CS2040S
Data Structures and Algorithms

Welcome!

Plan of the Day

Trees

- Terminology
- Traversals
- Operations

Balanced Trees

- Height-balanced binary search trees
- AVL trees
- Rotations

Part 2

On the importance of being balanced

Part 2

On the importance of being balanced

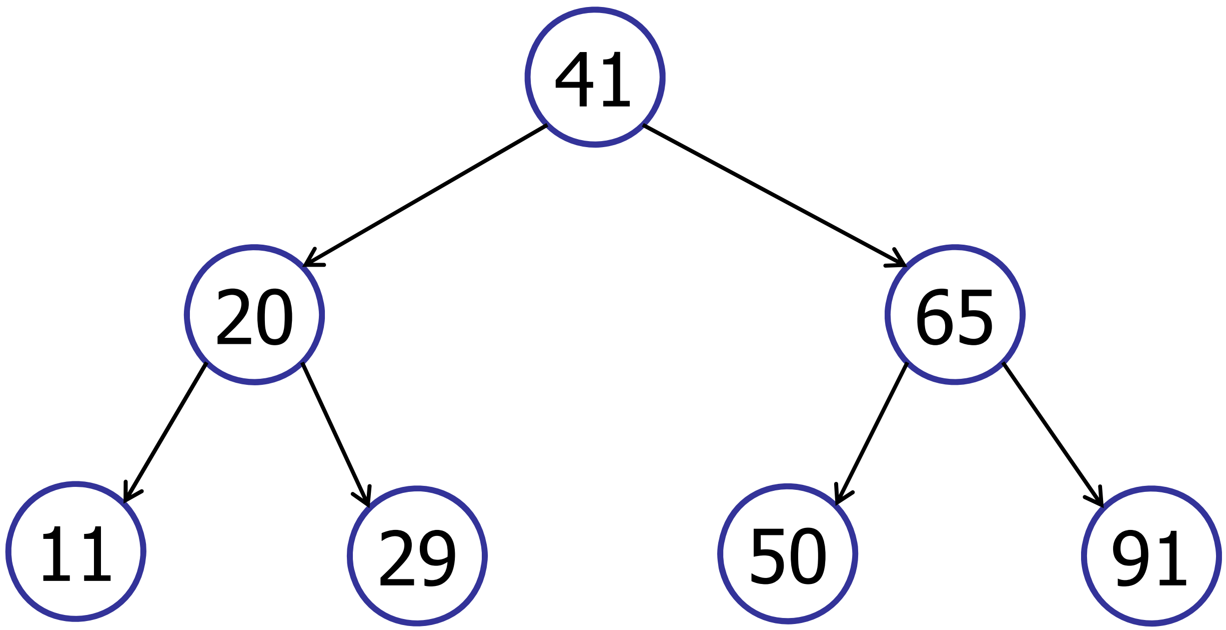
- Height-balanced binary search trees
- AVL trees
- Rotations

Dictionary Interface

A collection of (key, value) pairs:

interface	**IDictionary**	
`void`	`insert(Key k, Value v)`	*insert (k,v) into table*
`Value`	`search(Key k)`	*get value paired with k*
`Key`	`successor(Key k)`	*find next key > k*
`Key`	`predecessor(Key k)`	*find next key < k*
`void`	`delete(Key k)`	*remove key k (and value)*
`boolean`	`contains(Key k)`	*is there a value for k?*
`int`	`size()`	*number of (k,v) pairs*

Recap: 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

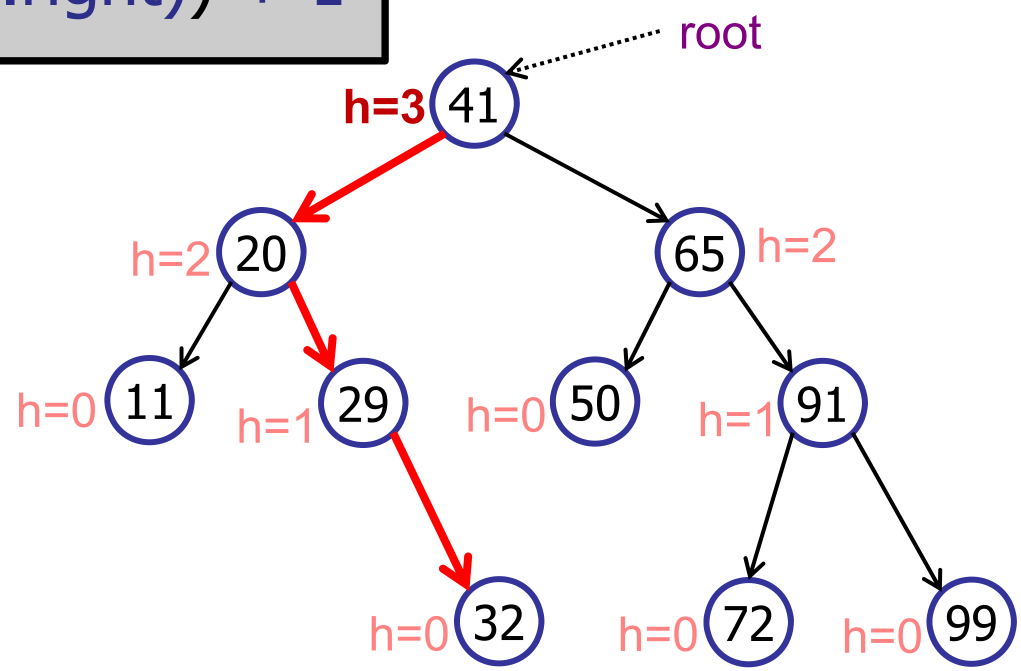
Binary Search Trees Heights

Height:

Number of edges on longest path from root to leaf.

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

$h(v) = \max(h(v.\mathrm{left}), h(v.\mathrm{right})) + 1$



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

Binary Search Tree

Modifying Operations

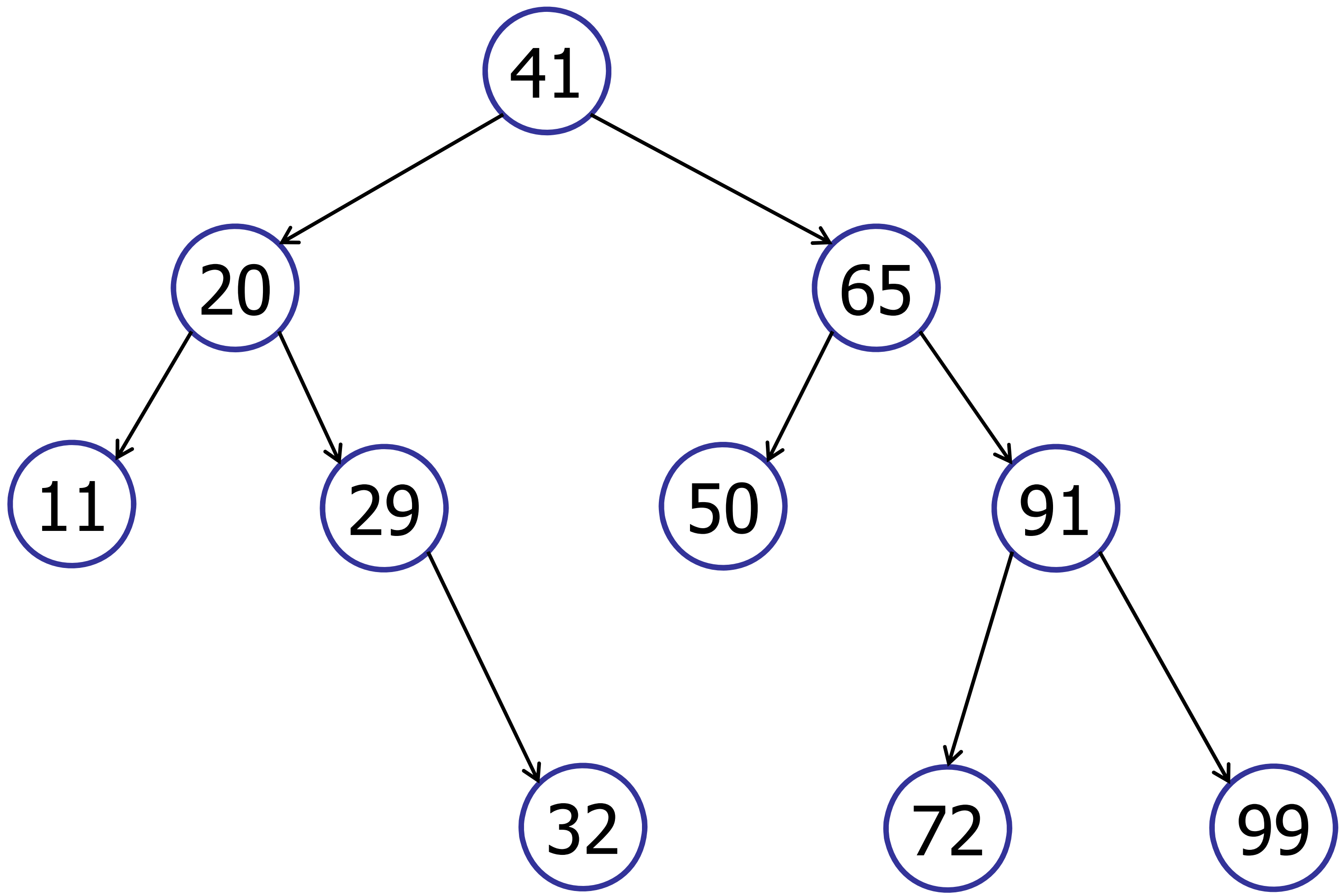
- insert
- delete

Query Operations:

- search
- predecessor, successor
- findMax, findMin
- in-order-traversal

Binary Search Tree

delete(v)

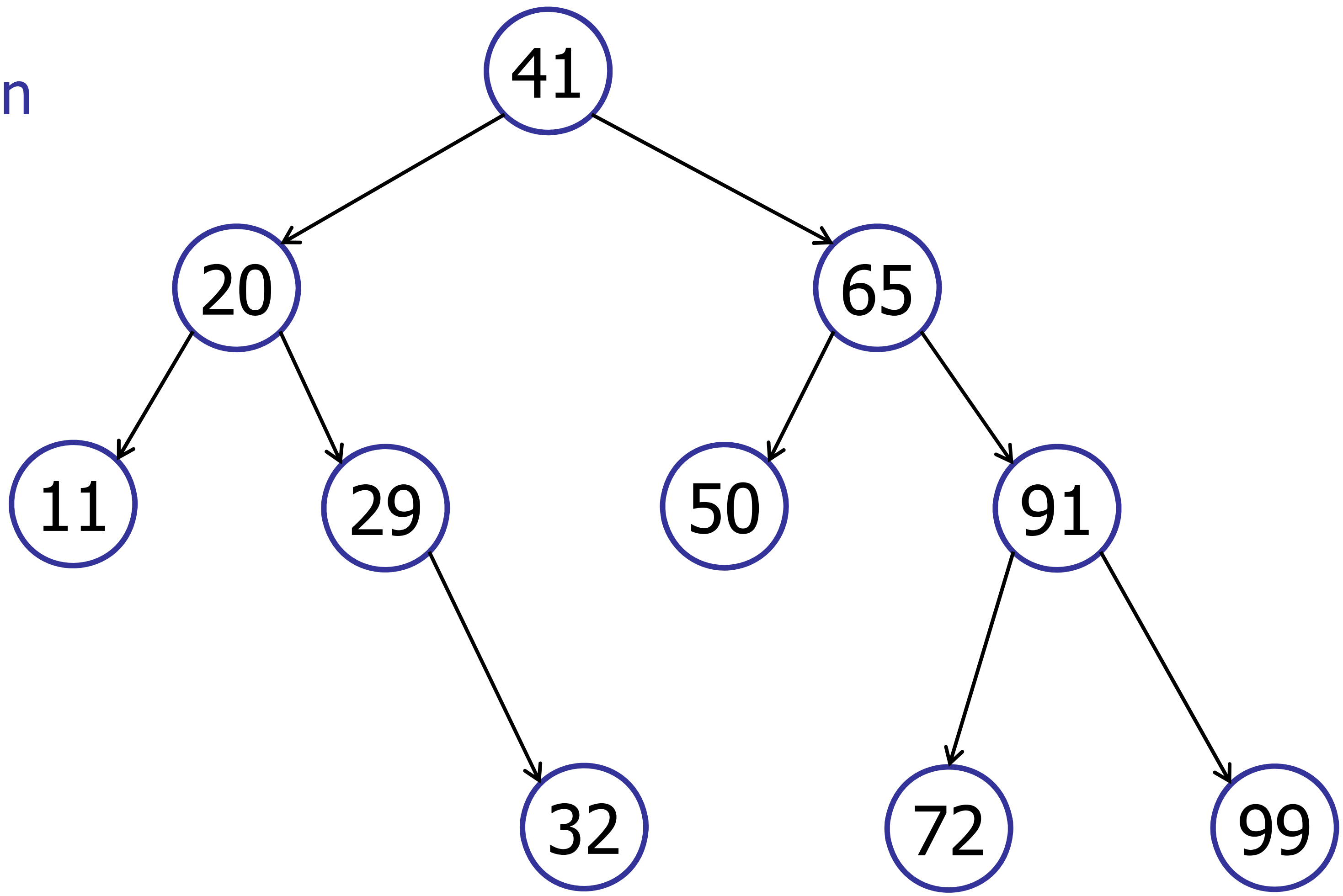


Binary Search Tree

delete(v)

Three cases:

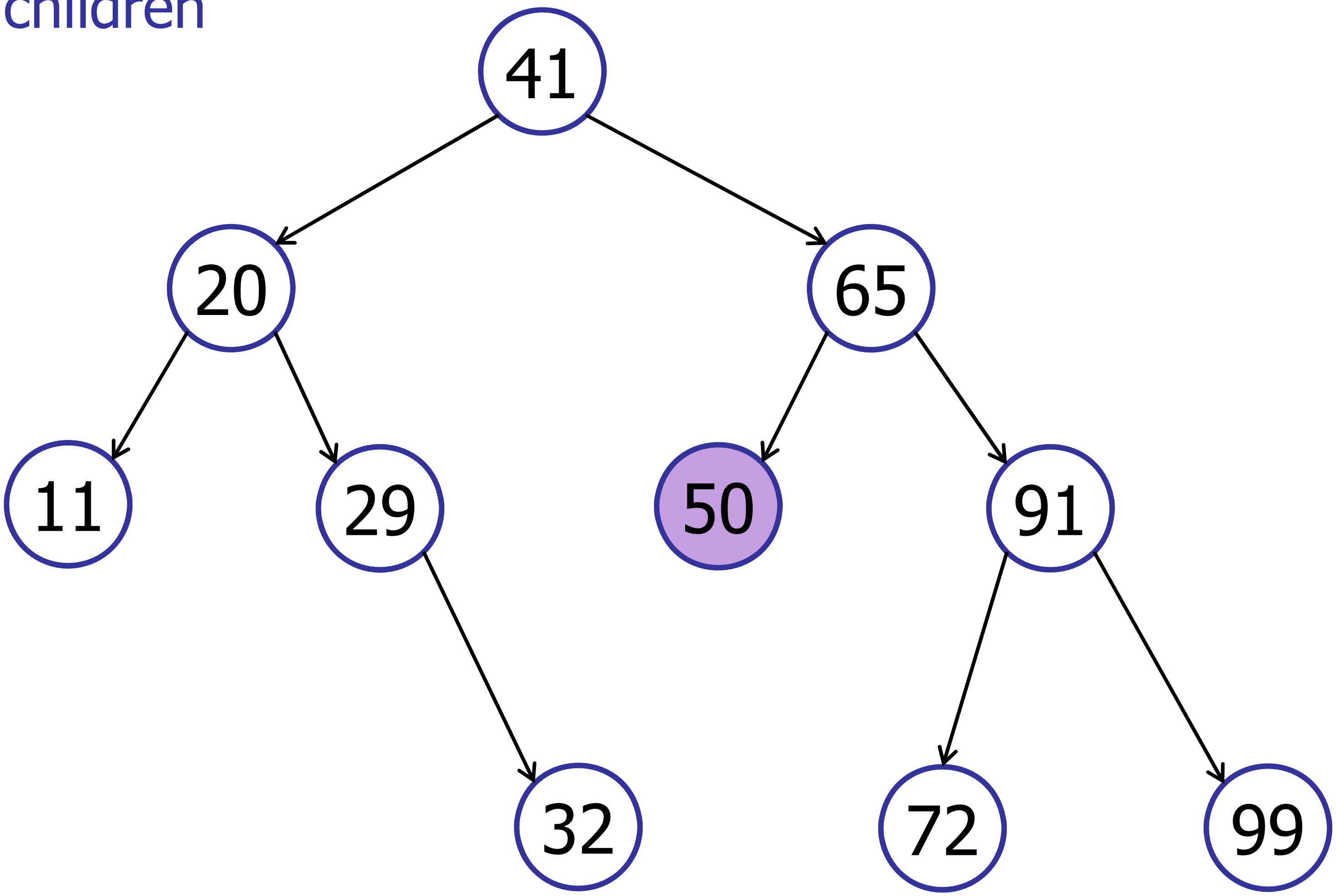
1. No children
2. 1 child
3. 2 children



Binary Search Tree

delete(50)

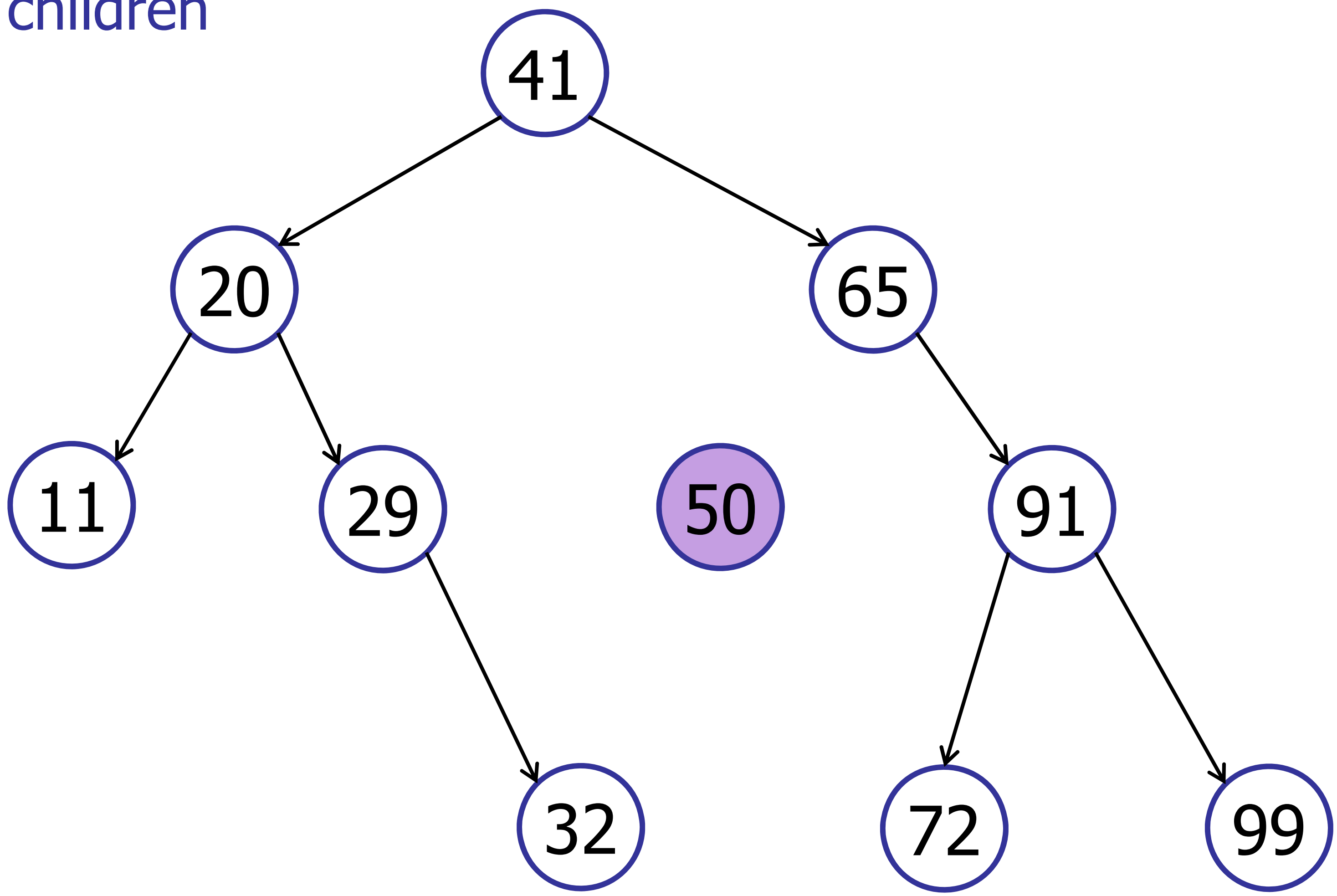
Case 1: No children



Binary Search Tree

delete(50)

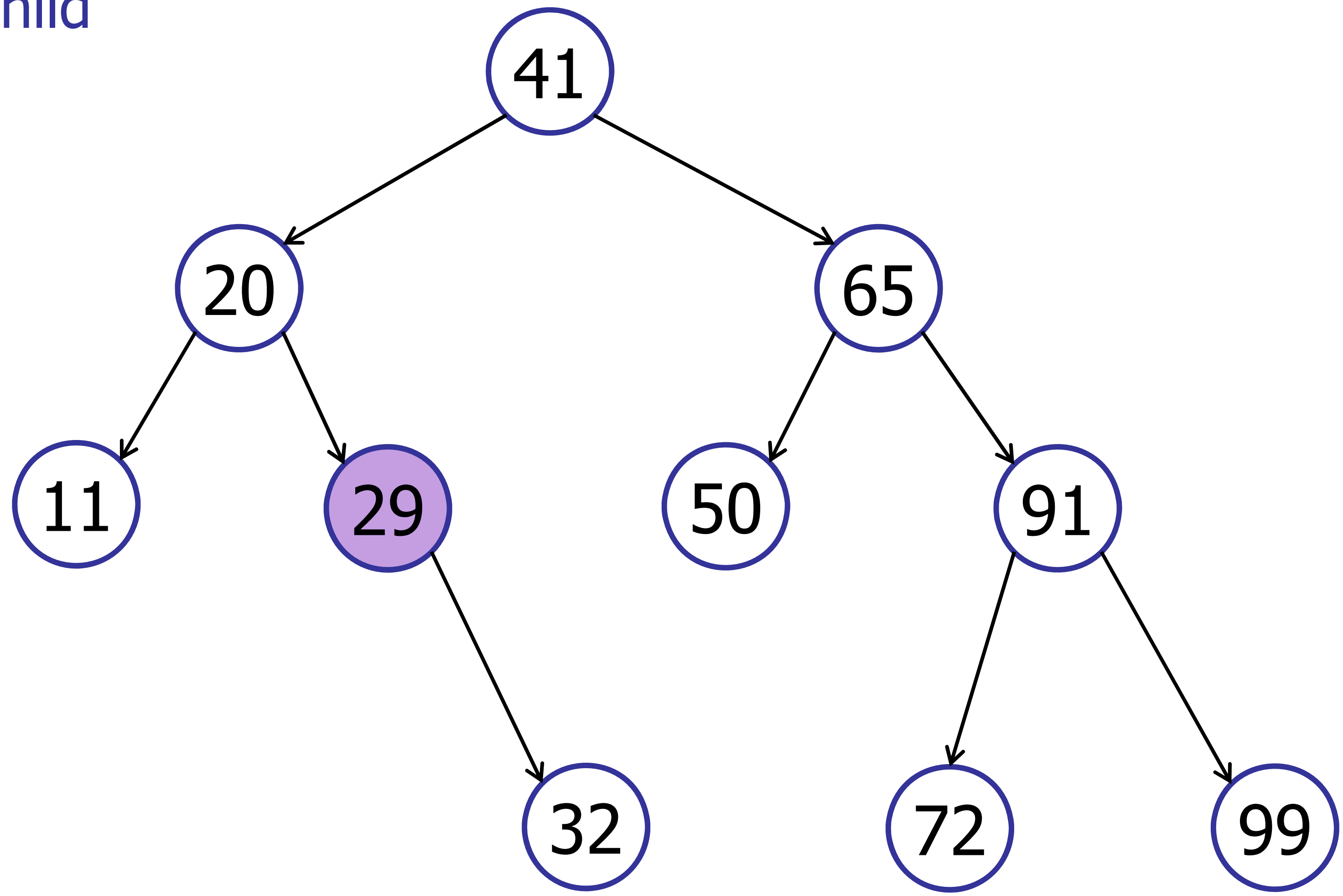
Case 1: No children



Binary Search Tree

delete(29)

