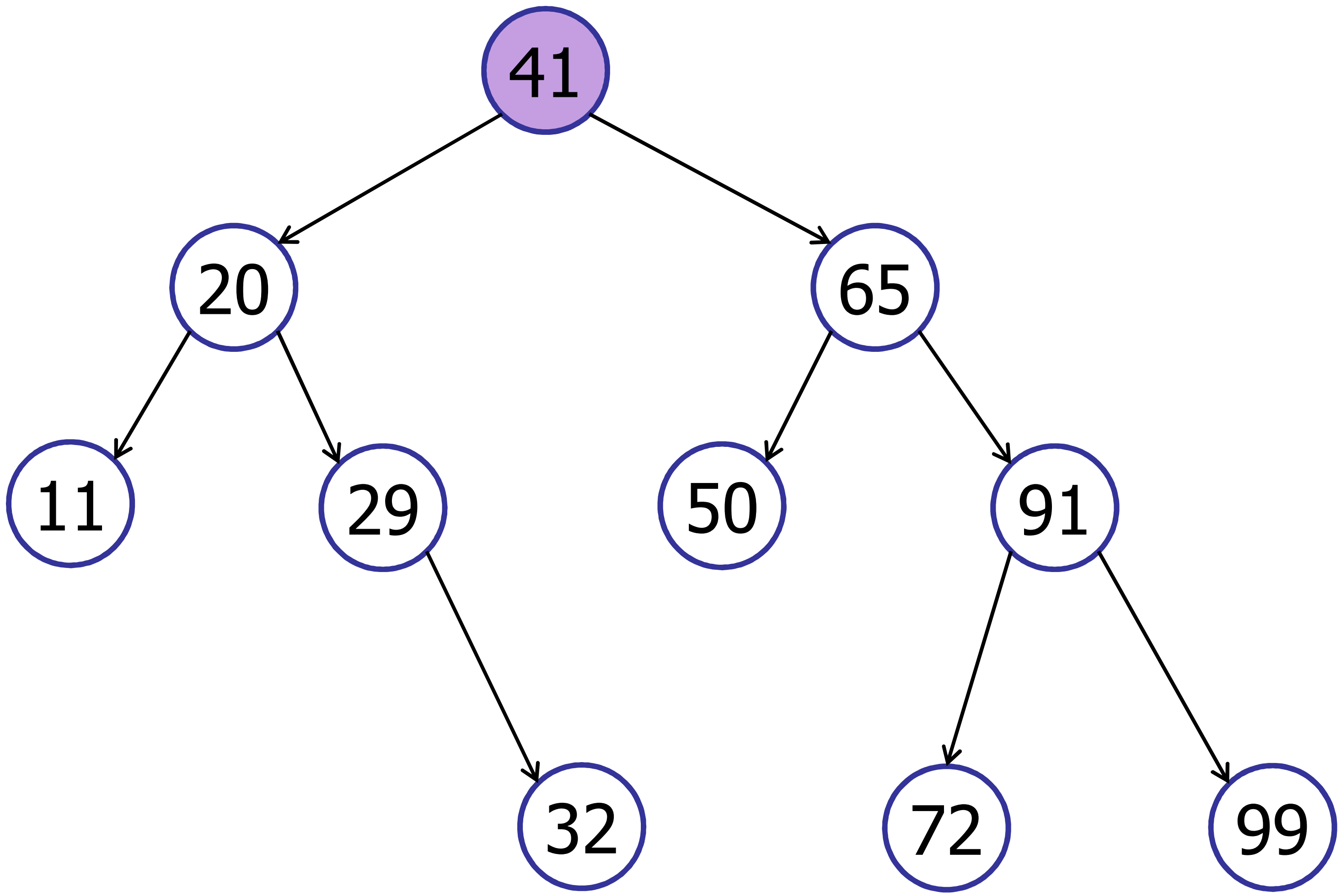
Binary Search Trees

in-order-traversal



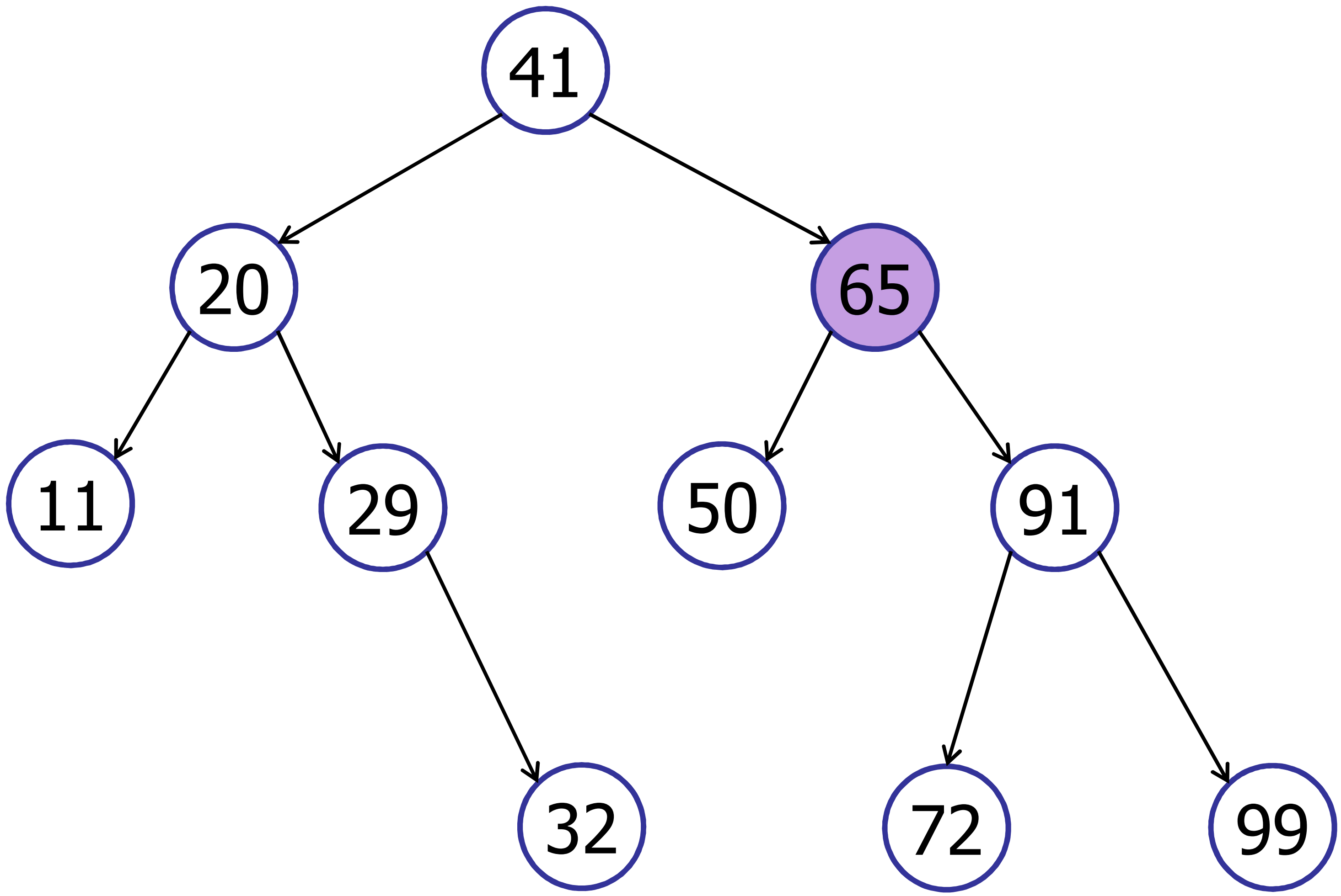
Binary Search Trees

in-order-traversal



Binary Search Trees

in-order-traversal



Binary Search Trees

in-order-traversal(v)

```
in-order-traversal(v.left);
output v.key;
in-order-traversal(v.right);
```

Running time: O(n)

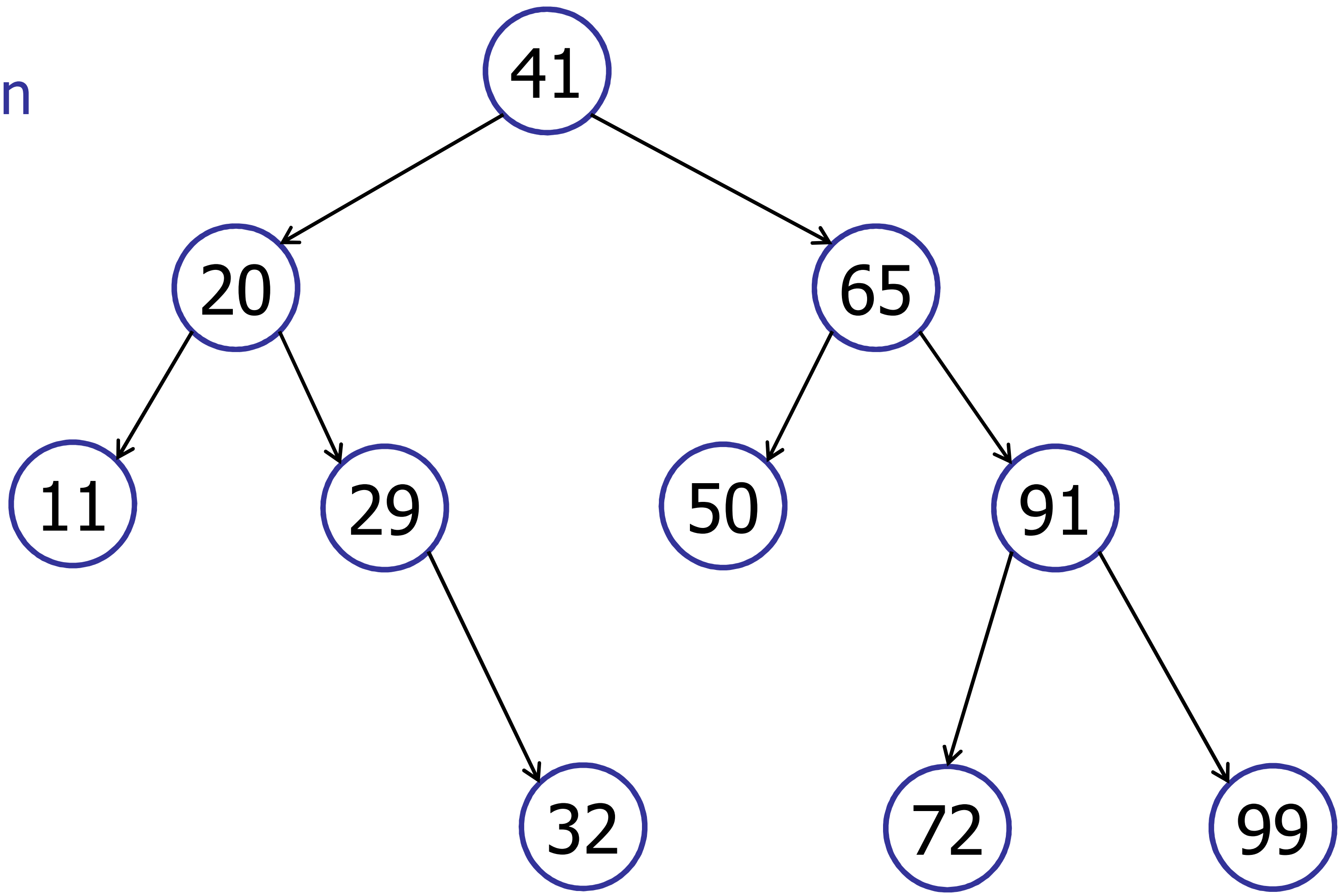
- visits each node at most once

Binary Search Tree

delete(v)

Three cases:

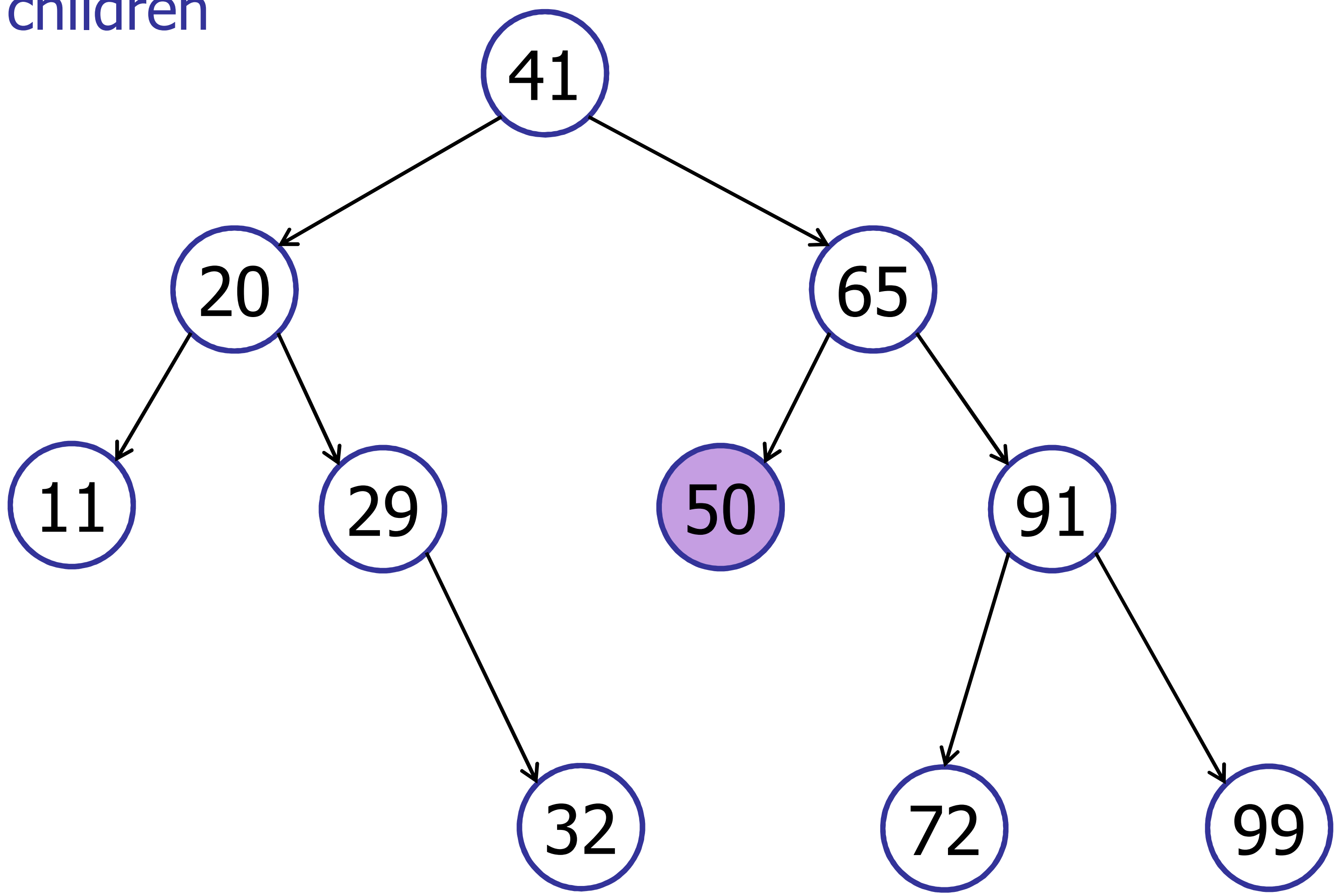
1. No children
2. 1 child
3. 2 children



Binary Search Tree

delete(50)

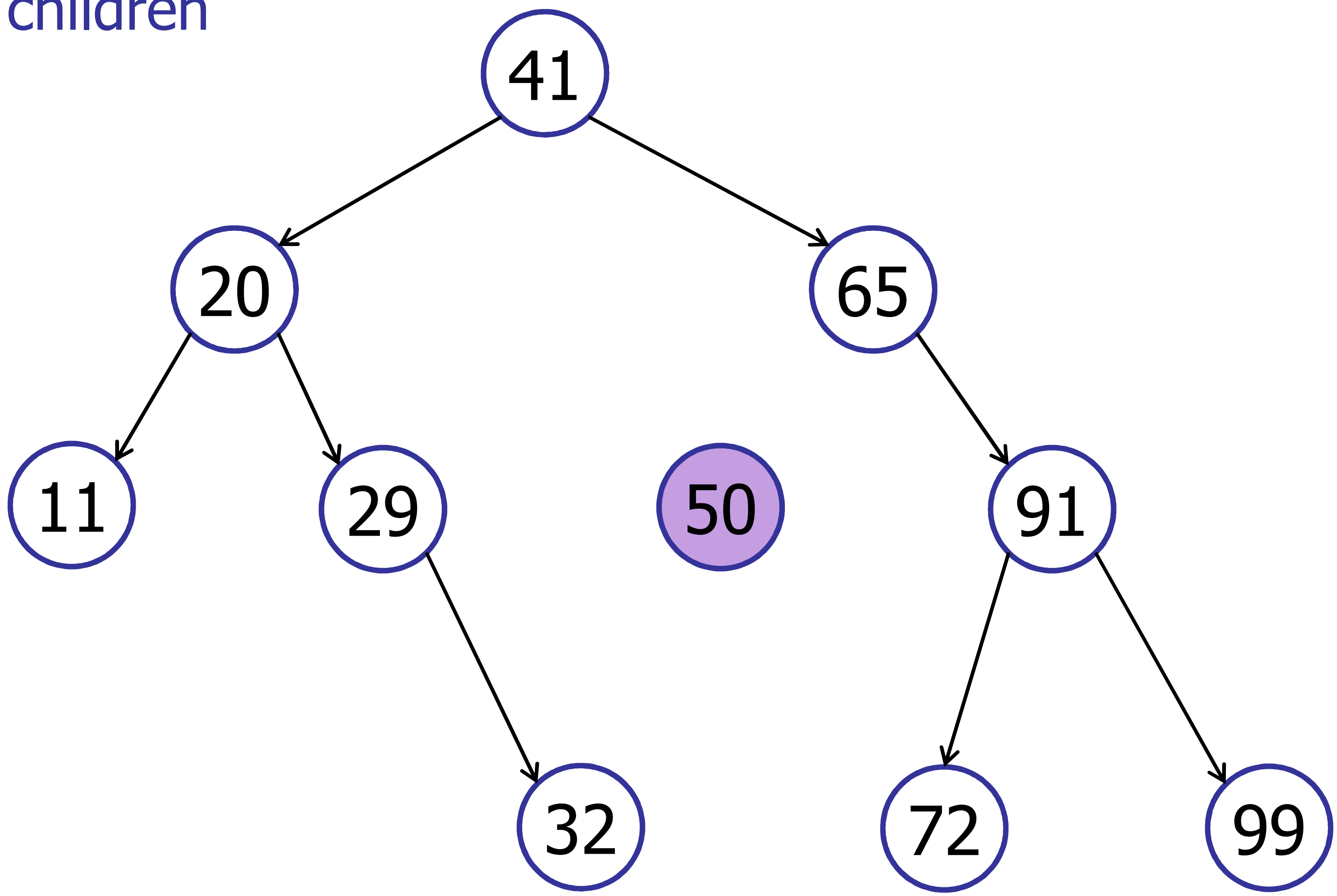
Case 1: No children



Binary Search Tree

delete(50)

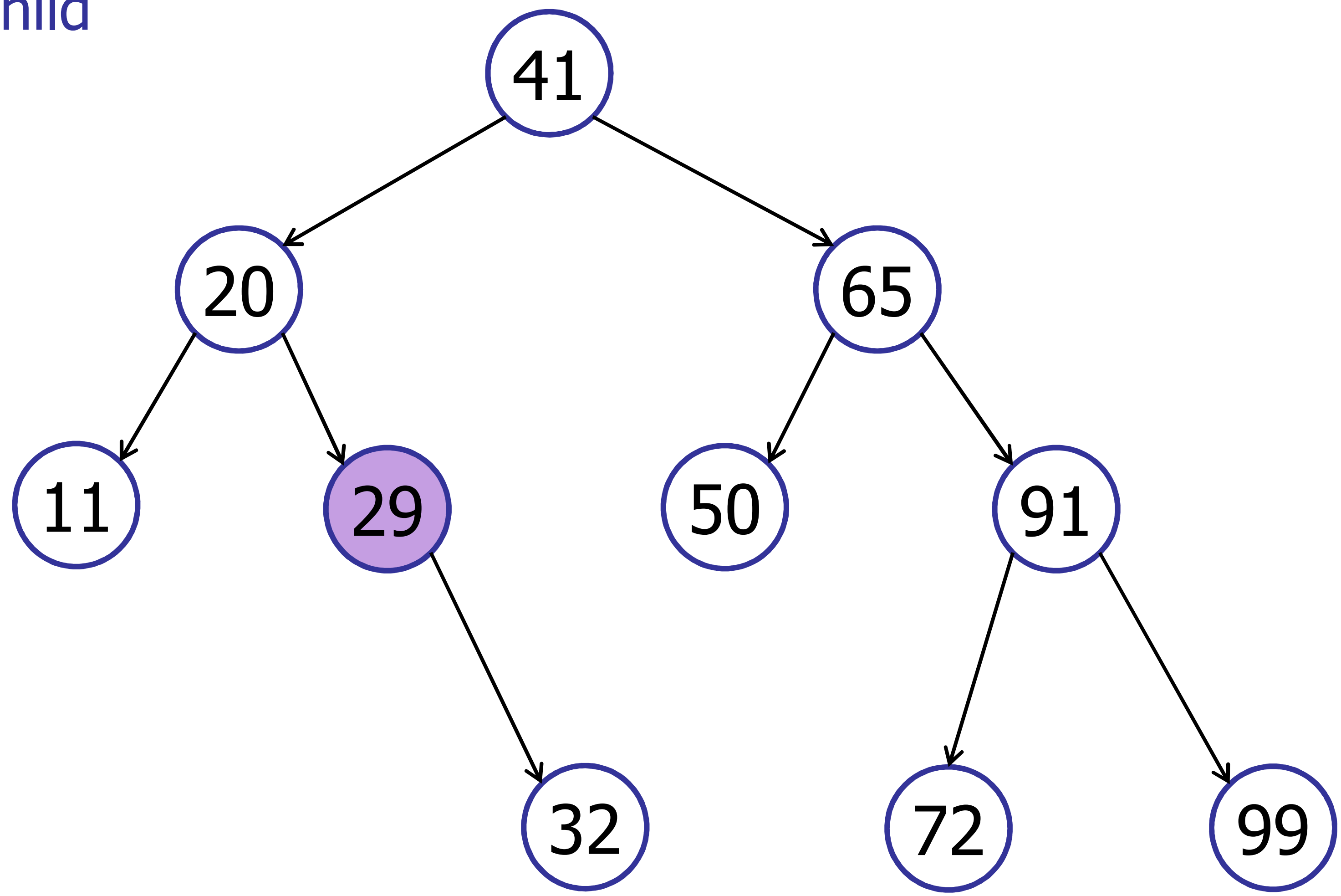
Case 1: No children



Binary Search Tree

delete(29)

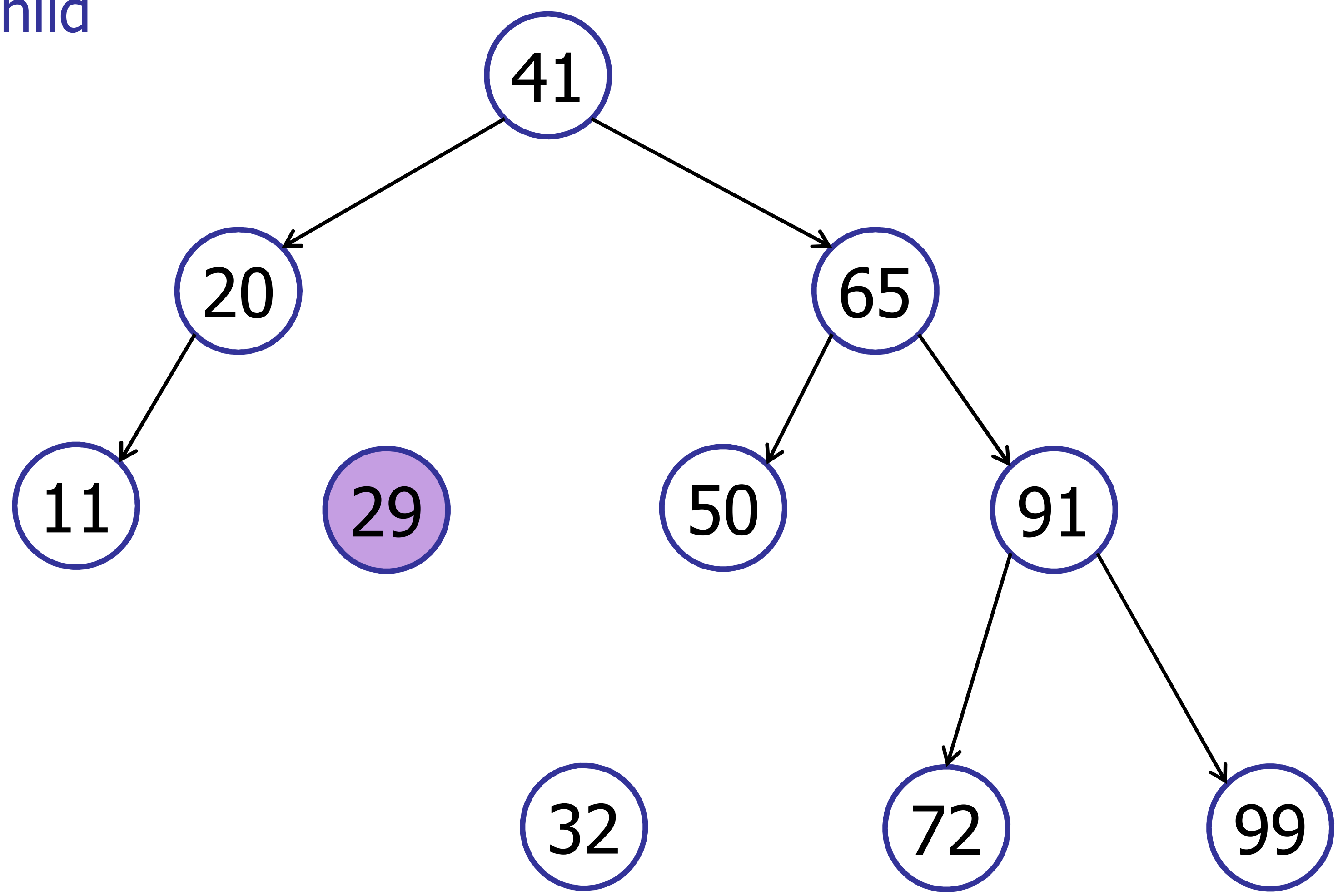
Case 2: 1 child



Binary Search Tree

delete(29)

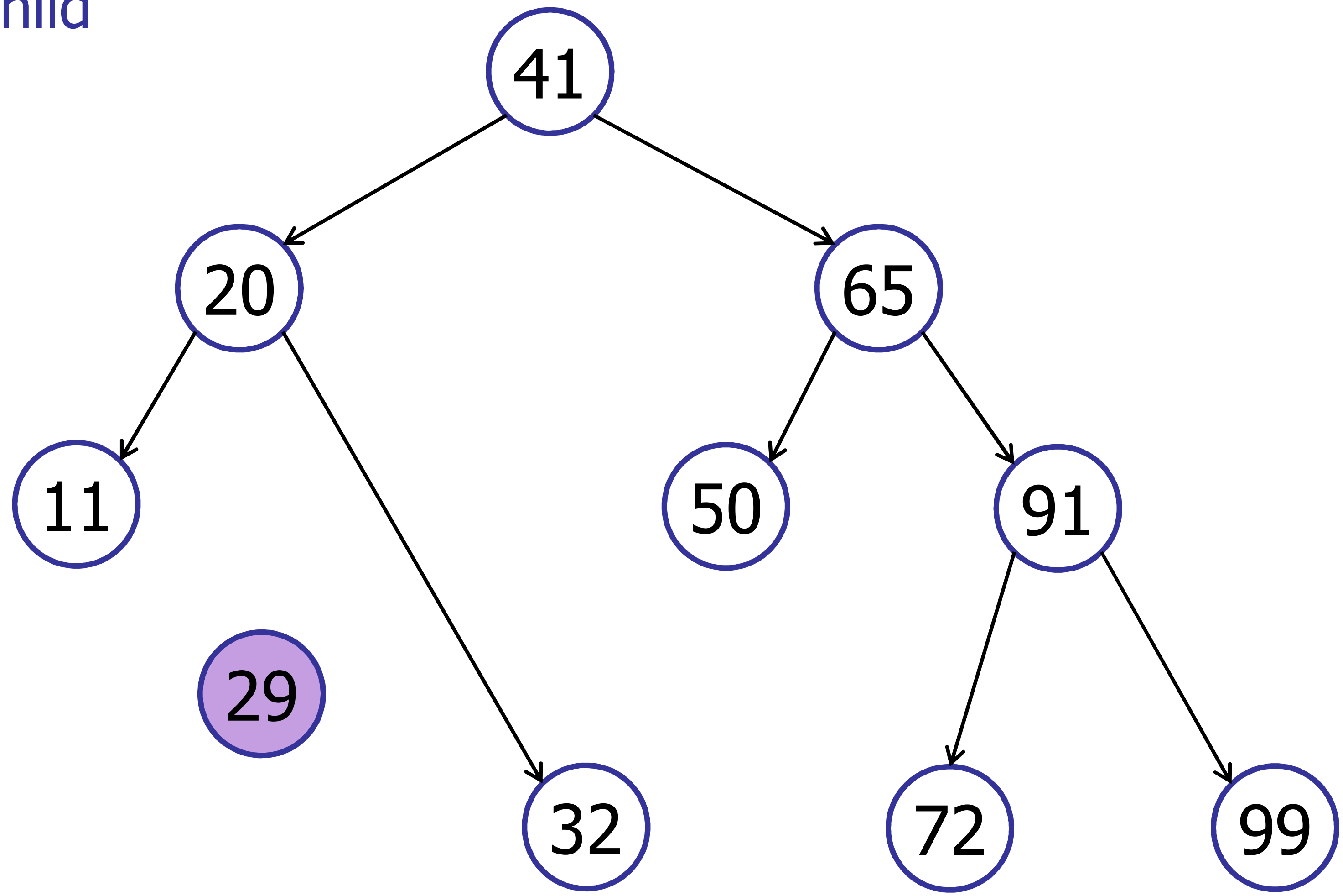
Case 2: 1 child



Binary Search Tree

delete(29)