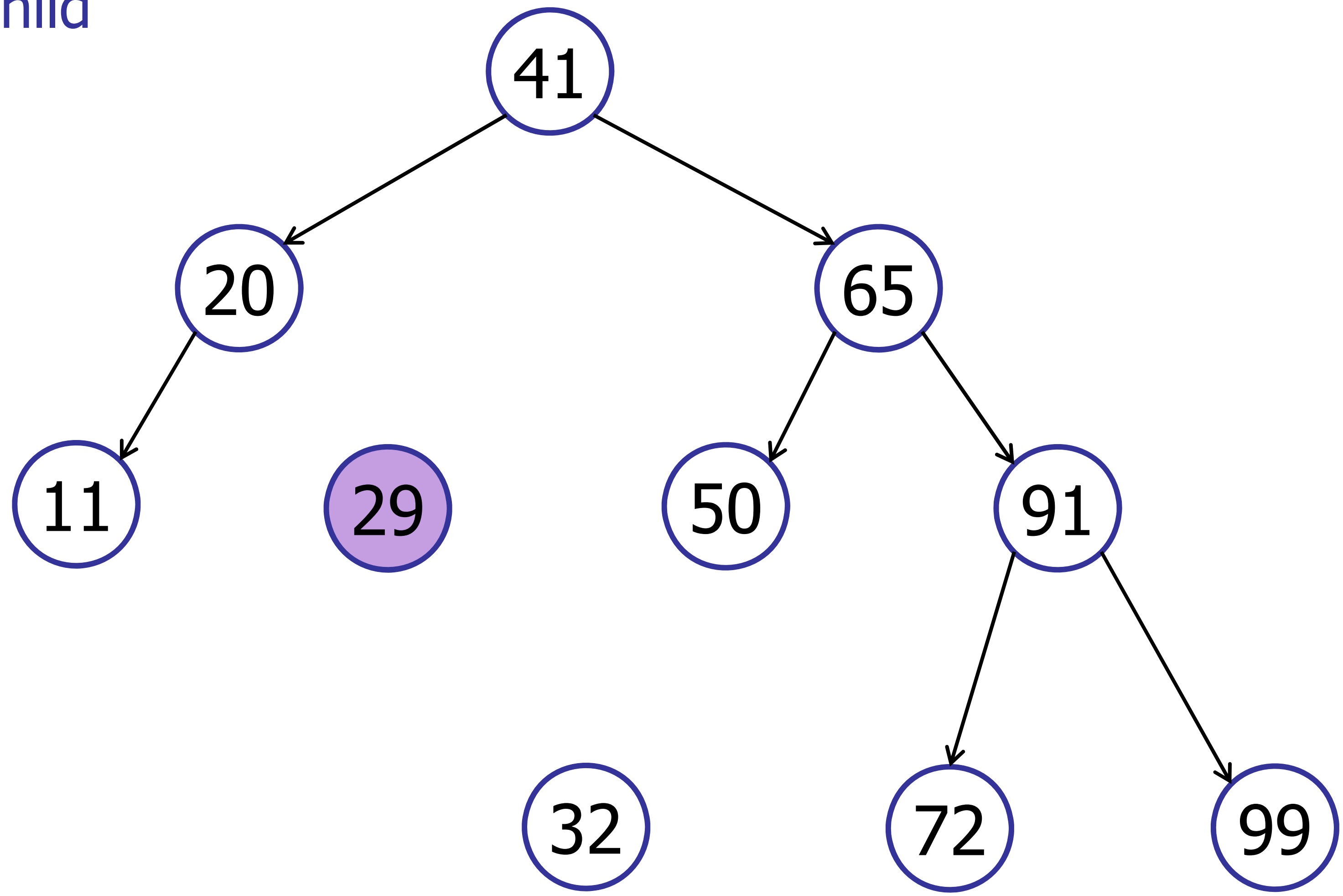
Case 2: 1 child



Binary Search Tree

delete(29)

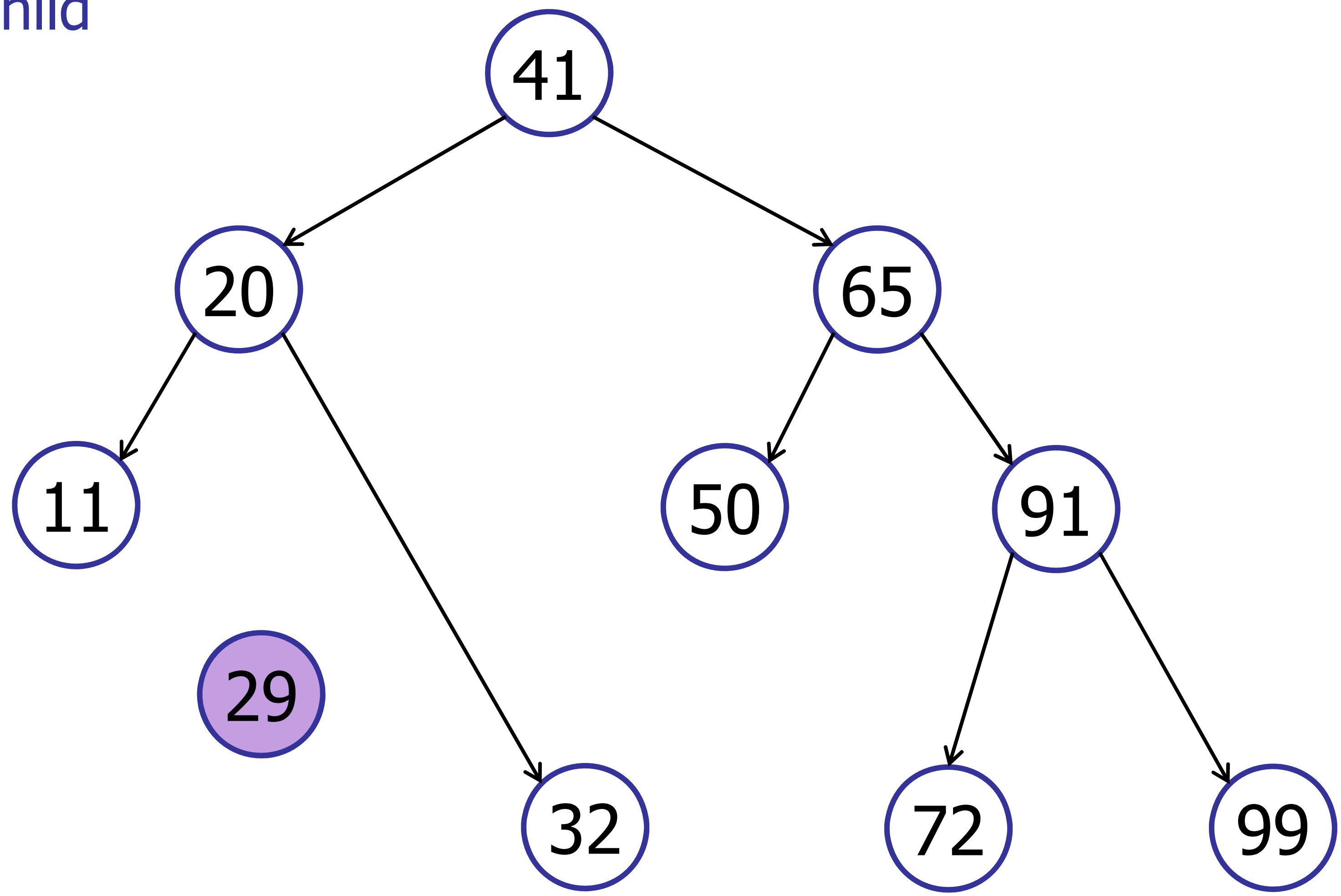
Case 2: 1 child



Binary Search Tree

delete(29)

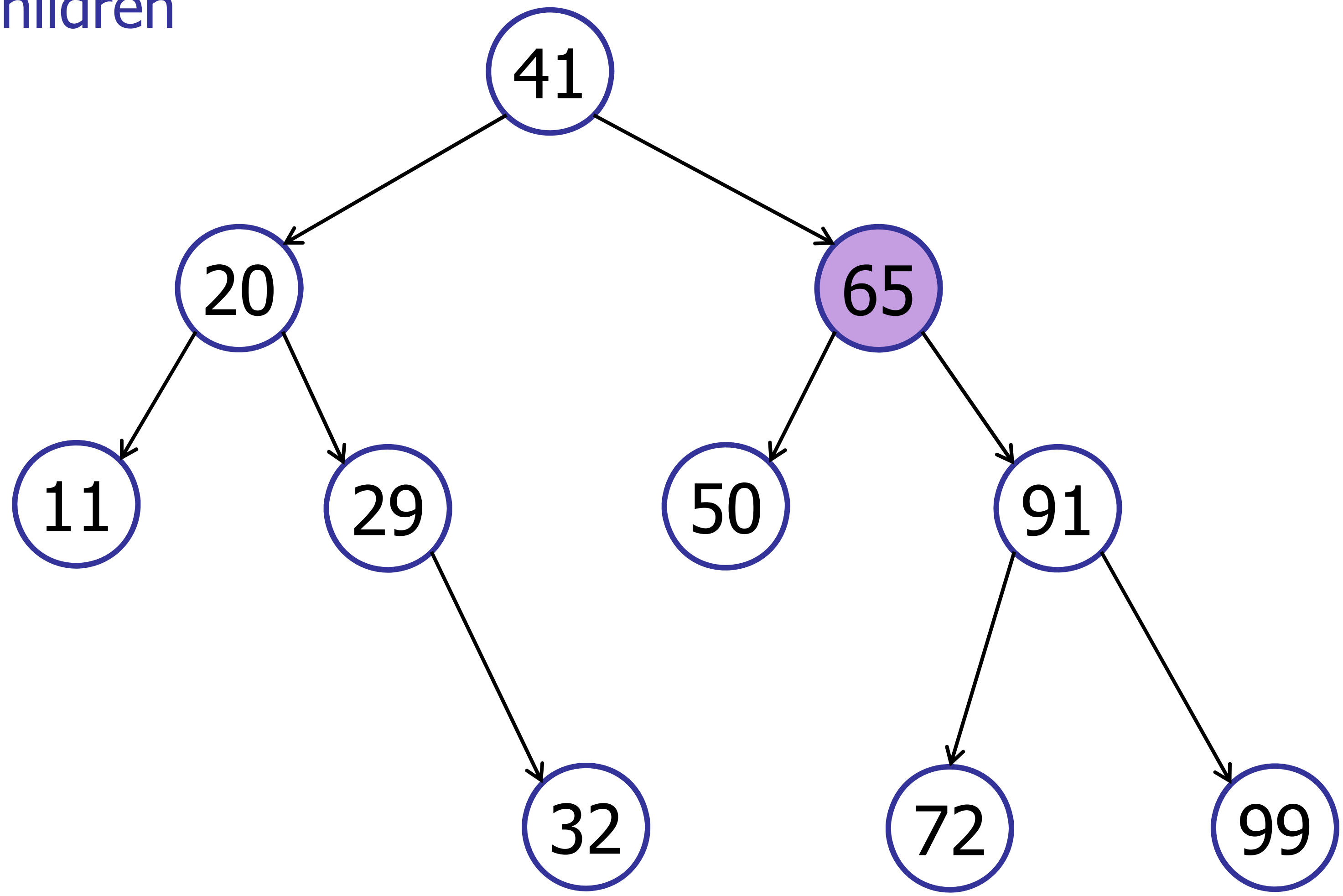
Case 2: 1 child



Binary Search Tree

delete(65)

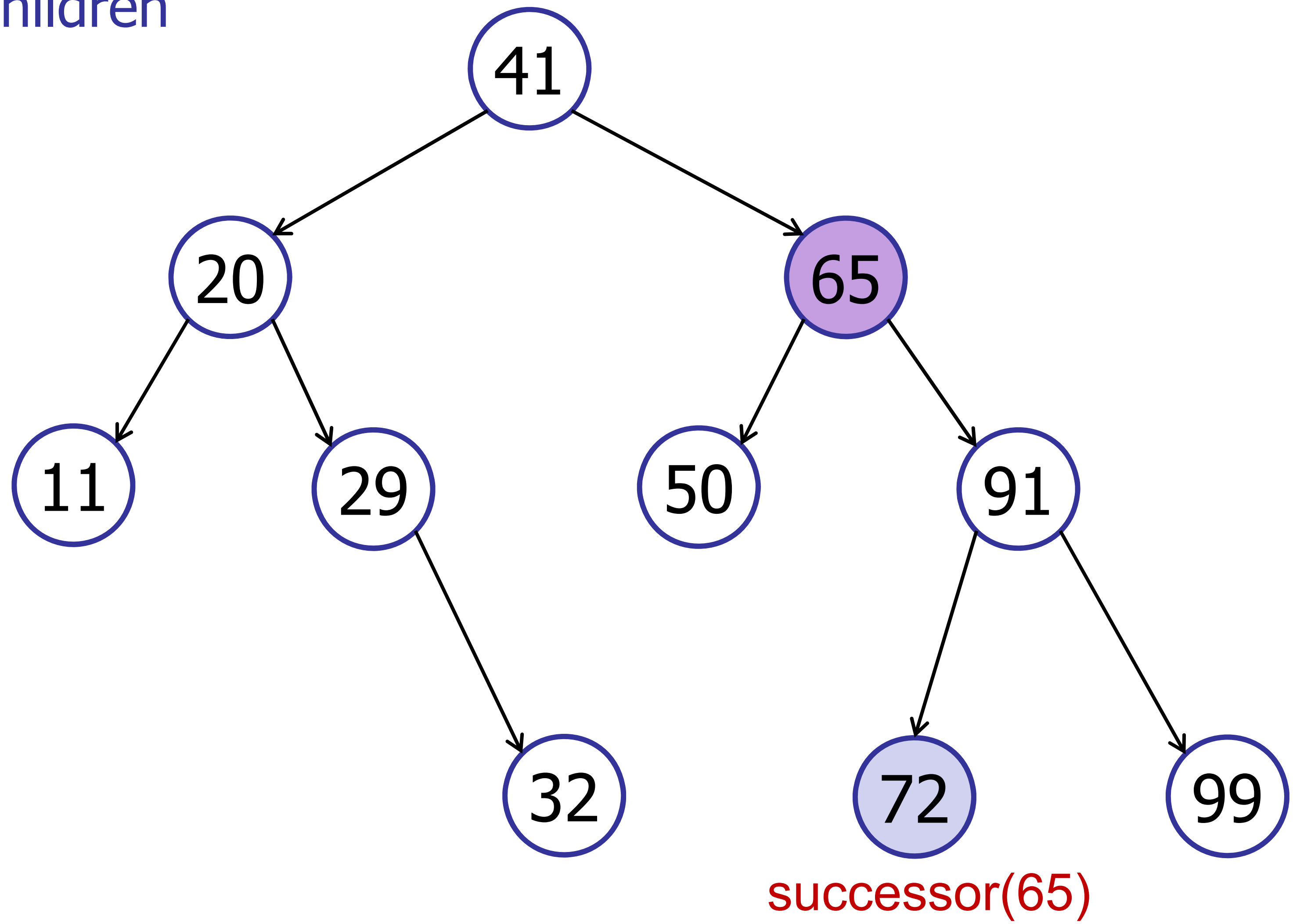
Case 3: 2 children



Binary Search Tree

delete(65)

Case 3: 2 children

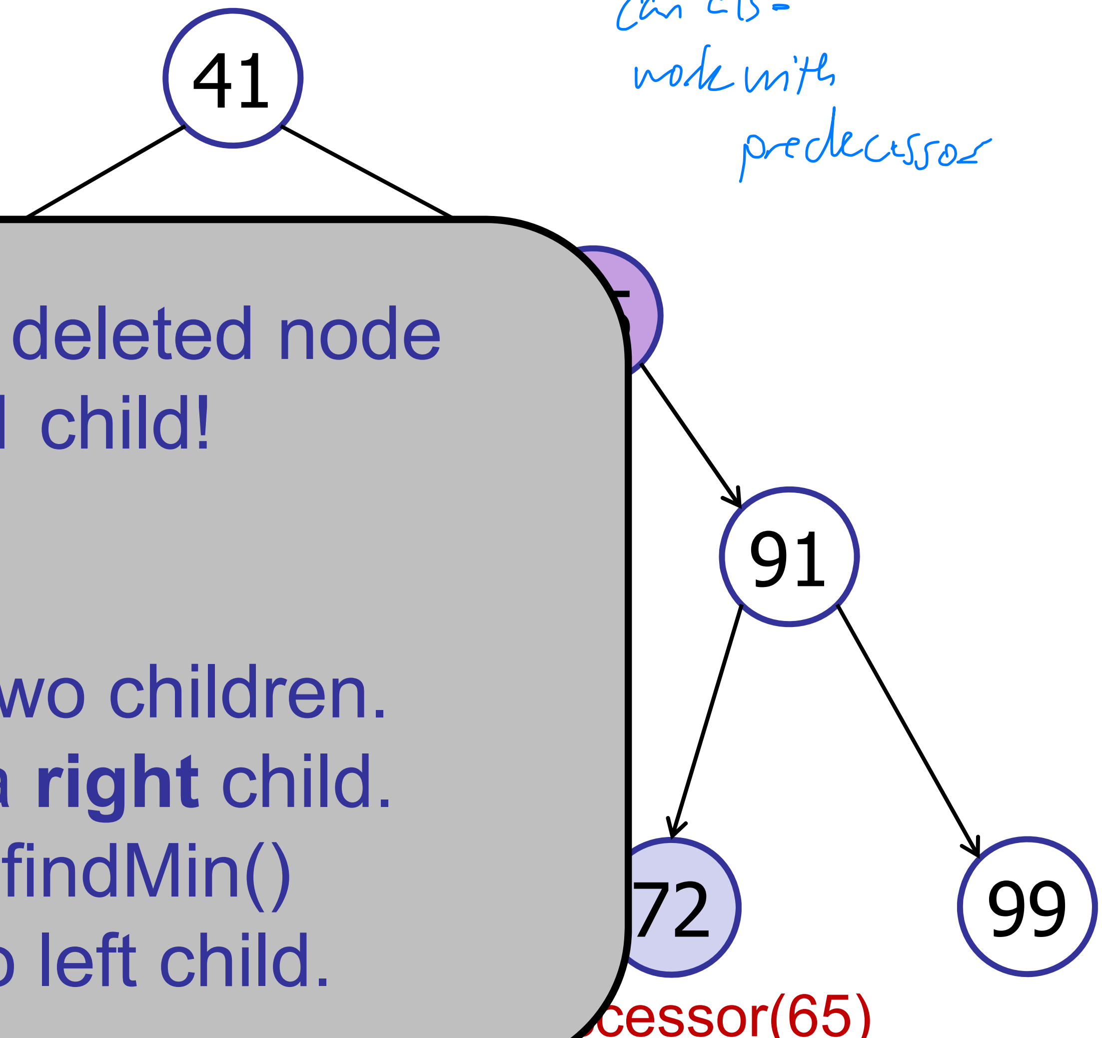


Binary Search Tree

delete(65)

Case 3: 2 children

can also
node with
predecessor



Claim: successor of deleted node has at most 1 child!

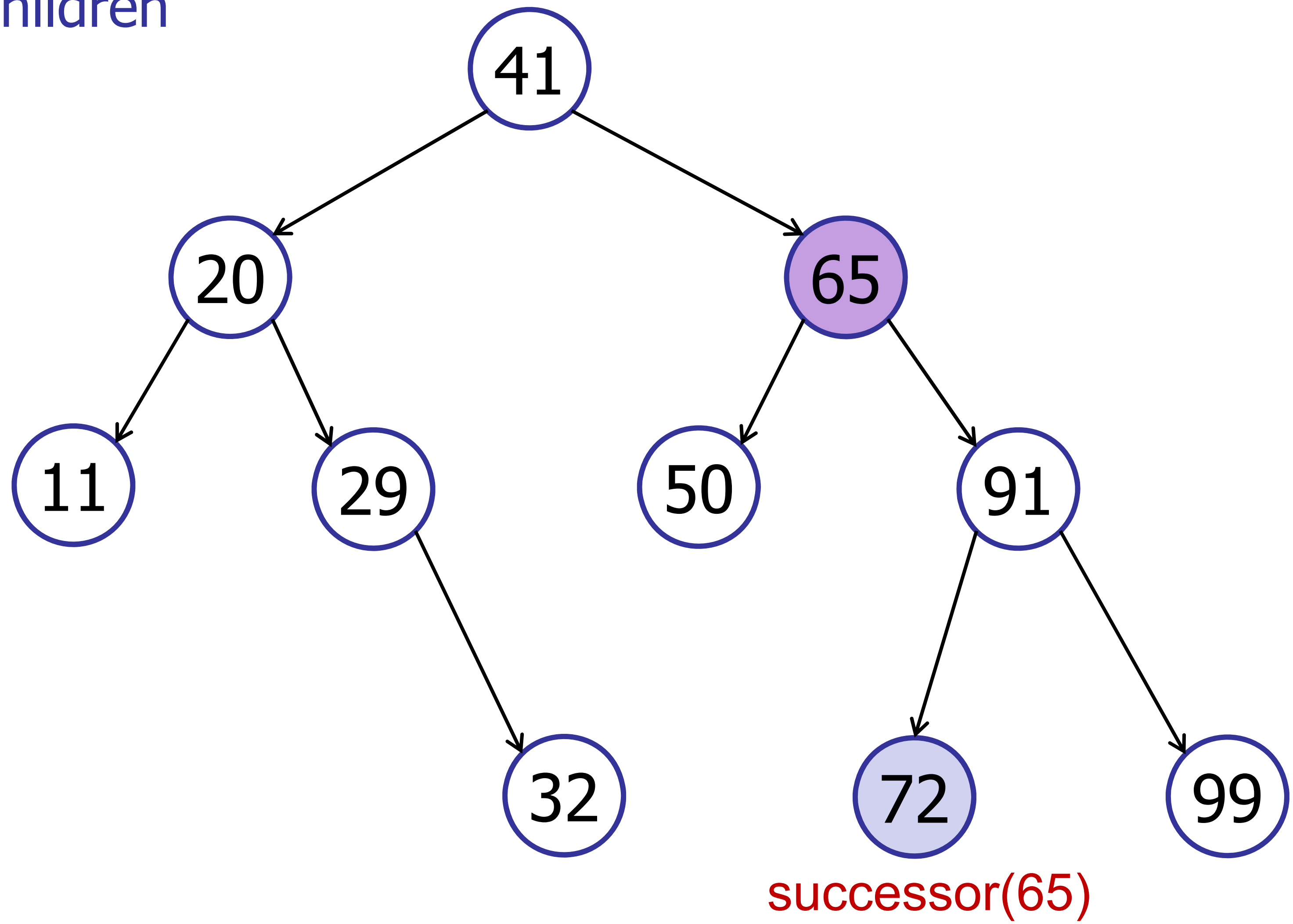
Proof:

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

Binary Search Tree

delete(65)

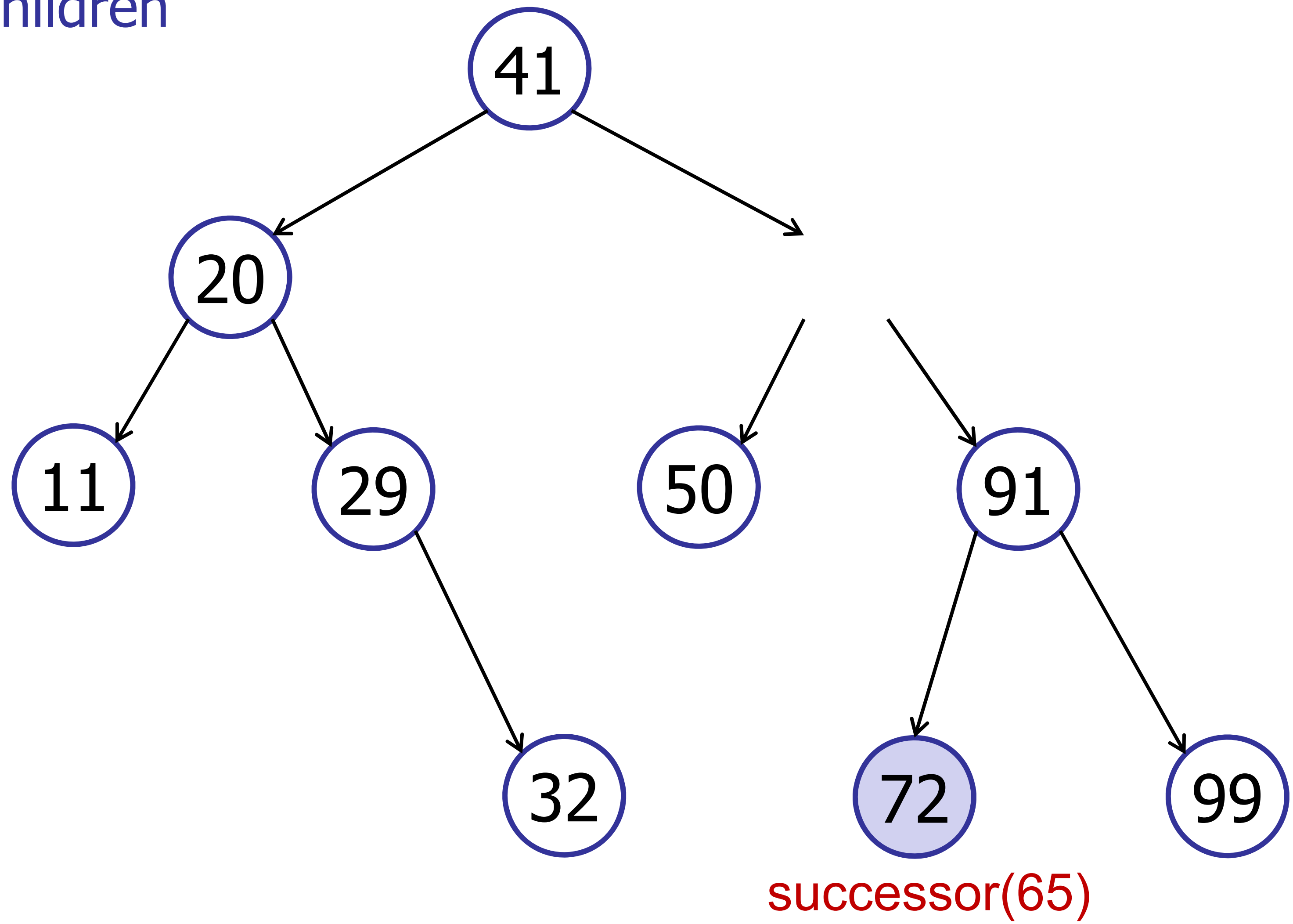
Case 3: 2 children



Binary Search Tree

delete(65)

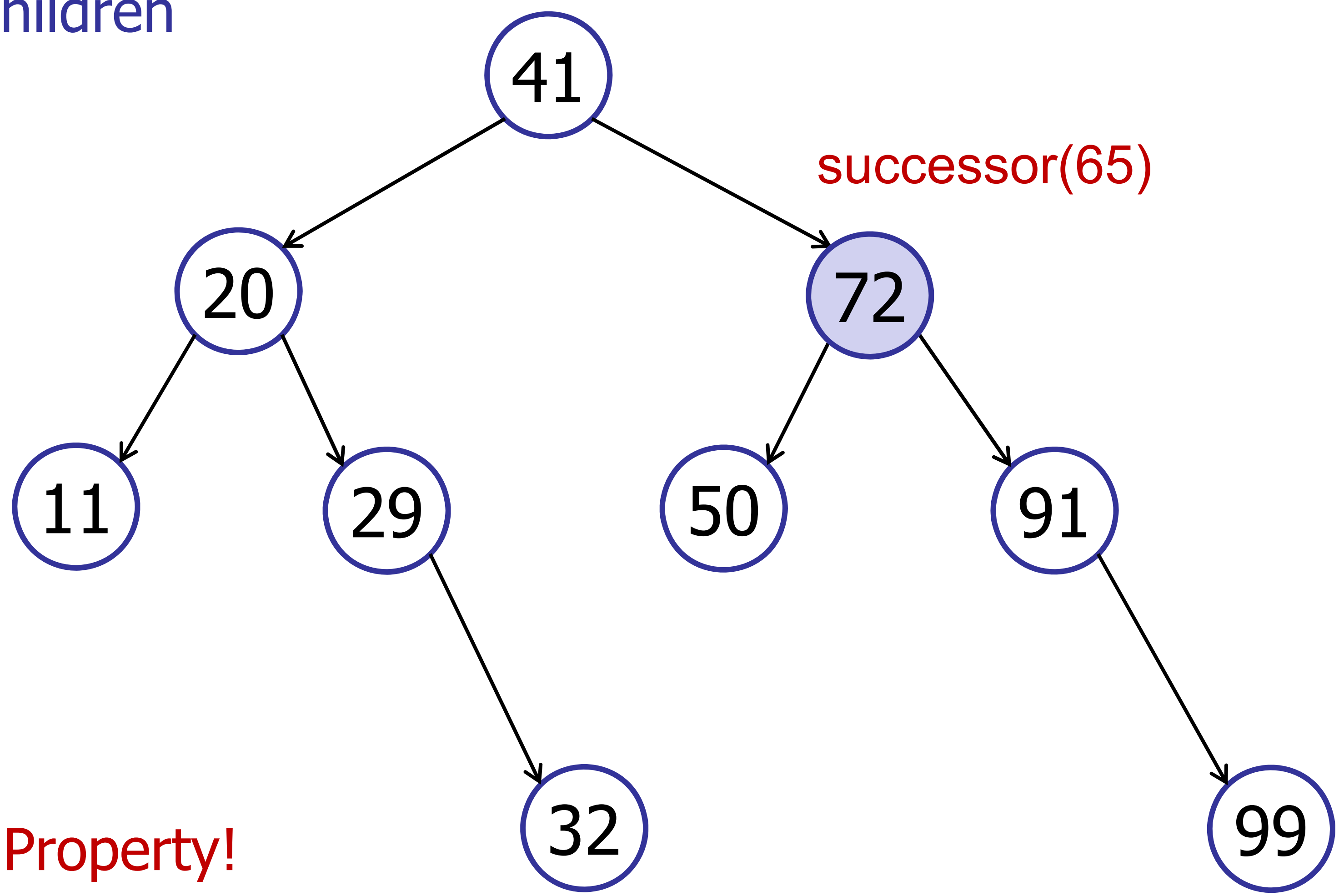
Case 3: 2 children



Binary Search Tree

delete(65)

Case 3: 2 children



Check BST Property!

Binary Search Tree

delete(v)

Running time: O(height)

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

delete(v)

Three cases:

1. No children:
   - remove v

2. 1 child:
   - remove v
   - connect child(v) to parent(v)

3. 2 children
   - Swap v with x = successor(v)
   - delete(v)
     - (which is in the original position of the successor)

Will this cause more calls for the function delete()?

Binary Search Tree

Modifying Operations

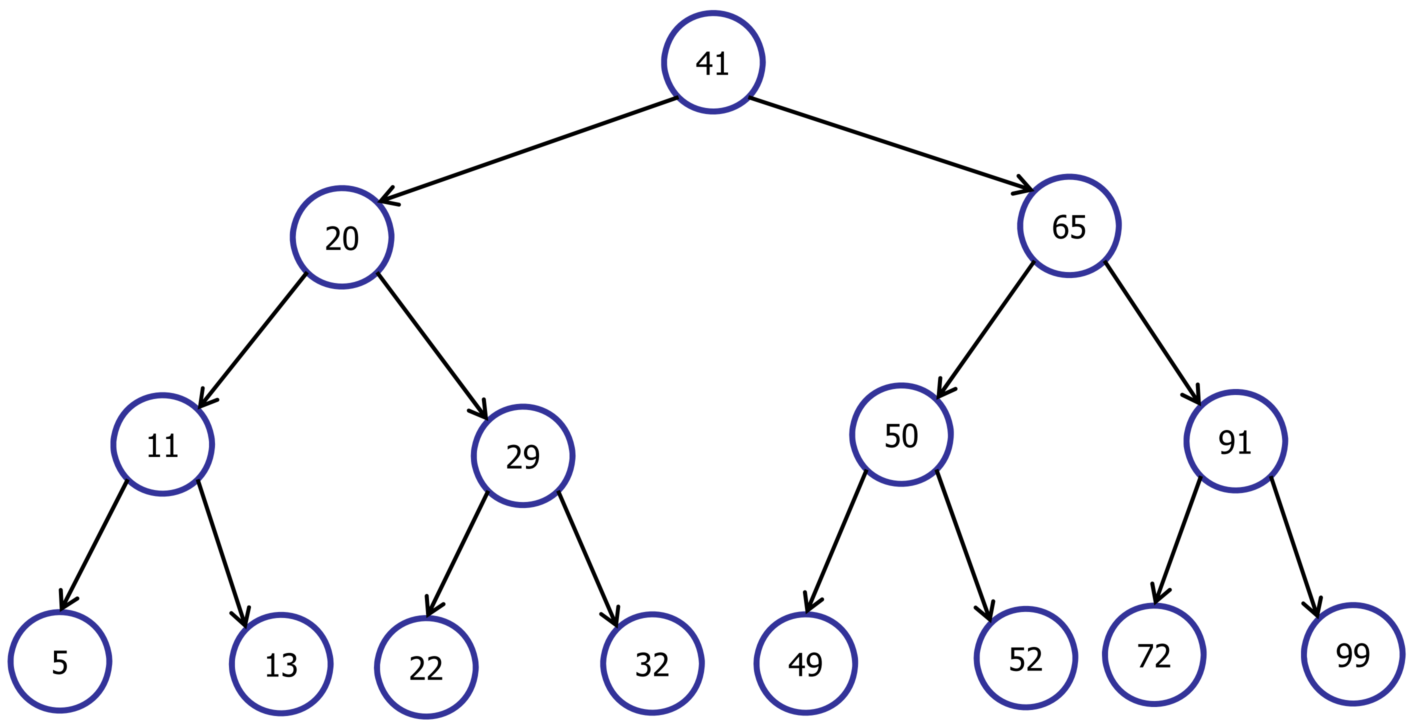
- insert: O(h)
- delete: O(h)

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



What is the largest possible height h?

1. $\theta(1)$
2. $\theta(\log n)$
3. $\theta(\mathrm{sqrt}(n))$
4. $\theta(n)$
5. $\theta(n^2)$

ARCHIPELAGO
is open

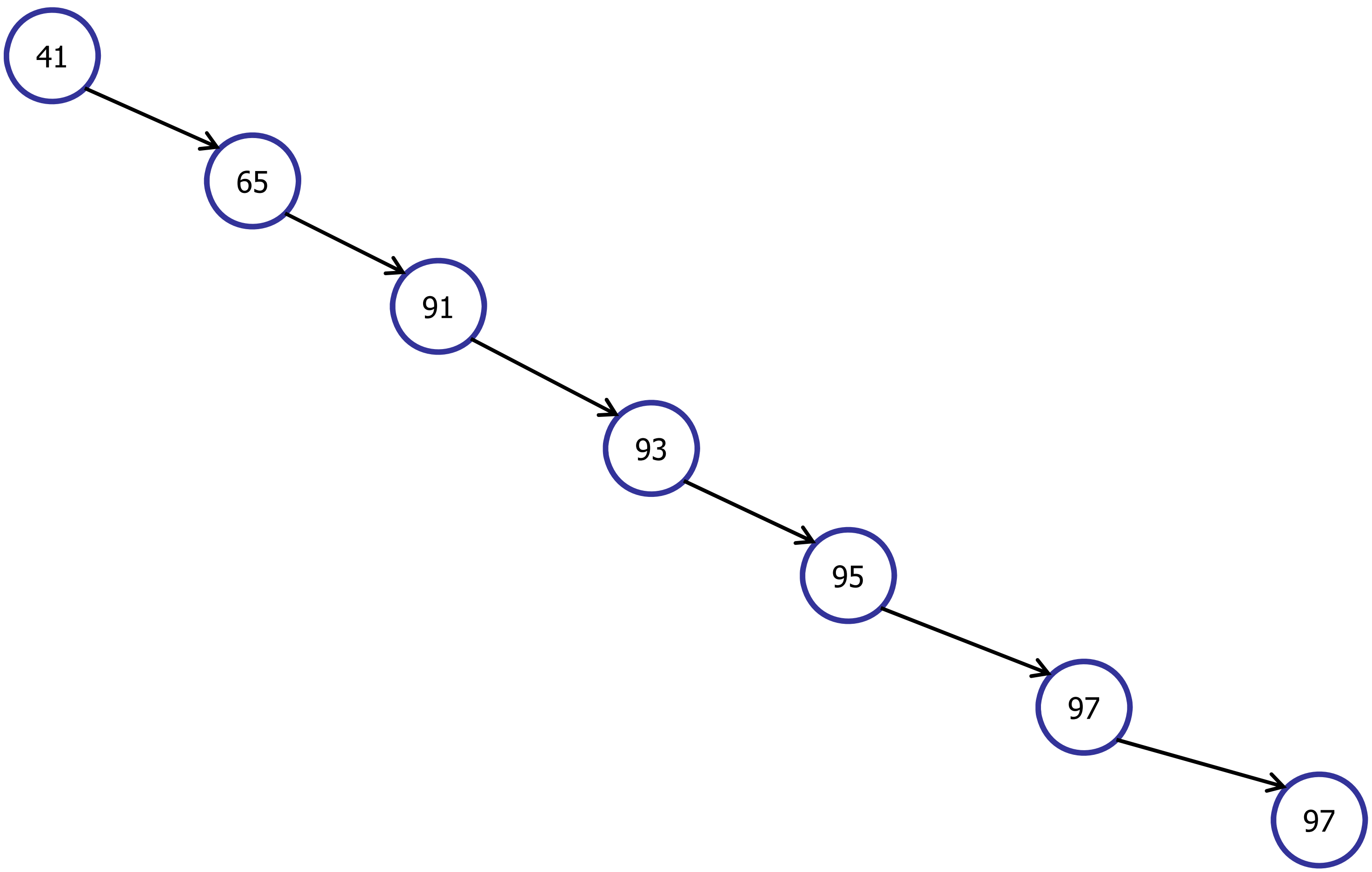
What is the largest possible height h?

1. $\theta(1)$
2. $\theta(\log n)$
3. $\theta(\mathrm{sqrt}(n))$
4. $\theta(n)$
5. $\theta(n^2)$

The Importance of Being Balanced

Operations take O(h) time

$h \leq n$



What is the smallest possible height h?

1. $\theta(1)$
2. $\theta(\log n)$
3. $\theta(\text{sqrt}(n))$
4. $\theta(n)$
5. $\theta(n^2)$

ARCHIPELAGO
is open

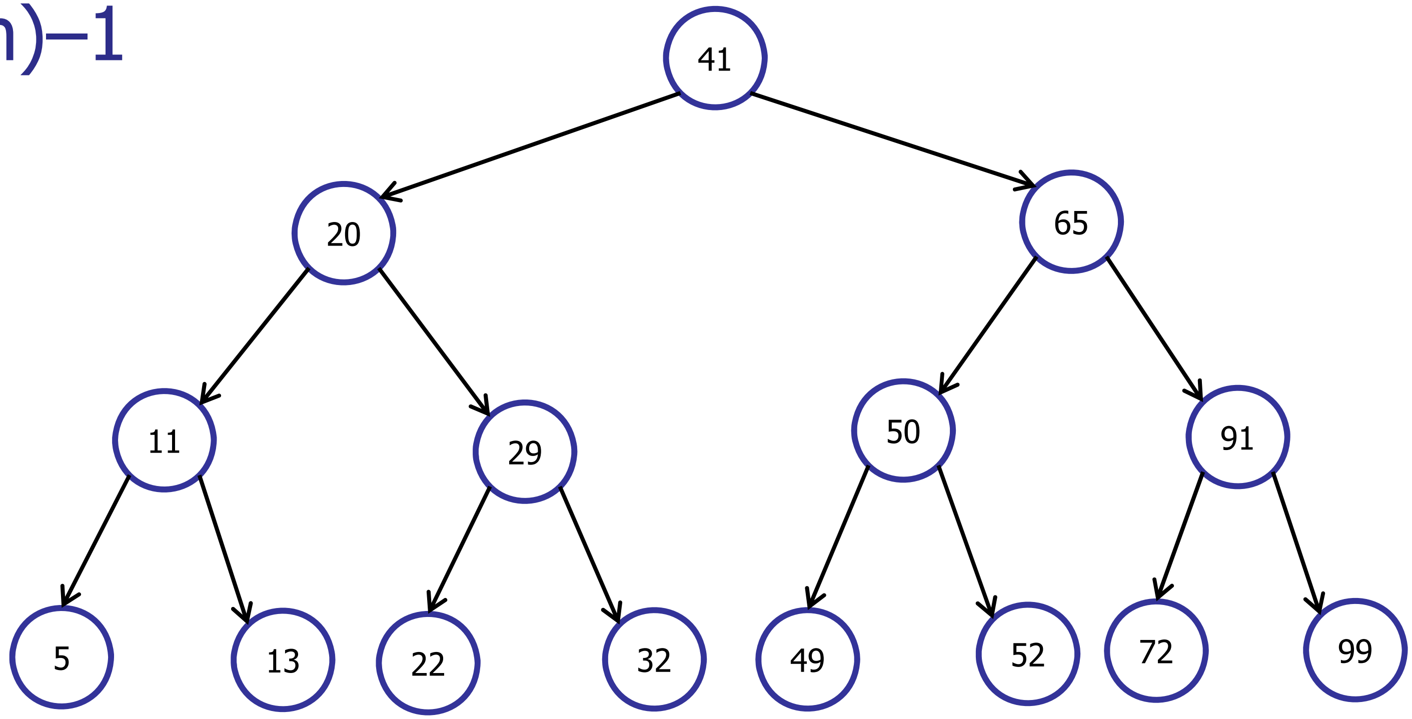
What is the smallest possible height h?

1. $\theta(1)$
2. $\theta(\log n)$
3. $\theta(\mathrm{sqrt}(n))$
4. $\theta(n)$
5. $\theta(n^2)$