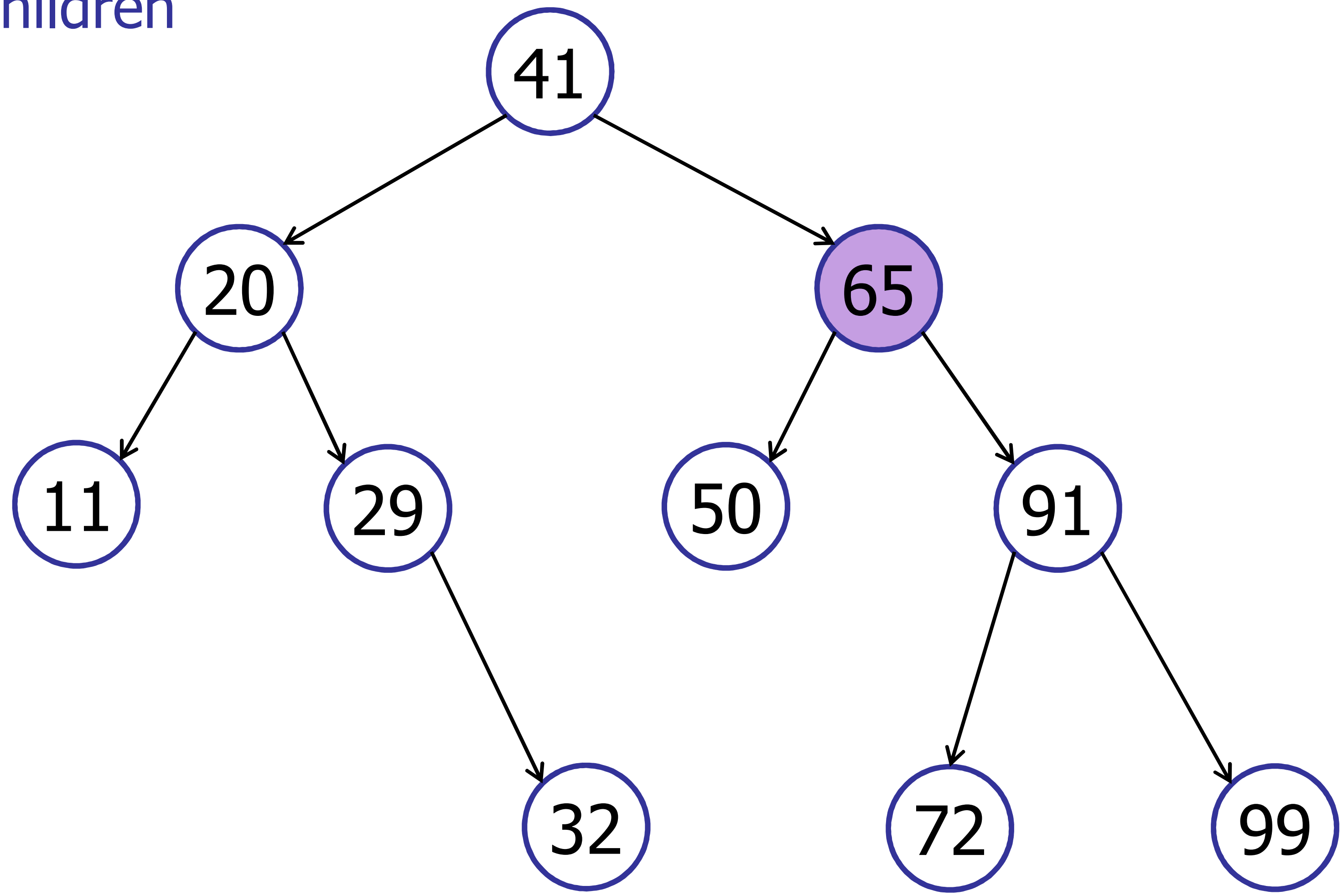
Case 2: 1 child



Binary Search Tree

delete(65)

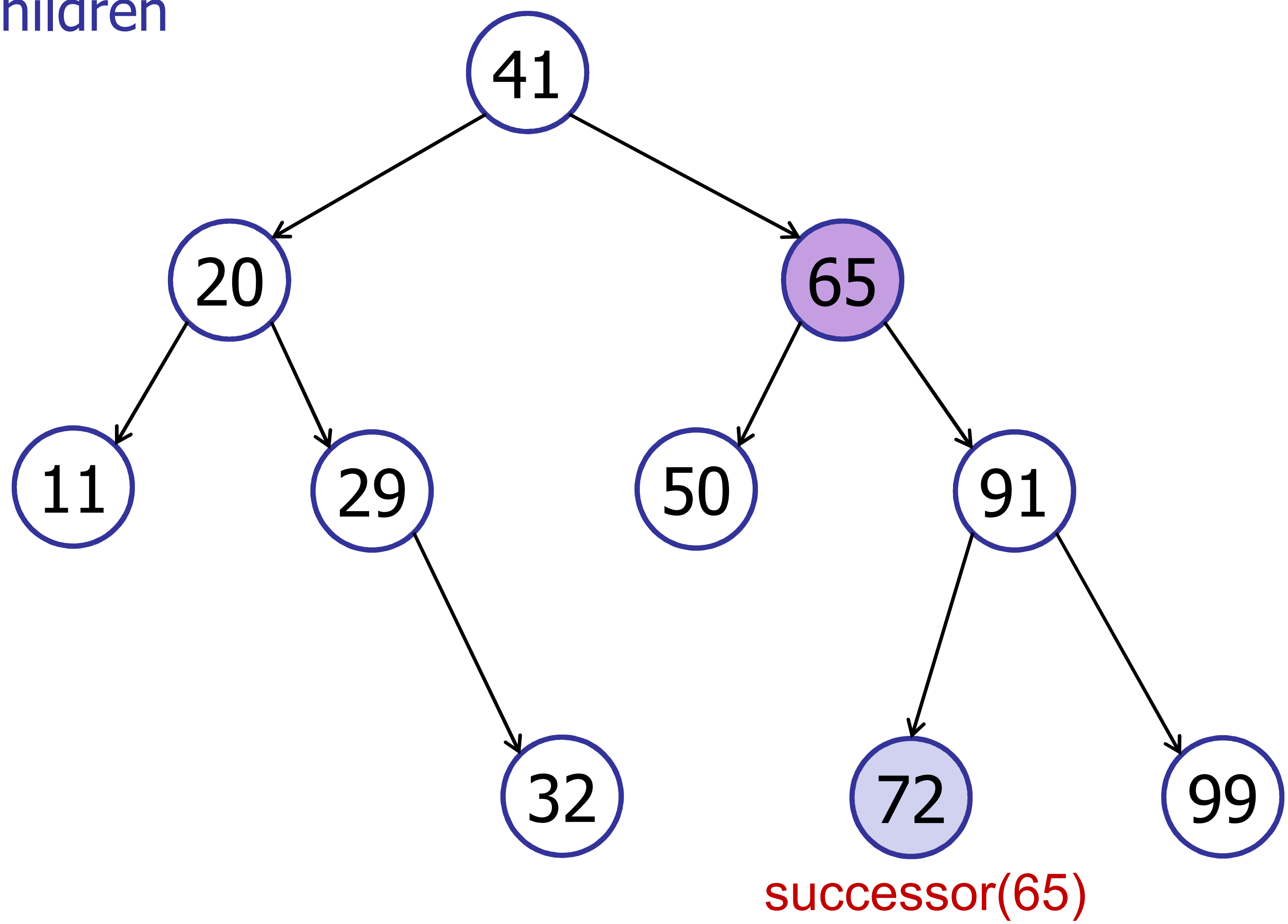
Case 3: 2 children



Binary Search Tree

delete(65)

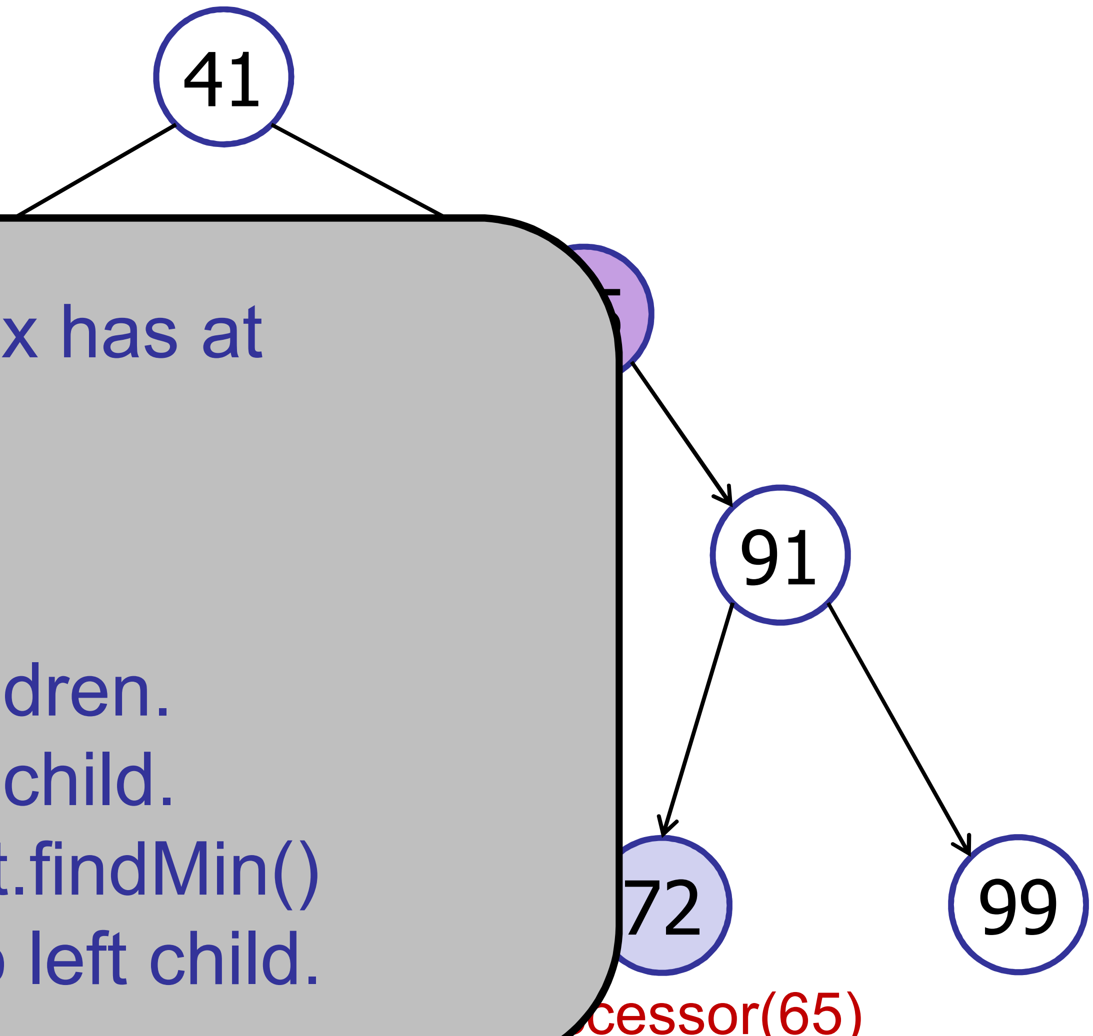
Case 3: 2 children



Binary Search Tree

delete(65)

Case 3: 2 children



Claim: successor of x has at most 1 child!

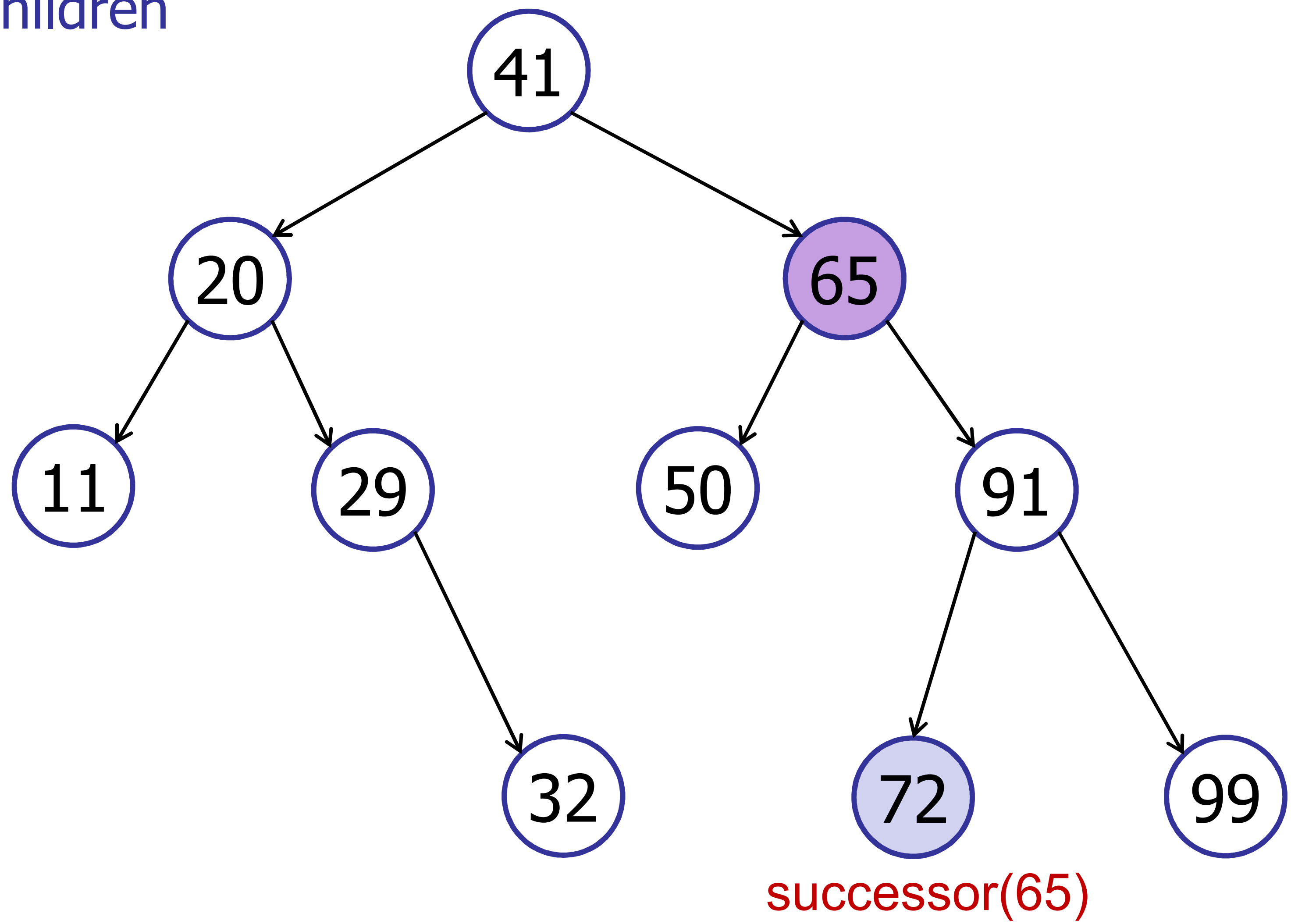
Proof:

- Node x has two children.
- Node x has a **right** child.
- successor(x) = right.findMin()
- min element has no left child.

Binary Search Tree

delete(65)

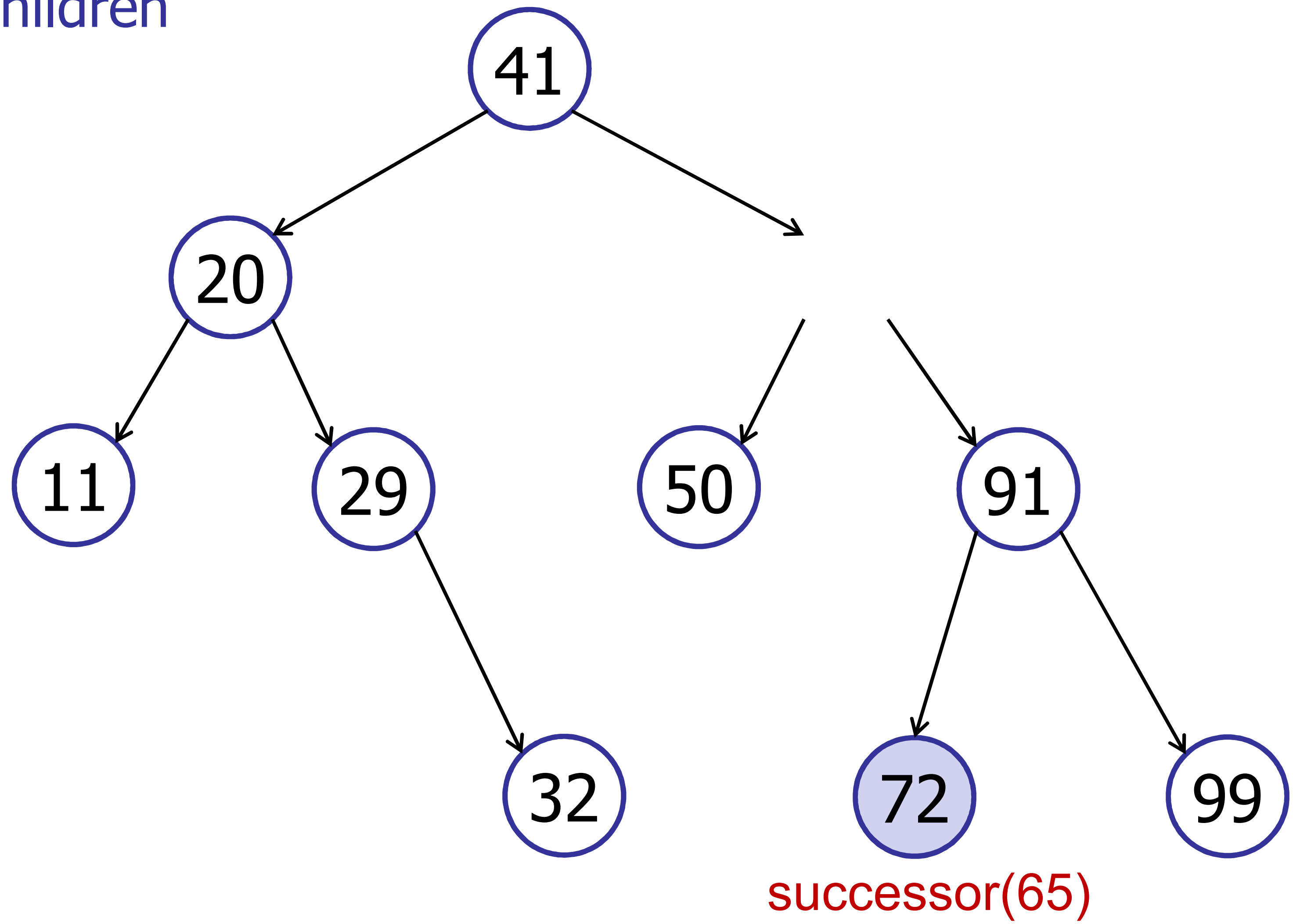
Case 3: 2 children



Binary Search Tree

delete(65)

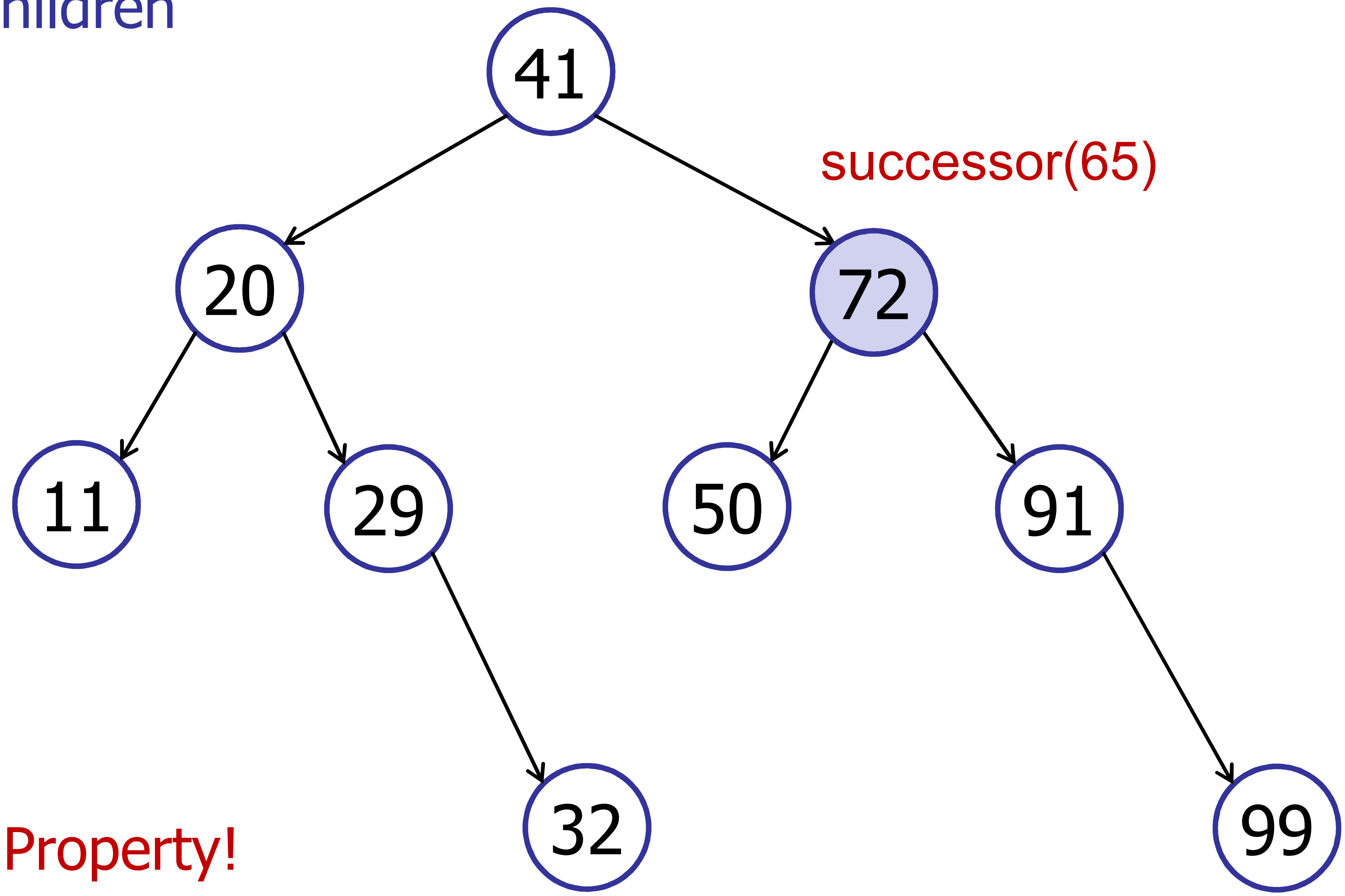
Case 3: 2 children



Binary Search Tree

delete(65)

Case 3: 2 children



Check BST Property!

Binary Search Tree

delete(v)

Running time: O(h)

Three cases:

1. No children:
   - remove v
2. 1 child:
   - remove v
   - connect child(v) to parent(v)
3. 2 children
   - x = successor(v)
   - delete(x)
   - remove v
   - connect x to left(v), right(v), parent(v)

Binary Search Tree

Modifying Operations

- insert: O(h)
- delete: O(h)

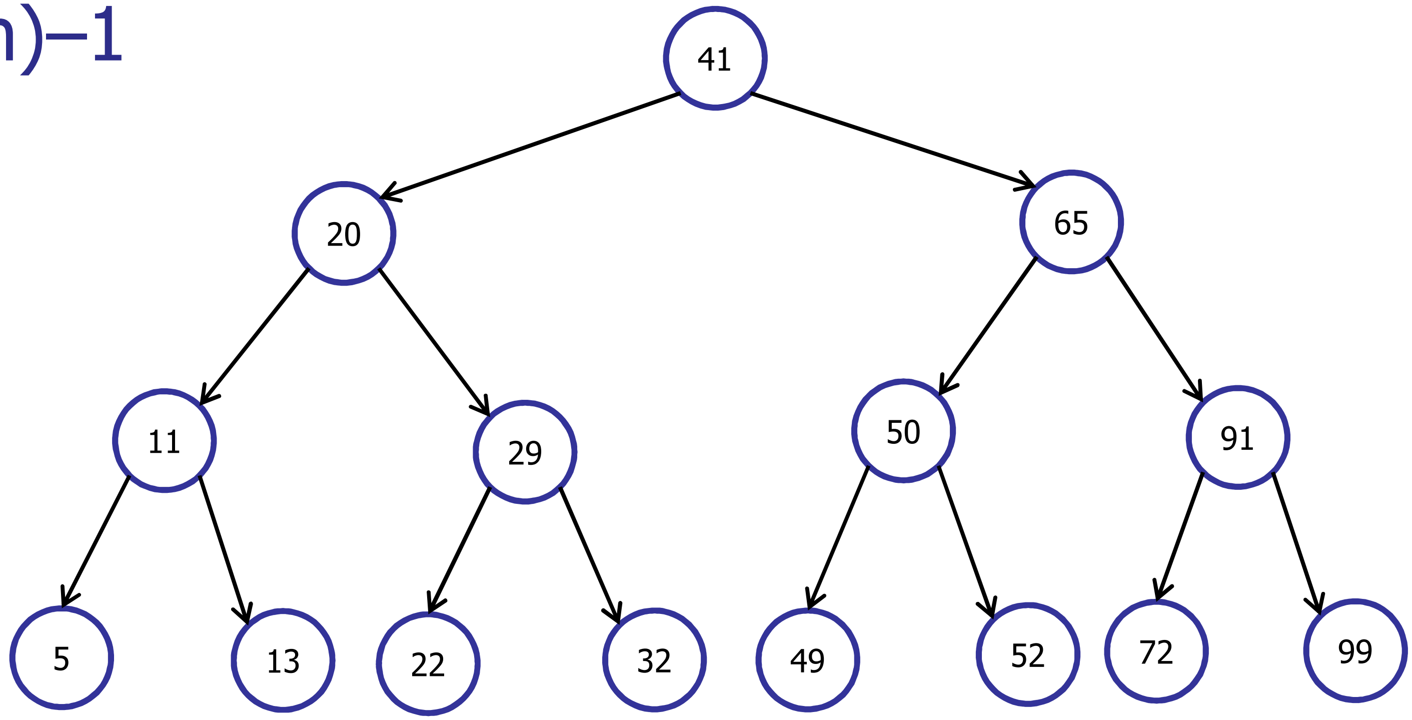
Query Operations:

- search: O(h)
- predecessor, successor: O(h)
- findMax, findMin: O(h)
- in-order-traversal: O(n)

The Importance of Being Balanced

Operations take O(h) time

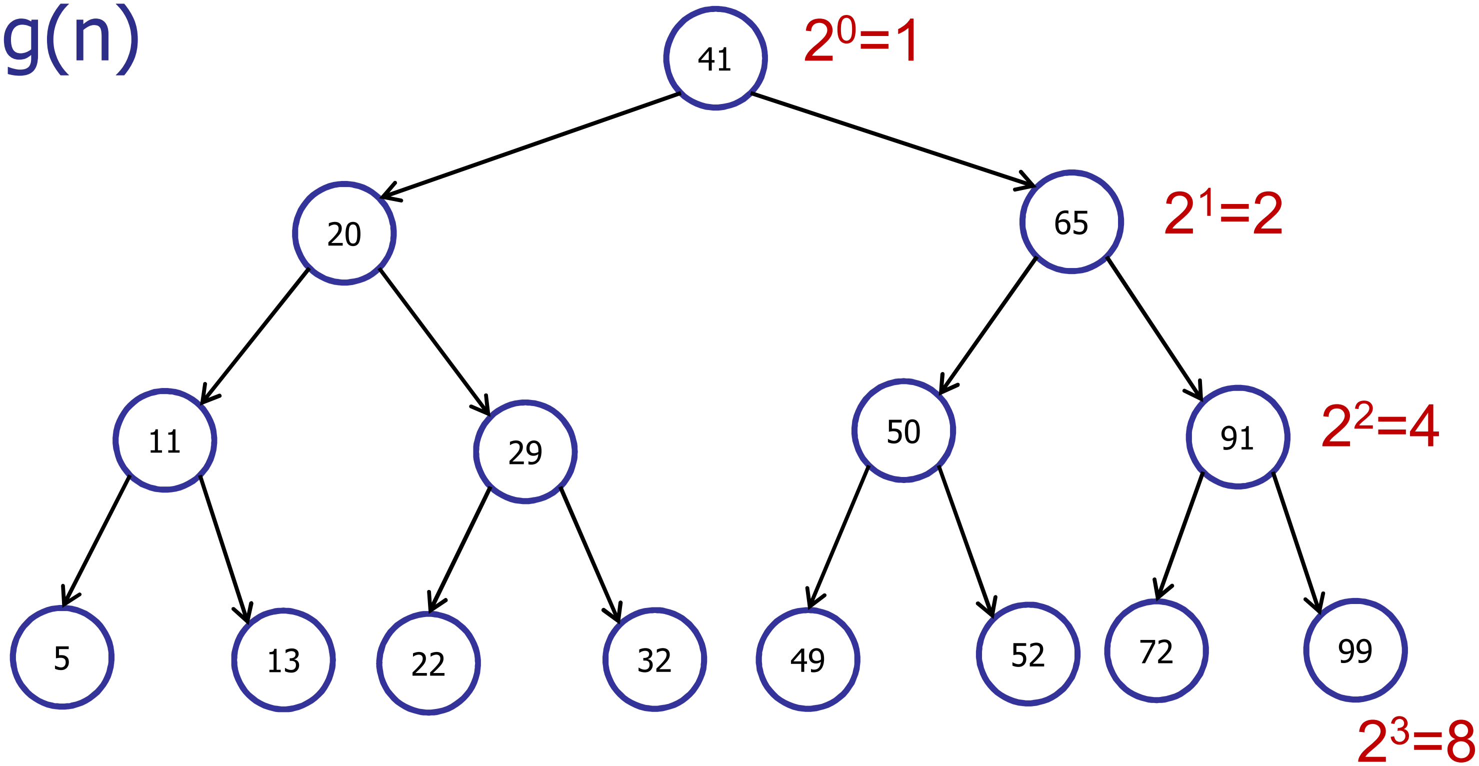
$h \geq \log(n) - 1$



The Importance of Being Balanced

Operations take O(h) time

$h+1 \geq \log(n)$



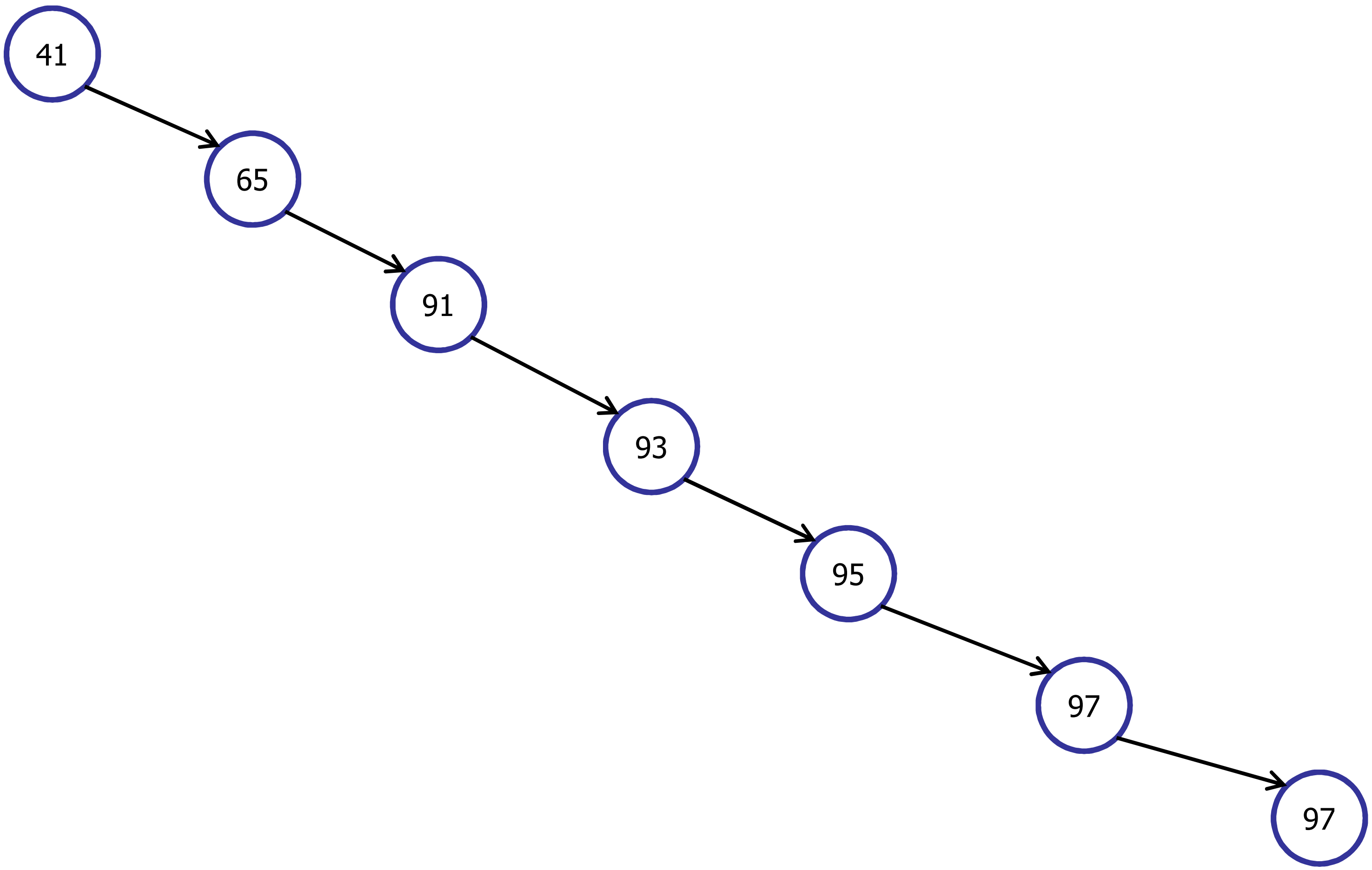
$$n \leq 1 + 2 + 4 + \ldots + 2^h$$

$$\leq 2^0 + 2^1 + 2^2 + \ldots + 2^h < 2^{h+1}$$

The Importance of Being Balanced

Operations take O(h) time

$h \leq n$



The Importance of Being Balanced

Operations take $O(h)$ time

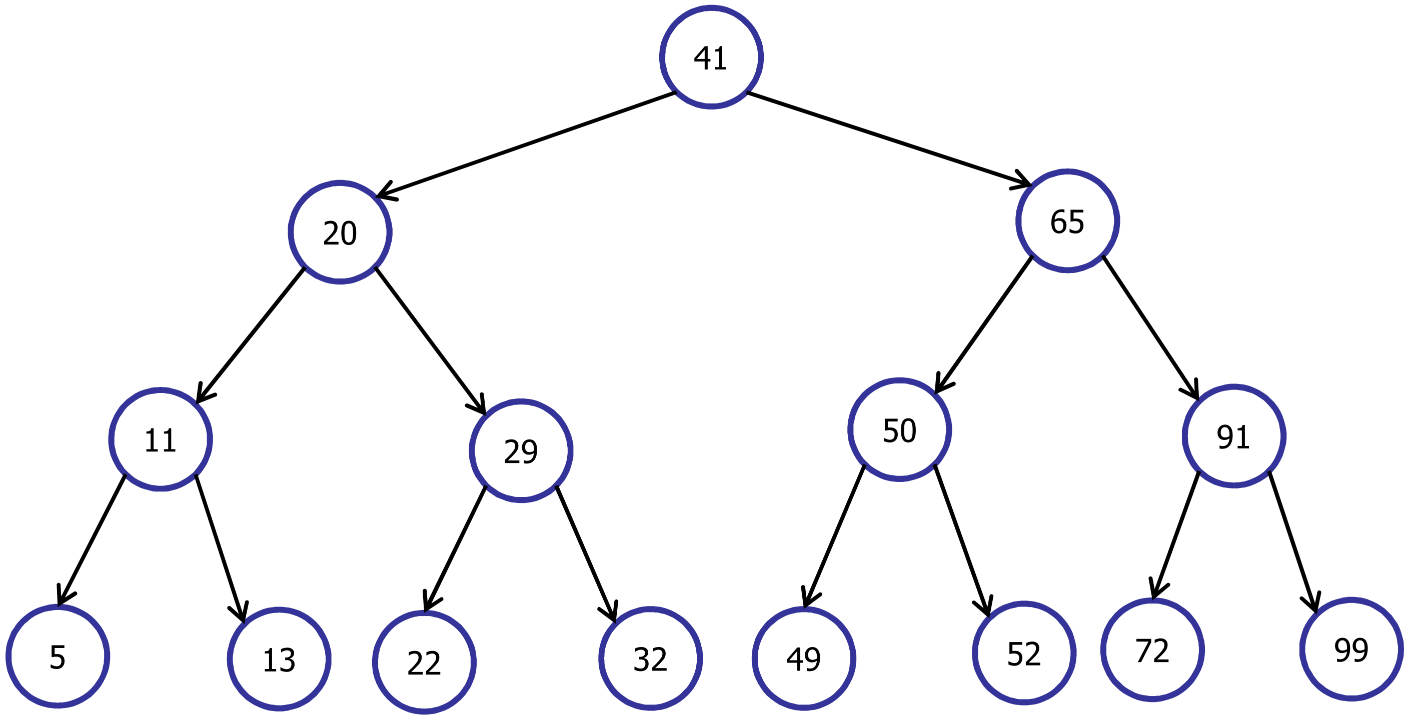
$\log(n) - 1 \le h \le n$

A BST is <u>balanced</u> if $h = O(\log n)$

On a balanced BST: all operations run in $O(\log n)$ time.