The Importance of Being Balanced

Operations take O(h) time

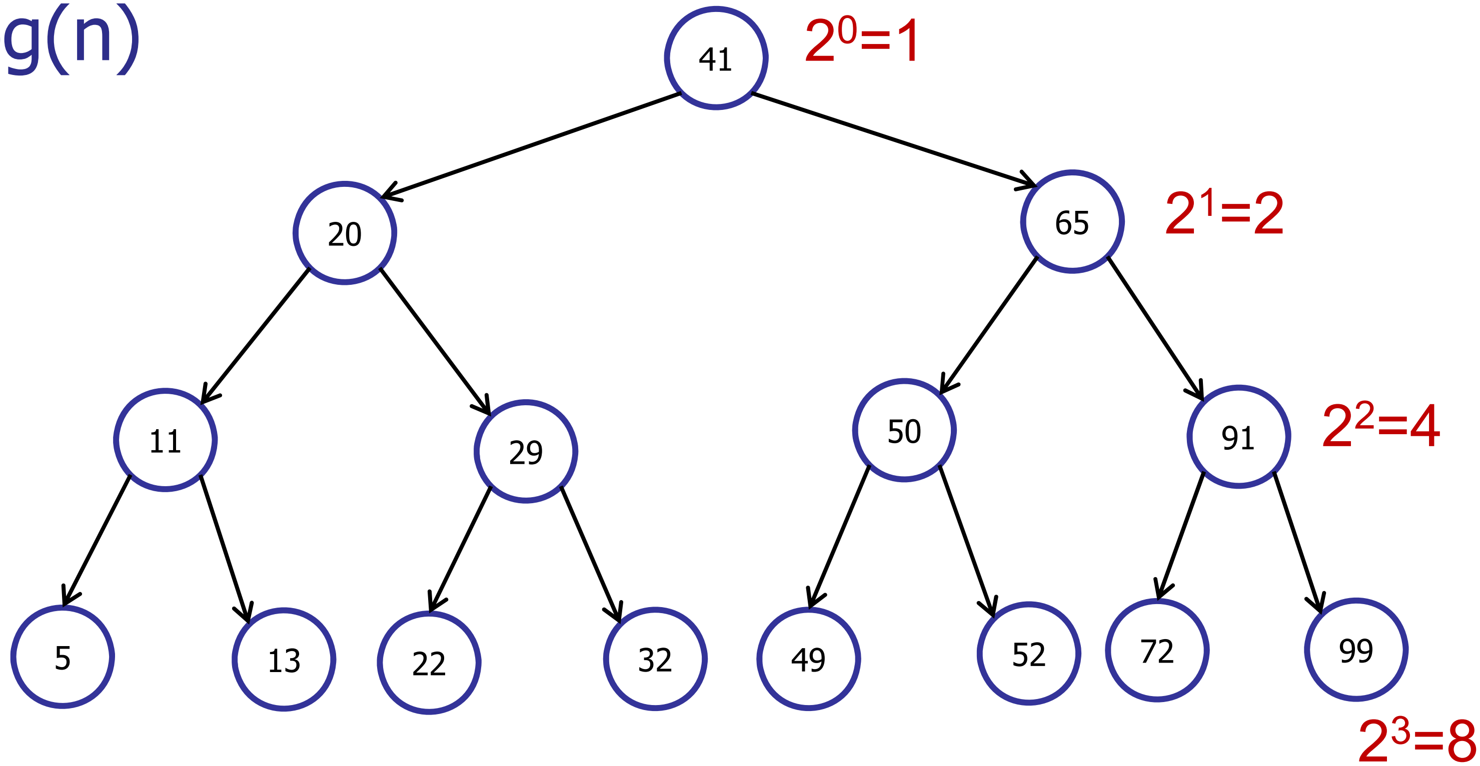
$h \geq \log(n) - 1$



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

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

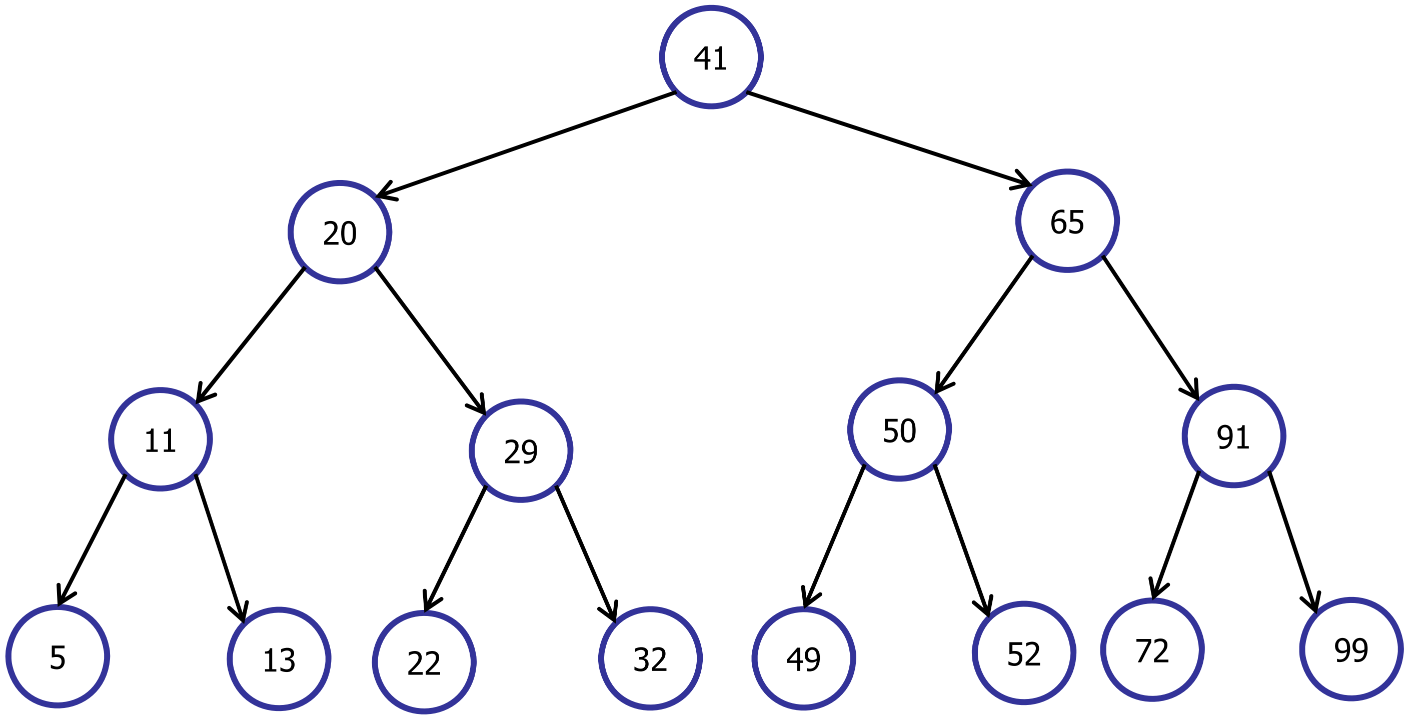
Key definition

A BST is balanced if $h = O(\log n)$

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

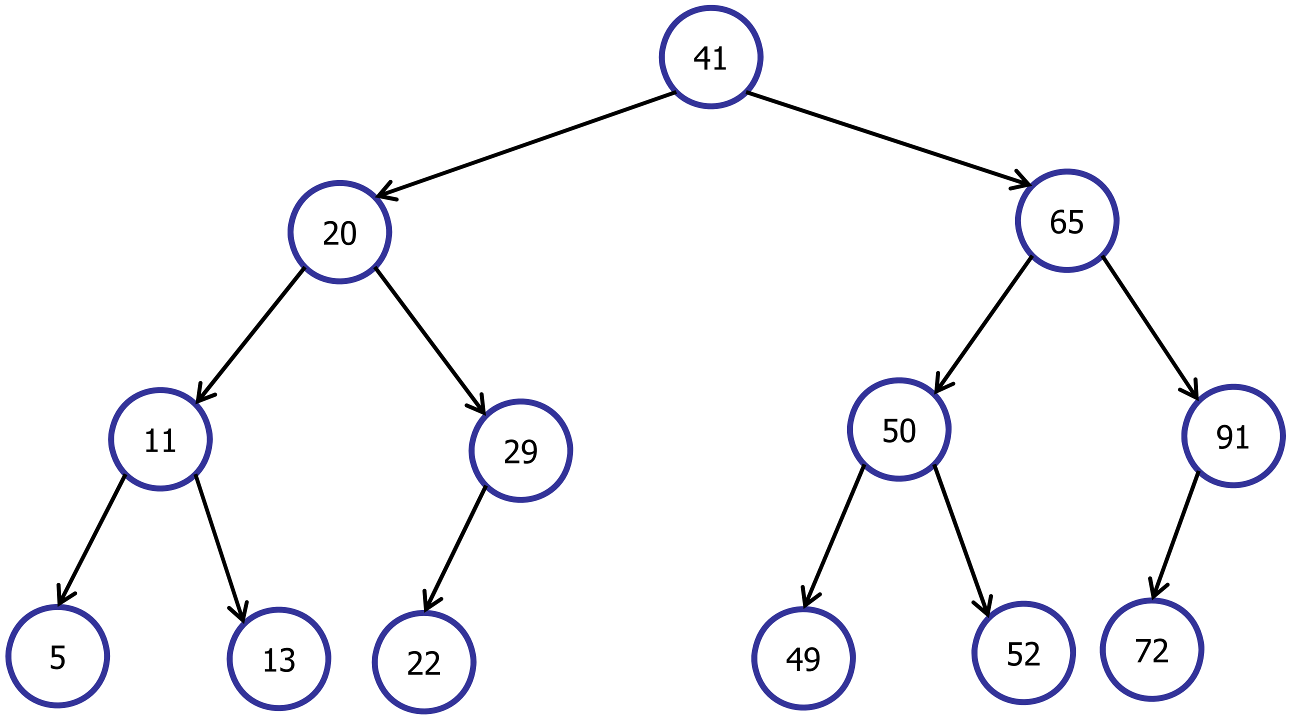
The Importance of Being Balanced

Perfectly balanced:



The Importance of Being Balanced

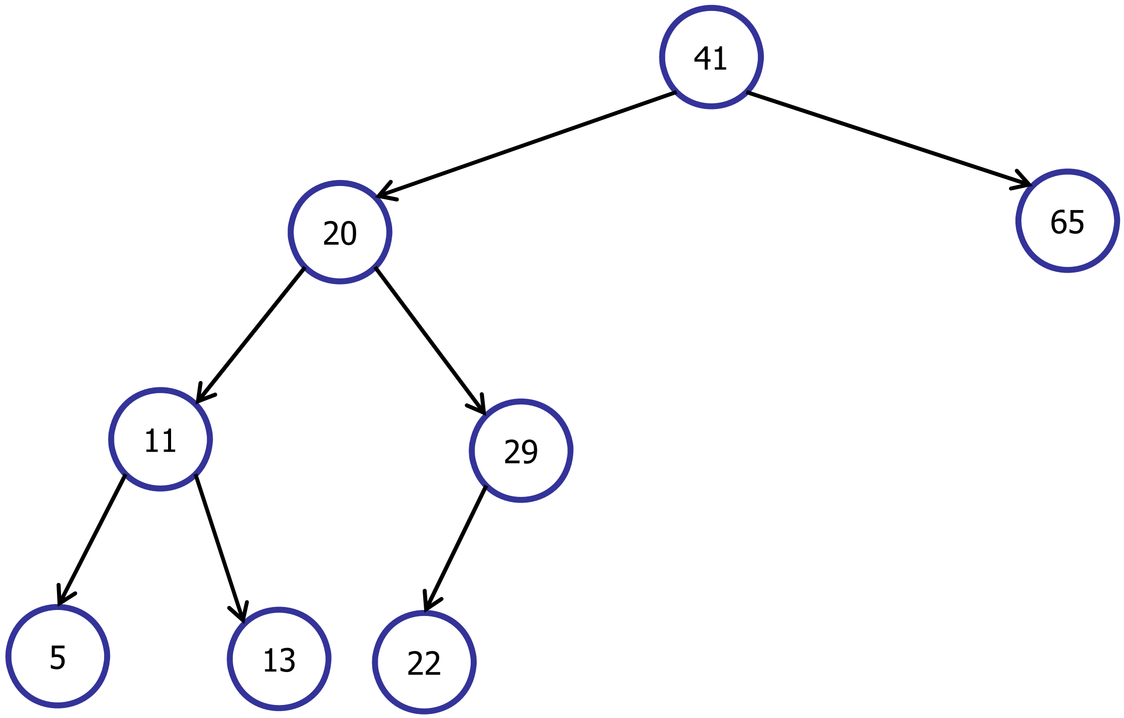
Almost perfectly balanced:



Every subtree has (almost) the same number of nodes.

The Importance of Being Balanced

Not perfectly balanced:



Left tree has 6, right tree has 1.

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)
- Scapegoat Trees (Anderson 1989)

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)
- Scapegoat Trees (Anderson 1989)

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.

Invariant

AVL Trees [Adelson-Velskii & Landis 1962]

AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

Step 1: Define Balance Condition

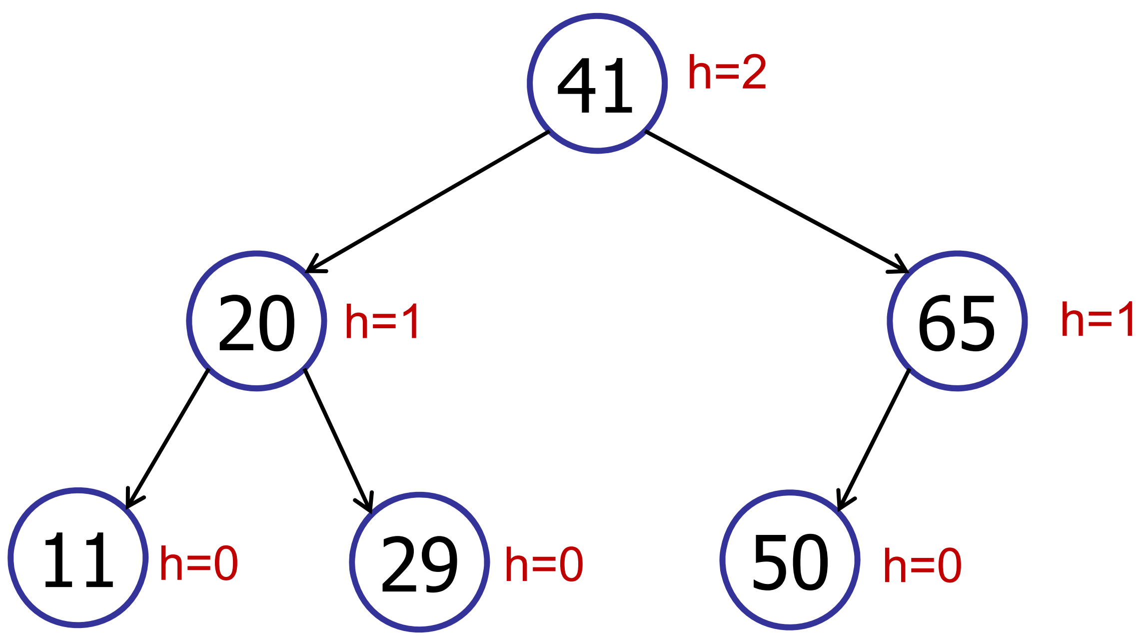
Step 2: Maintain Balance

AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

– In every node v, store height:

v.height = h(v)



AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

- In every node v, store height:

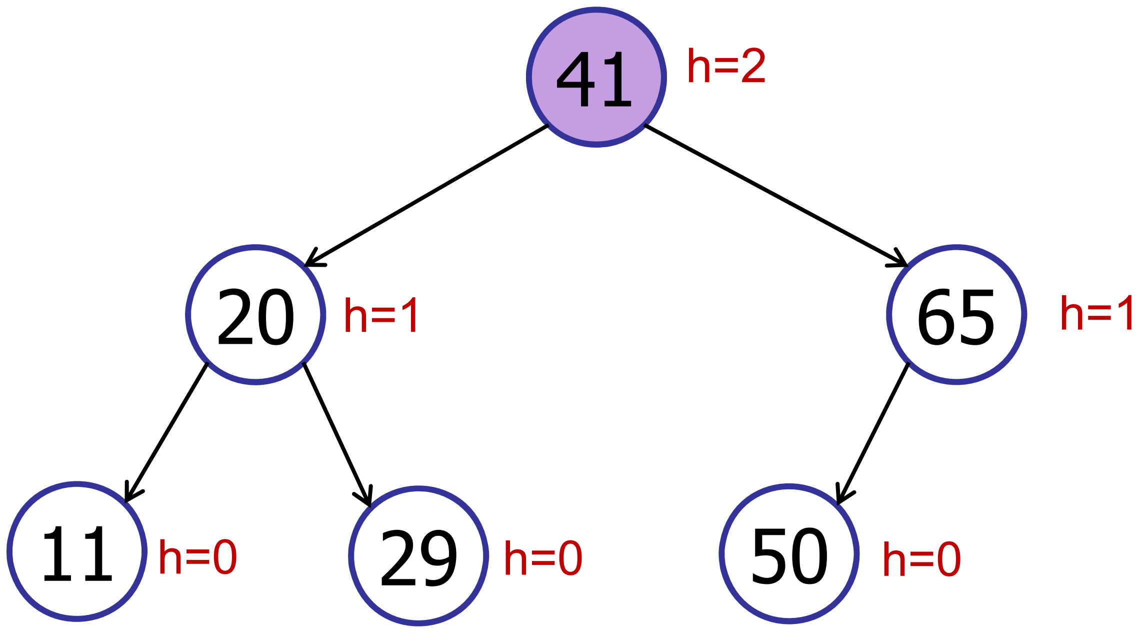
v.height = h(v)

- On insert & delete update height:

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

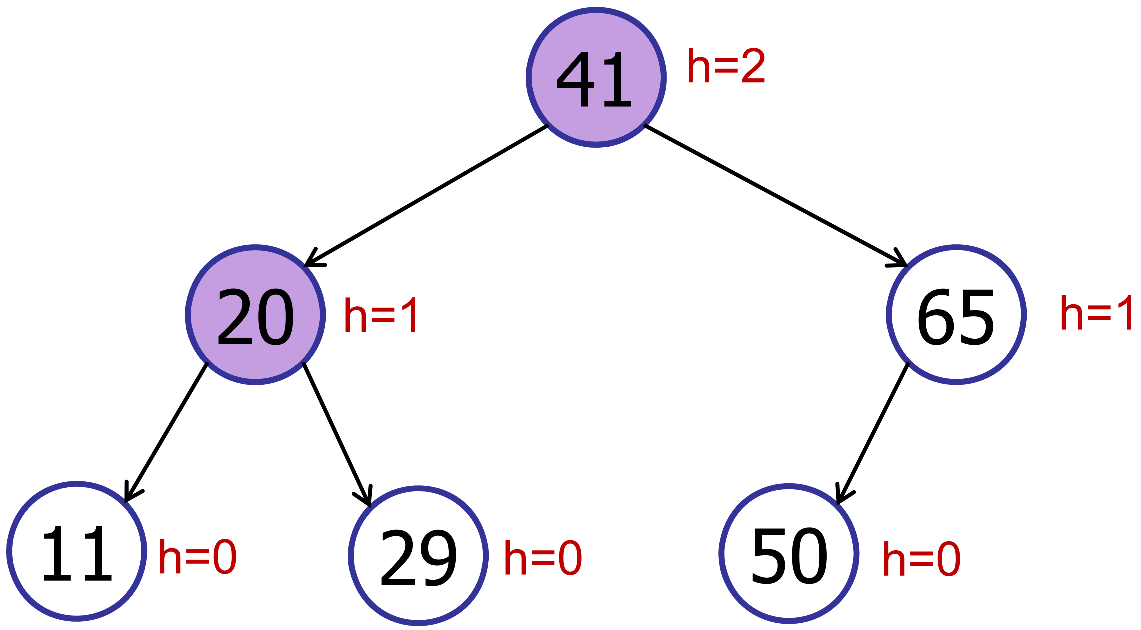
Binary Search Trees

insert(27)



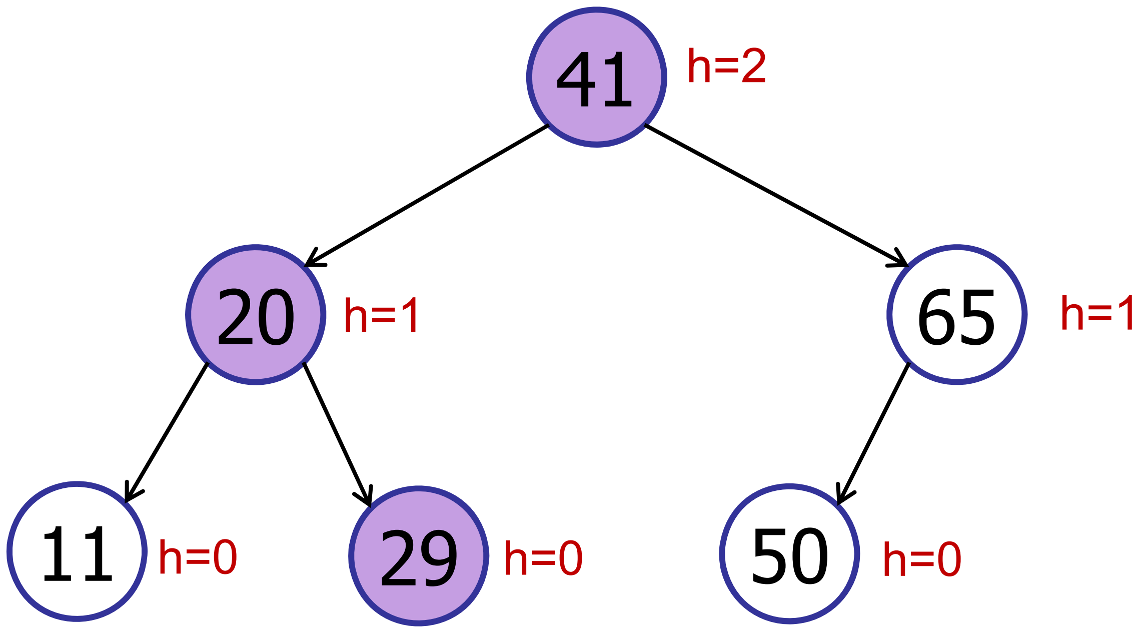
Binary Search Trees

insert(27)



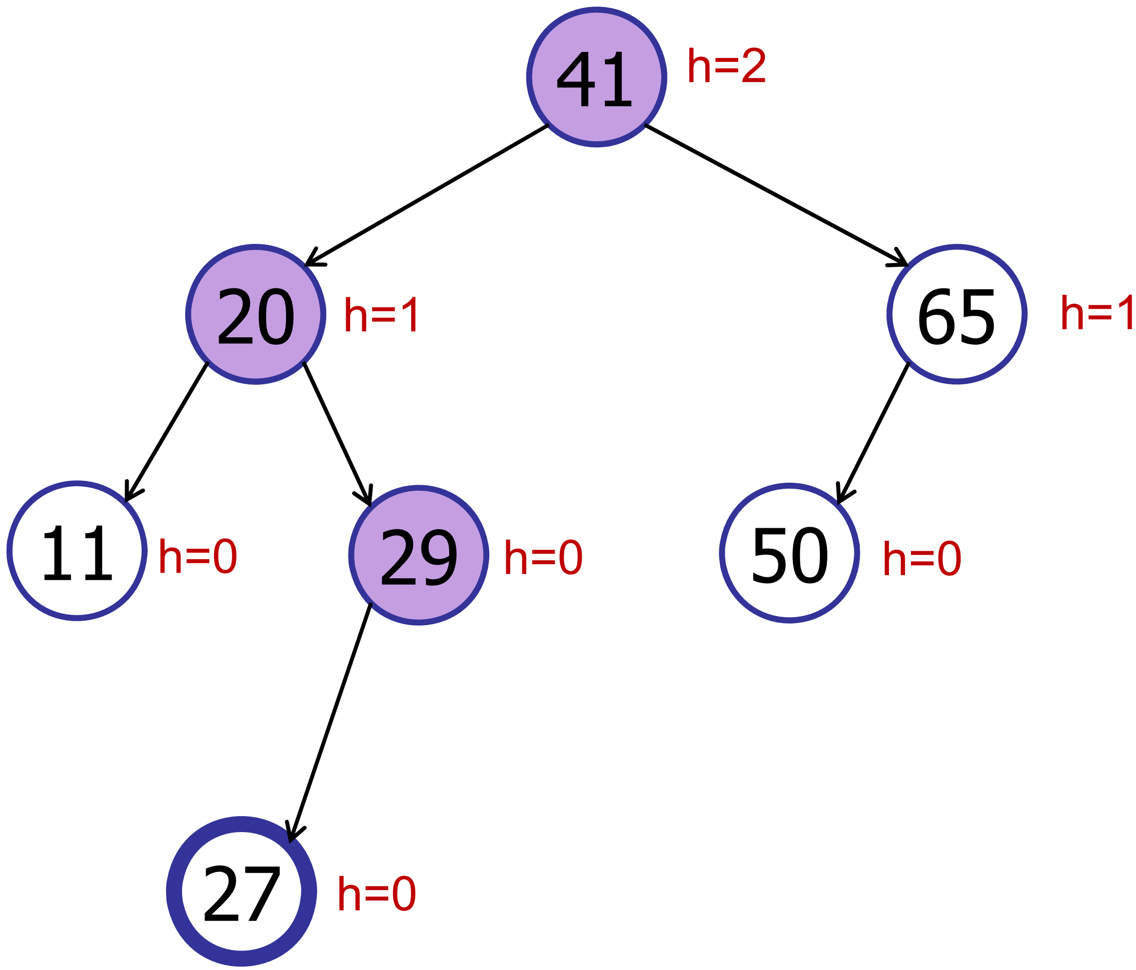
Binary Search Trees

insert(27)



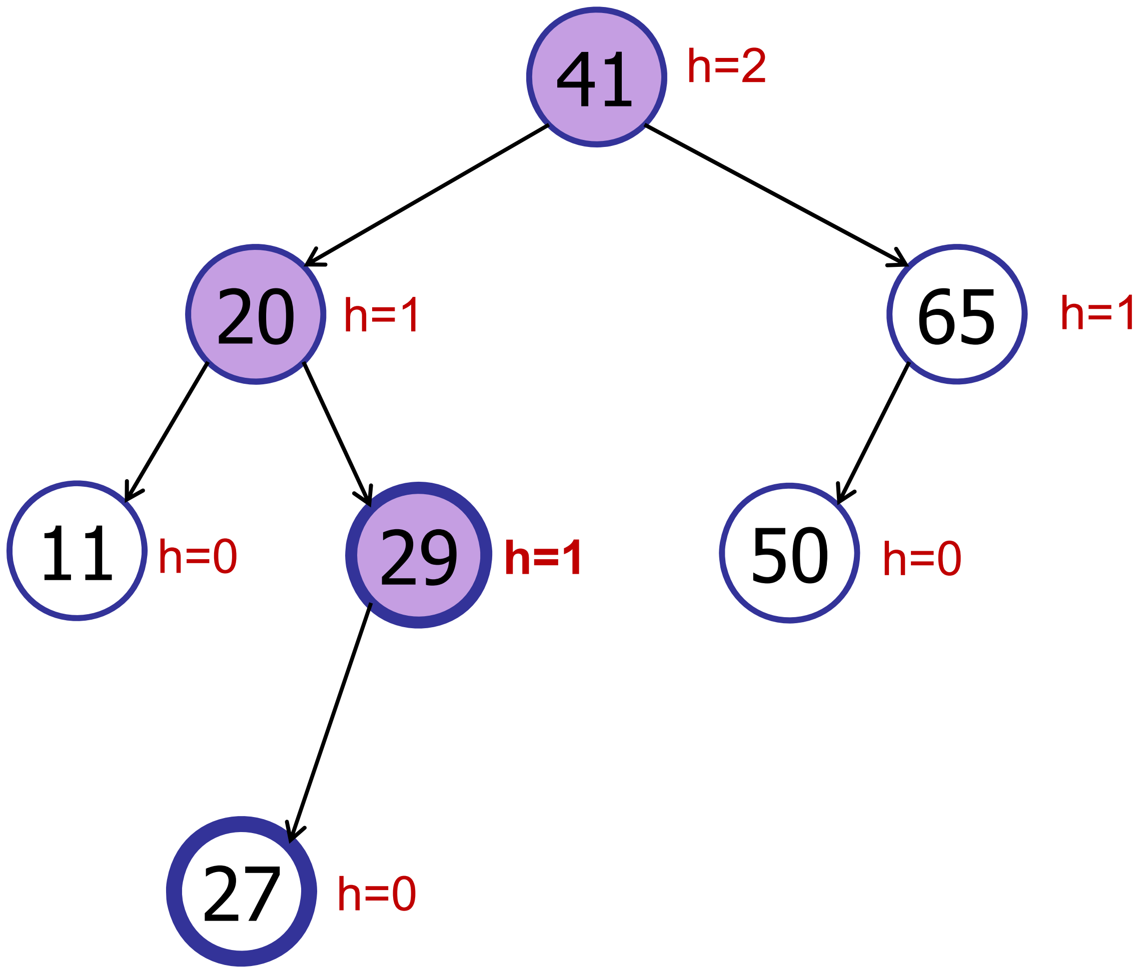
Binary Search Trees

insert(27)