The Importance of Being Balanced

Perfectly balanced:



PS3: given an array of keys, construct a perfectly balanced tree.

The Importance of Being Balanced

How to get a balanced tree:

- Define a good property of a tree.
- Show that if the good property holds, then the tree is balanced.
- After every insert/delete, make sure the good property still holds. If not, fix it.

AVL Trees [Adelson-Velskii & Landis 1962]

AVL Trees [Adelson-Velskii & Landis 1962]

Step 1: Augment

- In every node v, store height:

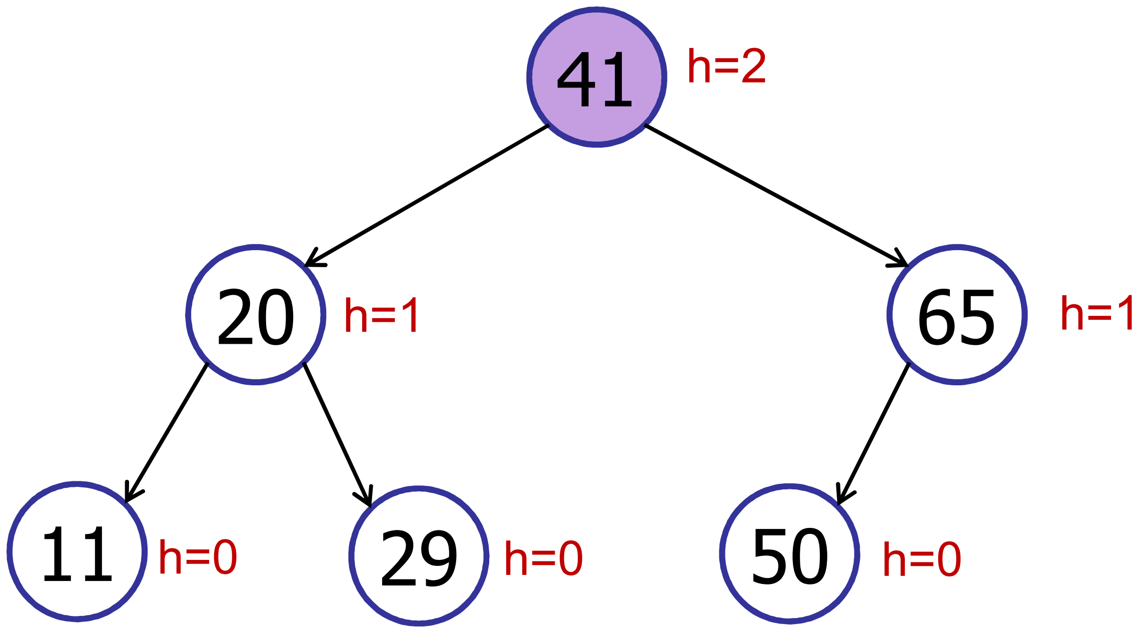
  v.height = h(v)

- On insert & delete update height:

```
insert(x)
        if (x < key)
                left.insert(x)
        else right.insert(x)
        height = max(left.height, right.height) + 1
```

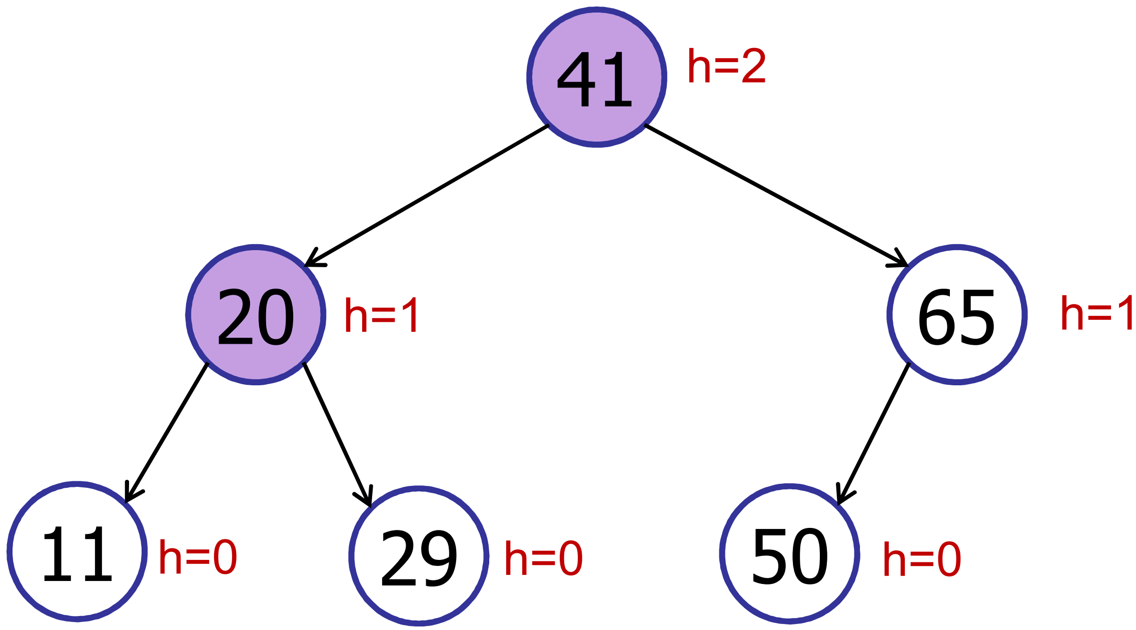
Binary Search Trees

insert(27)



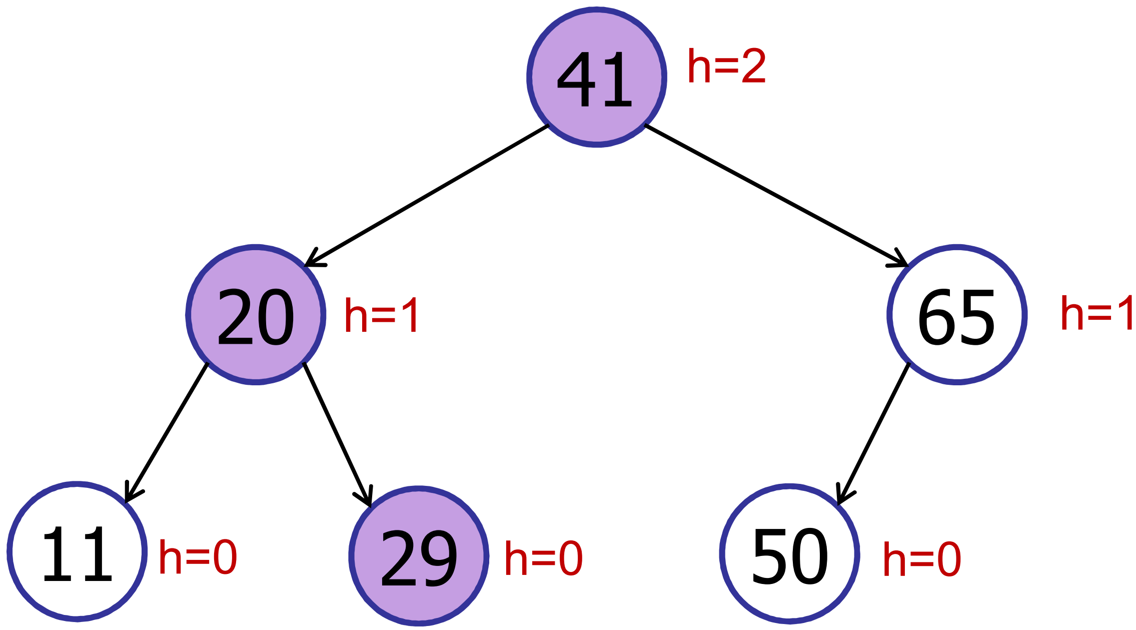
Binary Search Trees

insert(27)



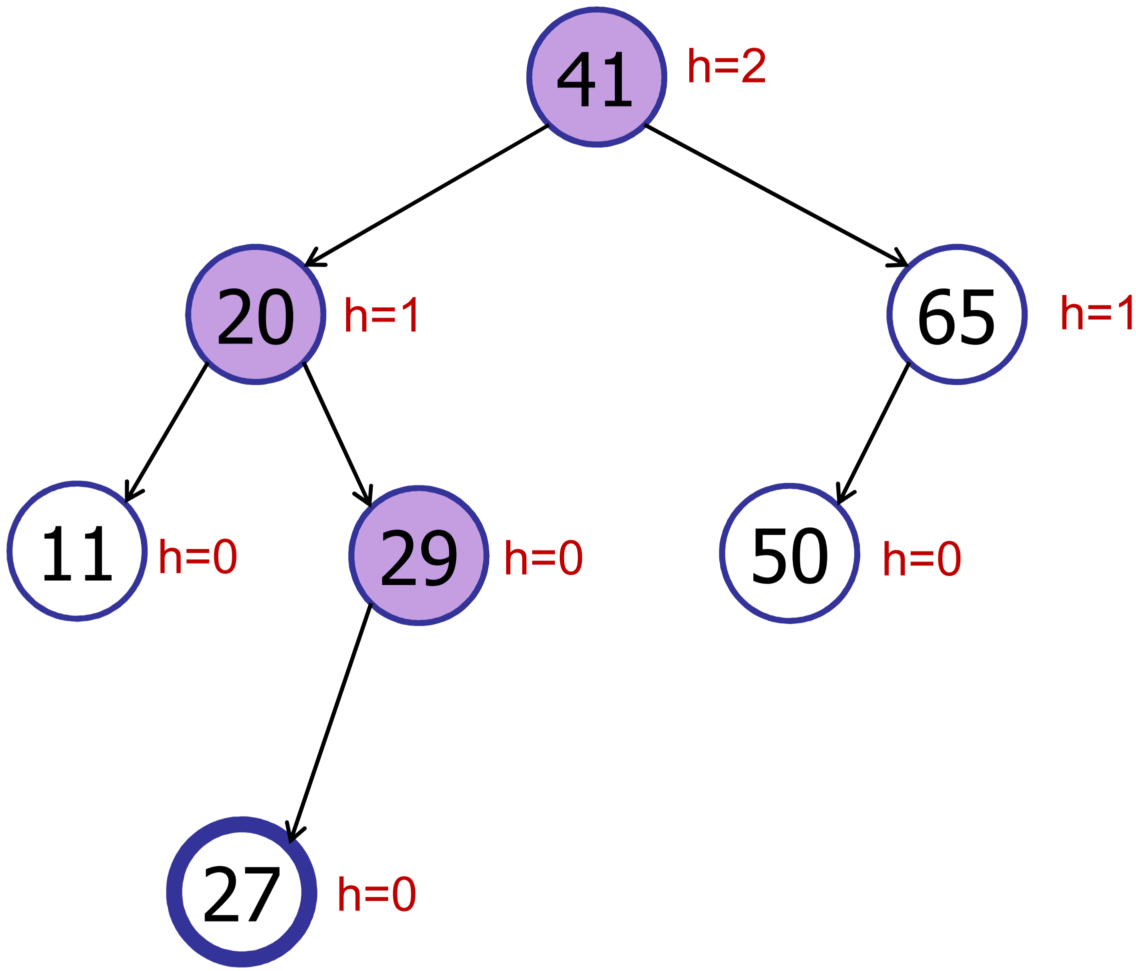
Binary Search Trees

insert(27)



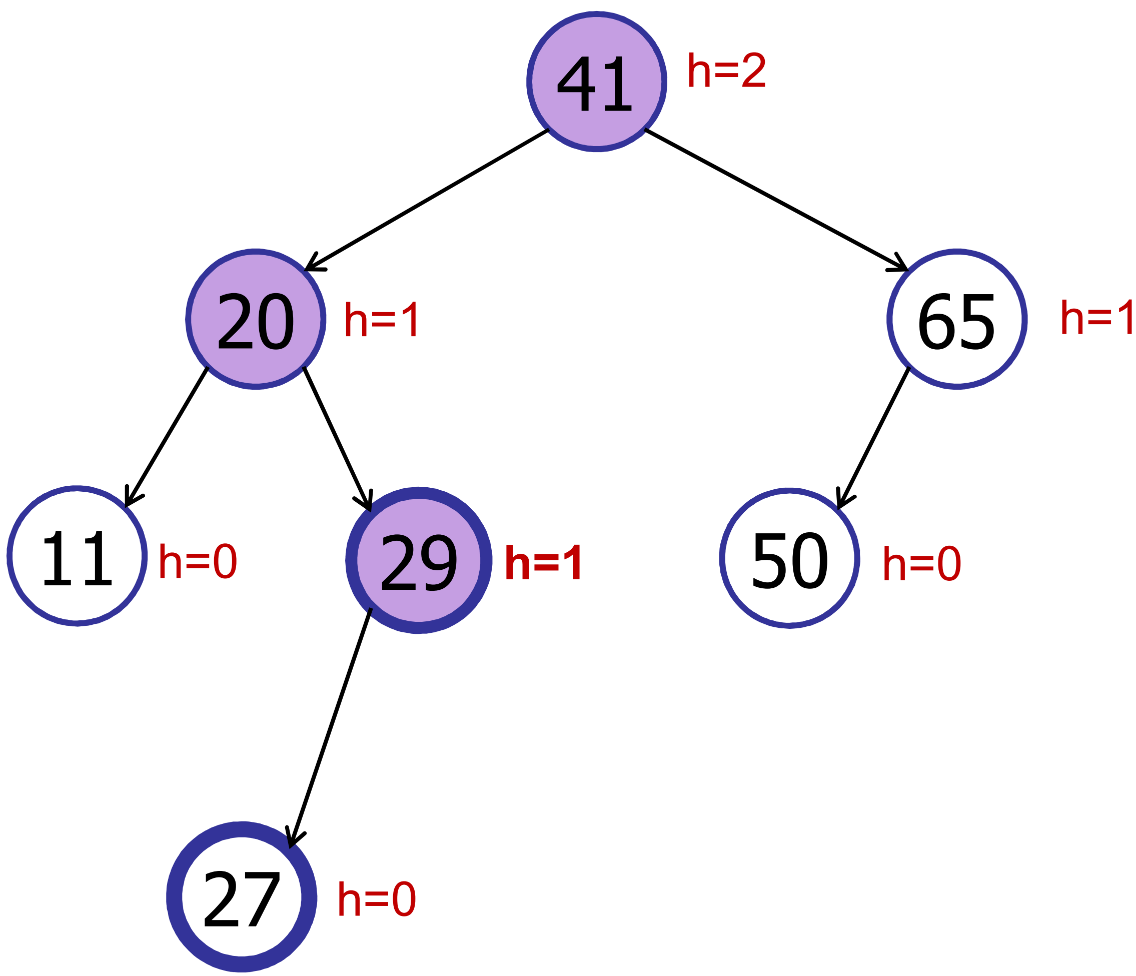
Binary Search Trees

insert(27)



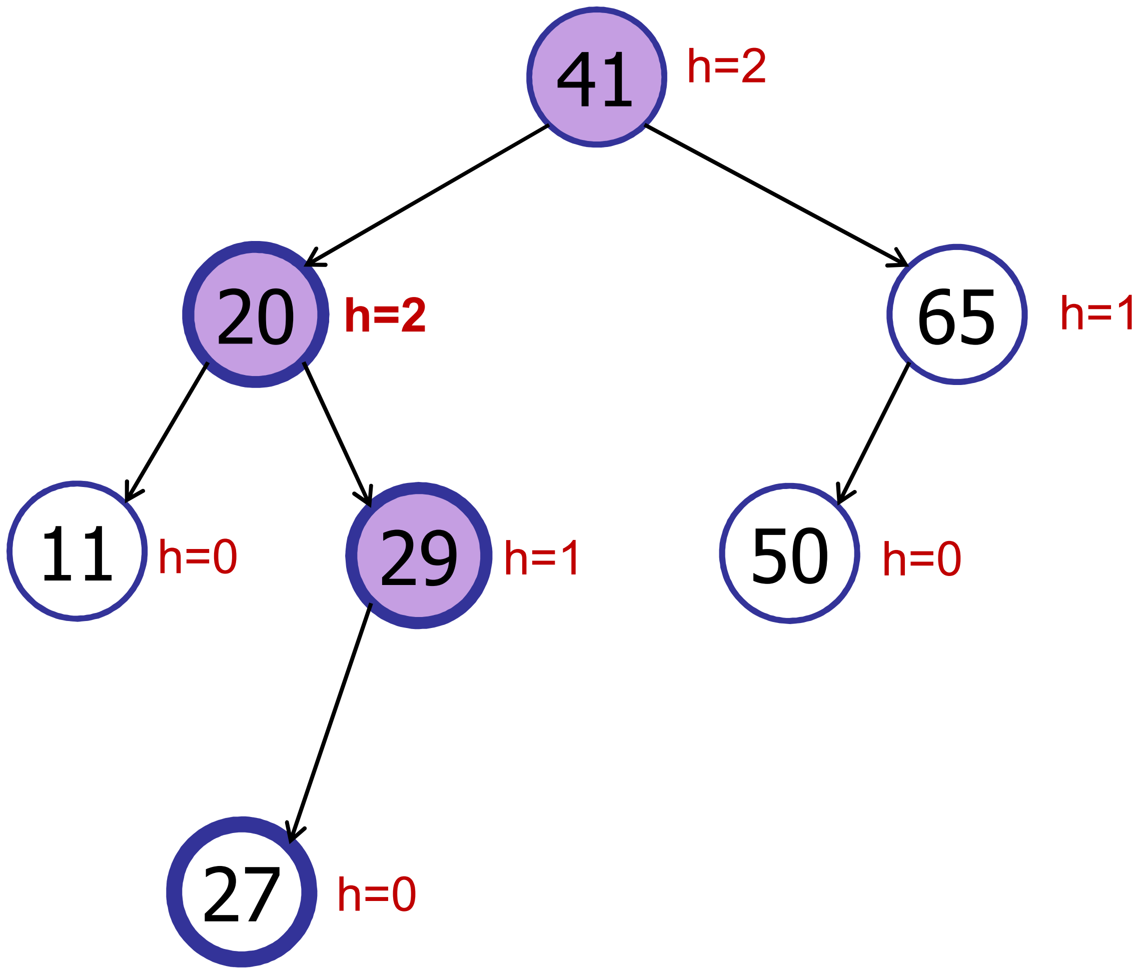
Binary Search Trees

insert(27)



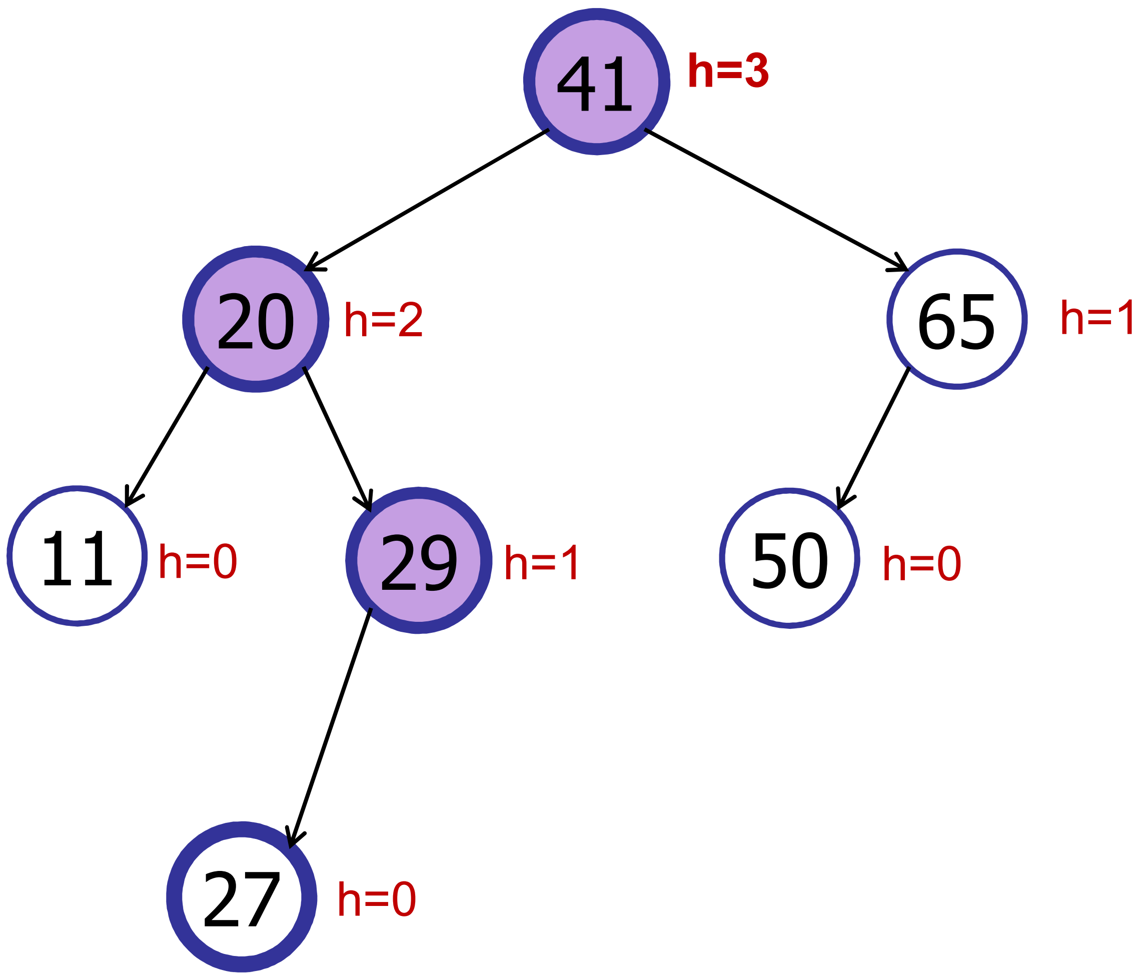
Binary Search Trees

insert(27)



Binary Search Trees

insert(27)



AVL Trees [Adelson-Velskii & Landis 1962]

Step 1: Augment

- In every node v, store height:

  v.height = h(v)

- On insert & delete update height:

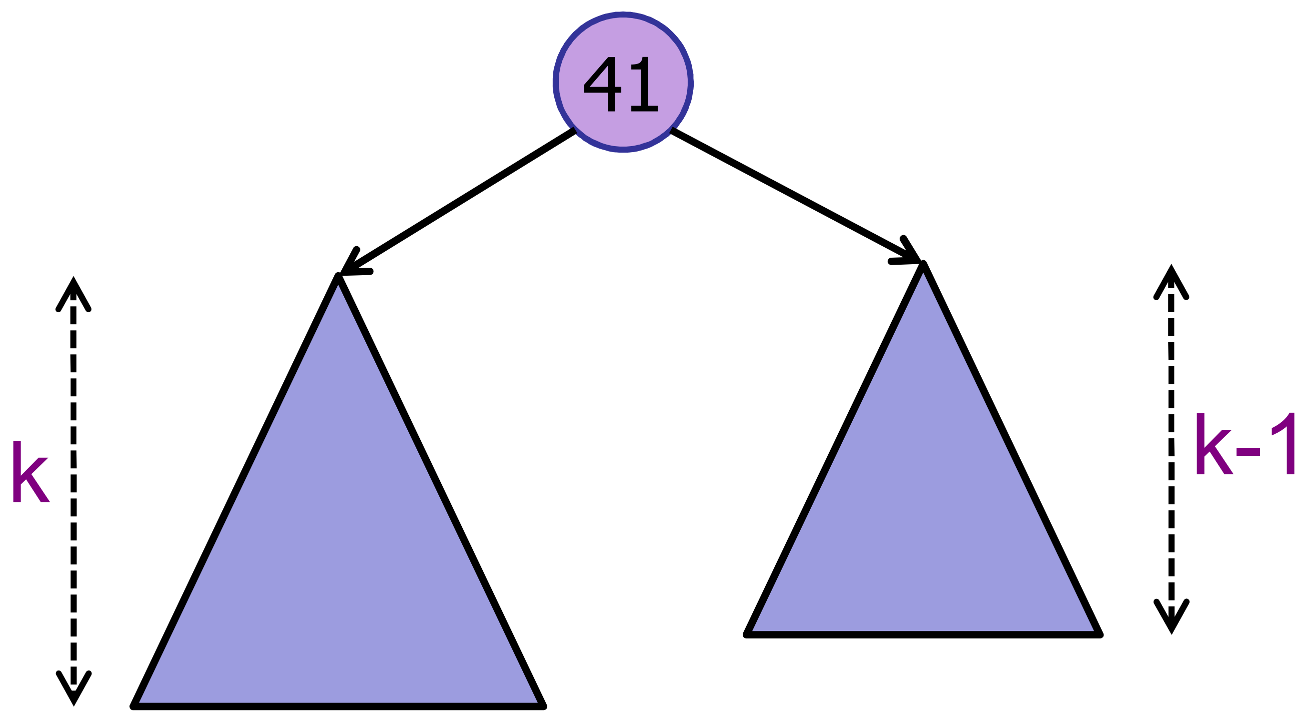
```
insert(x)
        if (x < key)
                left.insert(x)
        else right.insert(x)
        height = max(left.height, right.height) + 1
```

AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Define Invariant

- A node v is <u>height-balanced</u> if:

$$|v.left.height - v.right.height| \leq 1$$



AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Define Invariant

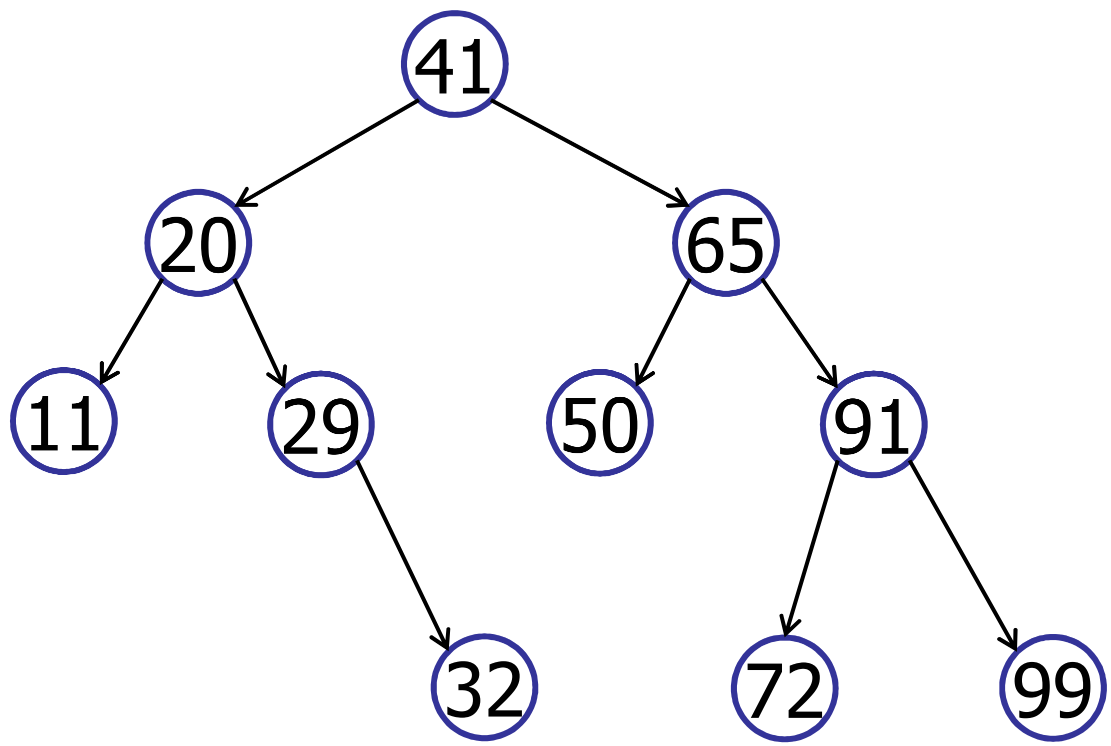
– A node v is <u>height-balanced</u> if:

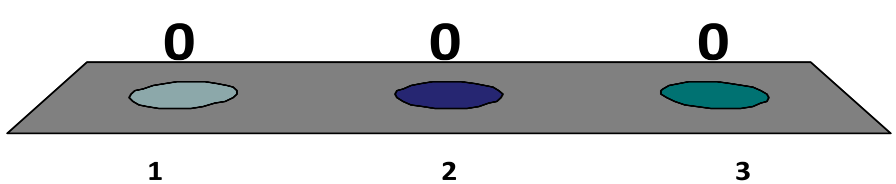
$$|v.left.height - v.right.height| \leq 1$$

– An binary search tree is <u>height balanced</u> if every node in the tree is height-balanced.

Is this tree height-balanced?

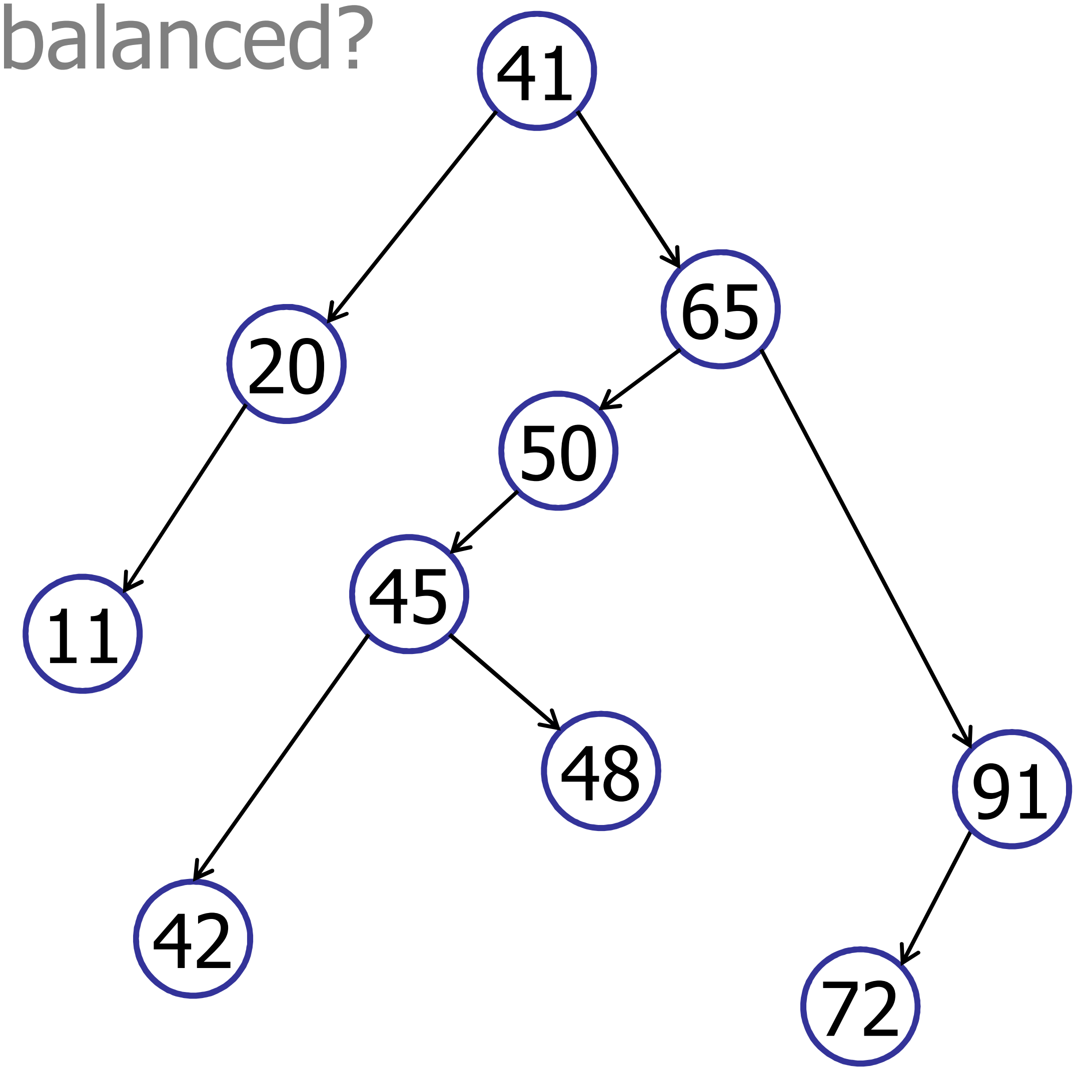
1. Yes
2. No
3. I'm confused.

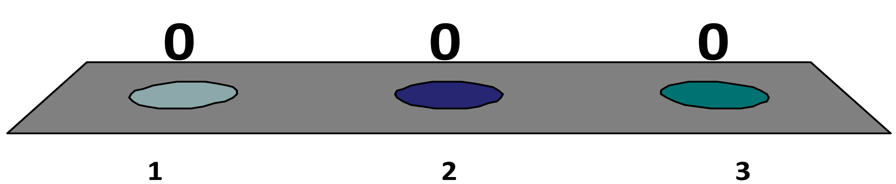




Is this tree height-balanced?

1. Yes
2. No
3. I'm confused.





Height-Balanced Trees

Claim:

A height-balanced tree with n nodes has height $h < 2\log(n)$.

Height-Balanced Trees

Proof:

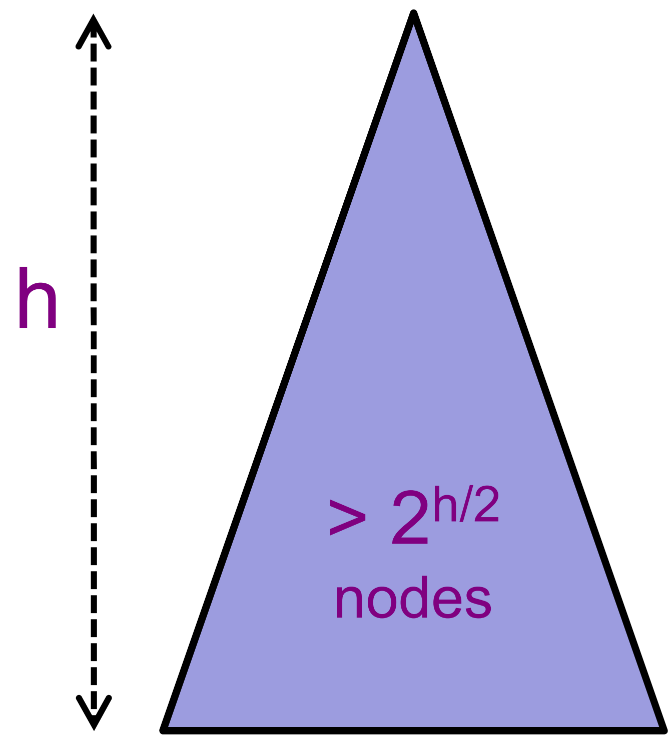
Let n_h be the minimum number of nodes in a height-balanced tree of height h.

Show:

$n_h > 2^{h/2}$

$\Rightarrow$

$2\log(n_h) > h$

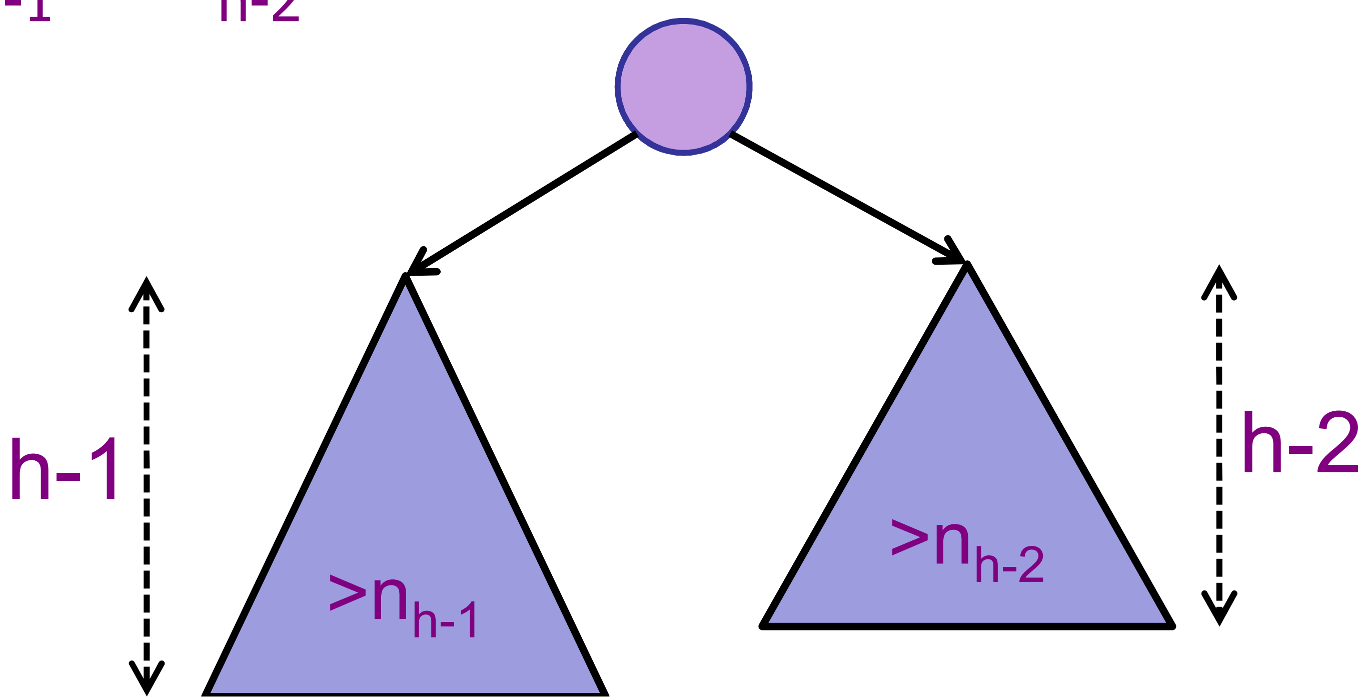


Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$n_h \geq 1 + n_{h-1} + n_{h-2}$



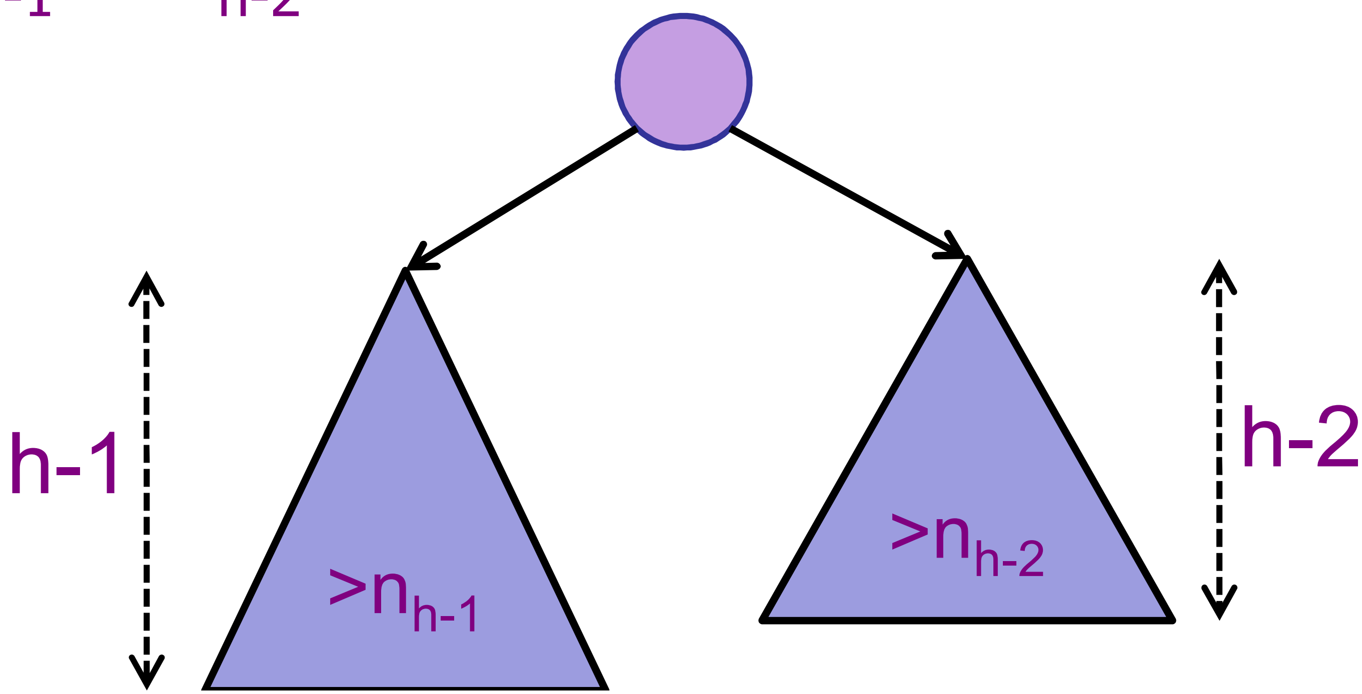
Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$n_h \geq 1 + n_{h-1} + n_{h-2}$$

$$\geq 2n_{h-2}$$



Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$
\begin{aligned}
n_h &\geq 1 + n_{h-1} + n_{h-2} \\
&\geq 2n_{h-2} \\
&\geq 4n_{h-4} \\
&\geq 8n_{h-6} \\
&\geq \ldots
\end{aligned}
$$

Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$
\begin{aligned}
n_h &\geq 1 + n_{h-1} + n_{h-2} \\
&\geq 2n_{h-2} \\
&\geq 4n_{h-4} \\
&\geq 8n_{h-6} \\
&\geq \ldots
\end{aligned}
$$

Base case:
$n_0 = 1$

Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$
\begin{aligned}
n_h &\geq 1 + n_{h-1} + n_{h-2} \\
&\geq 2n_{h-2} \\
&\geq 2^{h/2}\, n_0 \\
&\geq 2^{h/2}
\end{aligned}
$$

Base case:
$n_0 = 1$

Assume n is even.

Height-Balanced Trees

Claim:

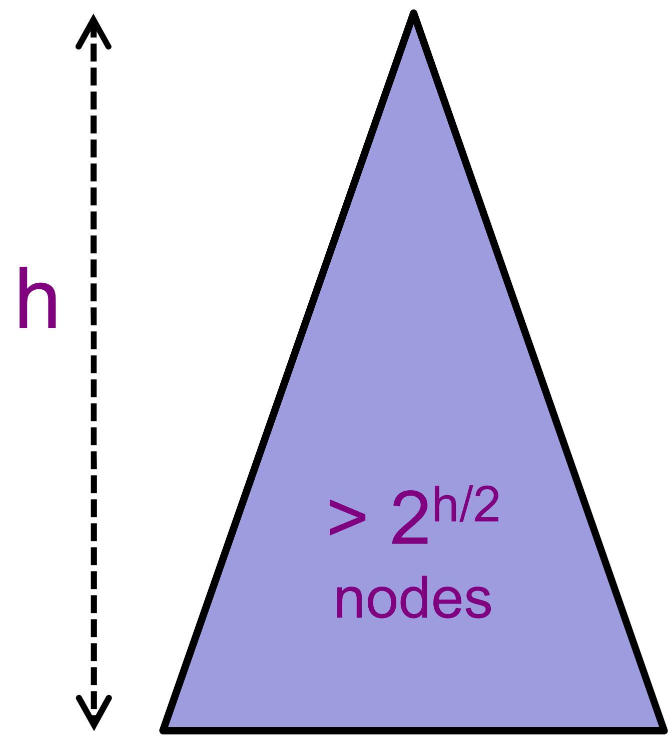
A height-balanced tree with n nodes has height $h < 2\log(n)$.

Show:

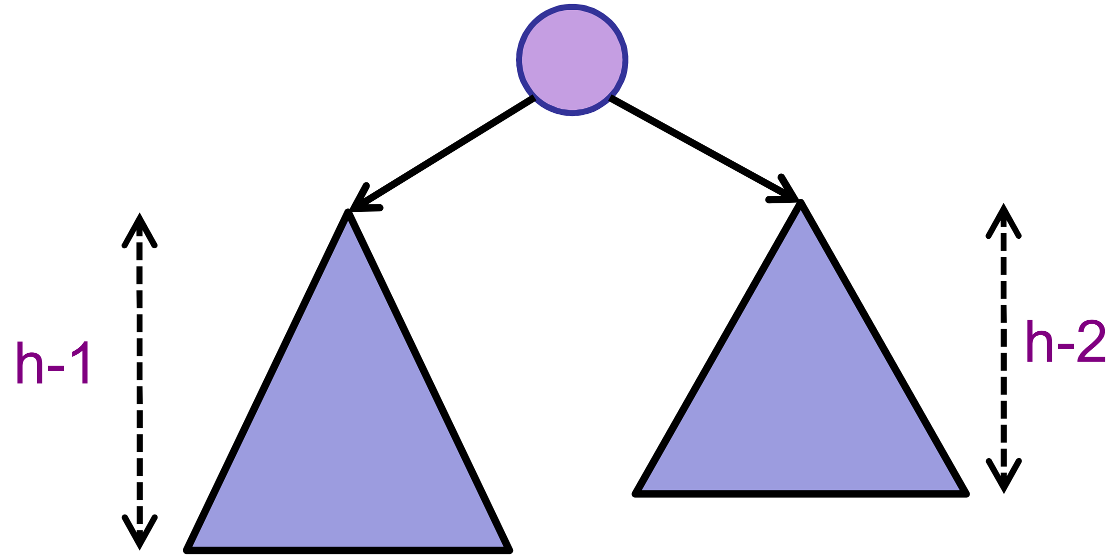
$n_h > 2^{h/2}$

$\Rightarrow$

$2\log(n_h) > h$



Height-Balanced Trees



Show (induction):

F_n = n^{th} Fibonacci number

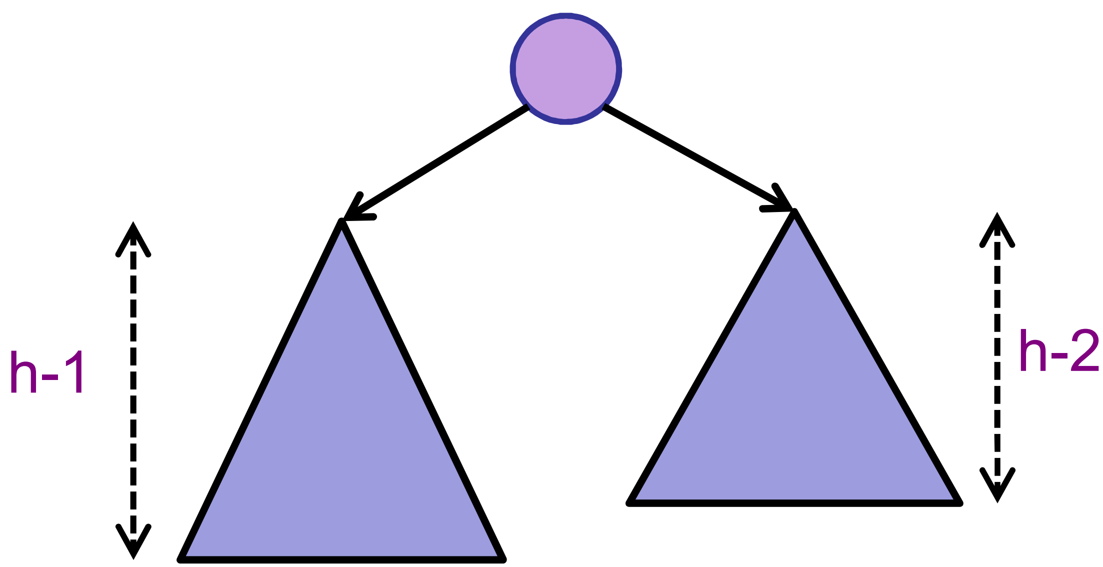
$n_h = F_{h+2} - 1 \cong \phi^{h+1}/\sqrt{5} - 1$ (rounded to nearest int)

$h \cong \log(n) / \log(\phi) \qquad \phi \cong 1.618$

Height-Balanced Trees

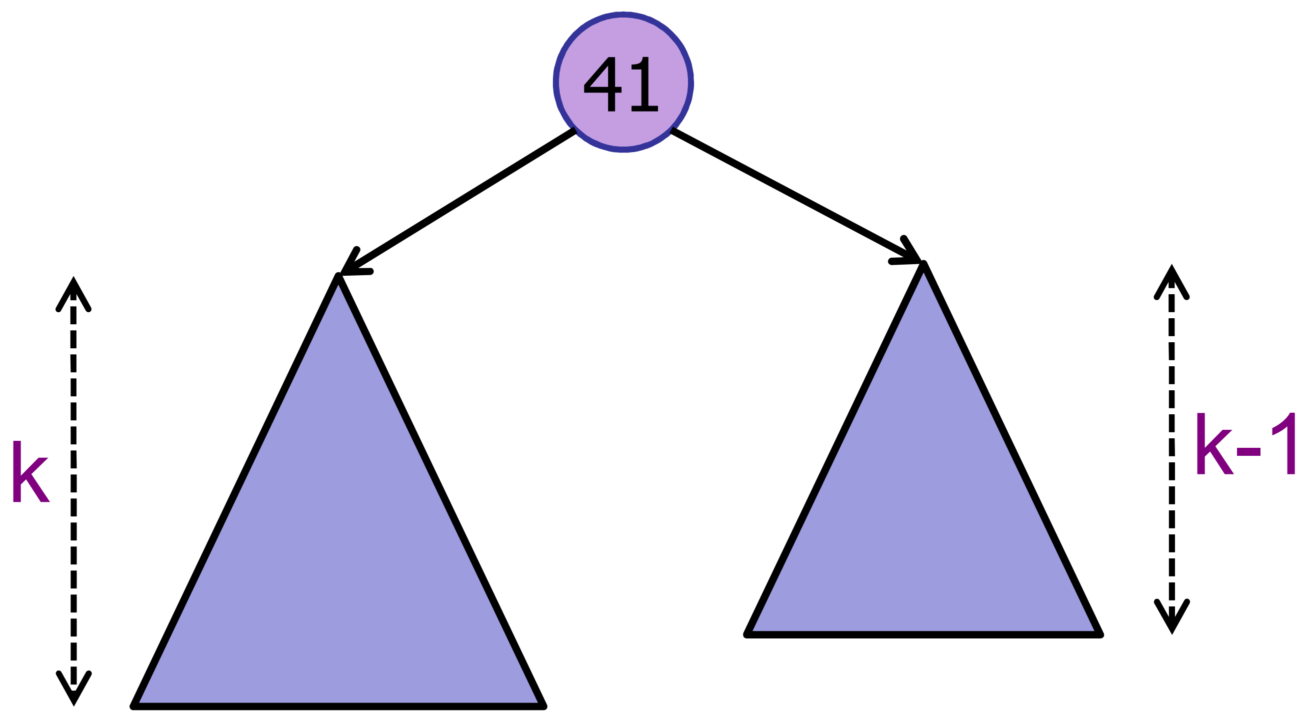
Claim:

A height-balanced tree is balanced, i.e., has height $h = O(\log(n))$.



AVL Trees [Adelson-Velskii & Landis 1962]

Step 3: Show how to maintain height-balance

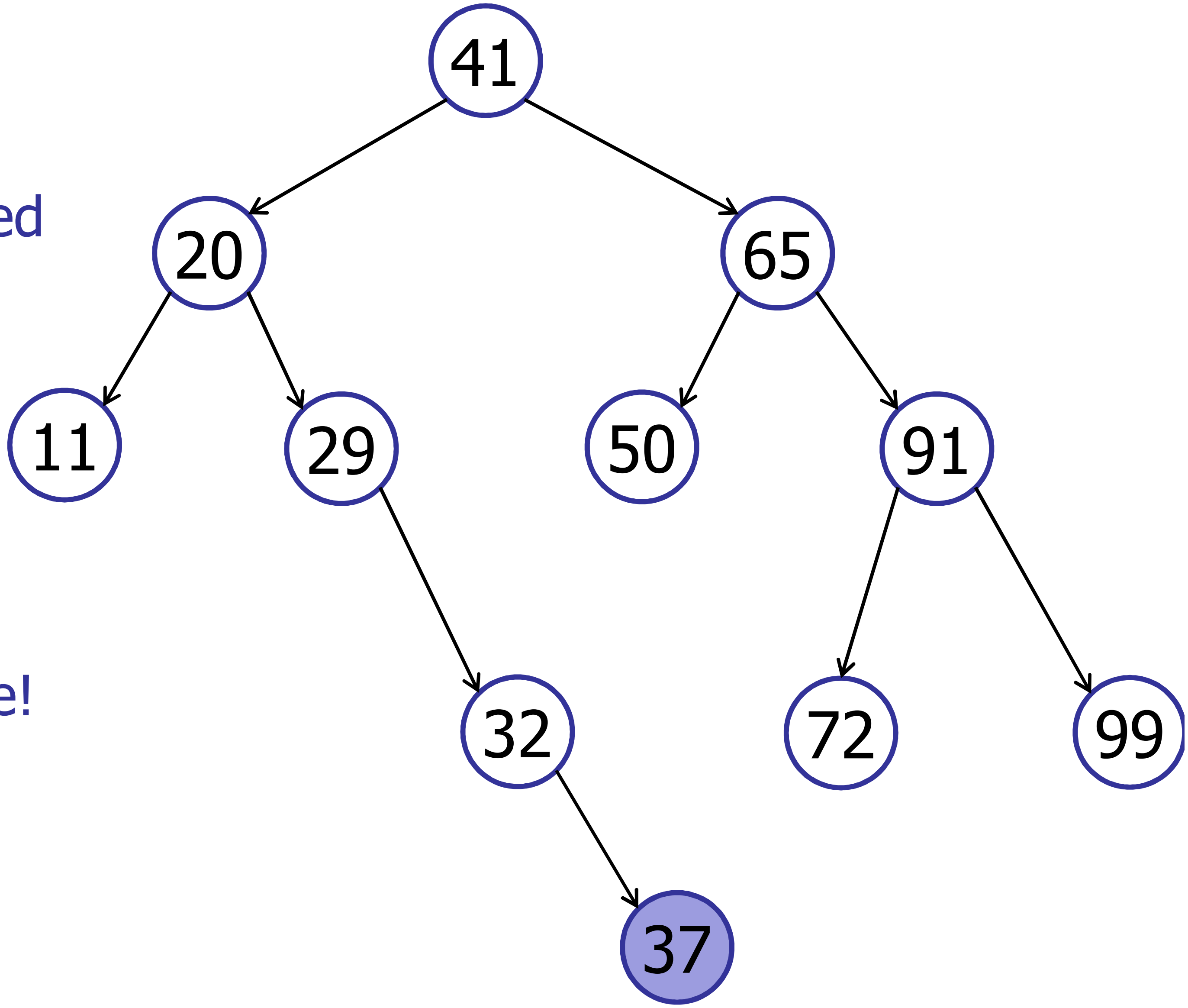


Inserting in an AVL Tree

insert(37)

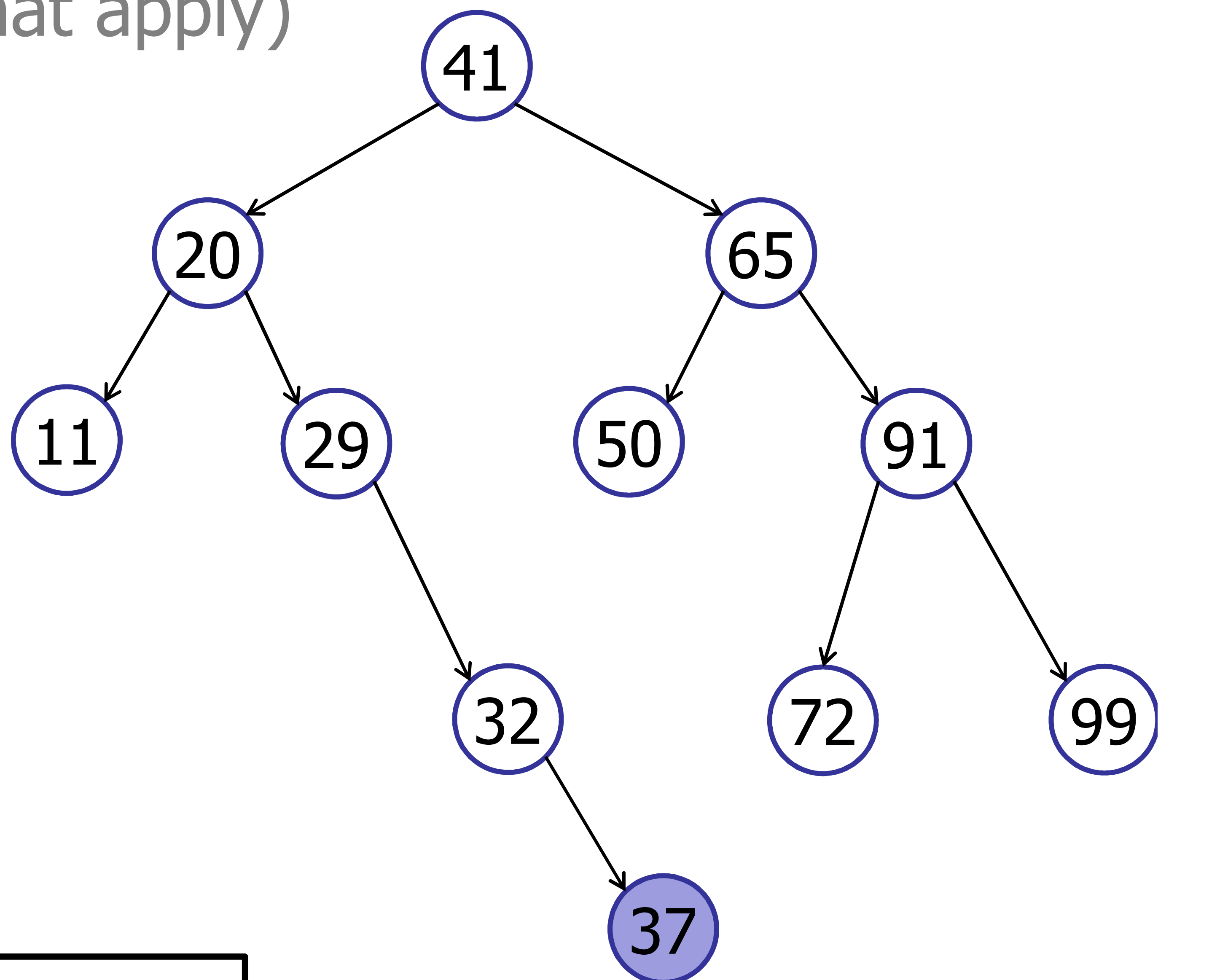
No longer balanced after insertion!

Need to rebalance!