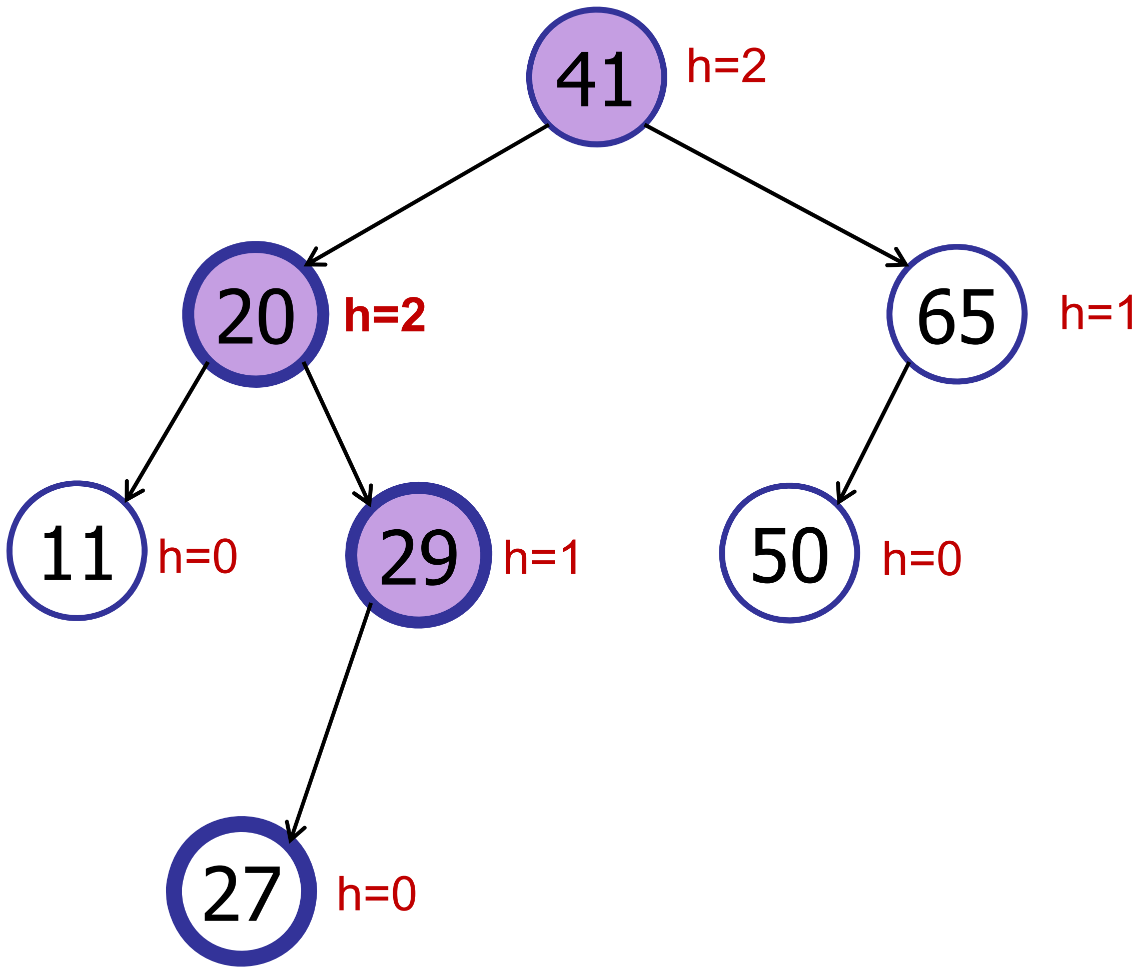
Binary Search Trees

insert(27)



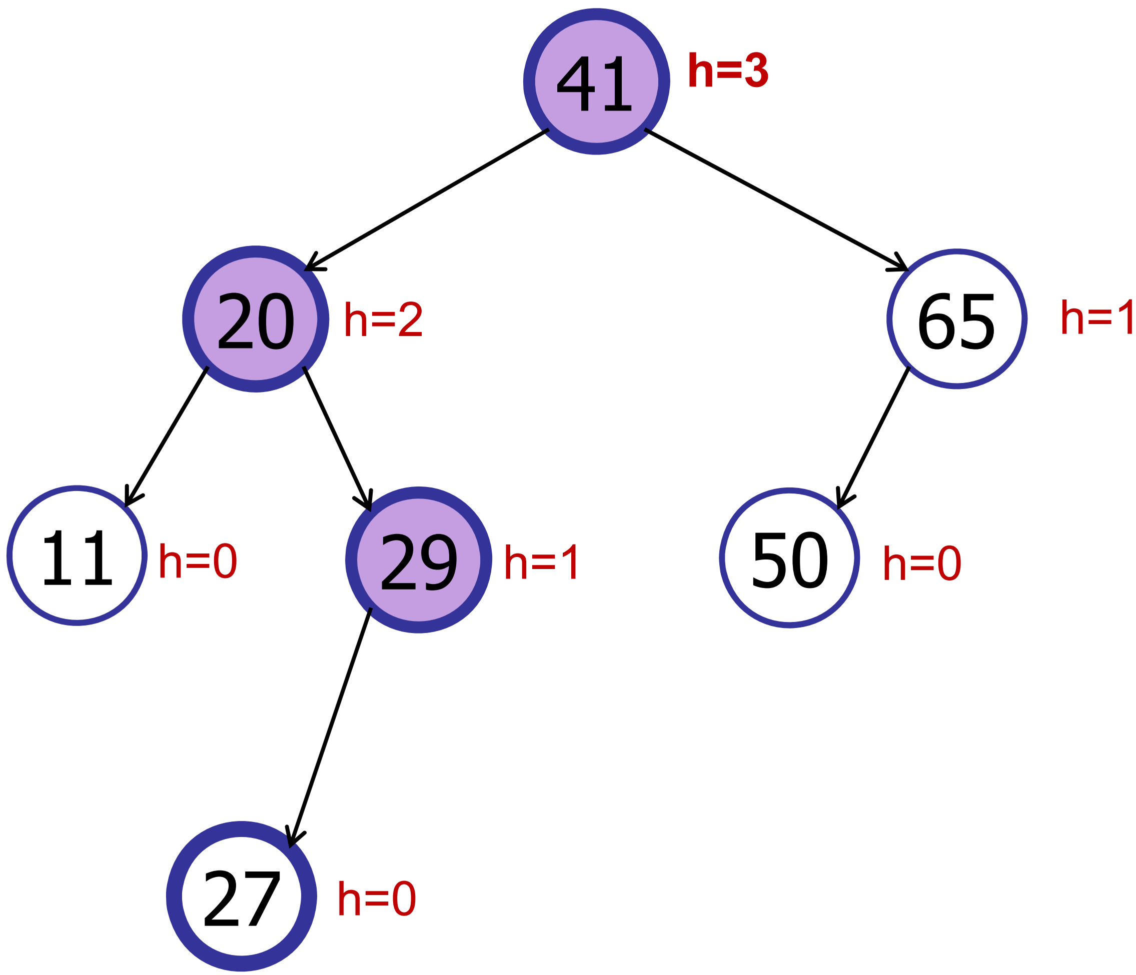
Binary Search Trees

insert(27)



Binary Search Trees

insert(27)



AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: 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 0: Augment

Step 1: Define Balance Condition

Step 2: Maintain Balance

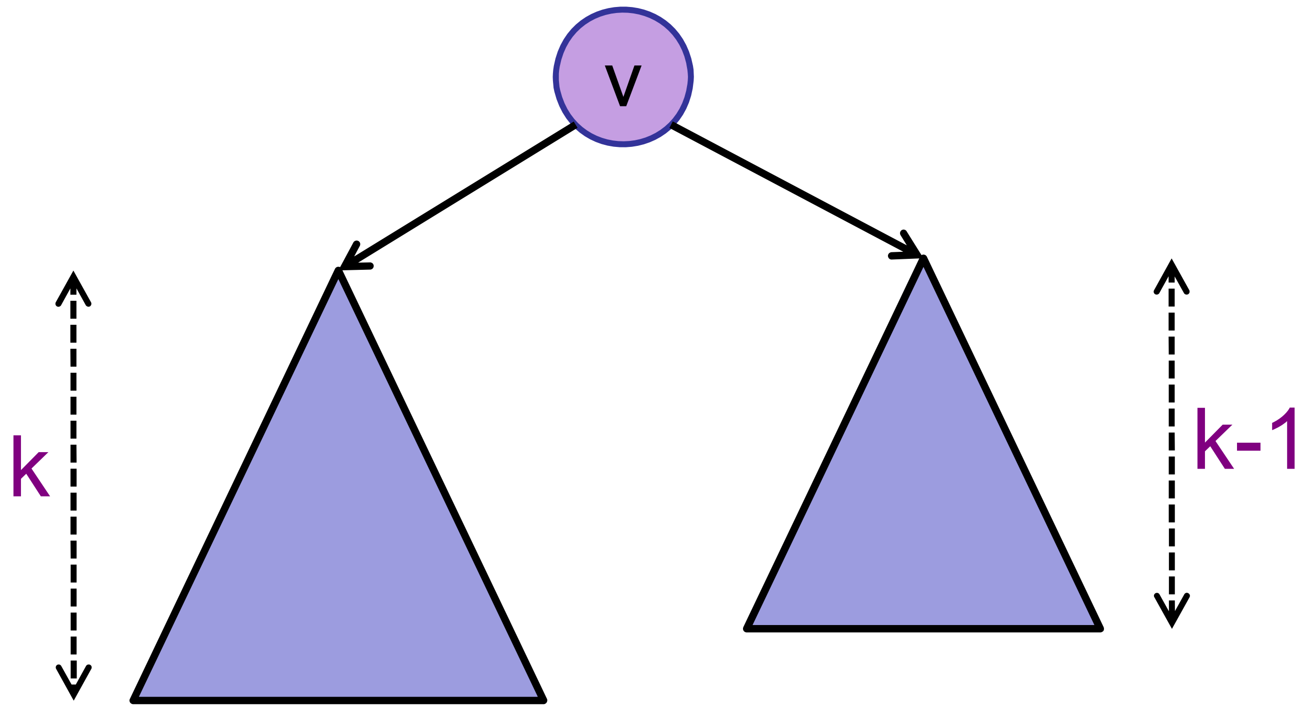
AVL Trees [Adelson-Velskii & Landis 1962]

Key definition

Step 1: Define Invariant

– A node v is **height-balanced** if:

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



AVL Trees [Adelson-Velskii & Landis 1962]

Step 1: Define Invariant

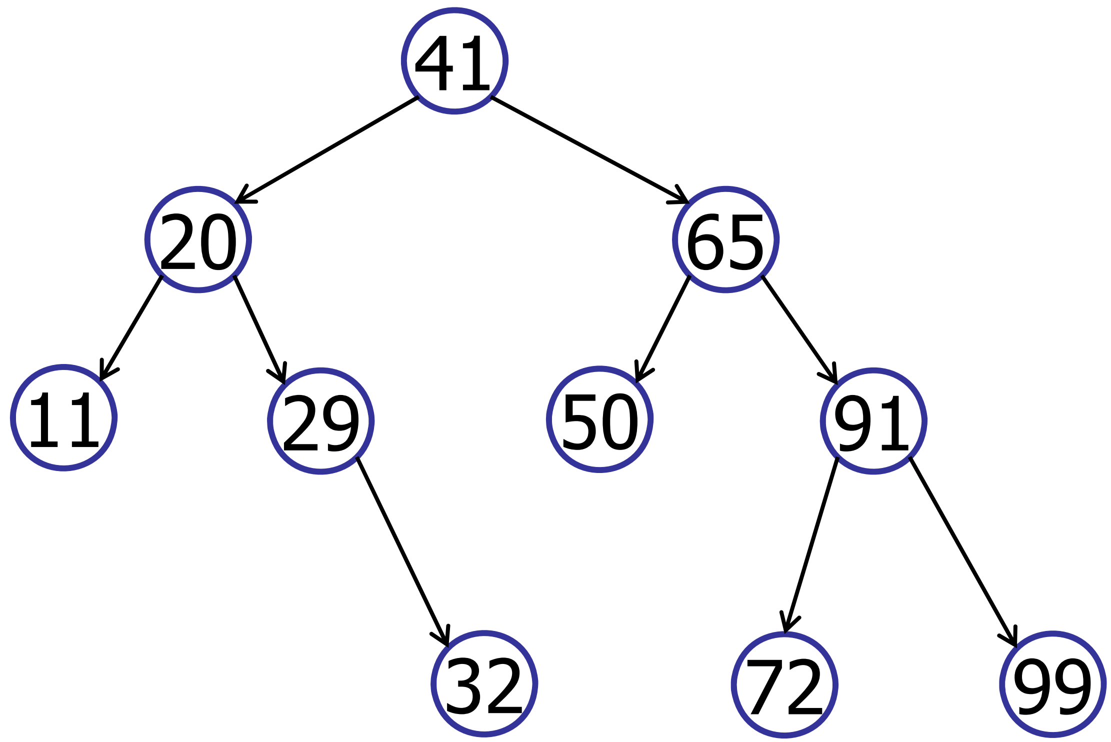
- A node v is height-balanced if:

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

- A binary search tree is height balanced if **every** node in the tree is height-balanced.

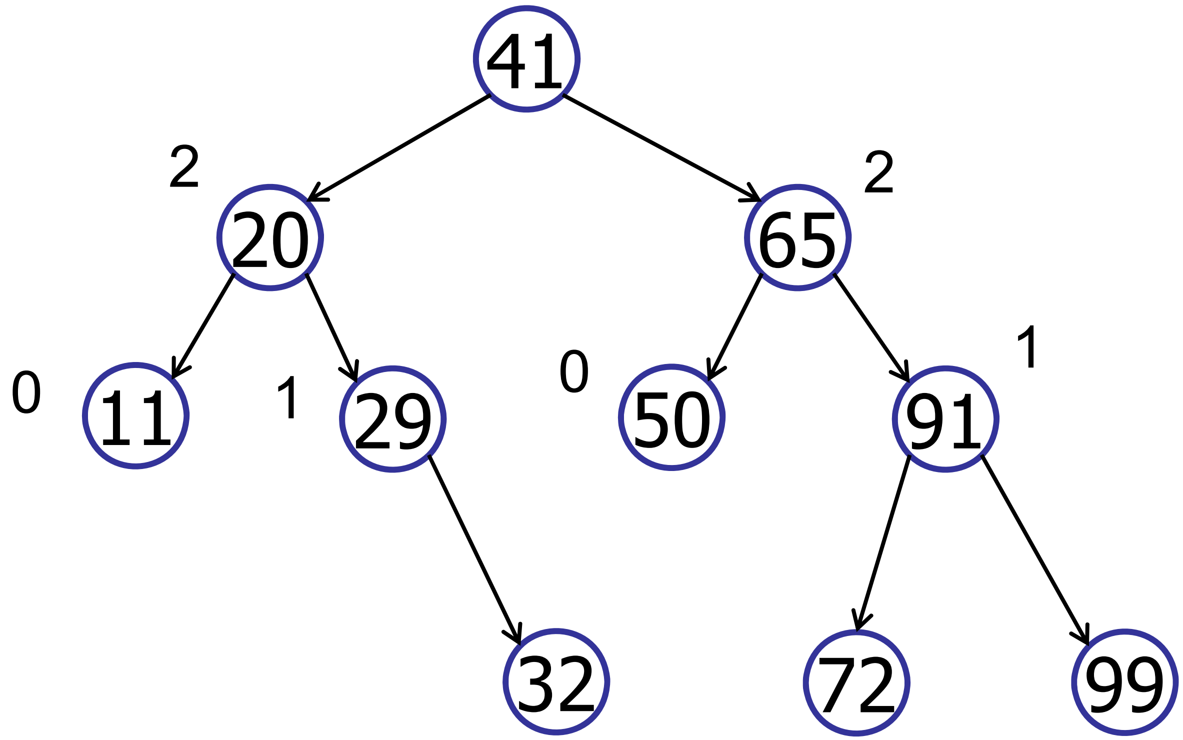
Is this tree height-balanced?

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



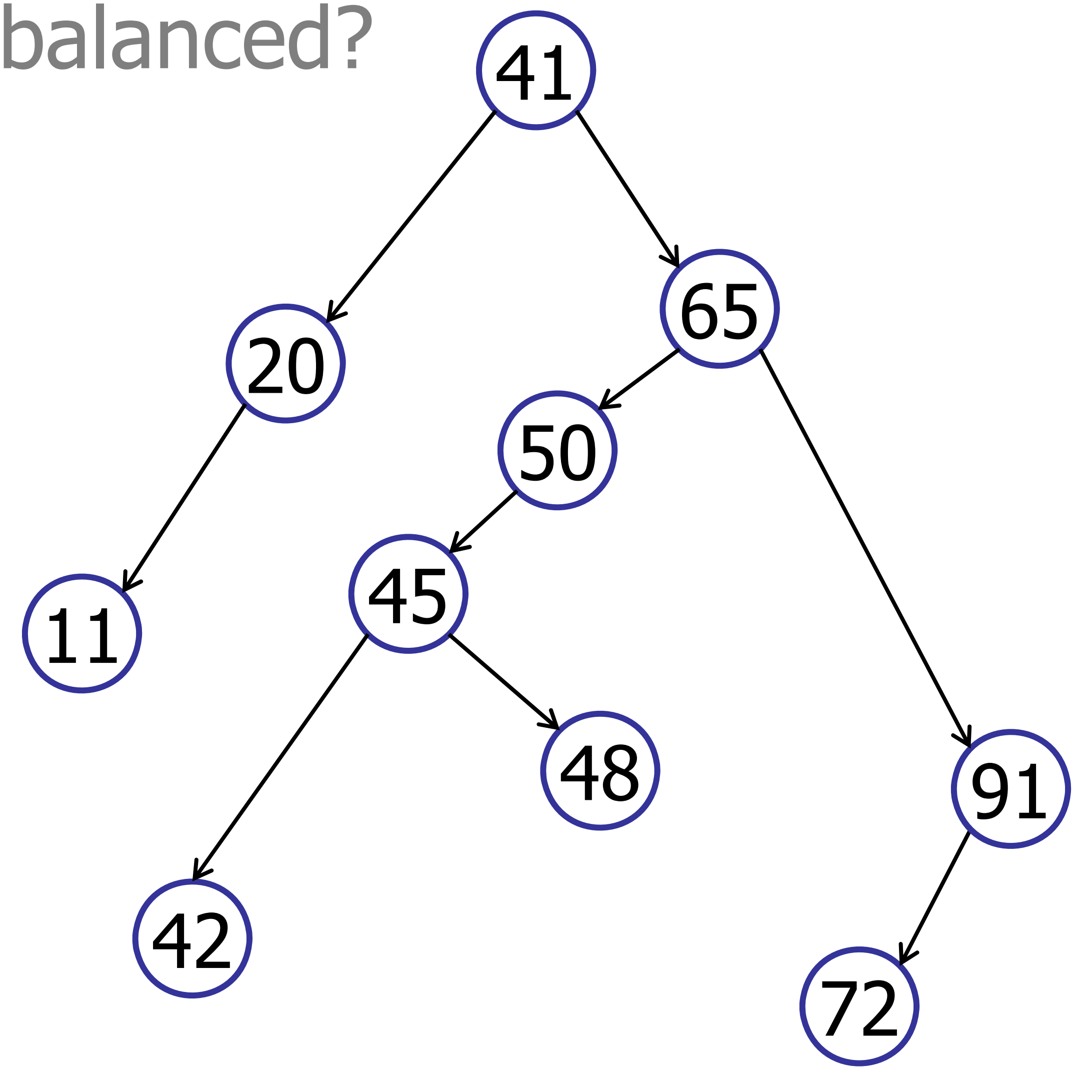
Is this tree height-balanced?

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



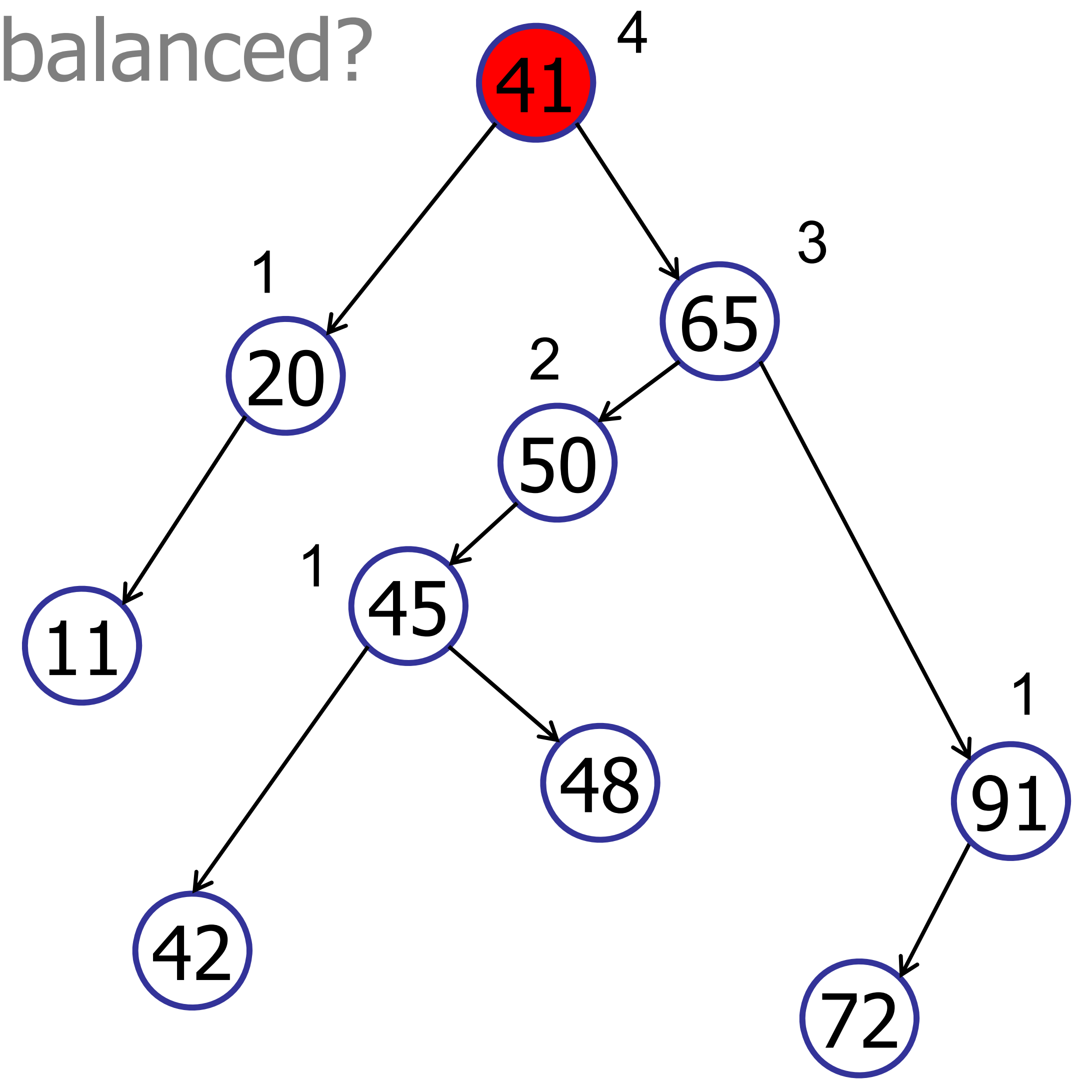
Is this tree height-balanced?

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 **at most** height $h < 2\log(n)$.

Height-Balanced Trees

Claim:

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

$\Leftrightarrow h/2 < \log(n)$

$\Leftrightarrow 2^{h/2} < 2^{\log(n)}$

$\Leftrightarrow 2^{h/2} < n$

A height-balanced tree with height h has **at least** $n > 2^{h/2}$ nodes

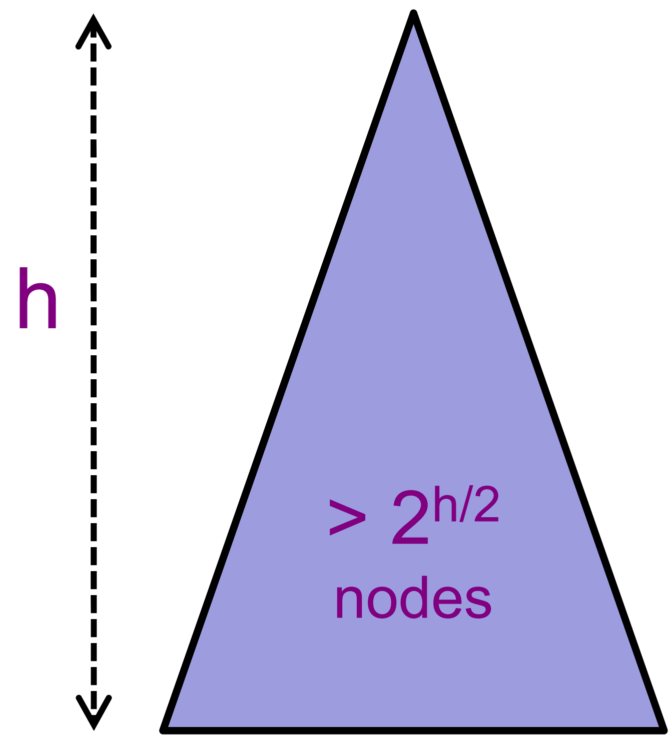
Height-Balanced Trees

Proof:

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

Show:

$n_h > 2^{h/2}$

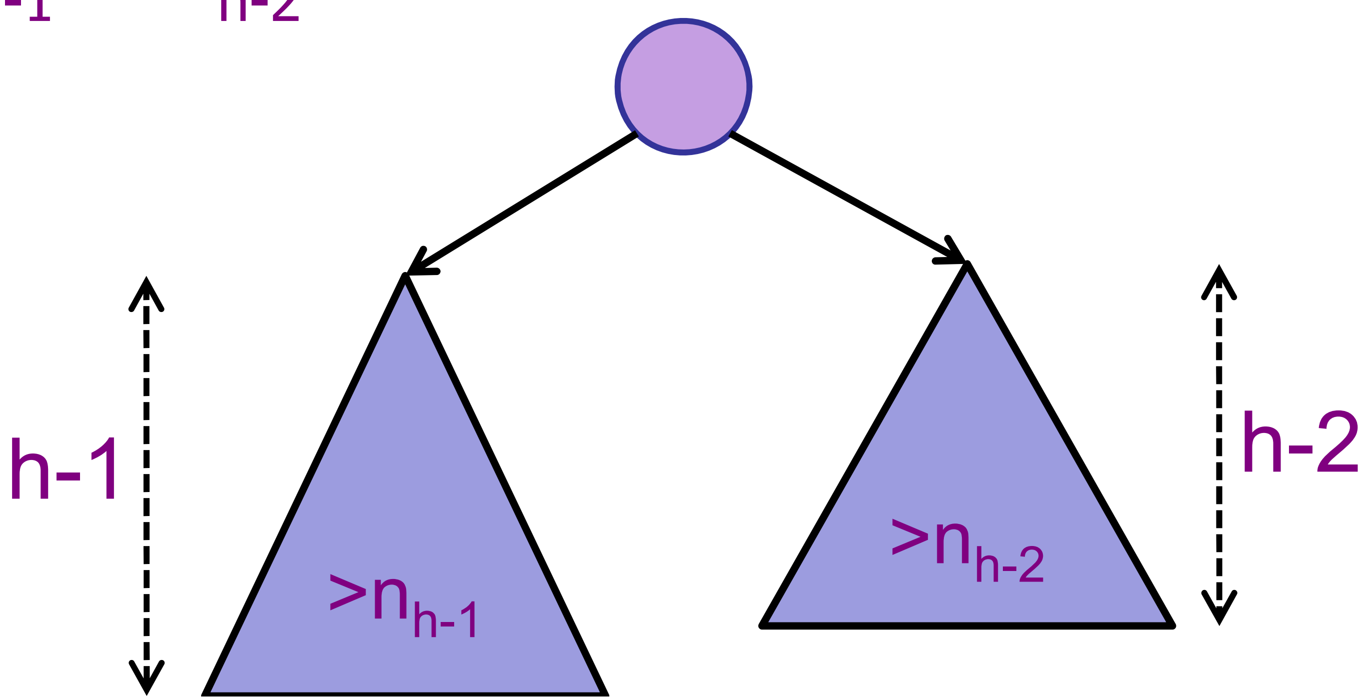


Height-Balanced Trees

Proof:

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

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



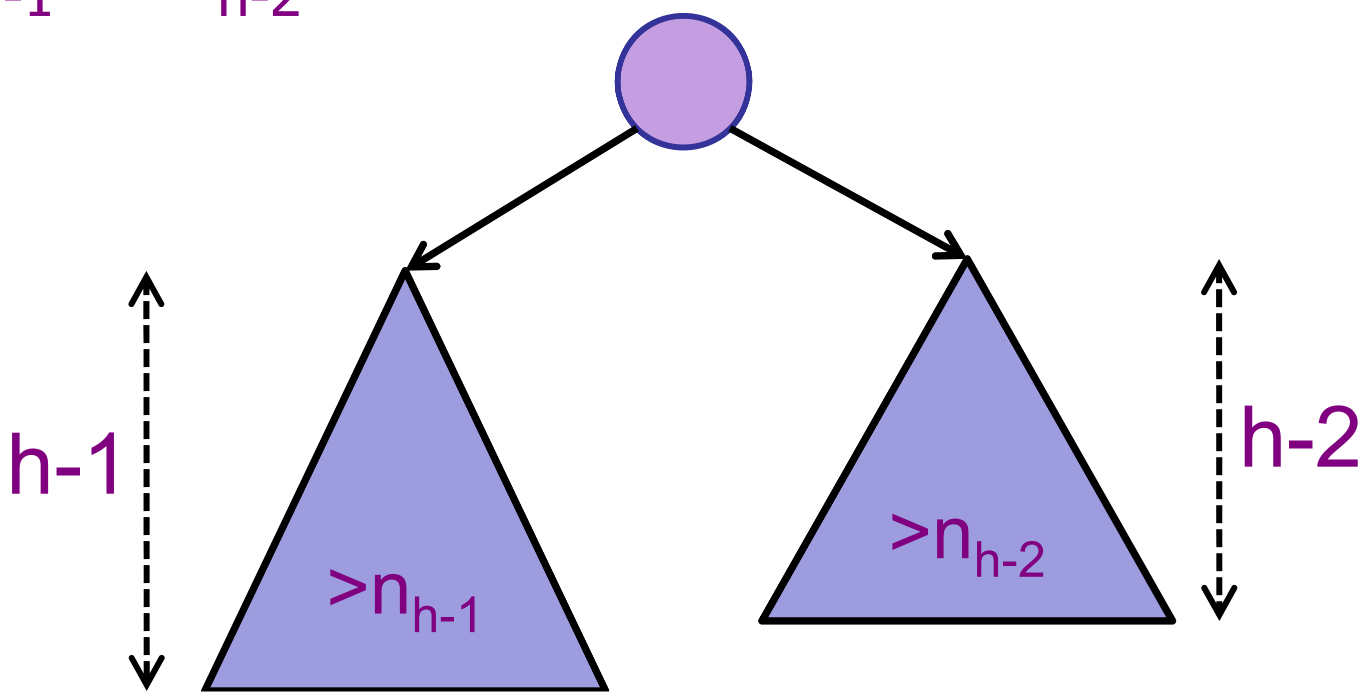
Height-Balanced Trees

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.

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

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

$$\geq 4n_{h-4}$$

$$\geq 8n_{h-6}$$

$$\geq \ldots$$

How many times?

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.

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

$$\geq 2^1 n_{h-2}$$

$$\geq 2^2 n_{h-4}$$

$$\geq 2^3 n_{h-6}$$

$$\geq \ldots \geq 2^k n_0$$

What is k?

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.

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

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

$$\geq 2^{h/2}\, n_0$$

$$\geq 2^{h/2}$$

Base case:
$n_0 = 1$

Height-Balanced Trees

Claim:

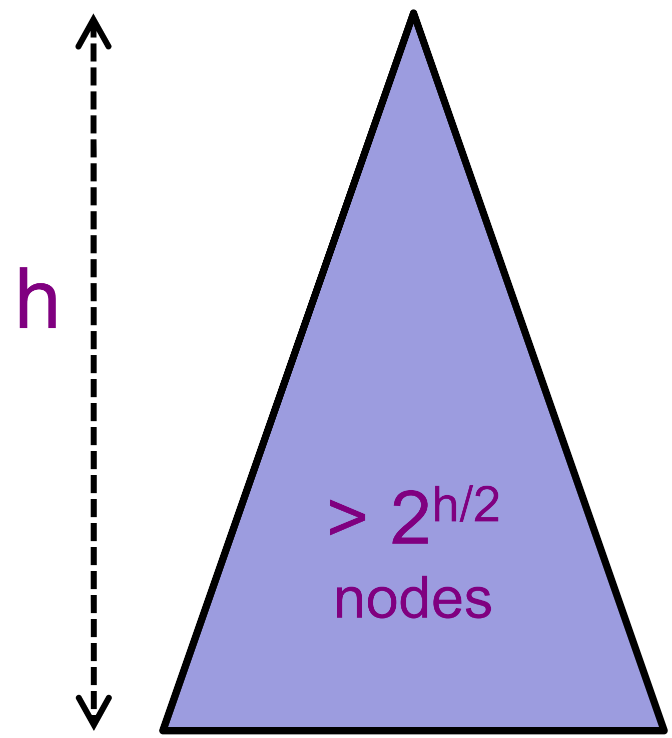
A height-balanced tree with n nodes has height $h < 2\log(n)$.

Show:

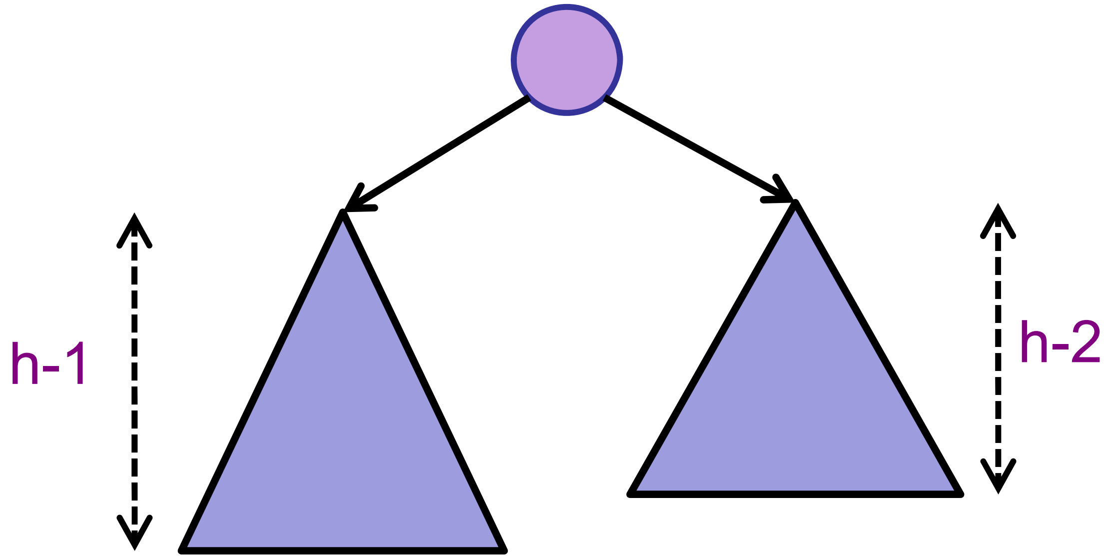
$n > 2^{h/2}$

$\Leftrightarrow$

$h < 2\log(n)$



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)

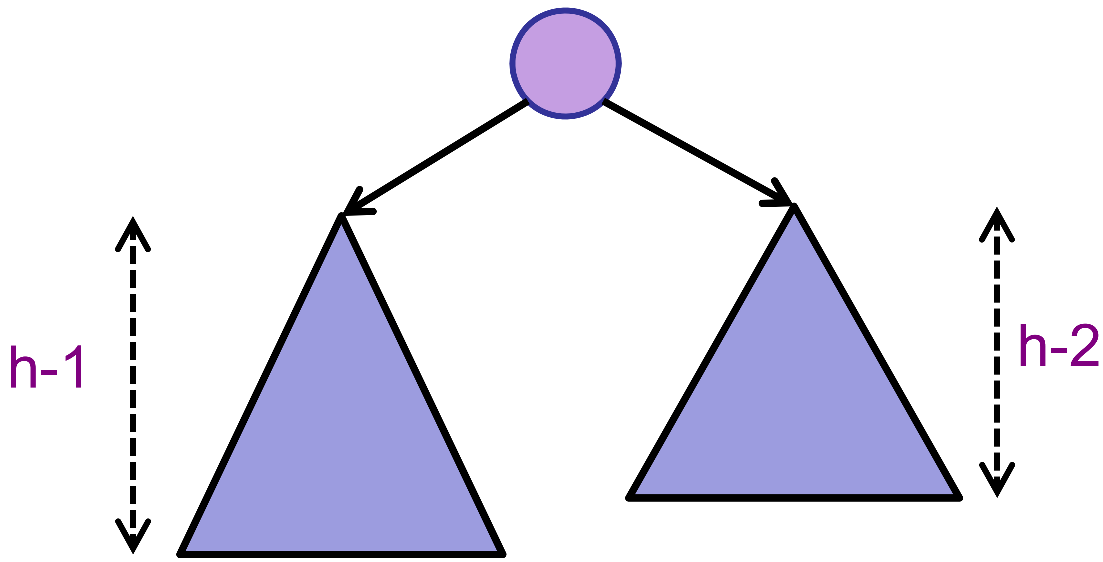
$h \cong \log(n) / \log(\phi)$ $\qquad$ $\phi \cong 1.618$

$h \cong 1.44 \log(n)$

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 0: Augment

Step 1: Define Balance Condition

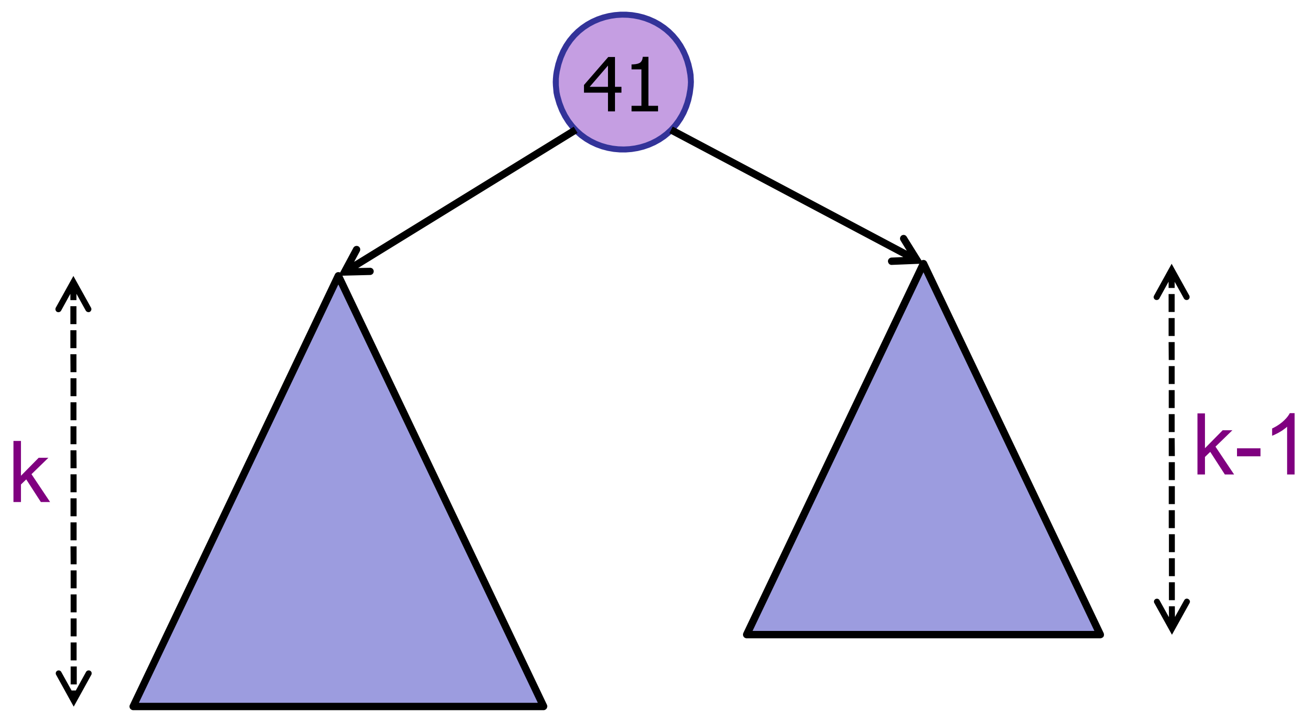
Step 2: Maintain Balance

It's good that we don't have to

Balance perfectly

AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Show how to maintain height-balance



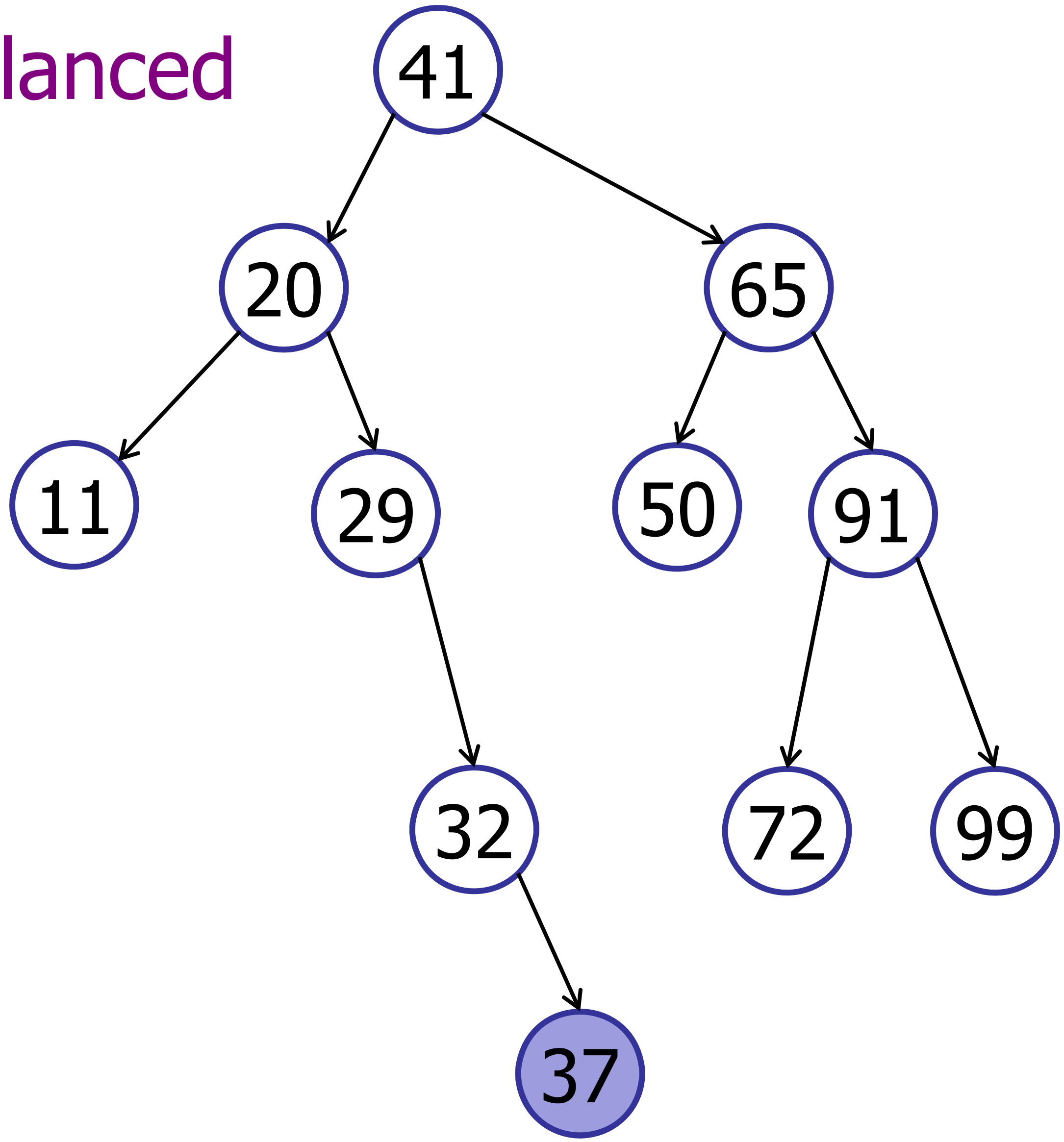
Inserting in an AVL Tree

Before insertion, balanced

insert(37)

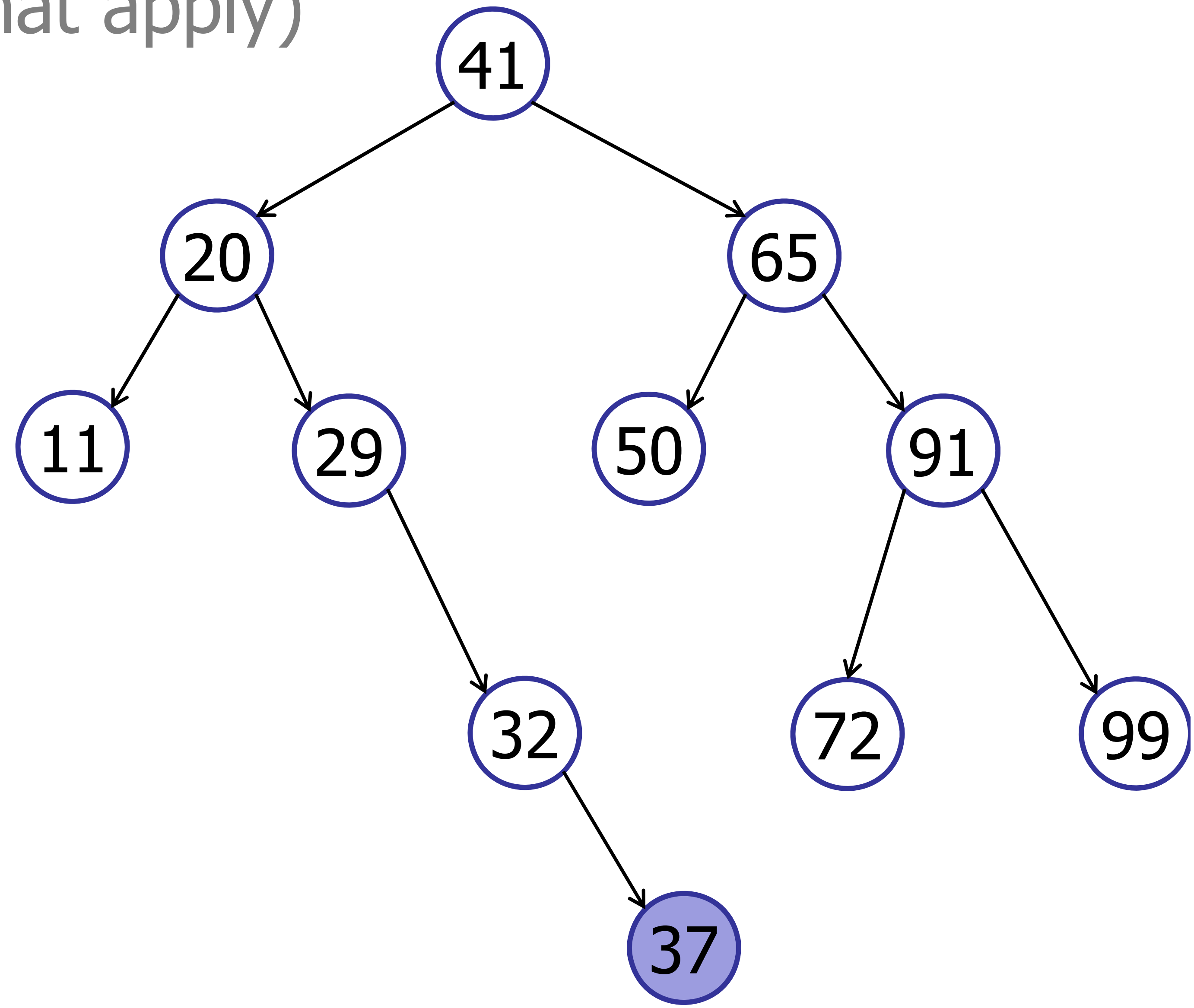
No longer balanced after insertion!

Need to rebalance!