Which nodes need rebalancing?
(click all that apply)

0 1. 41
0 2. 20
0 3. 11
0 4. 29
0 5. 32
0 6. 37
0 7. 65



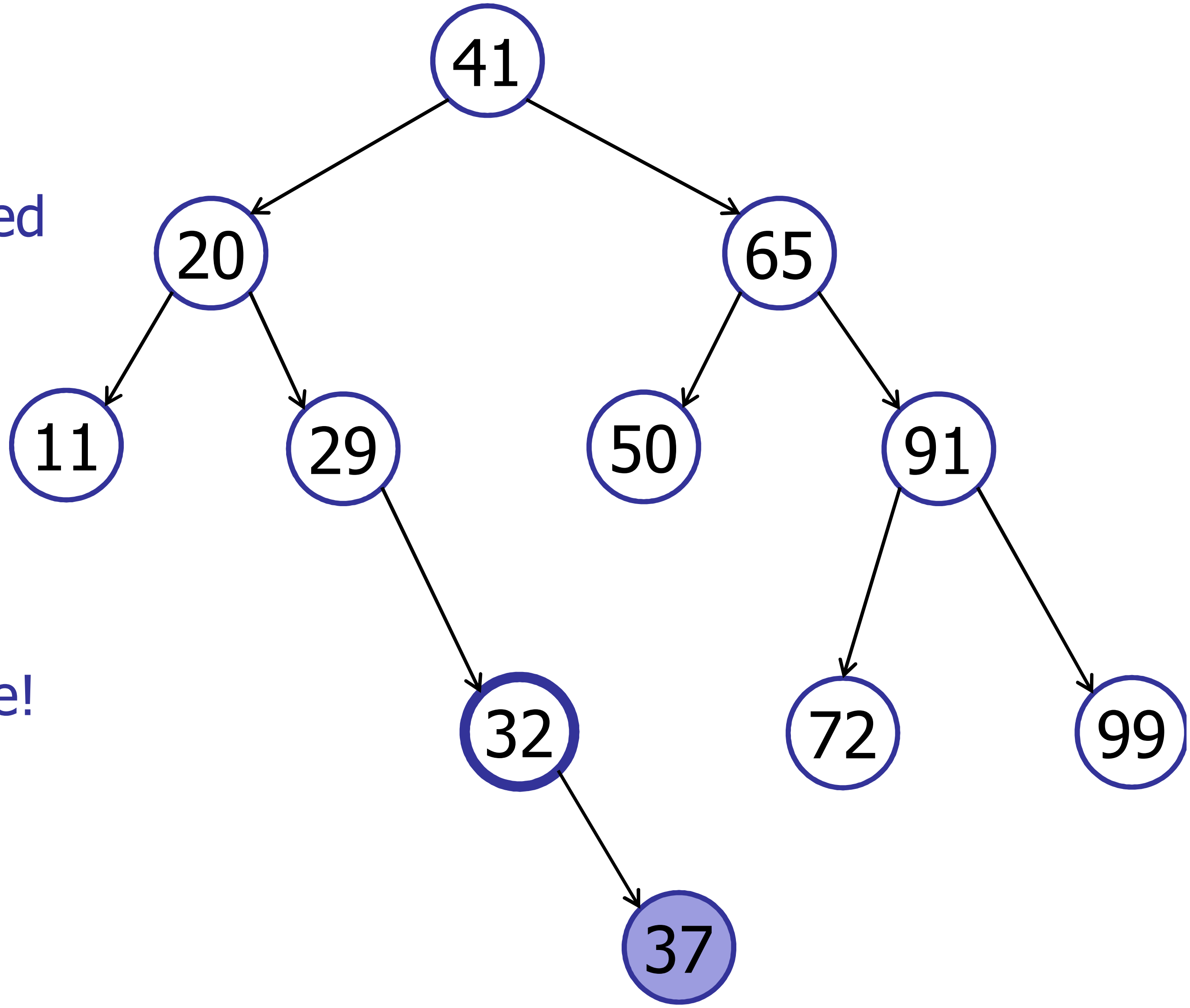
0 of 60

Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

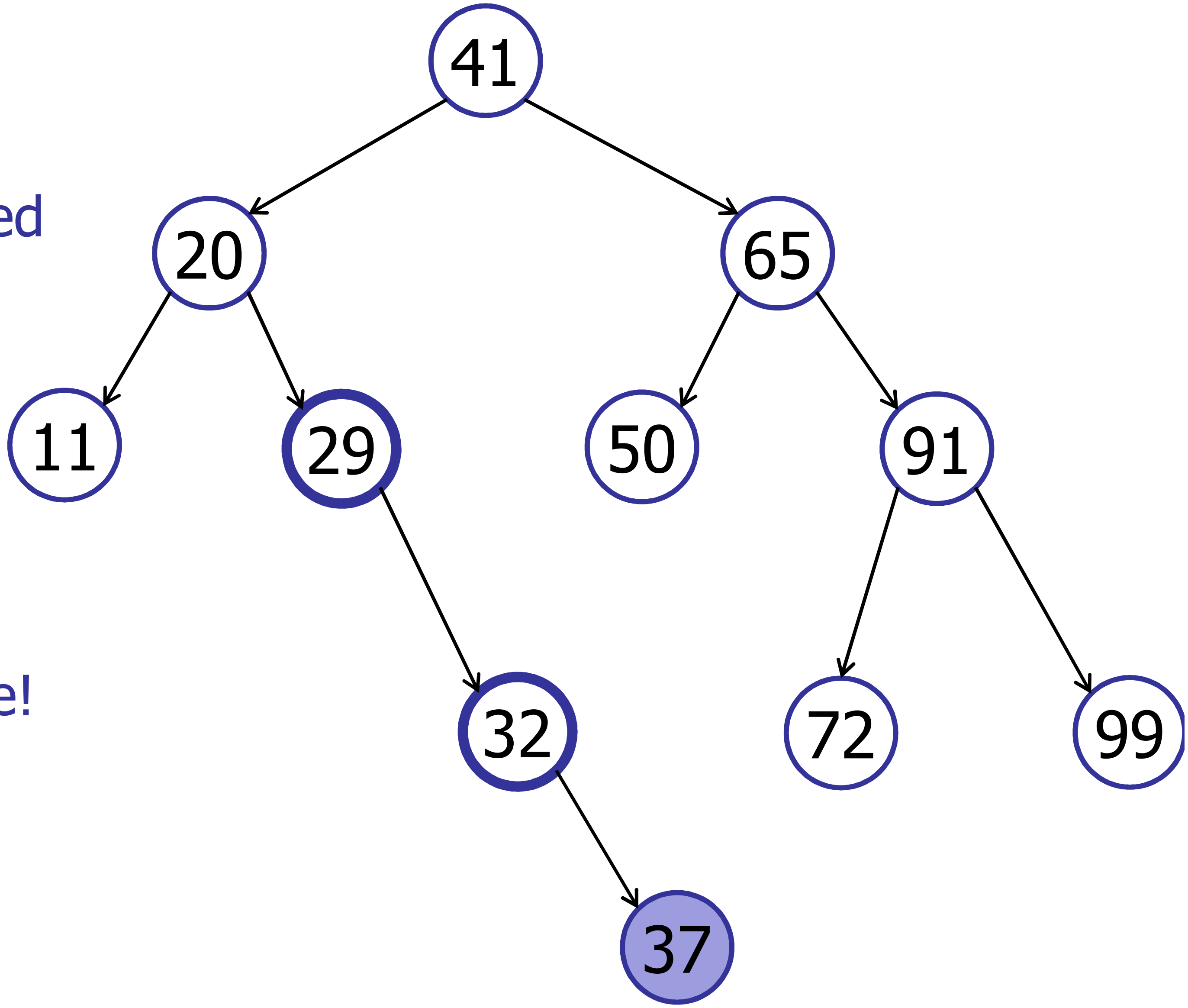


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

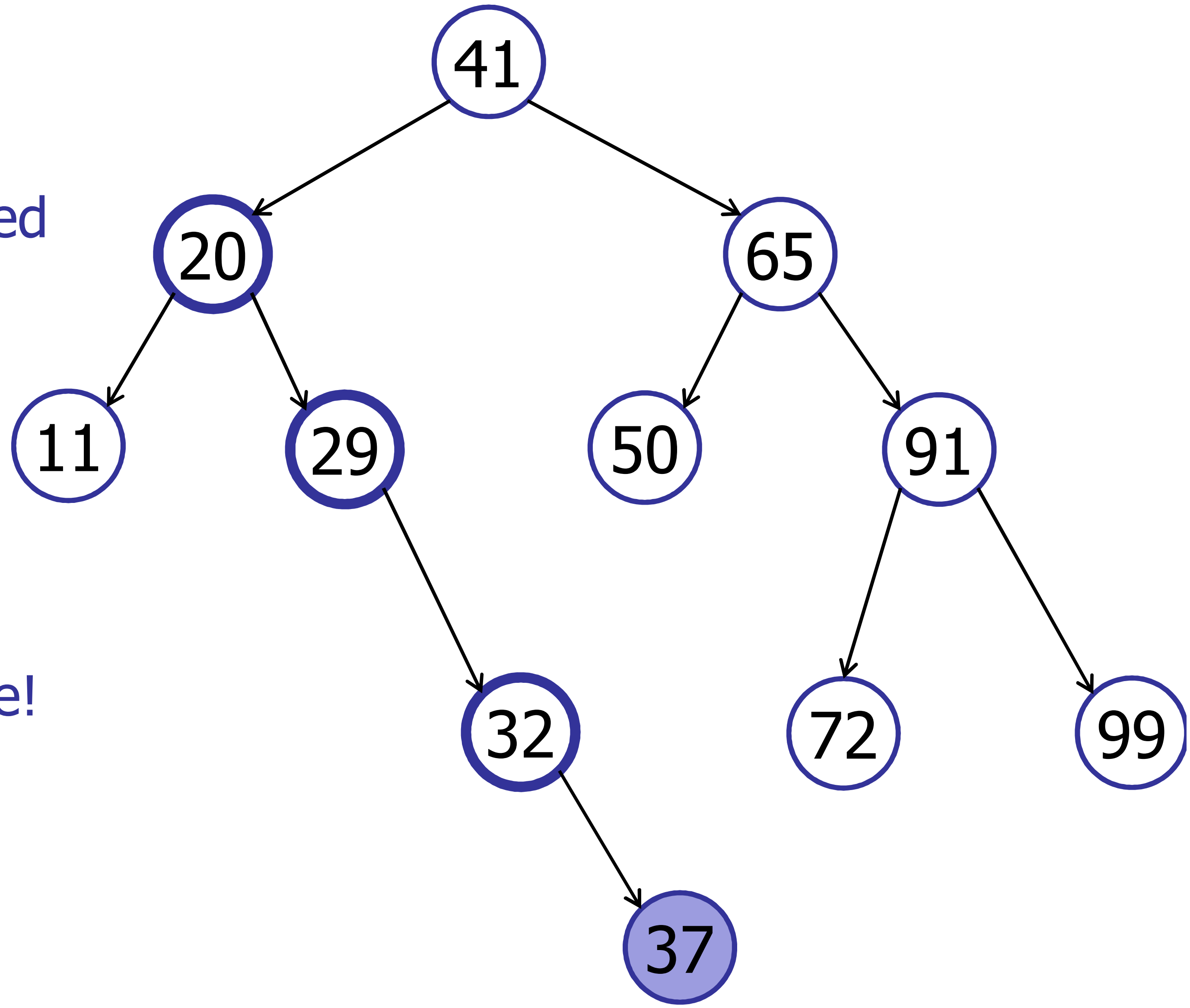


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

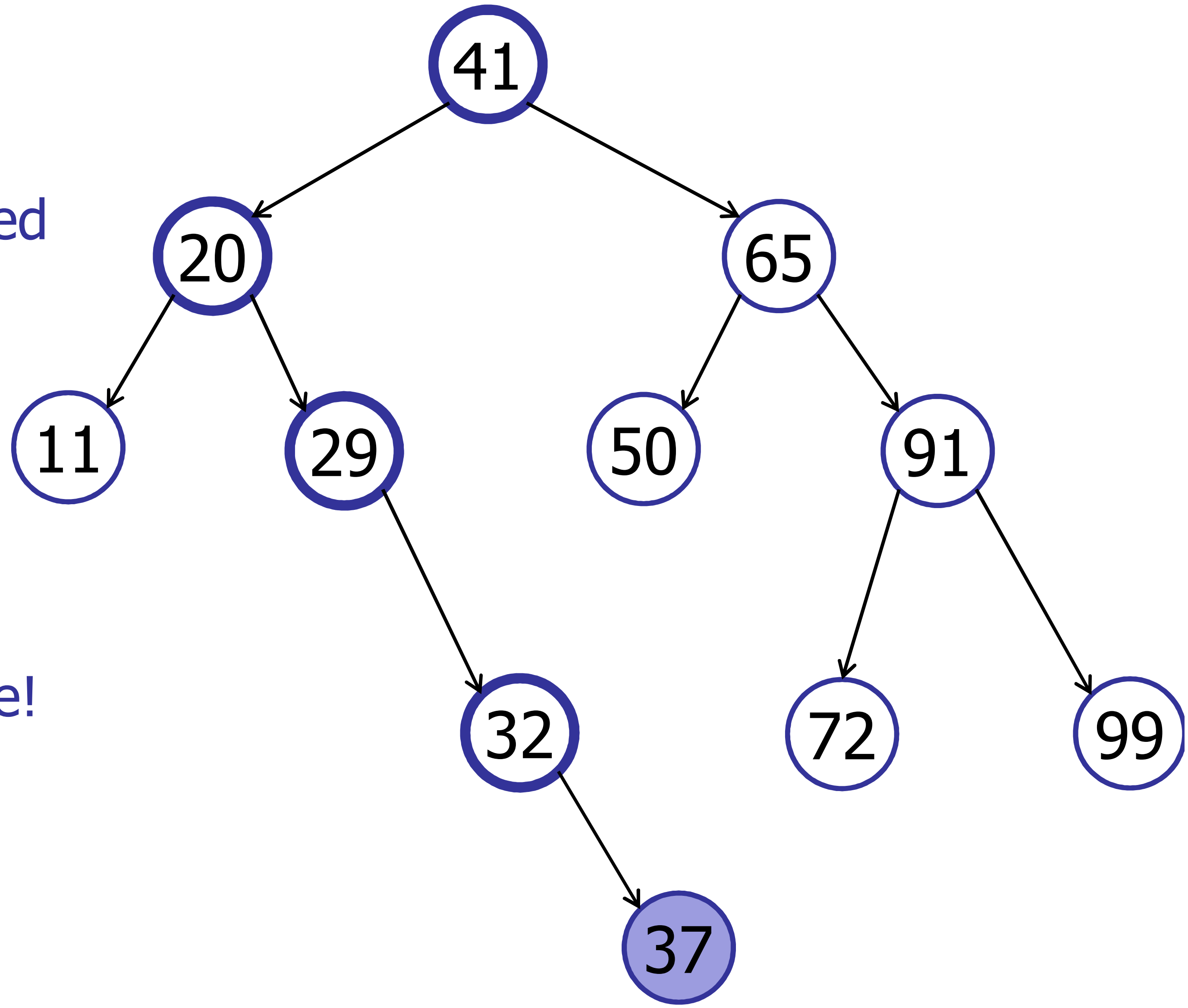


Inserting in an AVL Tree

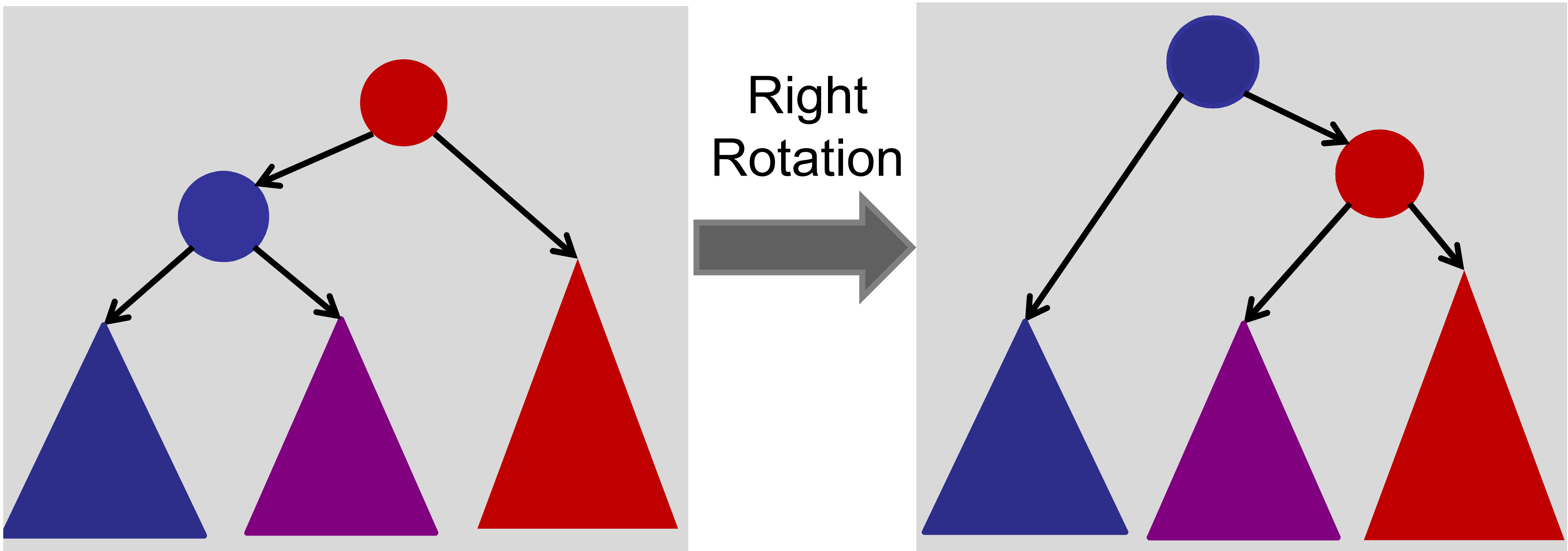
insert(37)

No longer balanced after insertion!

Need to rebalance!



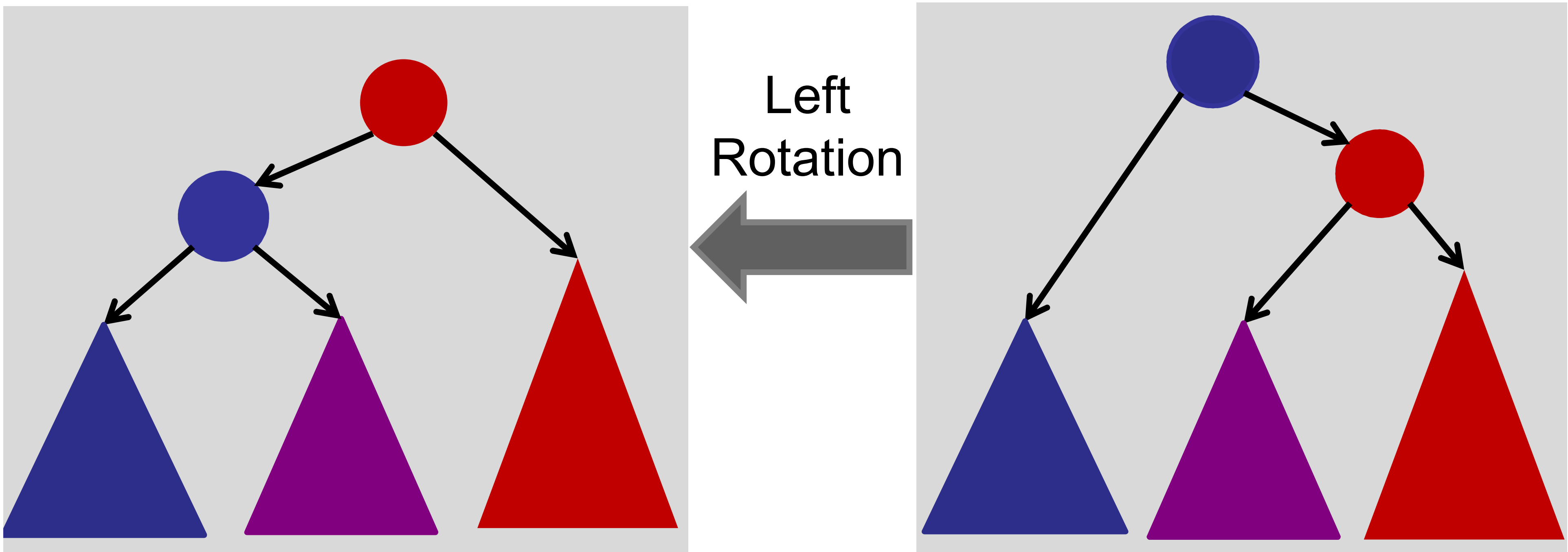
Tree Rotations



Rotations maintain ordering of keys.

$\Rightarrow$ Maintains BST property.

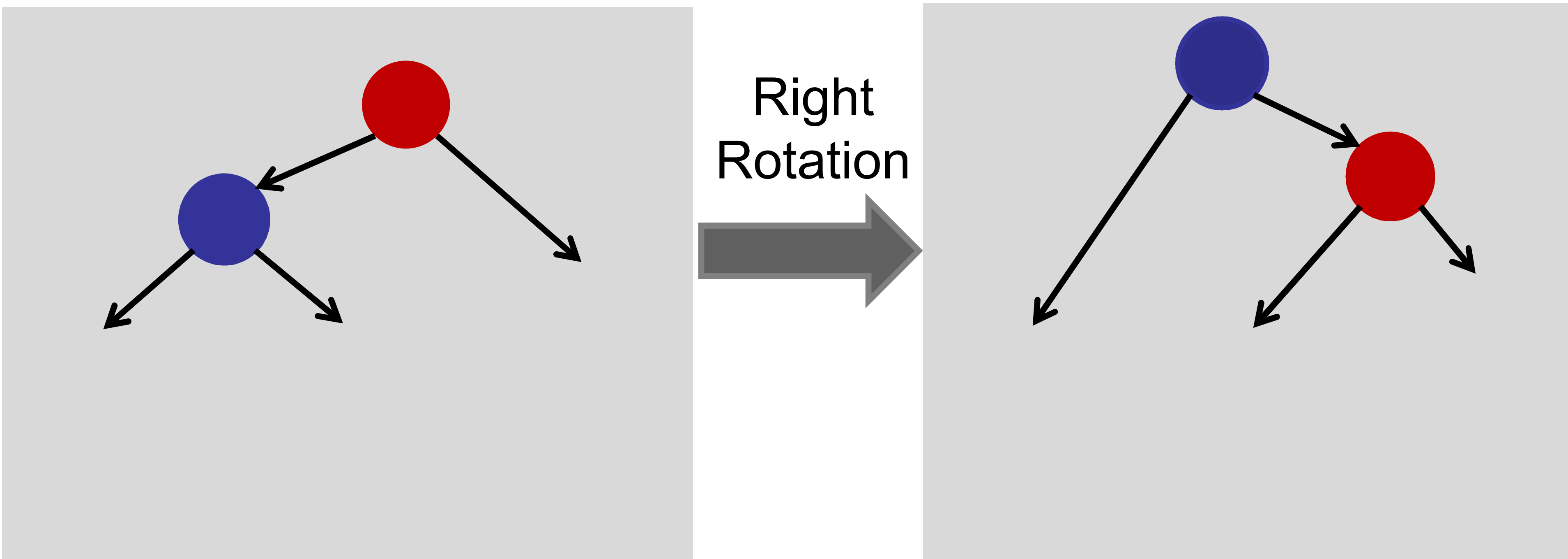
Tree Rotations



Rotations

```
right-rotate(v)          // assume v has left!=null
    w = v.left
    w.parent = v.parent
    v.parent = w
    v.left = w.right
    w.right = v
```

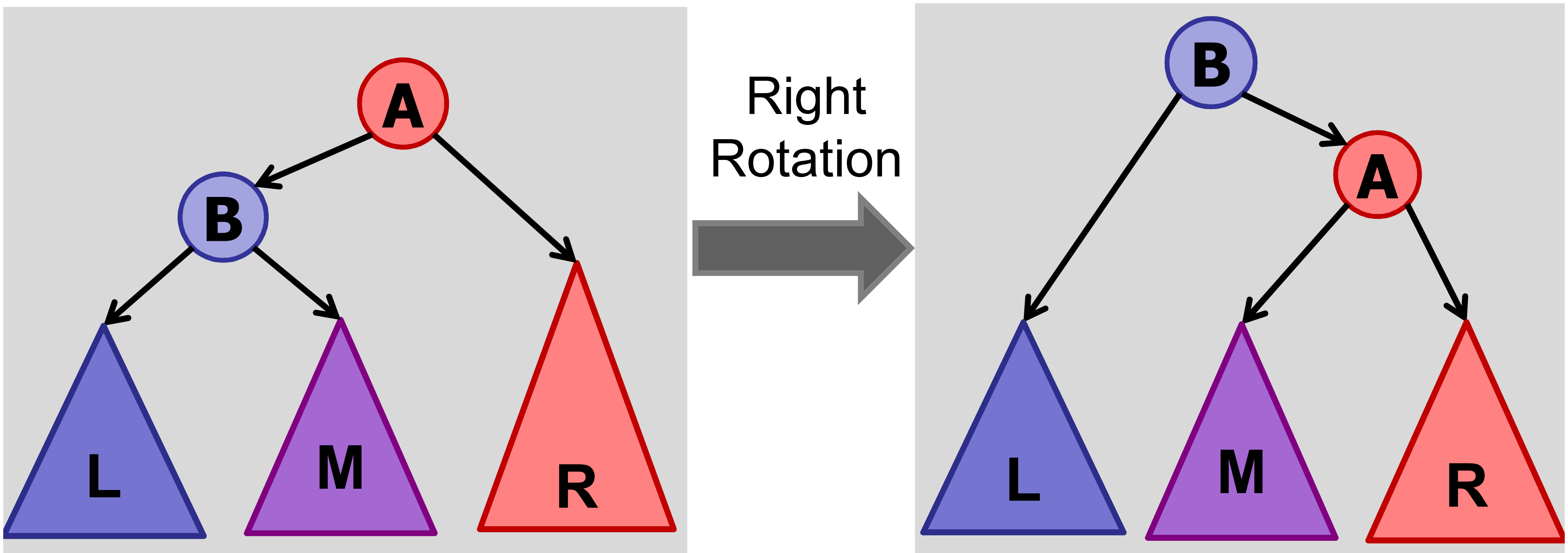
Tree Rotations



rotate-right requires a left child

rotate-left requires a right child

Tree Rotations

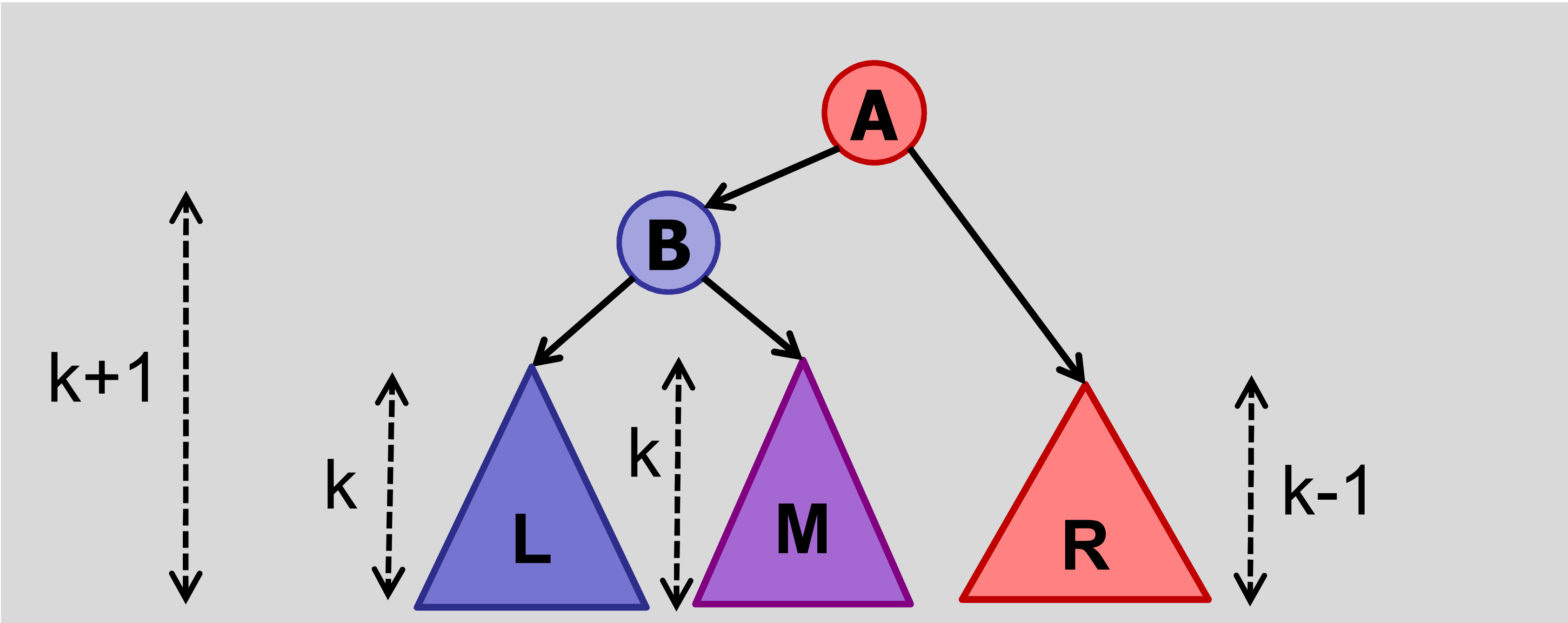


Use tree rotations to restore balance.

After insert, start at bottom, work your way up.

Assume tree is LEFT-heavy.

Tree Rotations

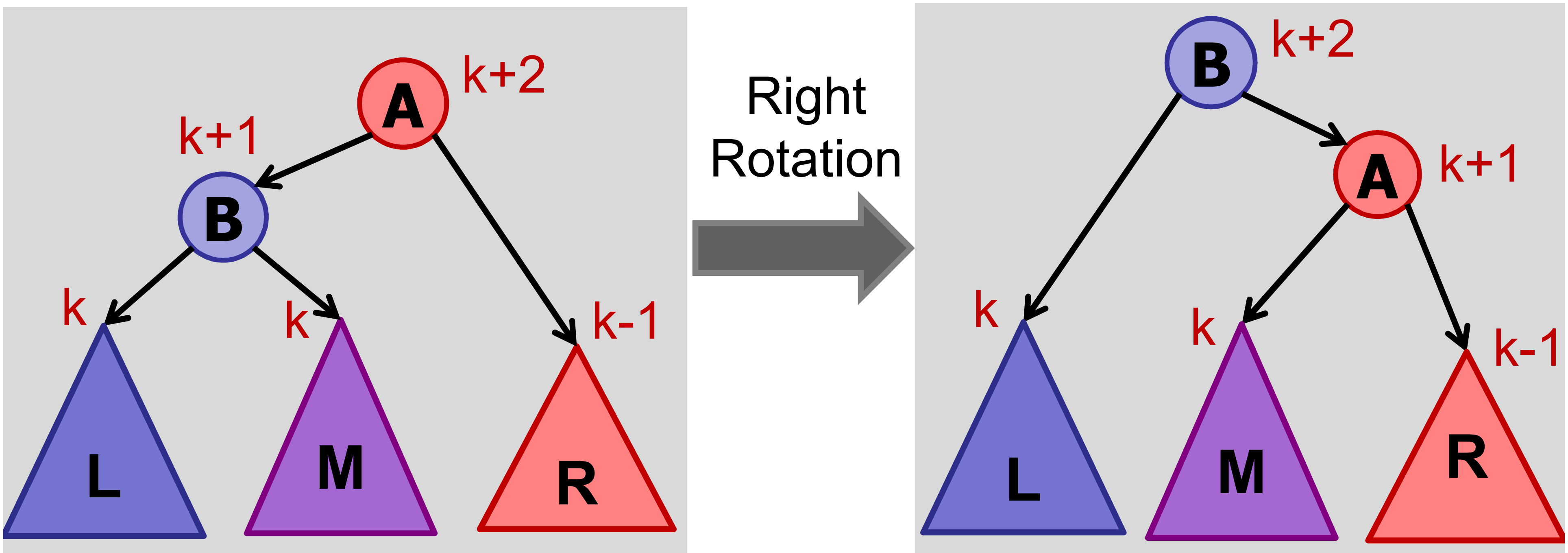


Assume **A** is the lowest node in the tree violating balance property.