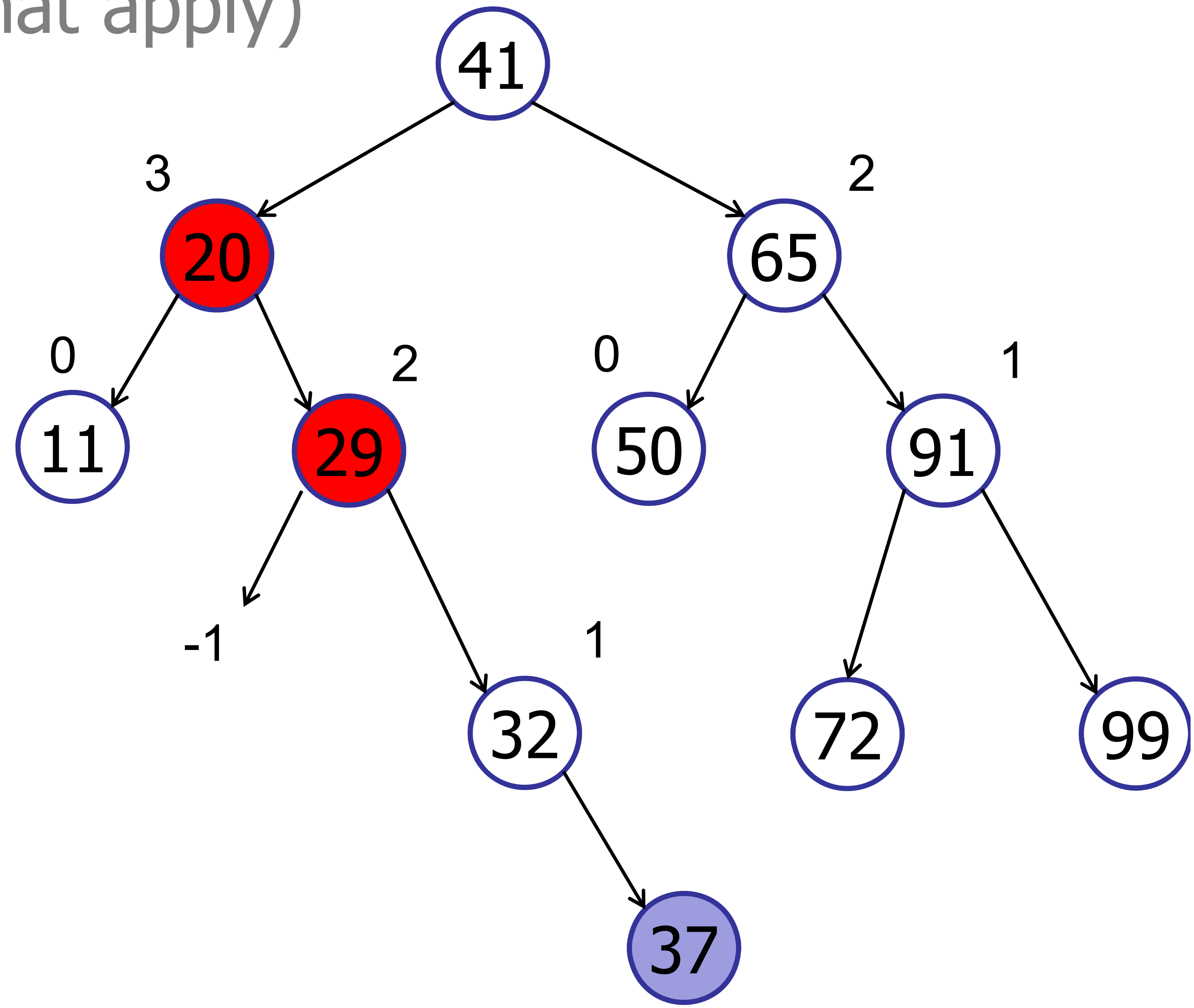
Which nodes need rebalancing?
(click all that apply)

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



Which nodes need rebalancing?
(click all that apply)

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

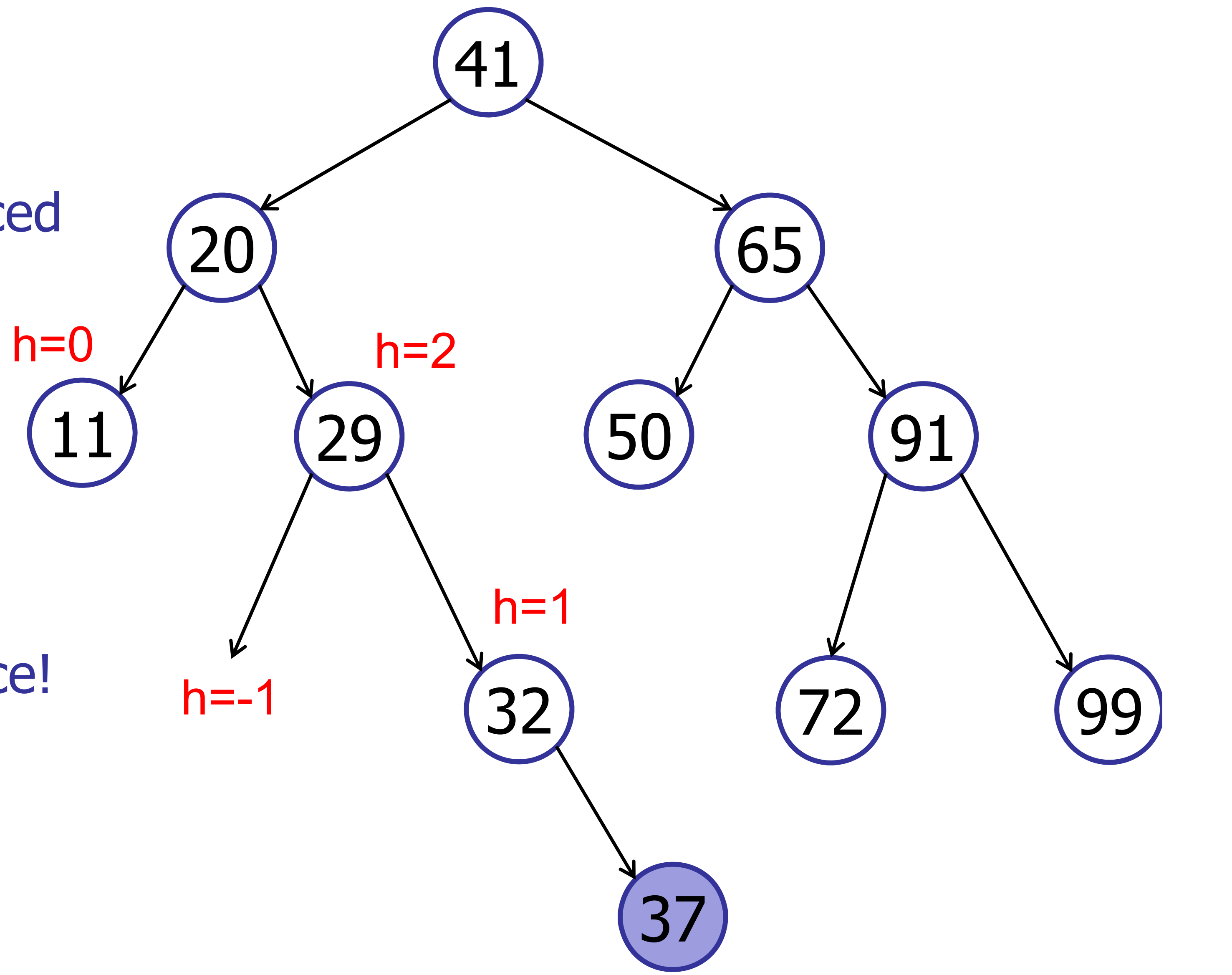


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

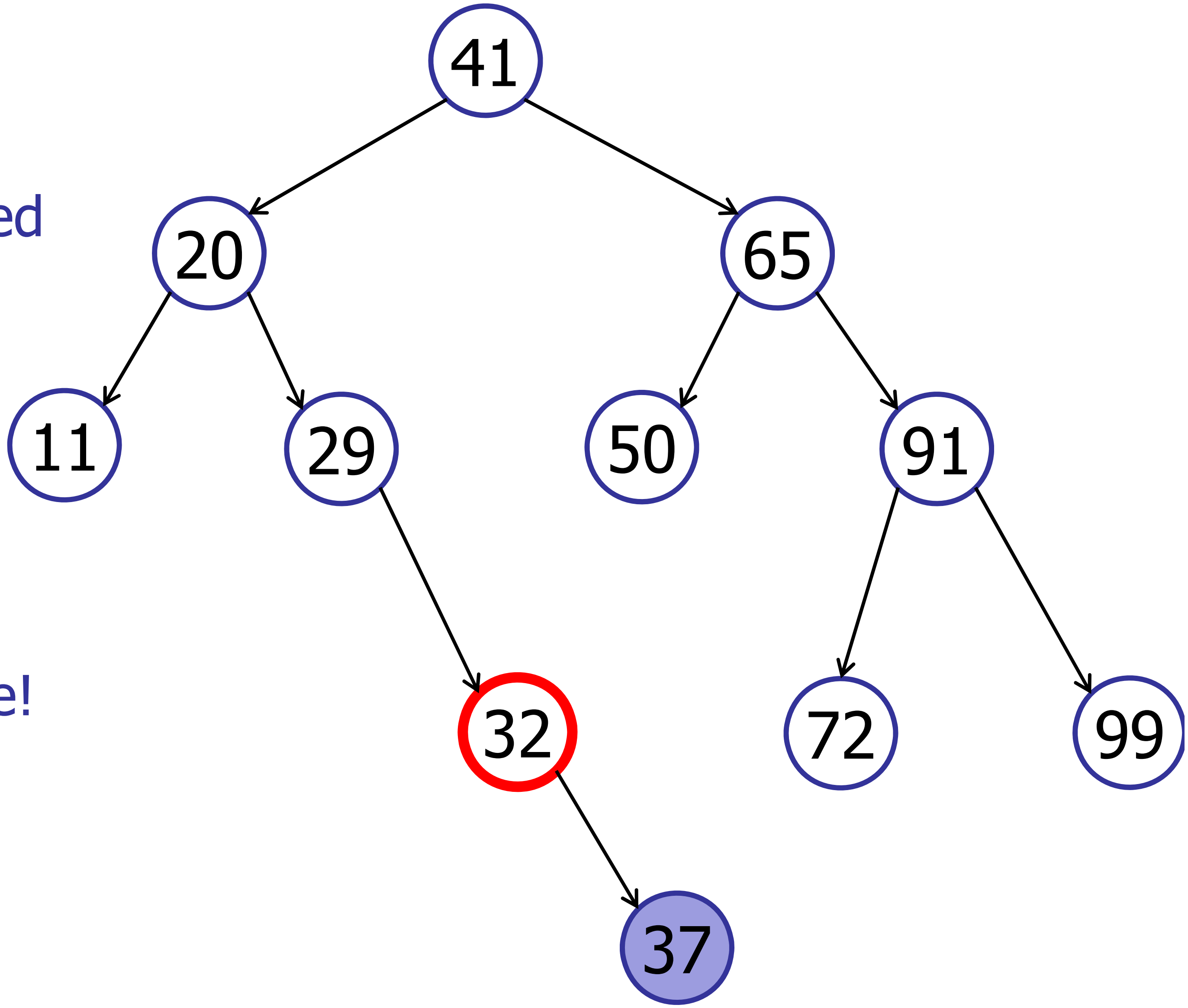


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

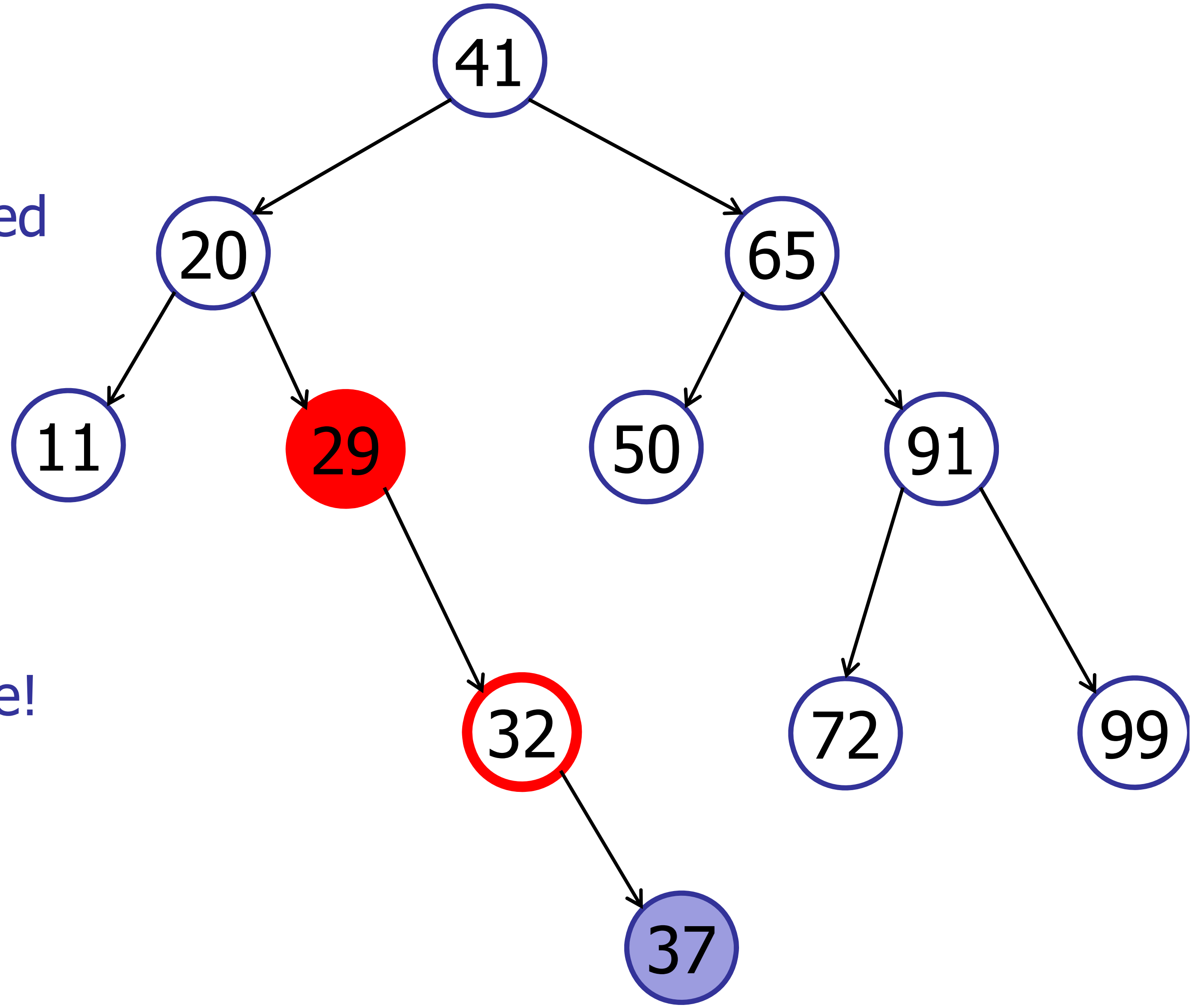


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

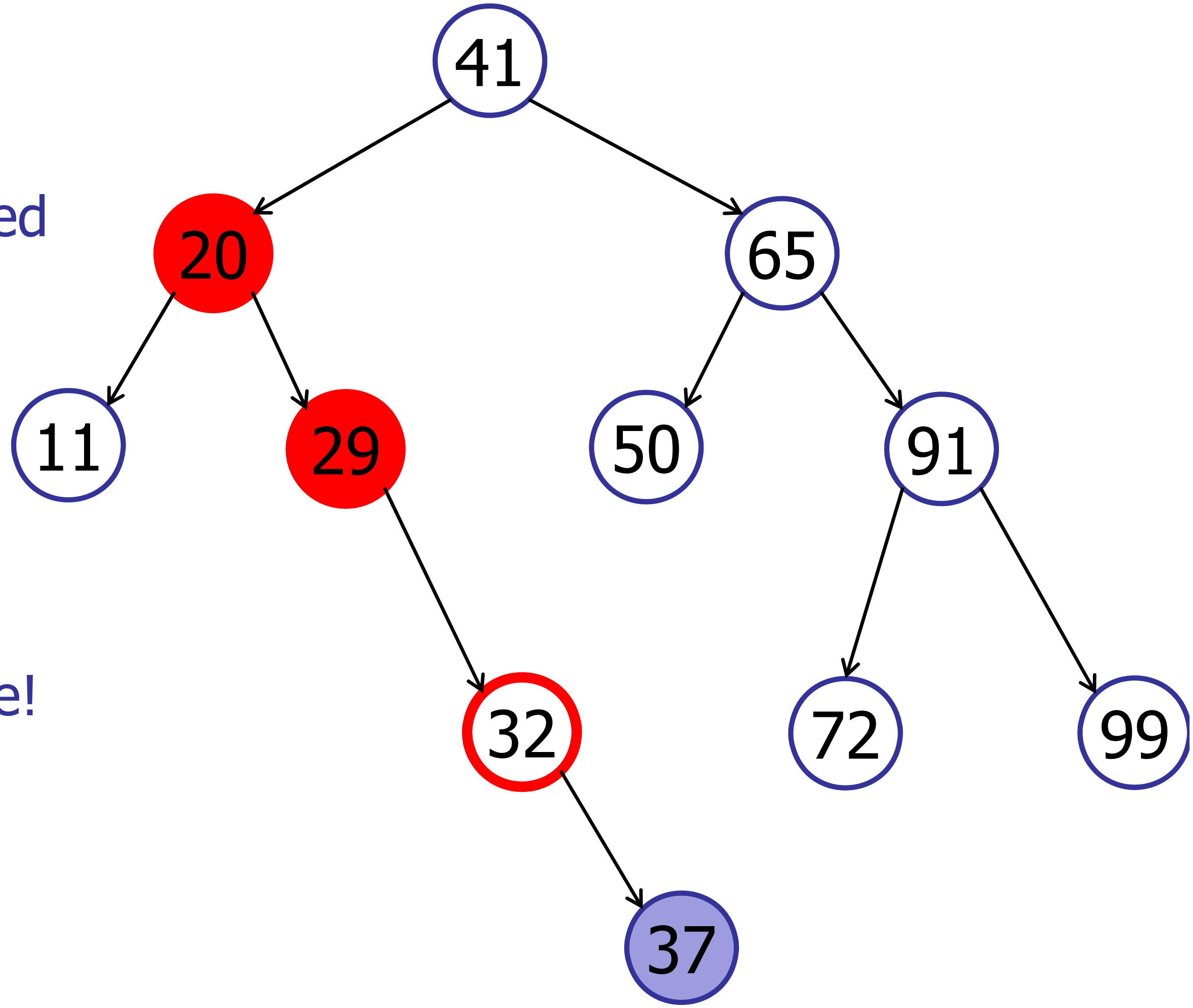


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

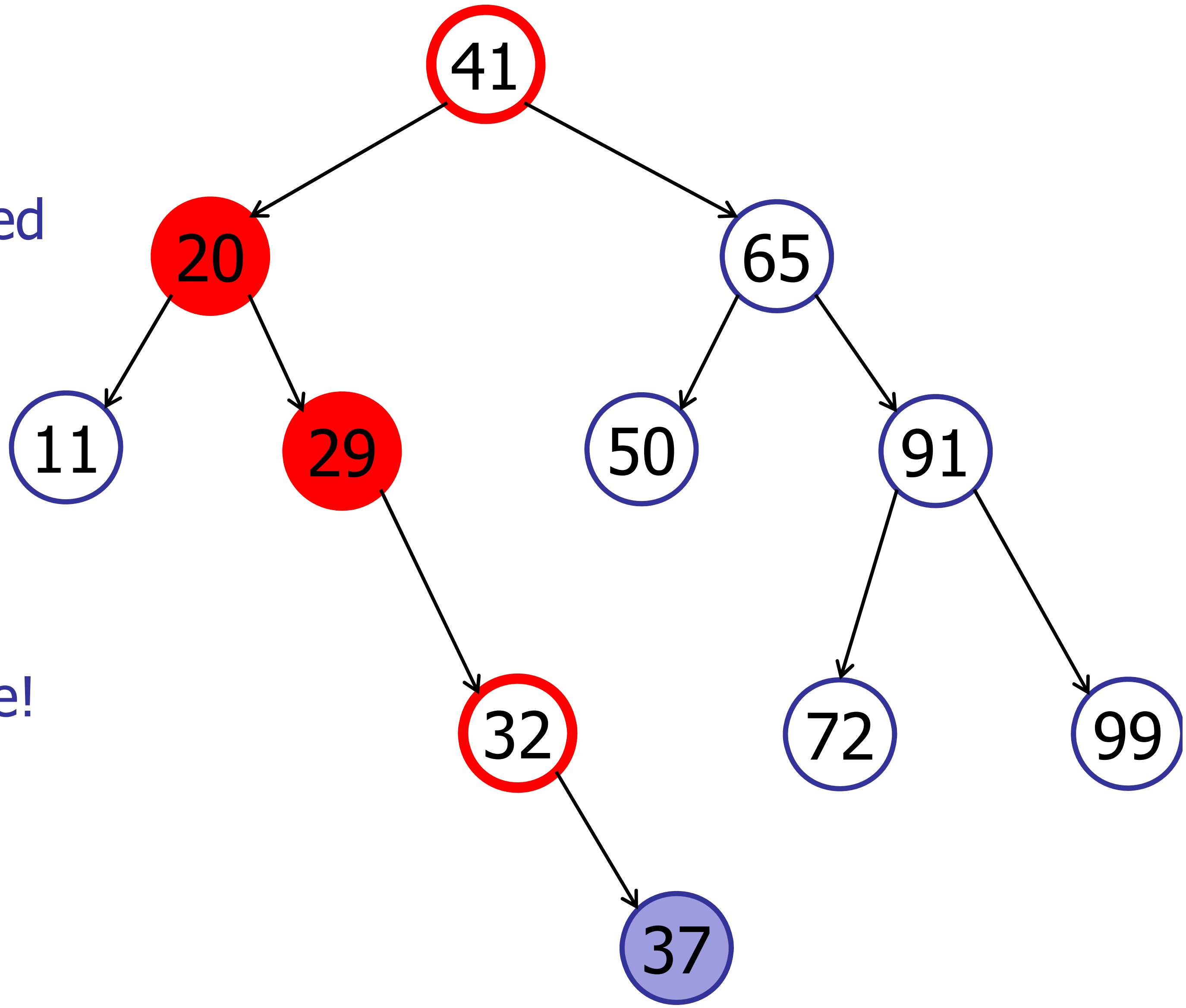


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

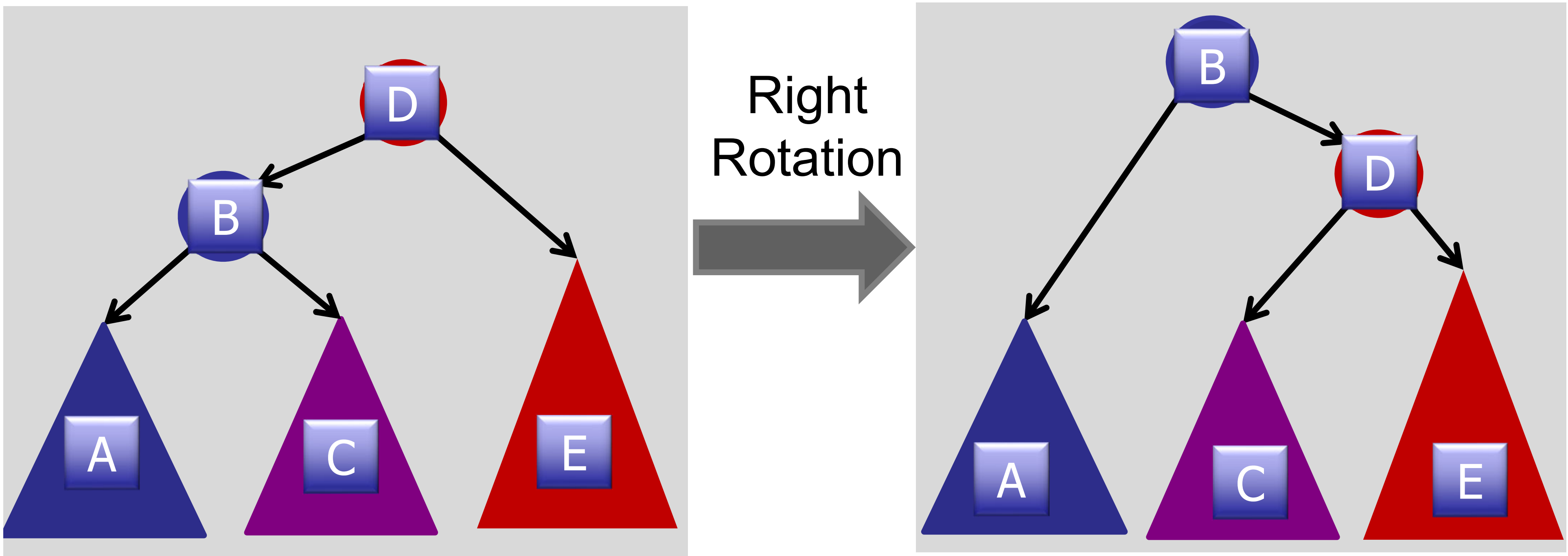
Need to rebalance!