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations

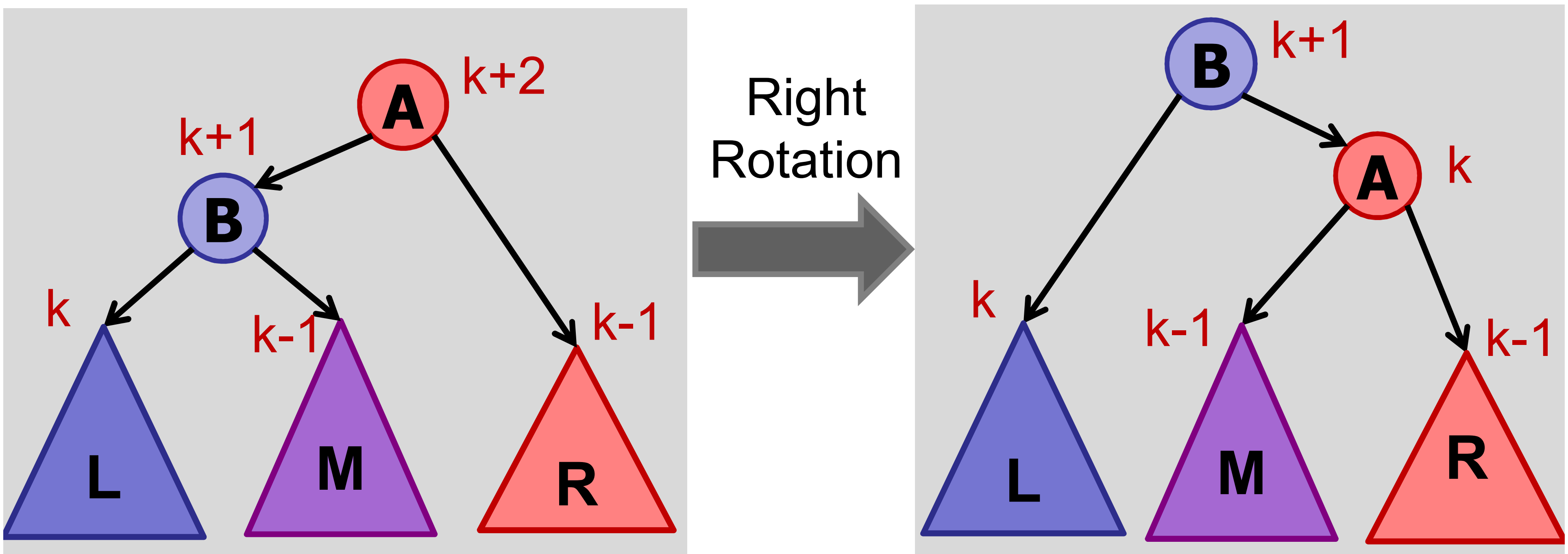


right-rotate:

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations

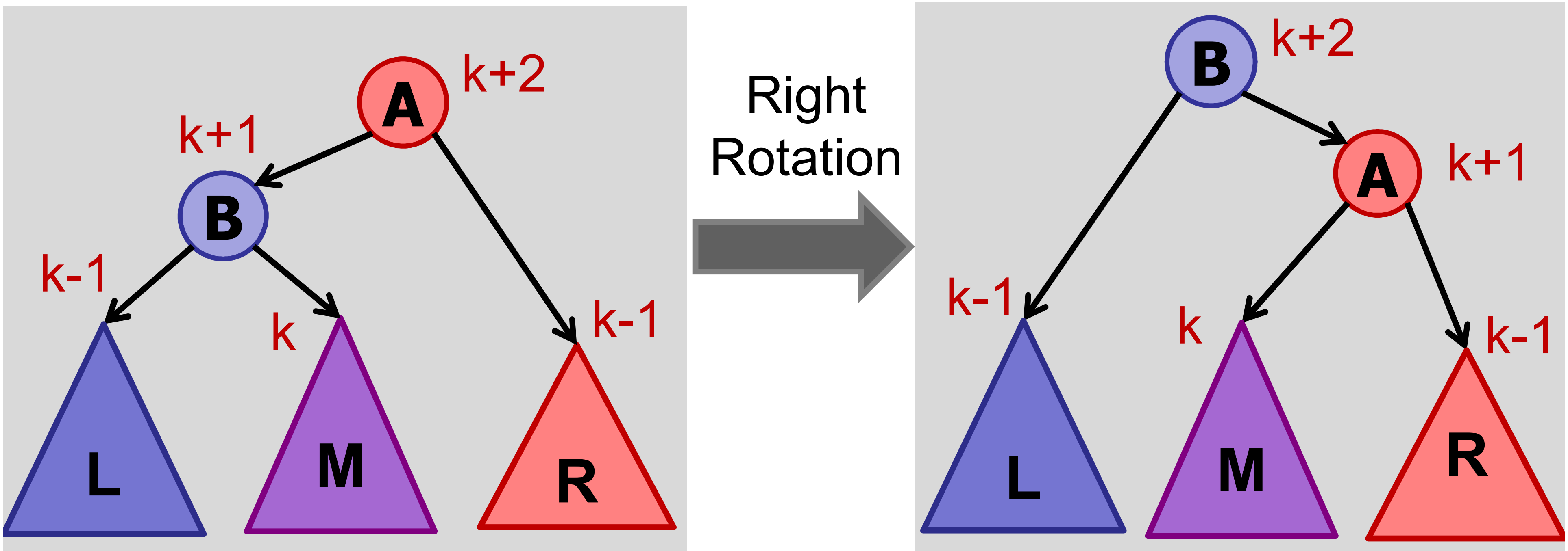


right-rotate:

Case 2: **B** is left-heavy: h(**L**) = h(**M**) +1

h(**R**) = h(**M**)

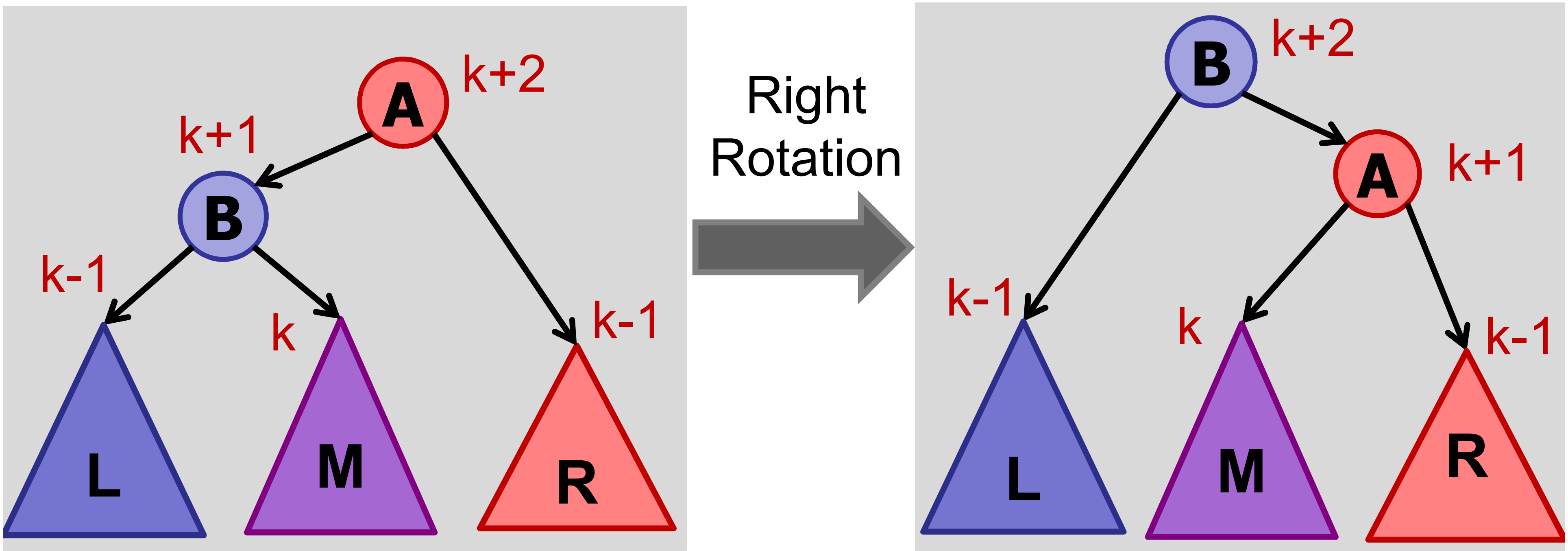
Tree Rotations



right-rotate:

Case 3: **B** is right-heavy: h(**L**) = h(**M**) – 1

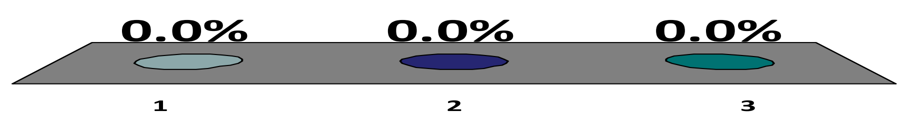
h(**R**) = h(**L**)



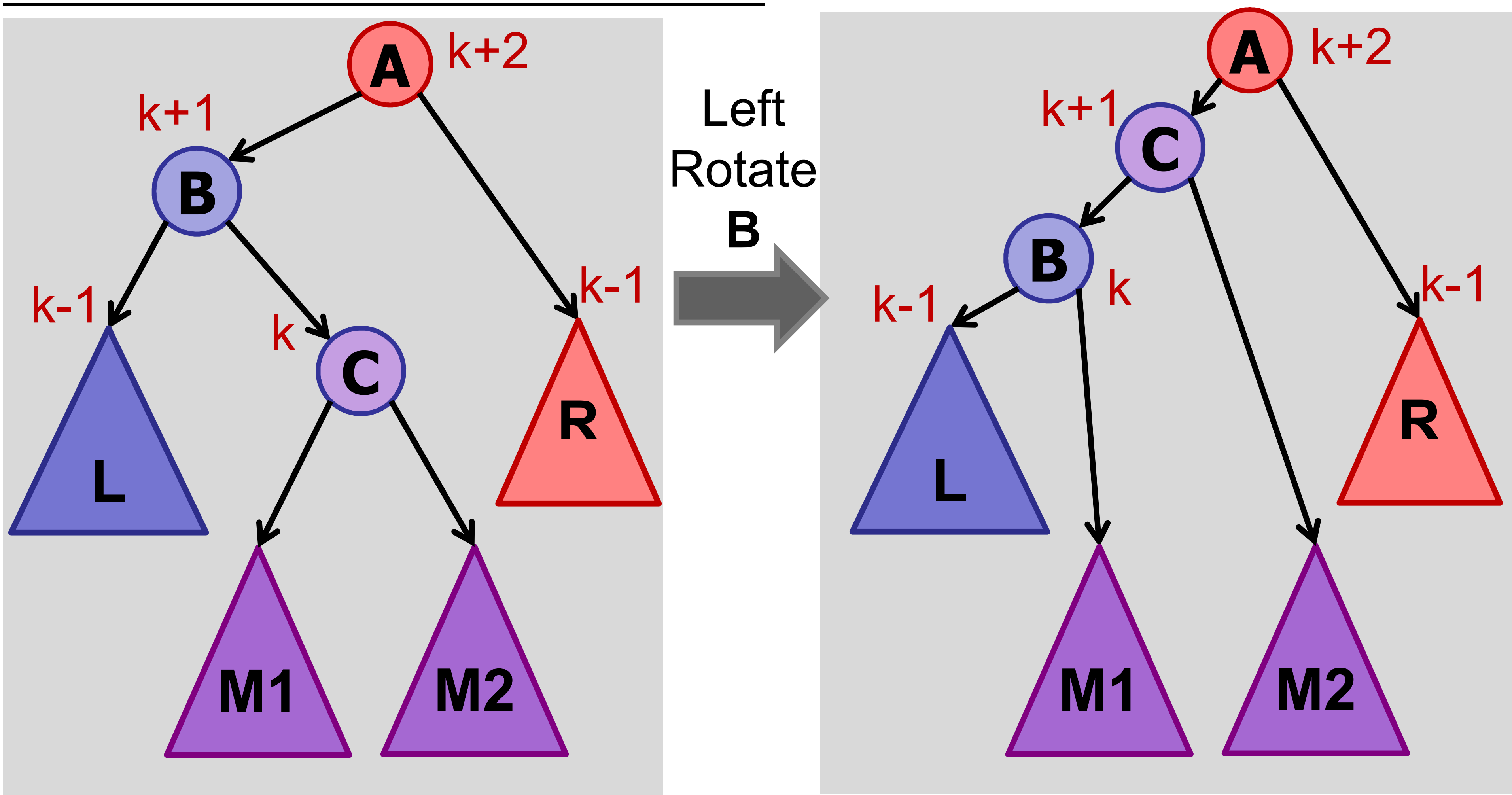
Are we done?

1. Yes.
2. No.
3. Maybe.

0 of 60

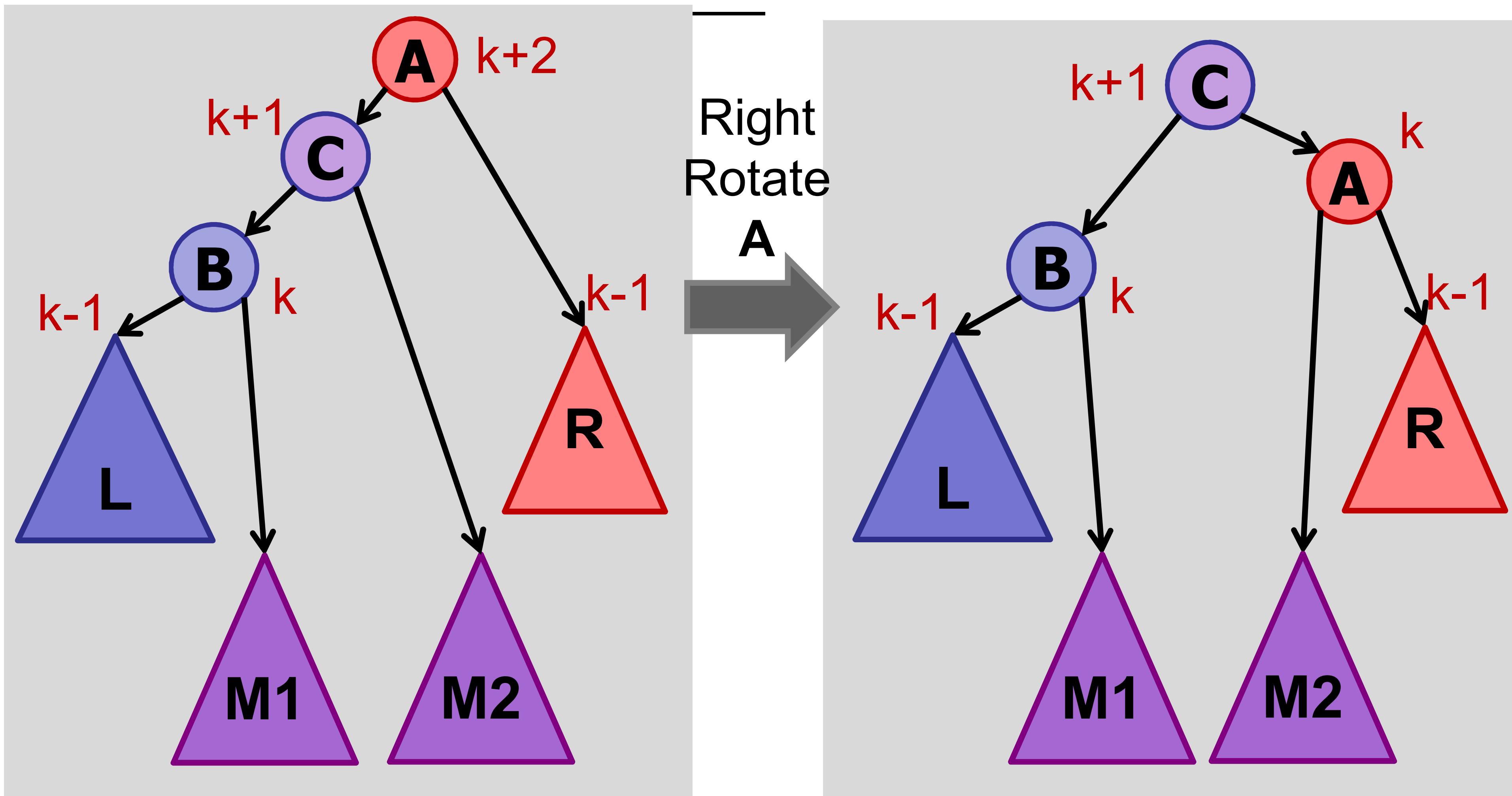


Tree Rotations



After left-rotate: **A** and **C** still out of balance.

Tree Rotations



After right-rotate: all in balance.

Rotations

Summary:

If v is out of balance and left heavy:

1. v.left is balanced: right-rotate(v)
2. v.left is left-heavy: right-rotate(v)
3. v.left is right-heavy: left-rotate(v.left)
   right-rotate(v)

If v is out of balance and right heavy:

Symmetric three cases….

Insert in AVL Tree

Summary:

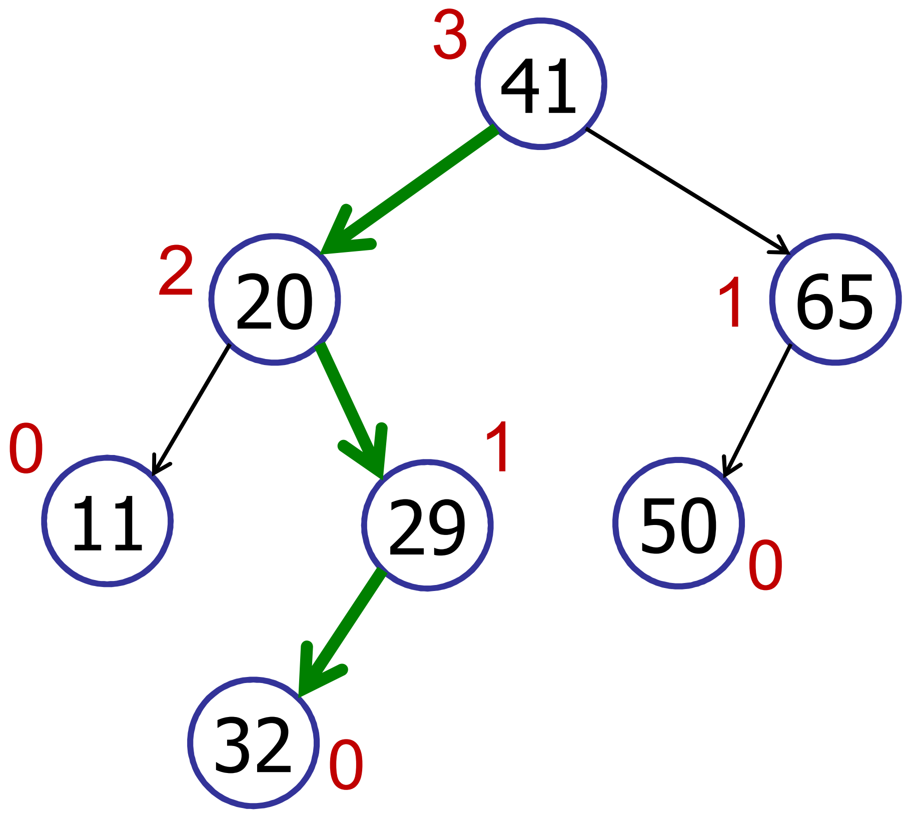
- Insert key in BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance.

Note: may need several rotations before done.

Note: delete is a little more complicated.

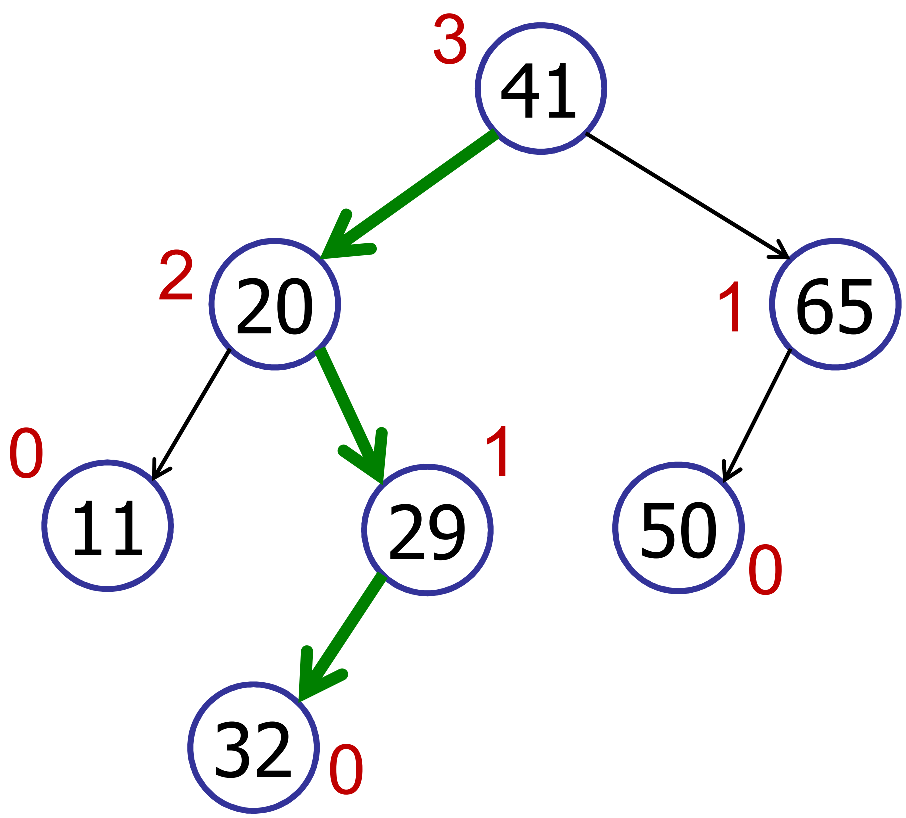
Example

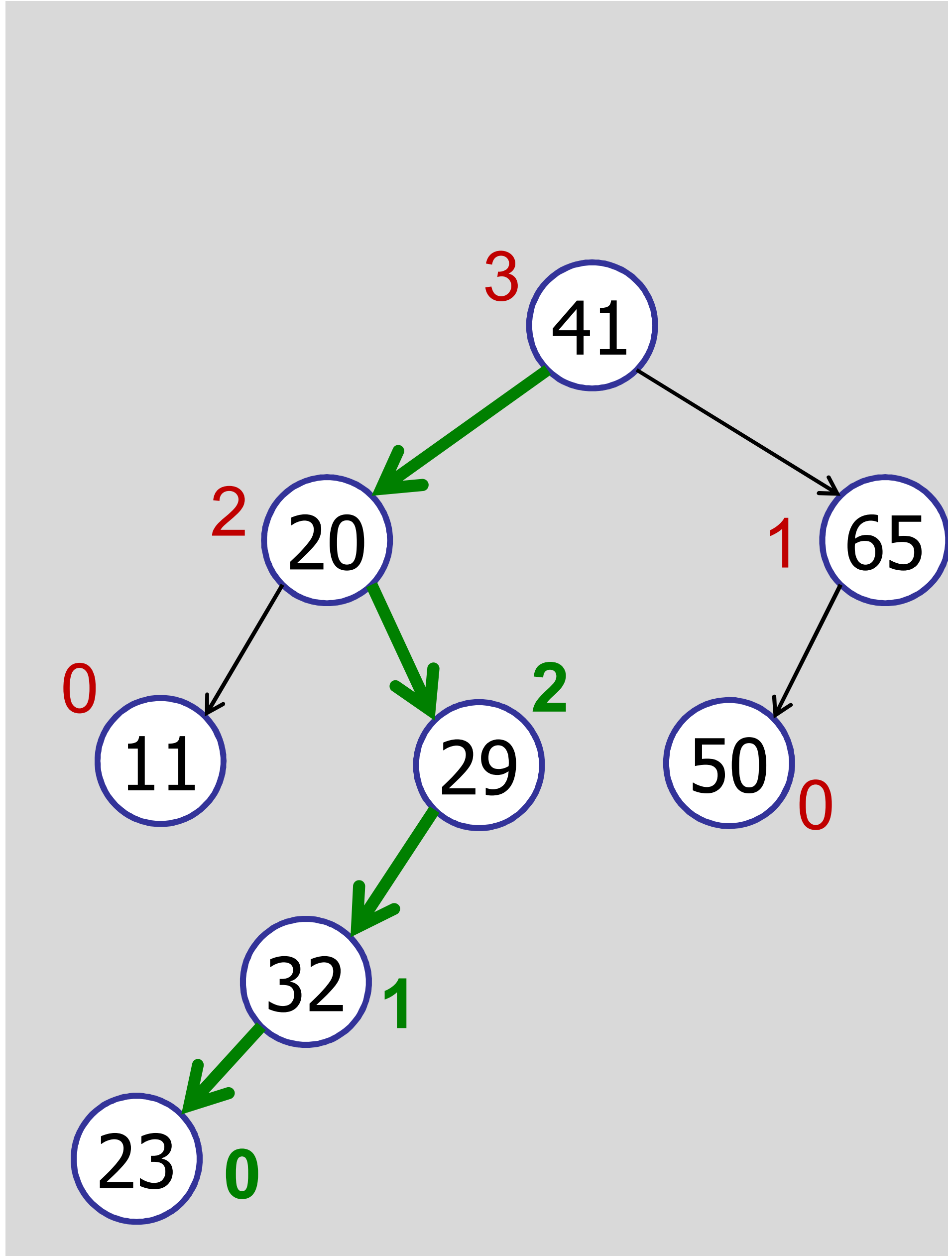
insert(23)



Example

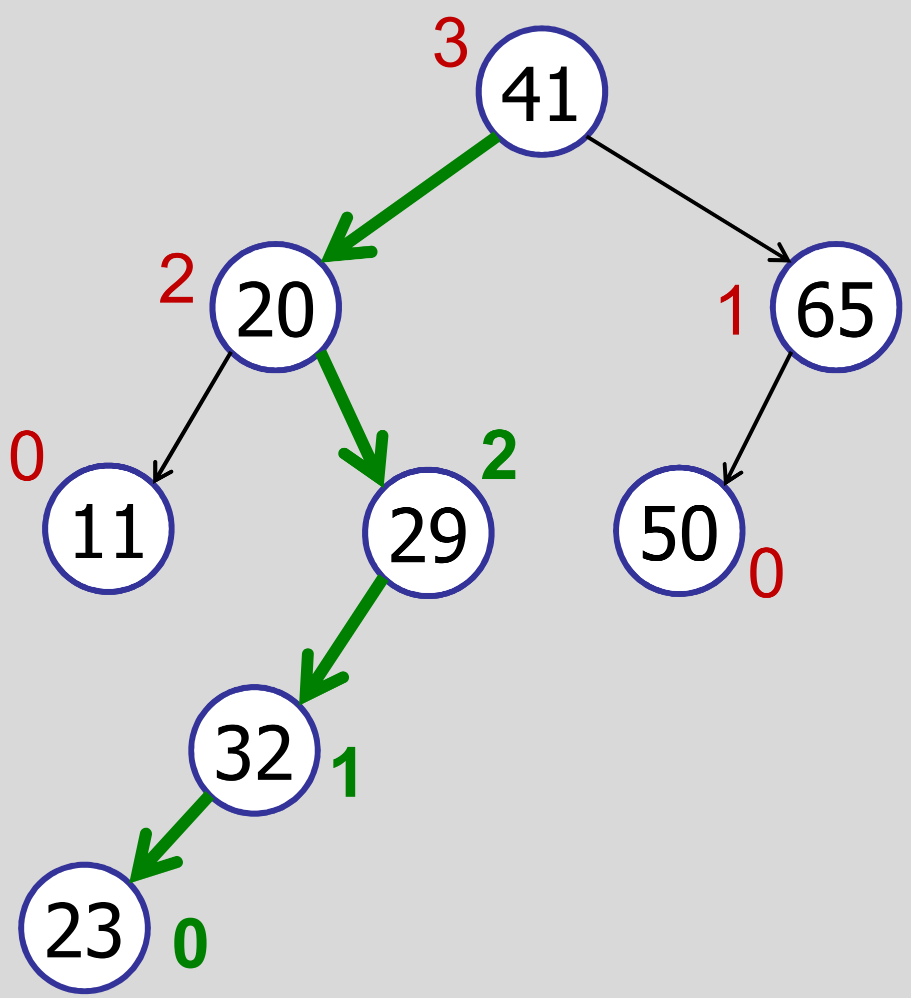
insert(23)

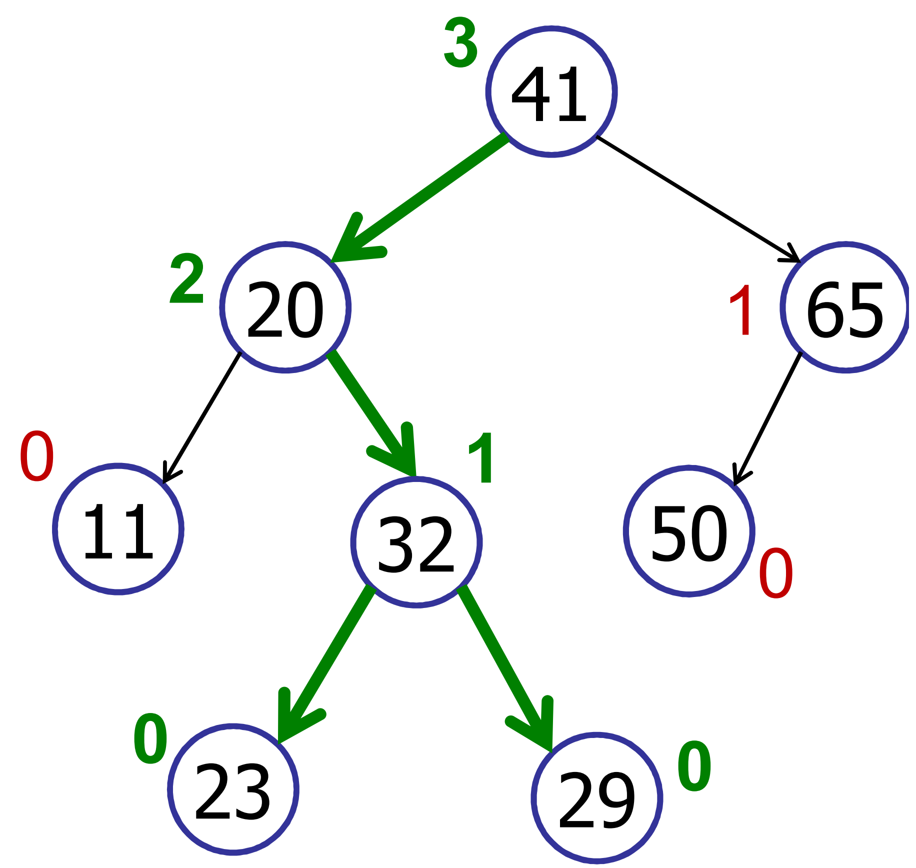




Example

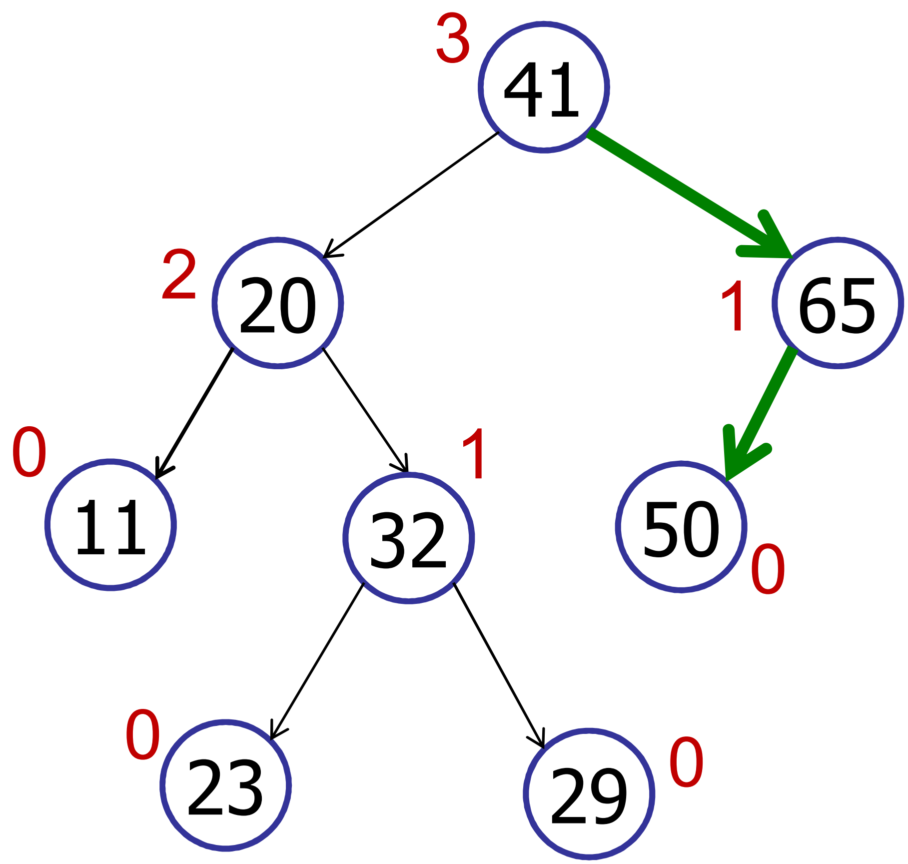
right-rotate(29)





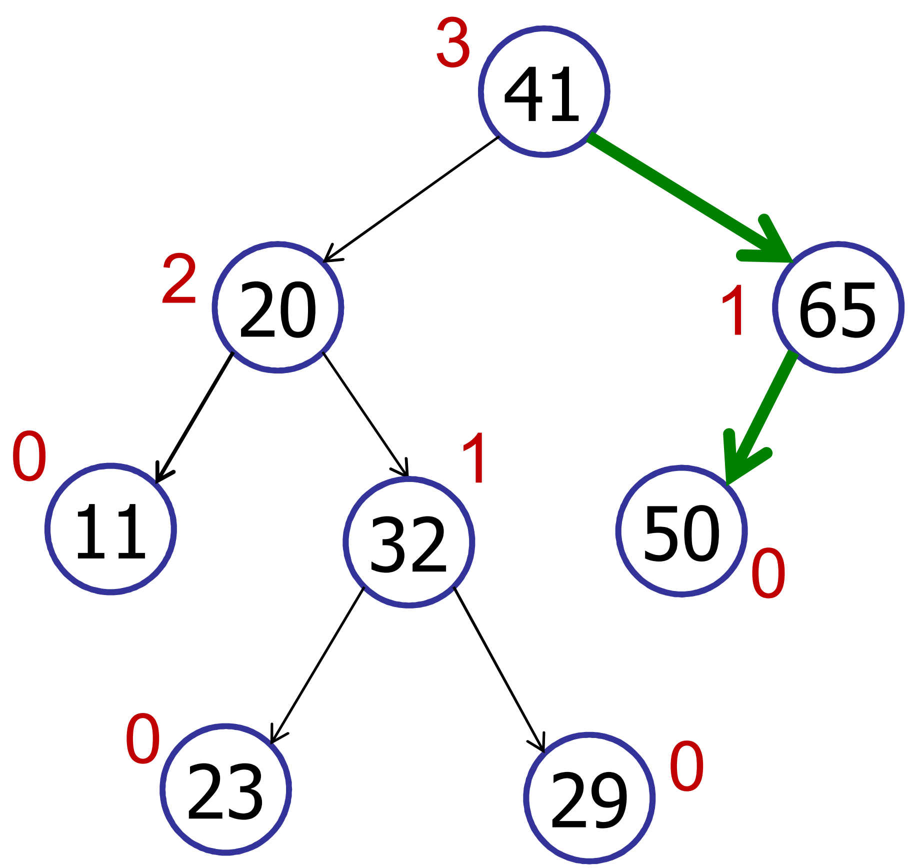
Example

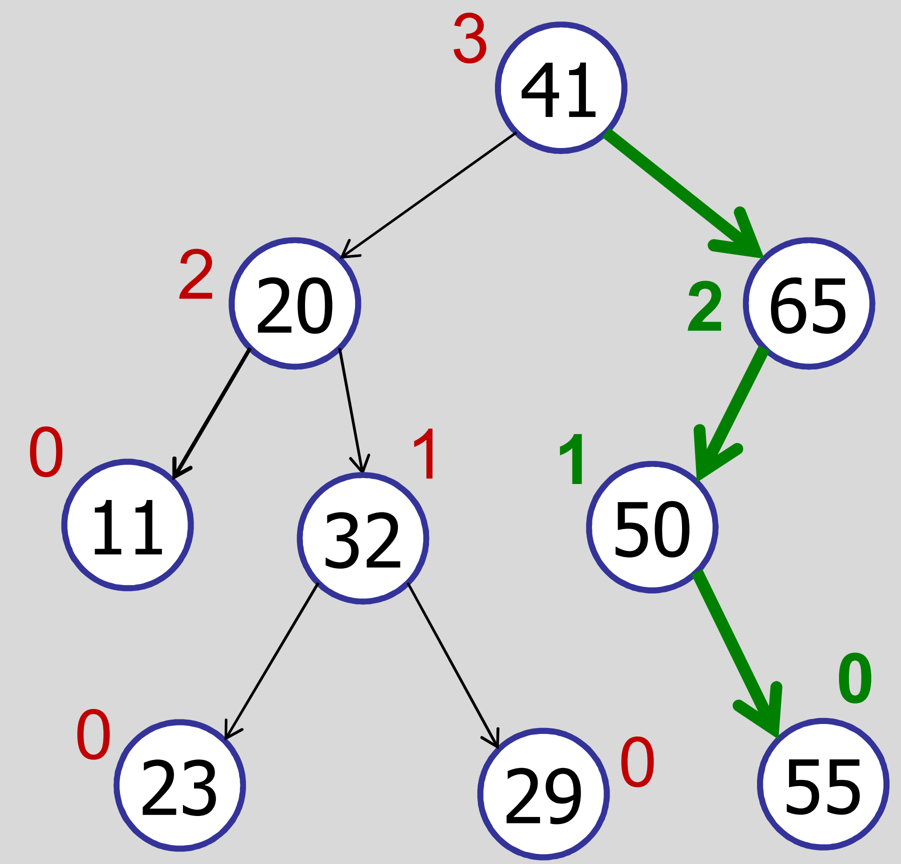
insert(55)



Example

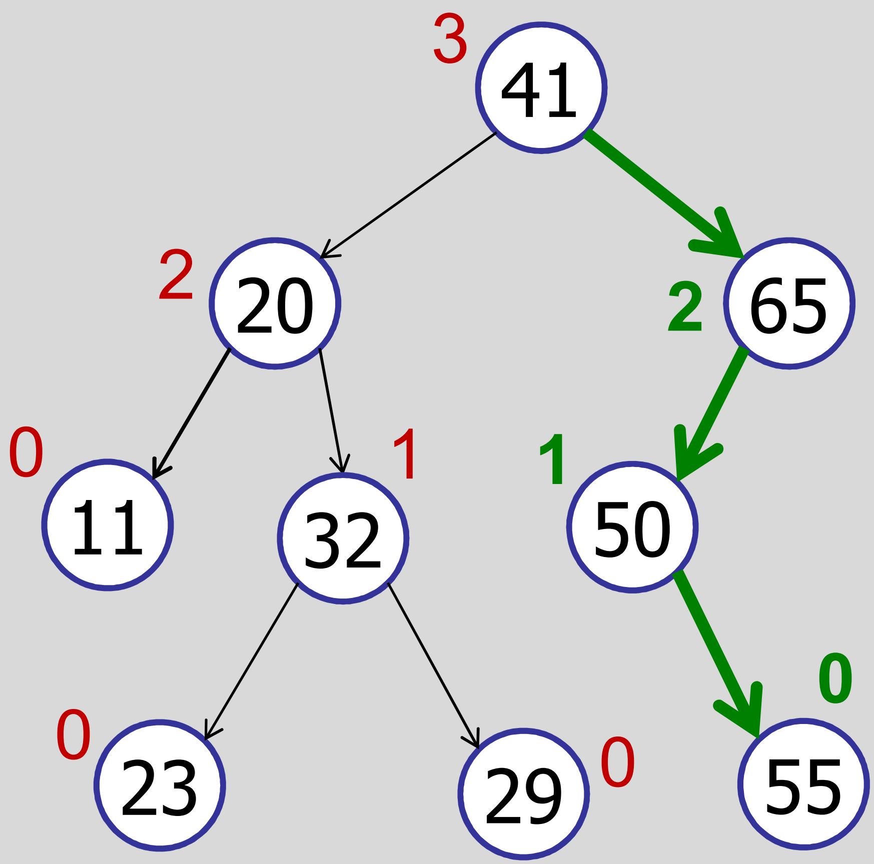
insert(55)

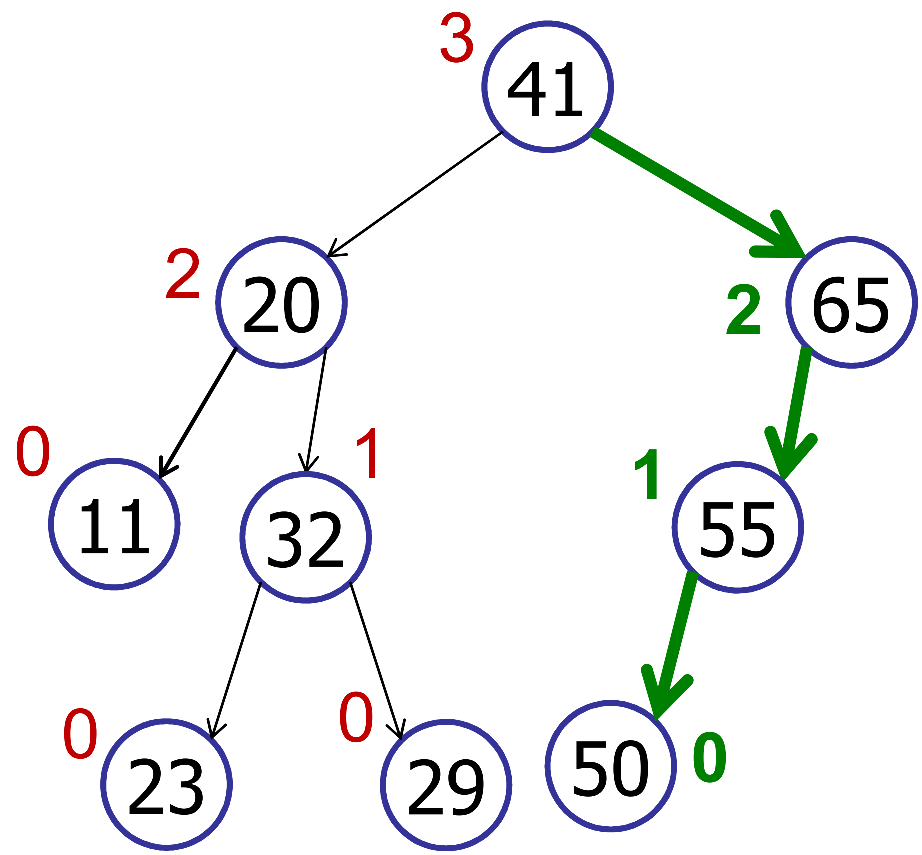




Example

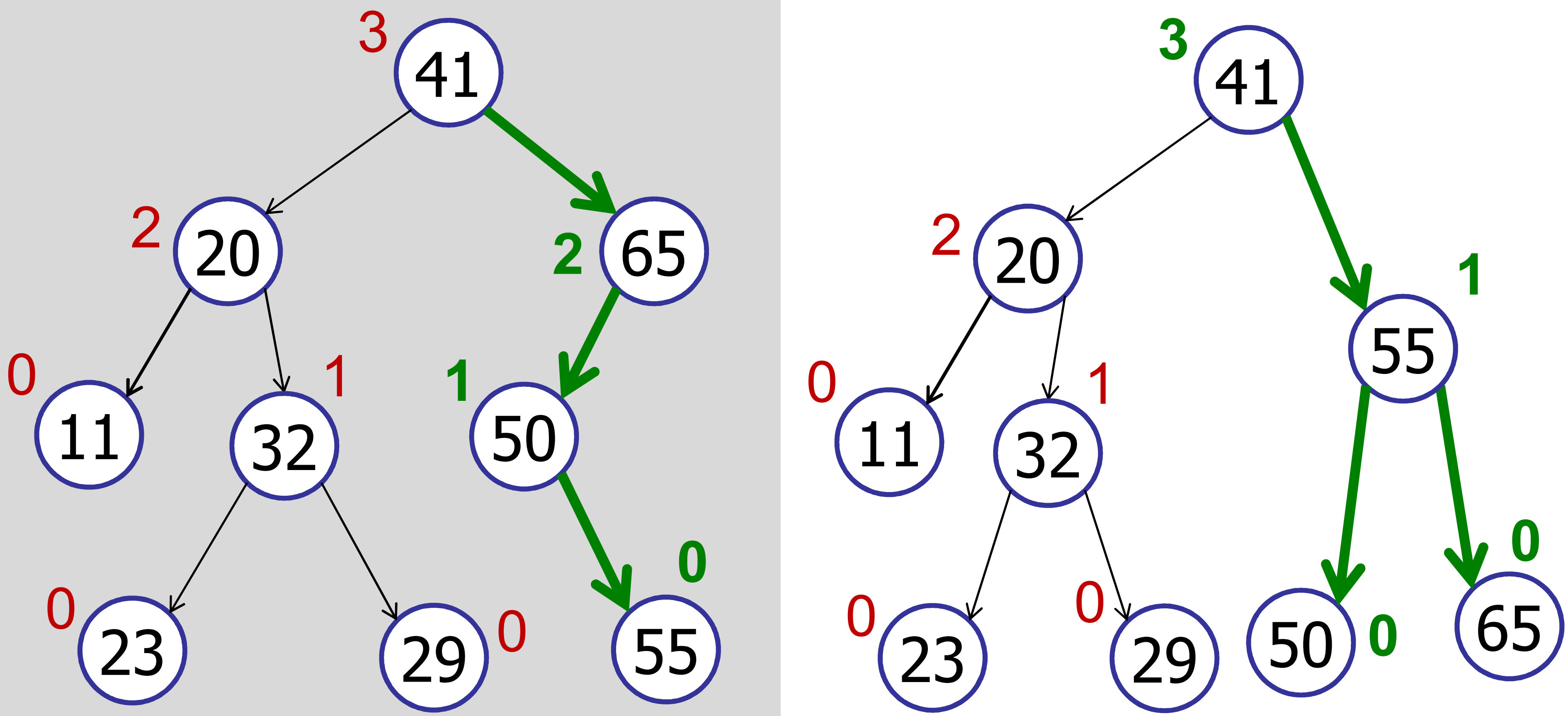
left-rotate(50)





Example

right-rotate(65)



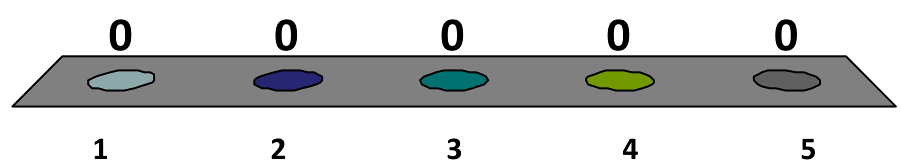
Balanced Search Trees

Many different flavors of balanced search trees

- AVL trees (Adelson-Velsii & Landis, 1962)
- B-trees / 2-3-4 trees (Bayer & McCreight, 1972)
- BB[α] trees (Nievergelt & Reingold 1973)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Treaps (Seidel and Aragon 1996)
- Skip Lists (Pugh 1989)

Quick review: a rotation costs:

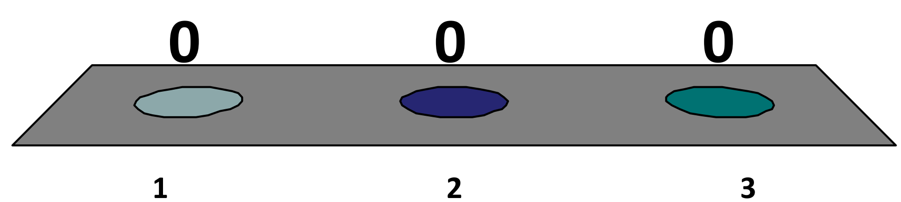
1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(2^n)$



0 of 60

Every insertion requires at least 1 rotation?

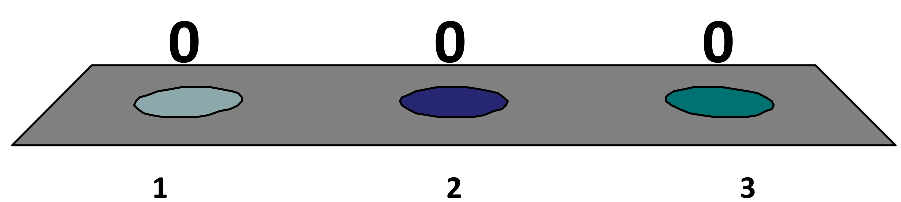
1. Yes
2. No
3. I don’t know



0 of 60

A tree is balanced if every node's children differ in height be at most 1?

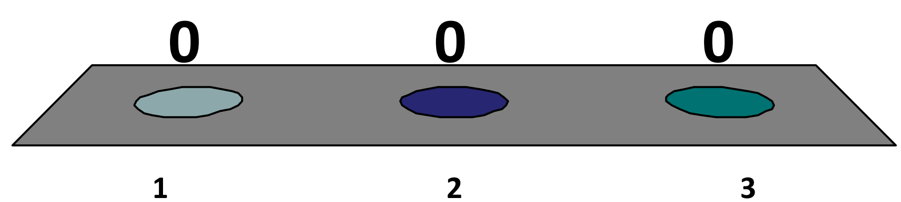
1. Yes
2. No
3. I don't know



0 of 60

A tree is balanced if every node either has two children or zero children?

1. Yes
2. No
3. I don’t know

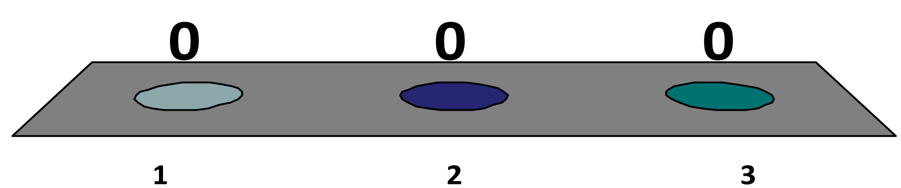


0 of 60

A tree is balanced if:
For every node, the number of keys in its heavier sub-tree is at most twice the number of keys in its lighter sub-tree.

1. Yes
2. No
3. I don’t know

0 of 60



Balanced Search Trees

Summary:

– The Importance of Being Balanced
– Height Balanced Trees
– Rotations
– AVL trees

Next time:

– Heaps
– Priority Queues

Augmented Search Trees

Many problems require storing additional data in the binary search tree:

- Dynamic order statistics (find median, etc.)
- Rank (find position in list)
- Interval trees
- Geometric data structures
- etc…

Augmented Search Trees

Dynamic Order Statistics

Implement a binary search tree that supports:

- insert(int key)
- search(int key)

and also:

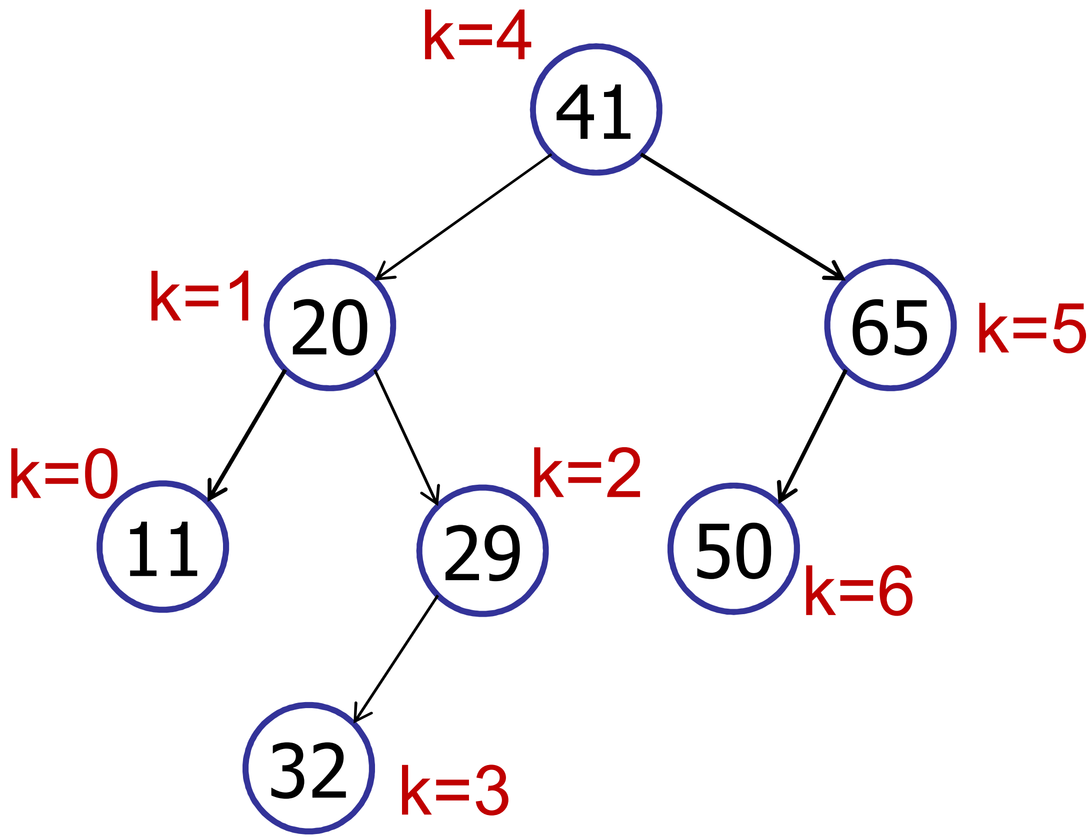
- select(int k)

7	**9**	**13**	**18**	**22**	**25**	**43**	**52**	**65**	**67**	**87**	**92**

select(4)

Dynamic Order Statistics

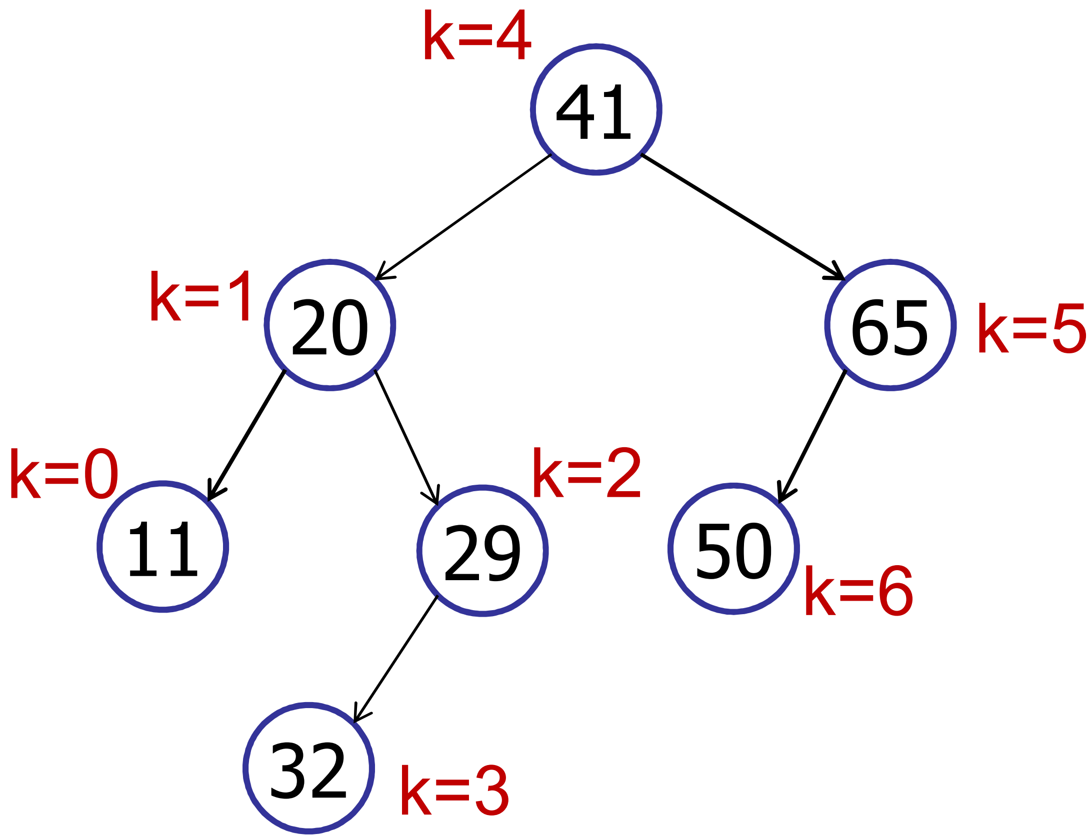
Option 1: store rank in every node



(Nota bene: k=rank, not height.)

Dynamic Order Statistics

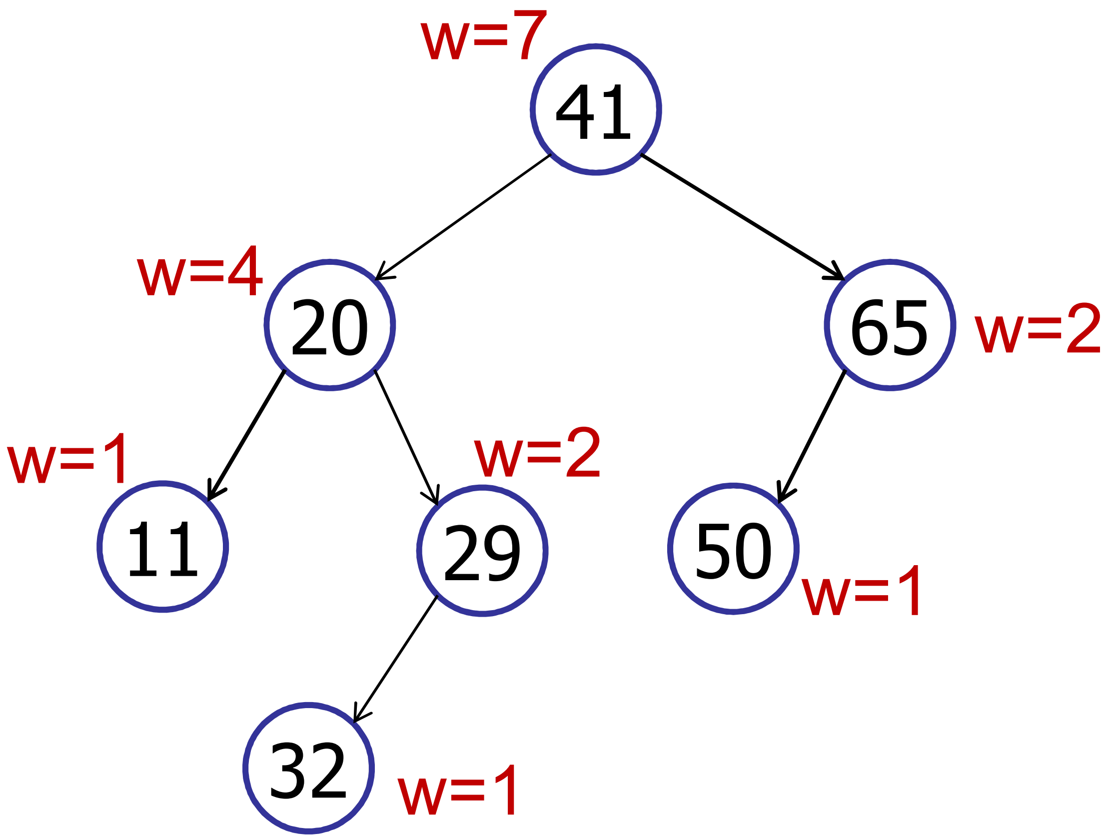
Option 1: store rank in every node



Problem: insert(5) requires updating all the ranks!

Dynamic Order Statistics

Option 2: store size of sub-tree in every node



Nota bene: w=weight, not height.

Dynamic Order Statistics

Option 2: store size of sub-tree in every node

The weight of a node is the size of the tree rooted at that node.

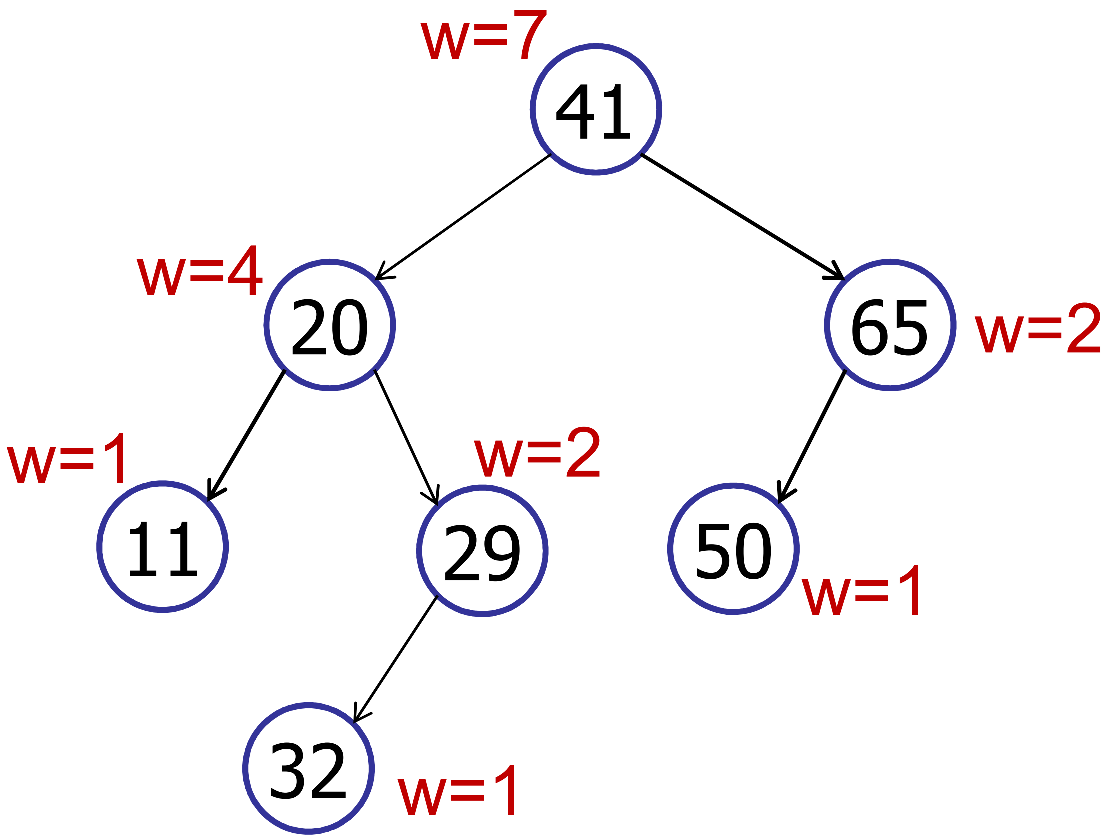
Define weight:

$w(\text{leaf}) = 1$

$w(v) = w(v.\text{left}) + w(v.\text{right}) + 1$

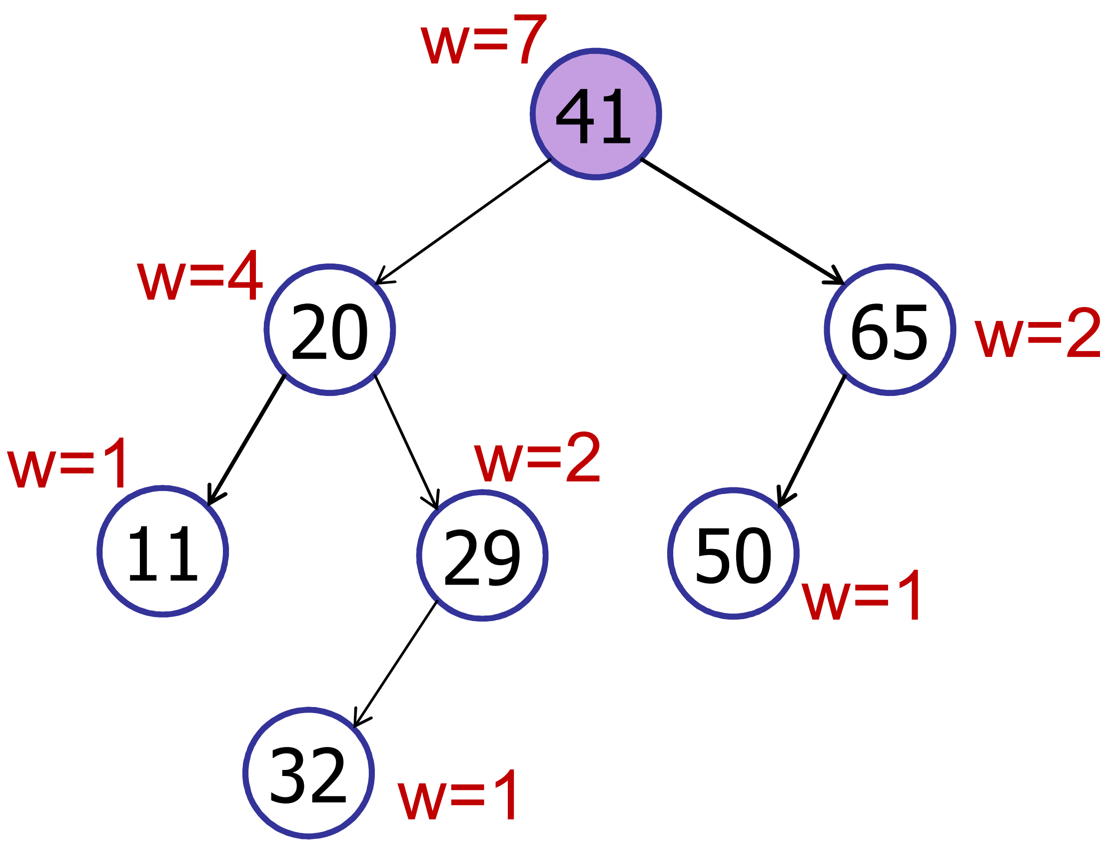
Dynamic Order Statistics

Option 2: store size of sub-tree in every node



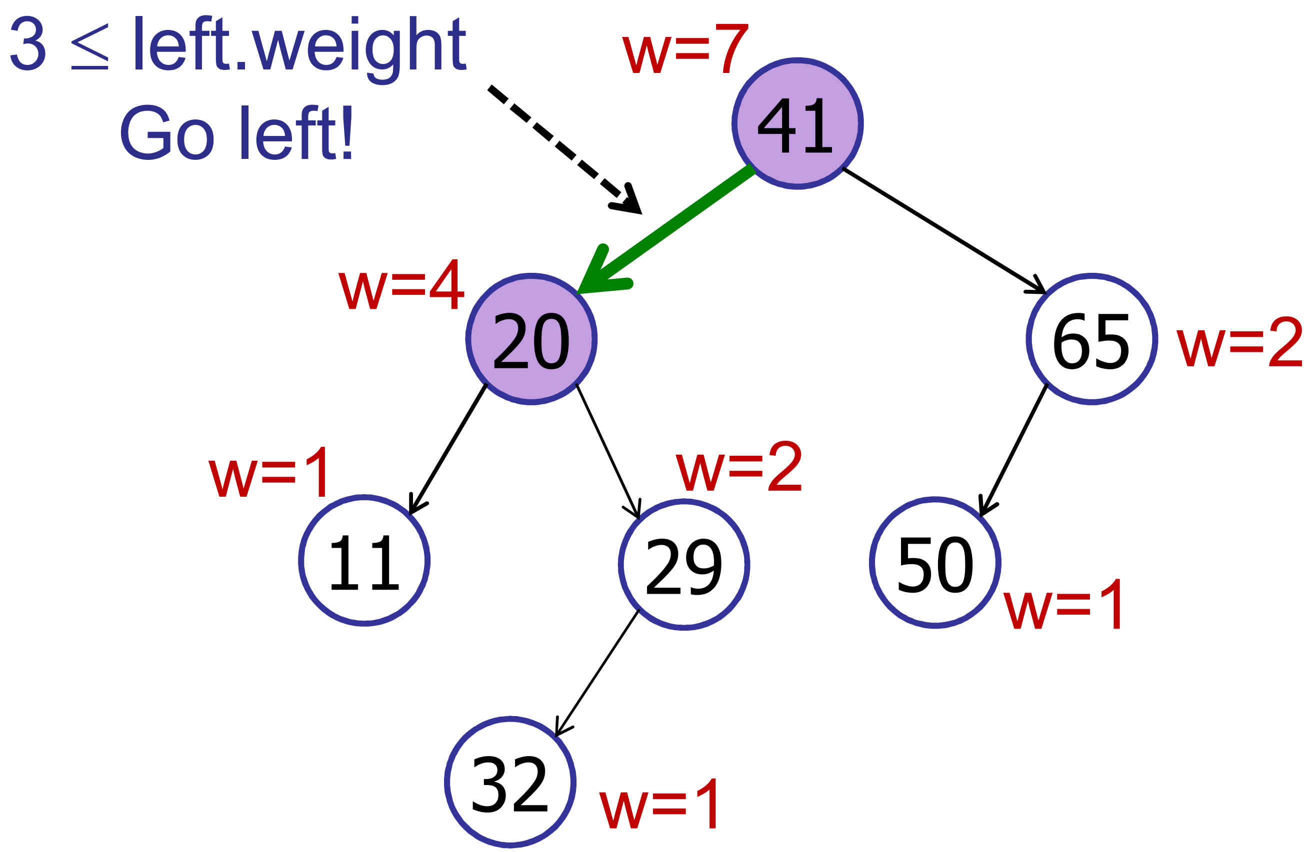
Dynamic Order Statistics

Example: select(3)



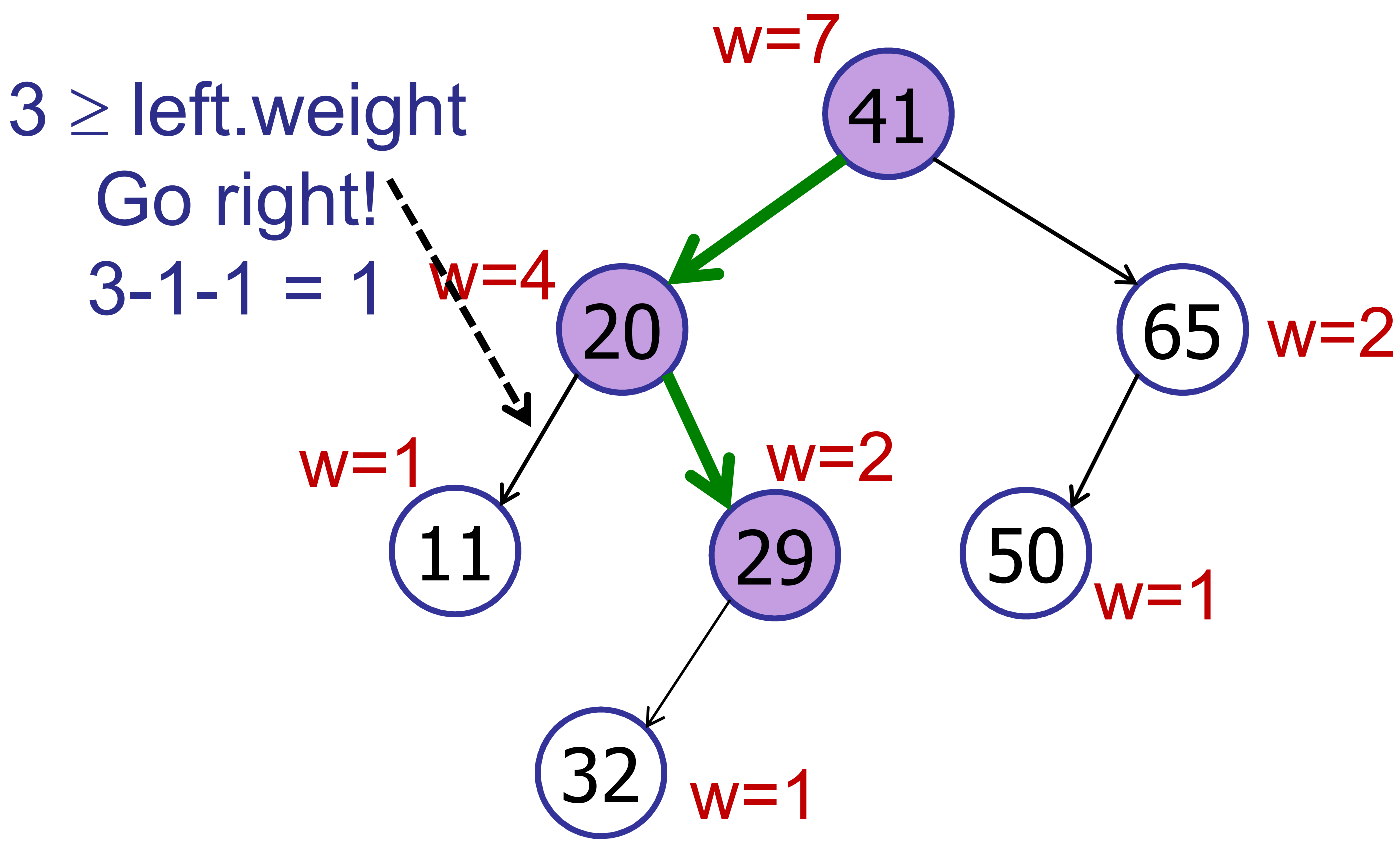
Dynamic Order Statistics

Example: select(3)



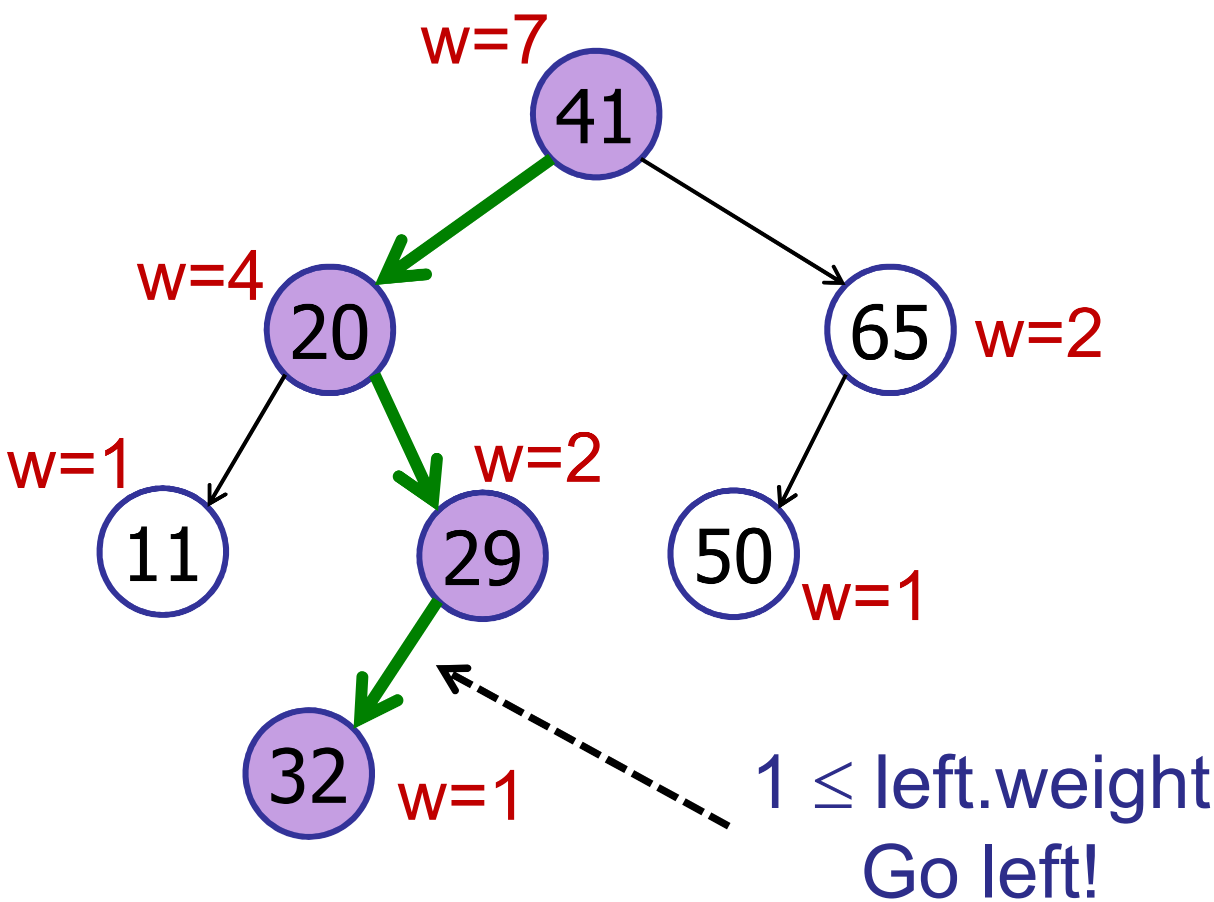
Dynamic Order Statistics

Example: select(3)



Dynamic Order Statistics

Example: select(3)



Dynamic Order Statistics

```
select(v, k)
    r = v.left.weight + 1;
    if (k==r) then
        return v;
    else if (k < r) then
        return select(v.left, k);
    else if (k > r) then
        return select(v.right, k-r);
```

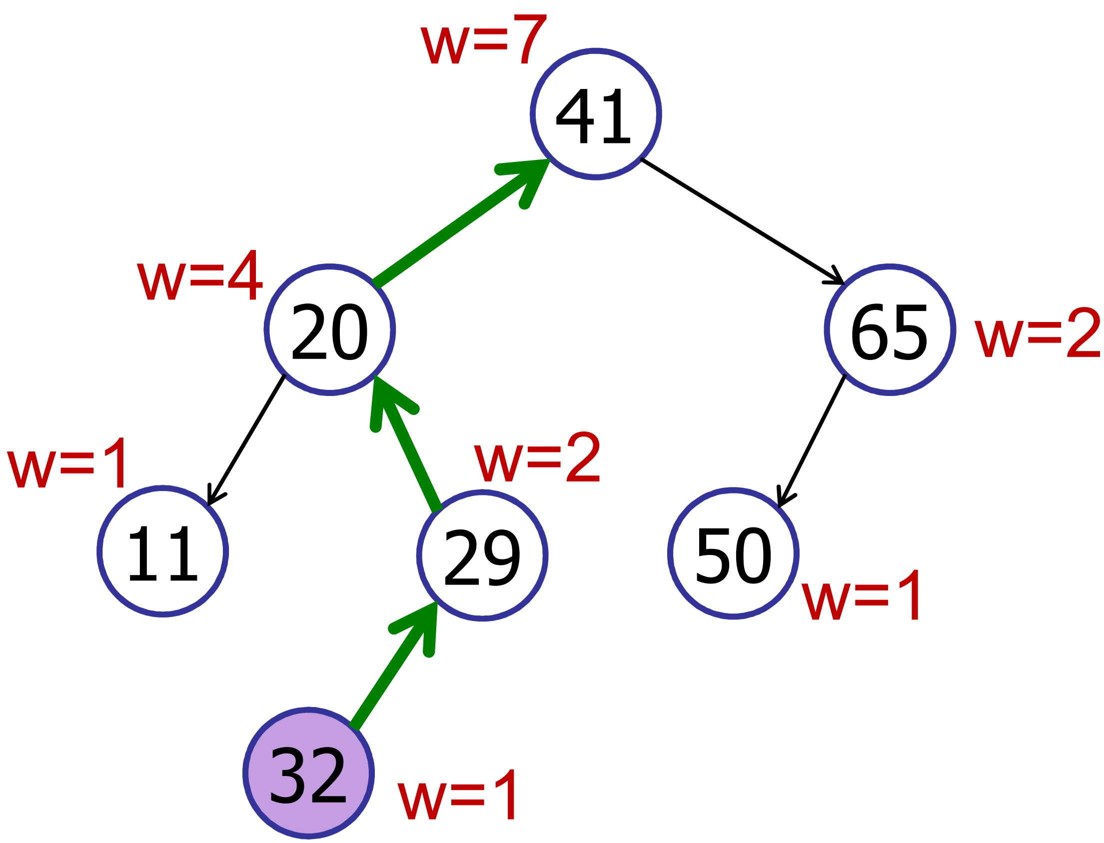
Dynamic Order Statistics

Rank(v) : computes the rank of a node v

```
rank(v)
    r = v.left.weight + 1;
    while (v != root) do
        if v is right child then
            r += y.parent.left.weight + 1
        y = y.parent
    return r;
```

Dynamic Order Statistics

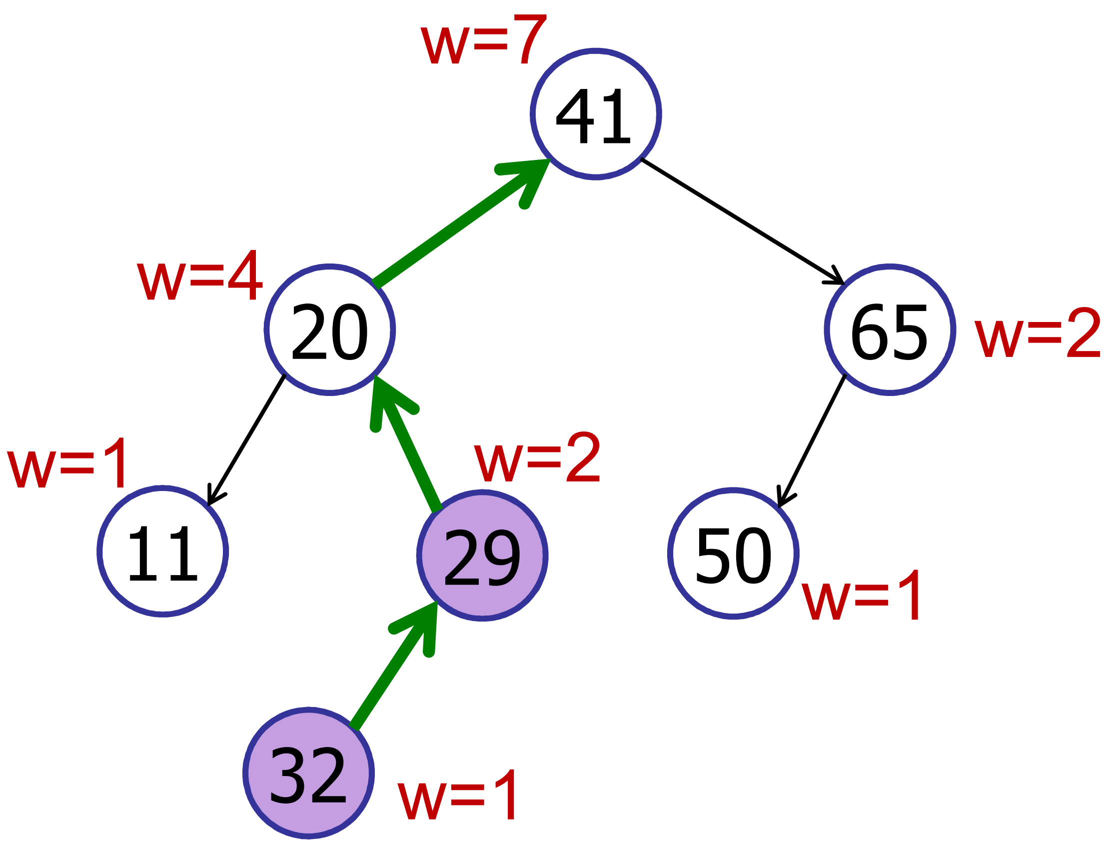
Example: rank(32)



rank = 1

Dynamic Order Statistics

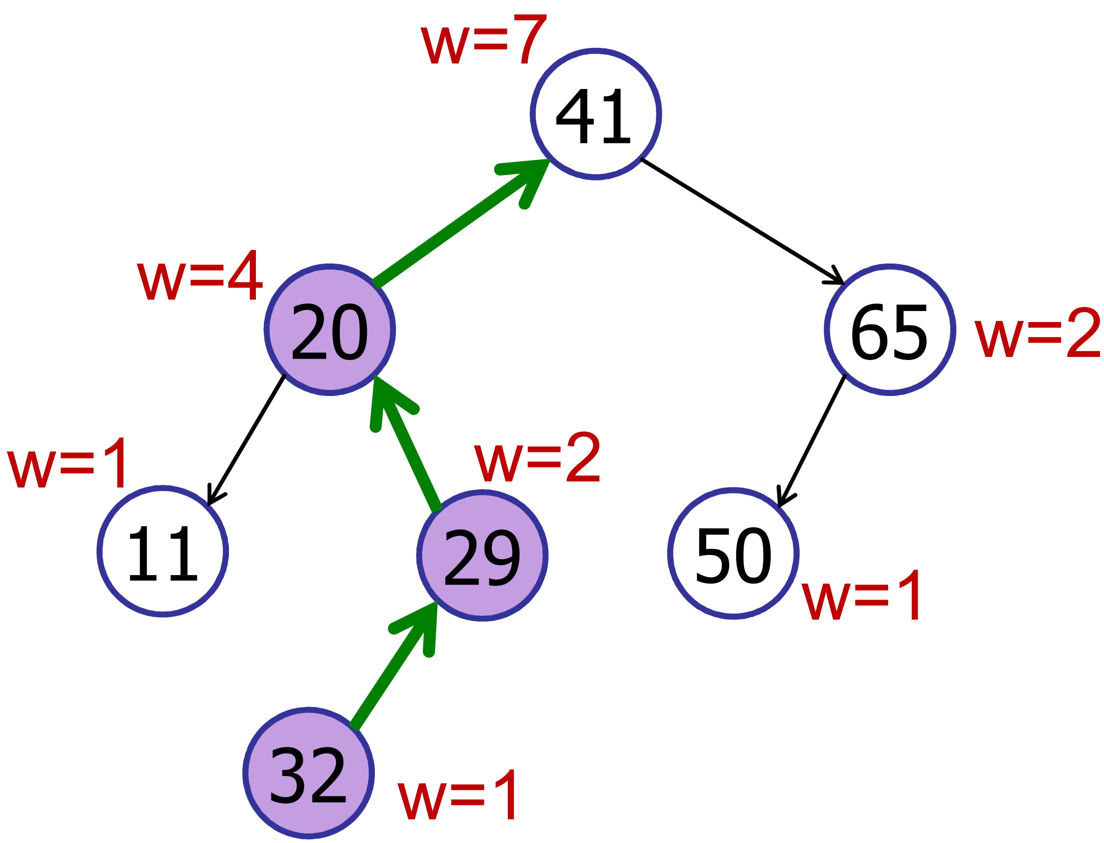
Example: rank(32)



rank = 1

Dynamic Order Statistics

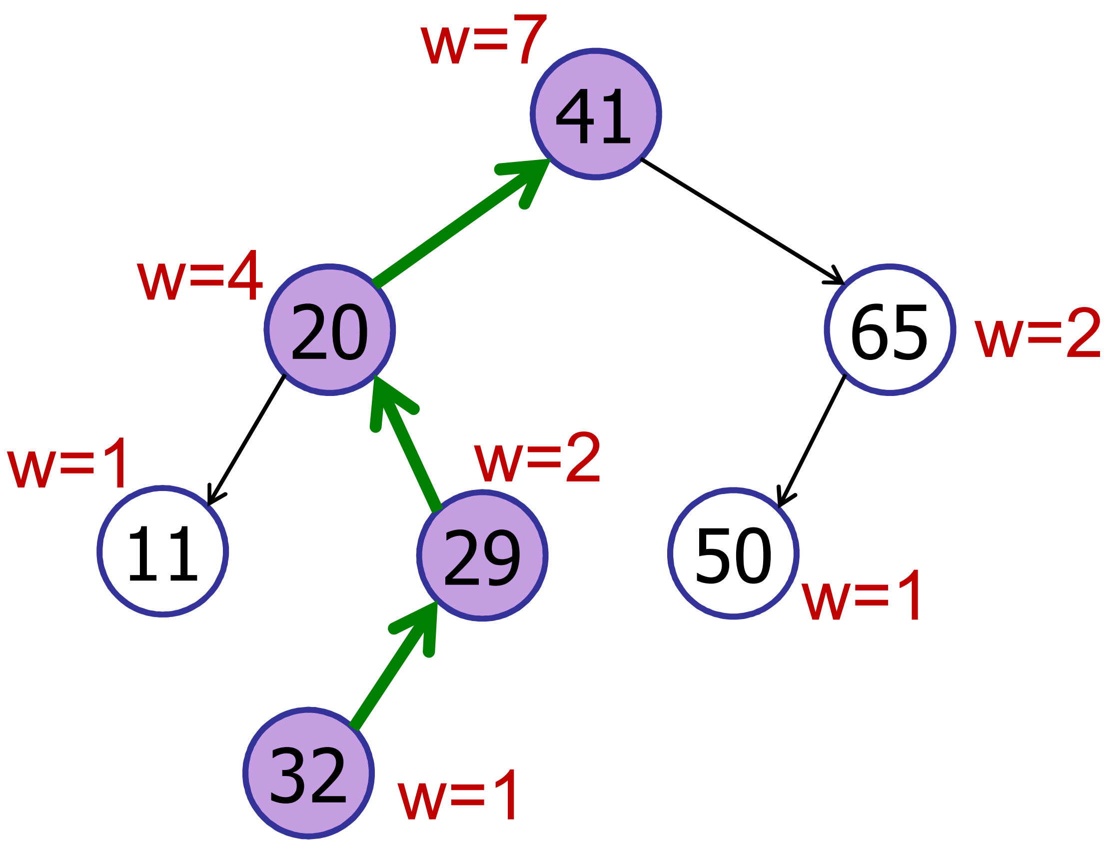
Example: rank(32)



rank = 1 + 2

Dynamic Order Statistics

Example: rank(32)

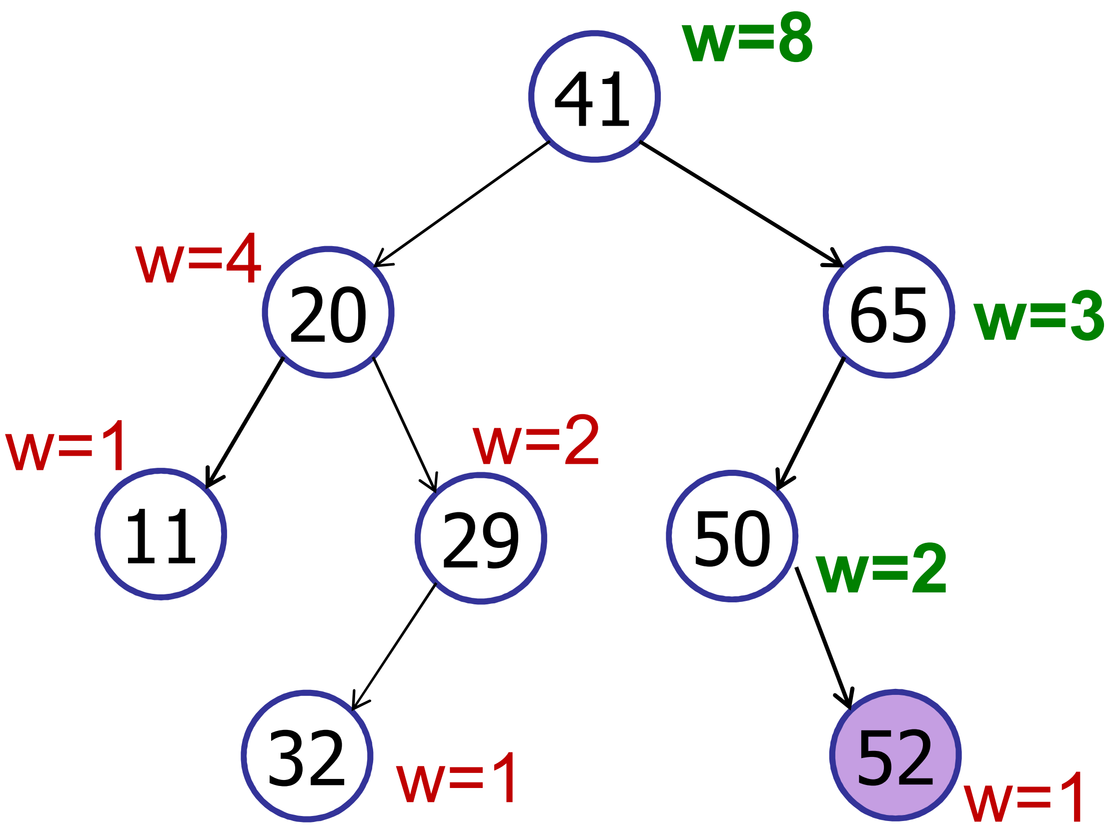


rank = 1 + 2 = 3

Augmented Trees

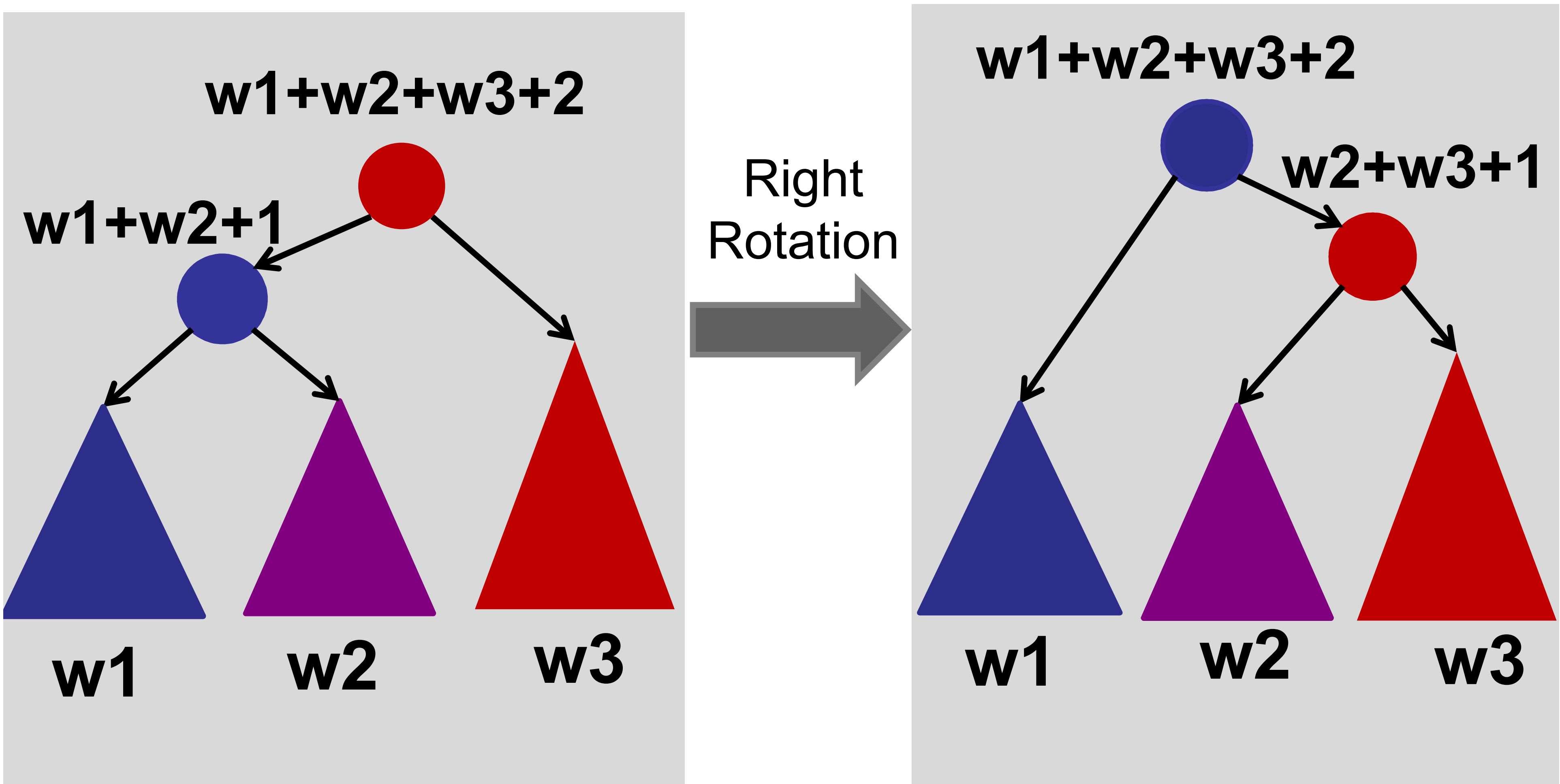
Maintain weight during insertions:

– Just like maintaining height…



Augmented Trees

Maintain weight during rotations:



Balanced Search Trees

Summary:

- The Importance of Being Balanced
- Height Balanced Trees
- Rotations
- AVL trees
- Augmented Search Trees

Next time:

- Heaps
- Priority Queues