Trick to rebalance the tree

Tree rotation!

Tree Rotations

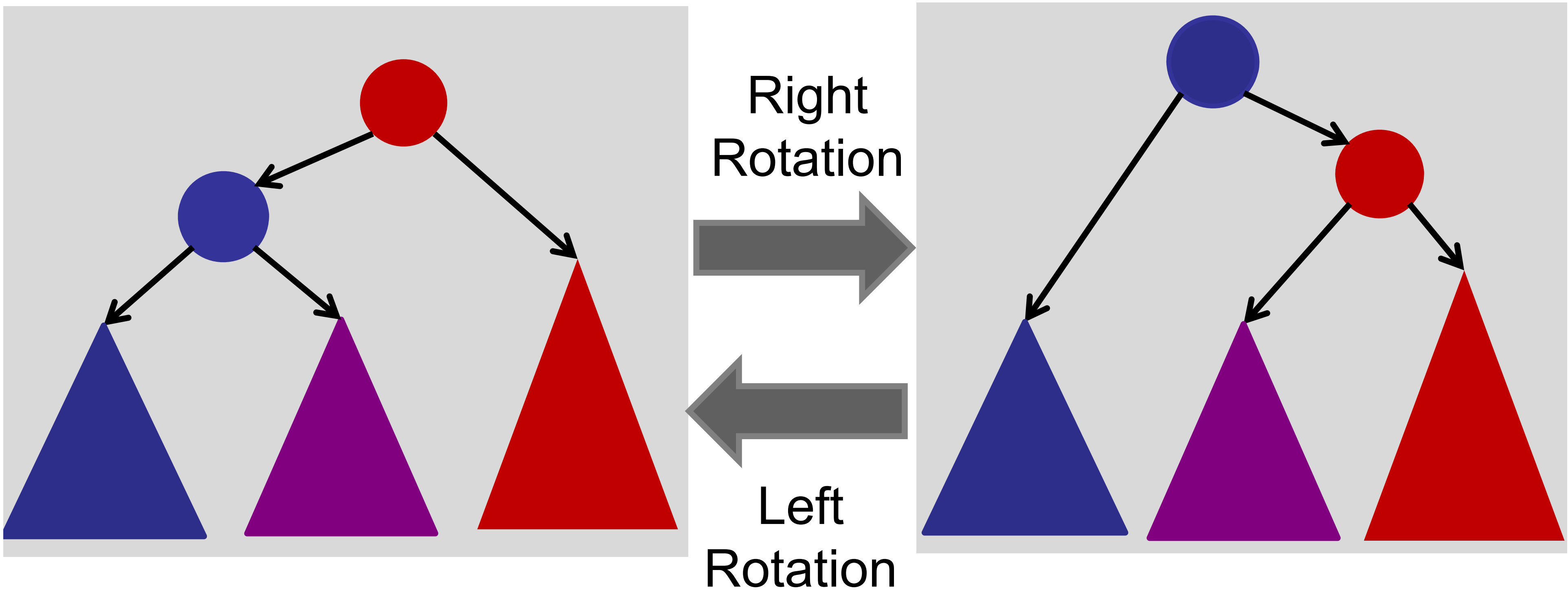


$A < B < C < D < E$

Rotations maintain ordering of keys.

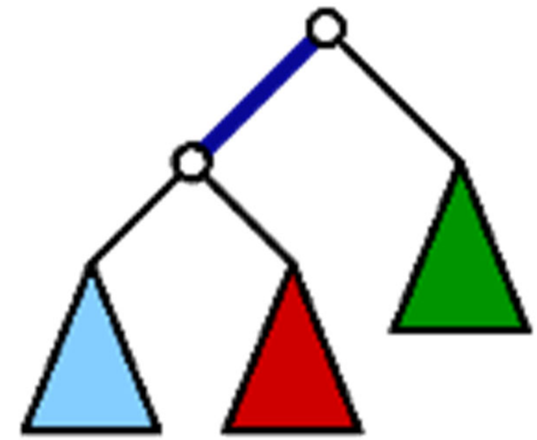
$\Rightarrow$ Maintains BST property.

Tree Rotations

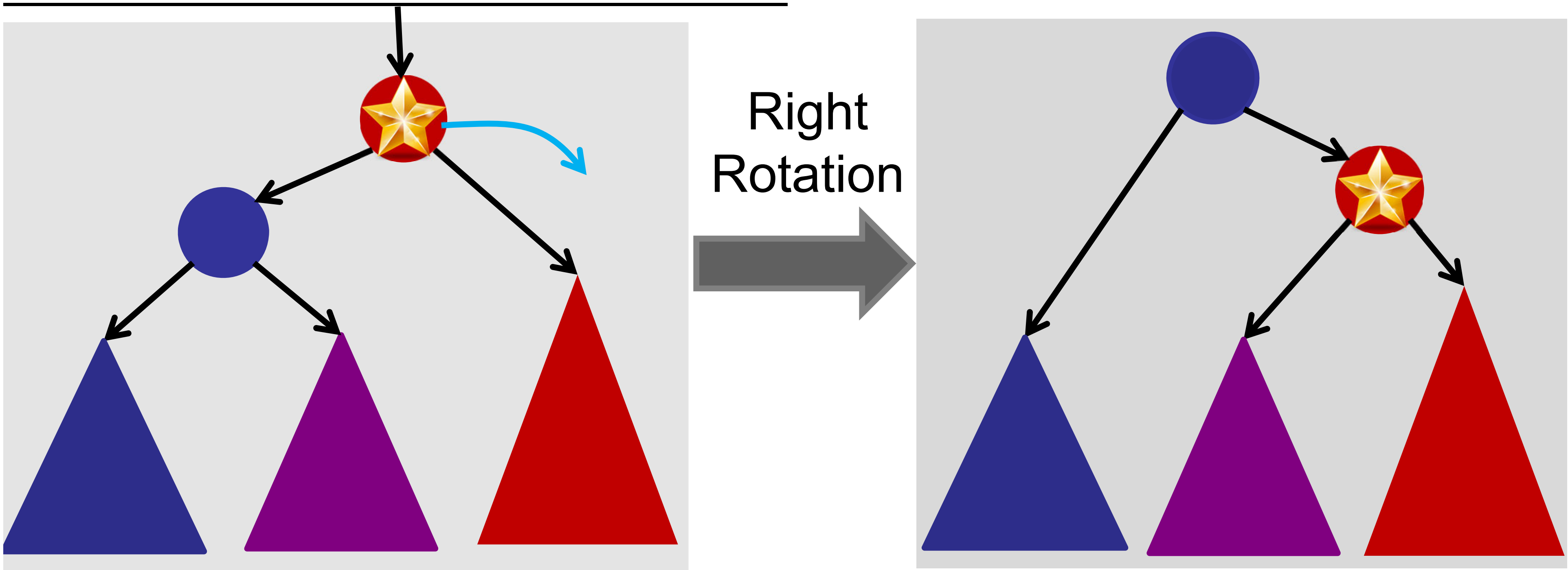


Wait….

What is a left rotation and what is a right rotation!?

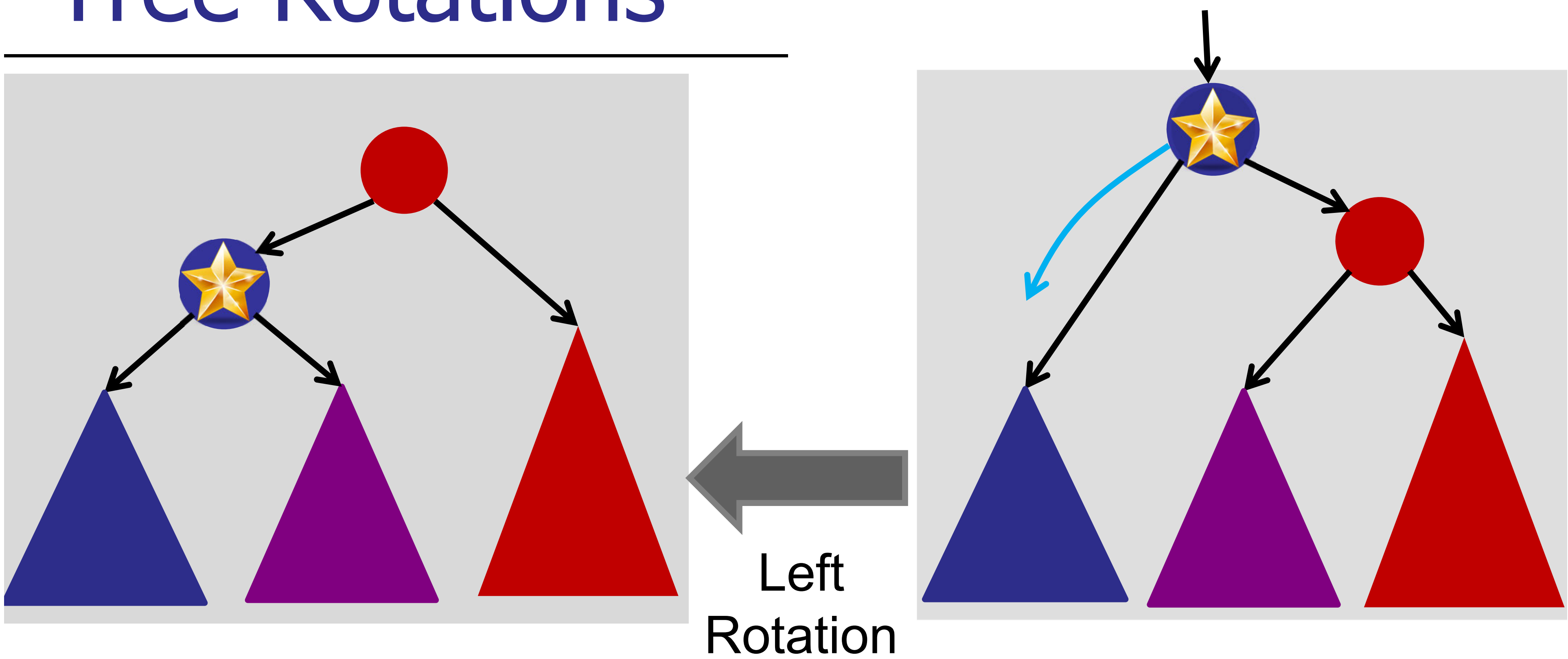


The way to remember it



The root of the subtree moves right

Tree Rotations

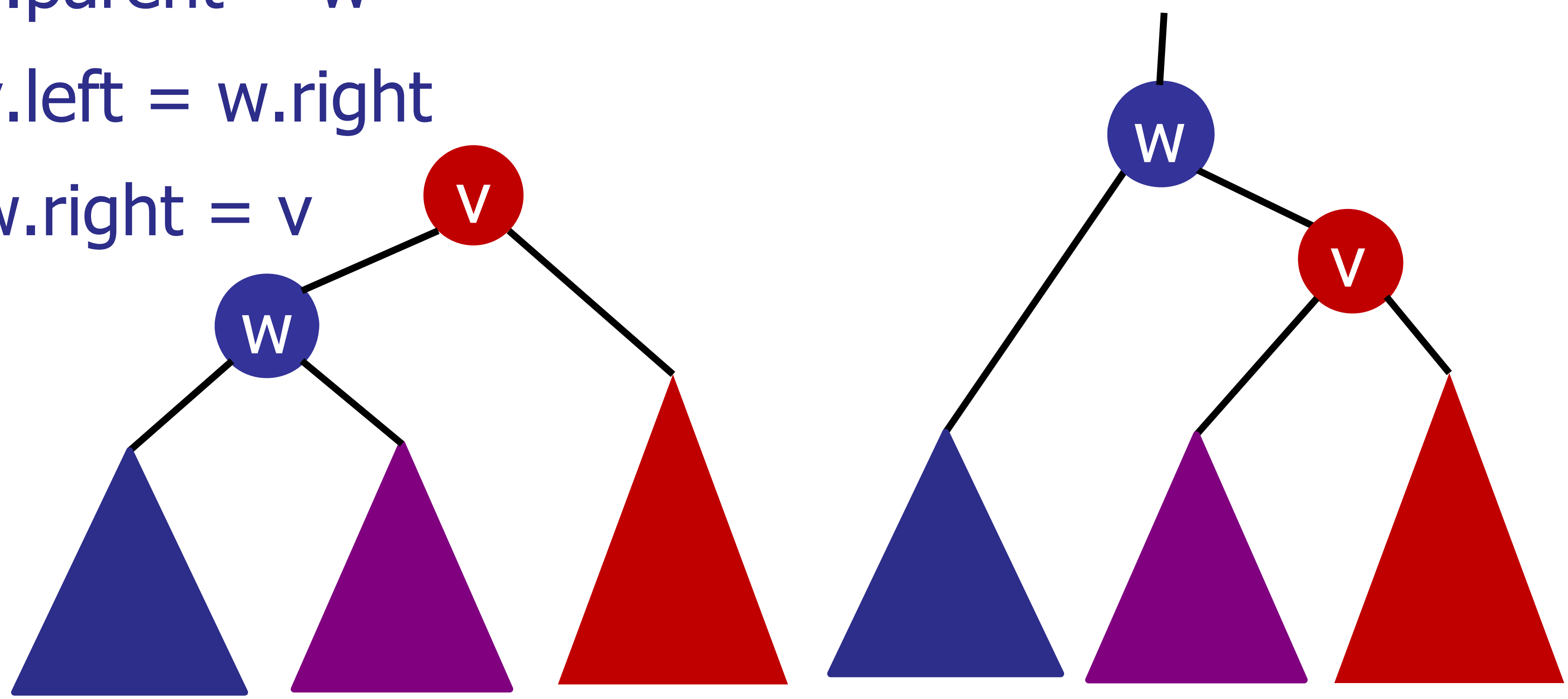


The root of the subtree moves left

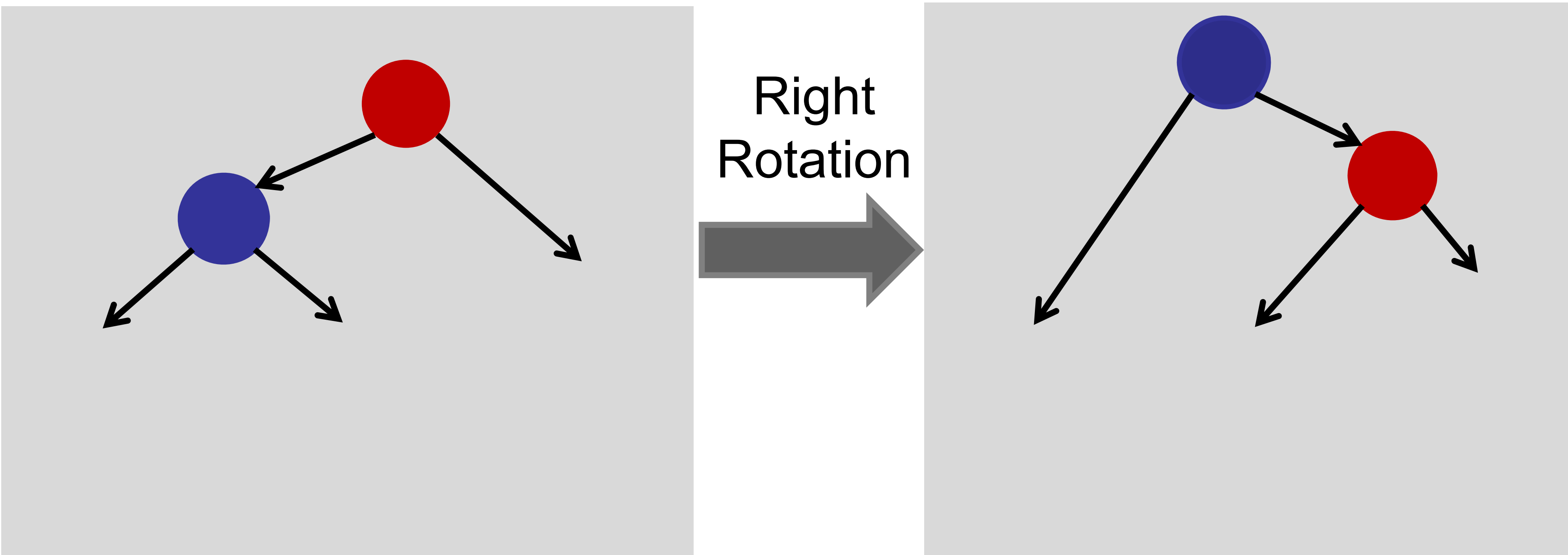
Rotations

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
```



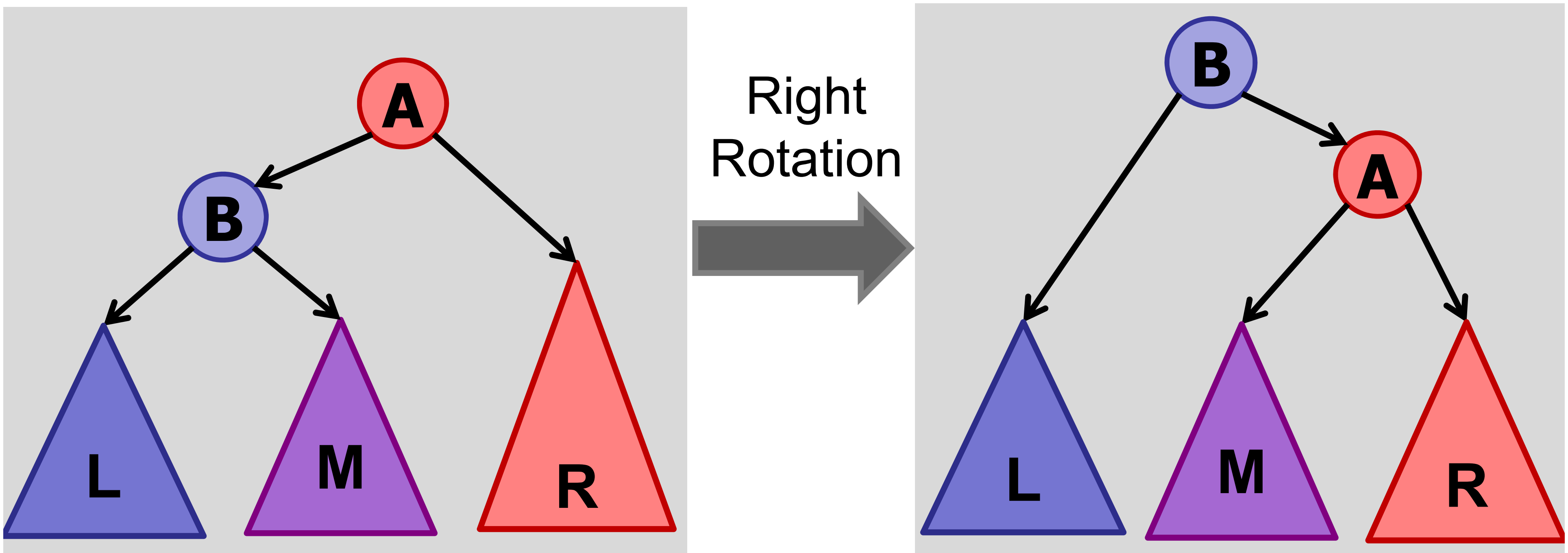
Tree Rotations



rotate-right requires a left child

rotate-left requires a right child

Tree Rotations



After insert:

Use tree rotations to restore balance.

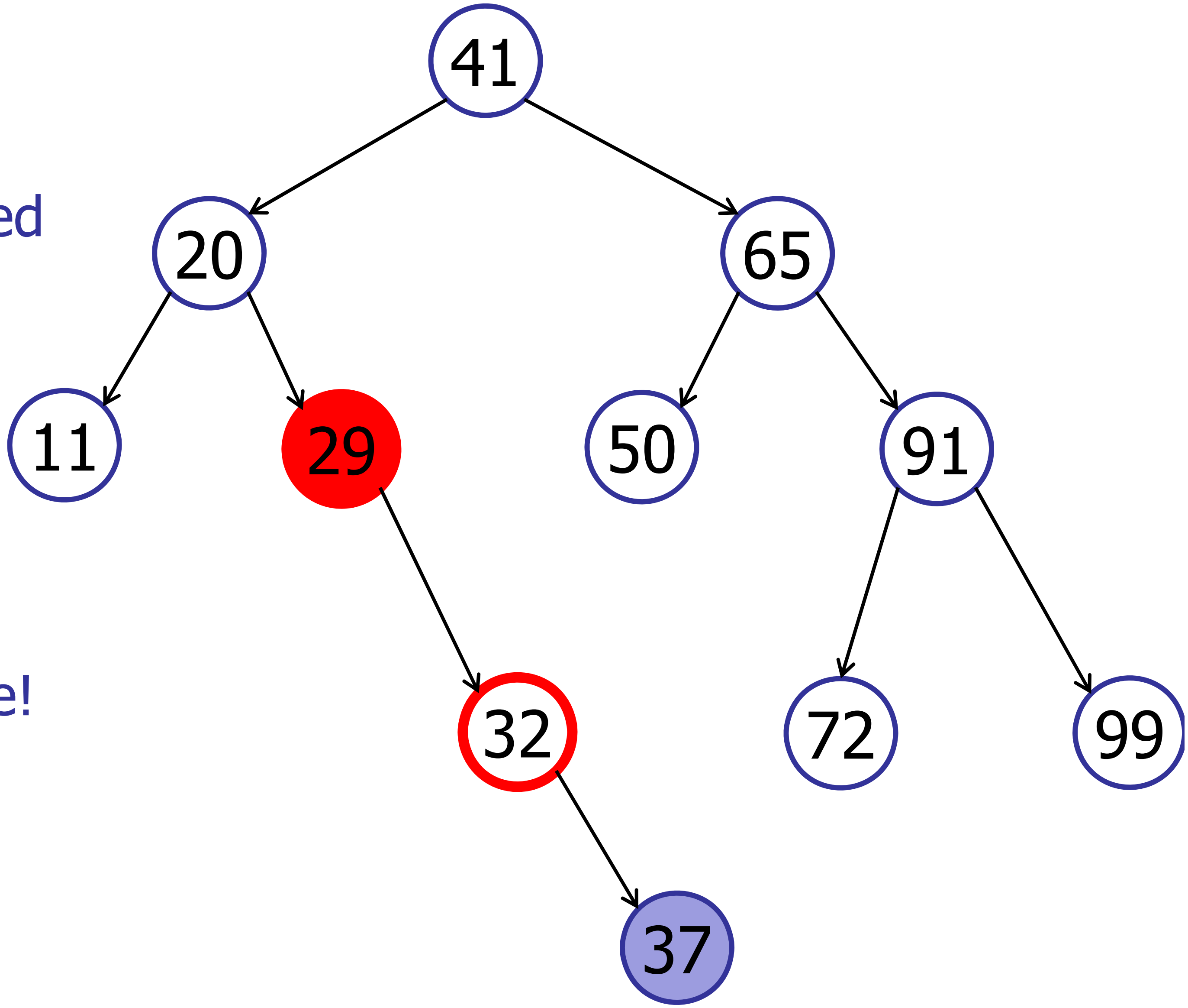
Height is out-of-balance by 1

Inserting in an AVL Tree

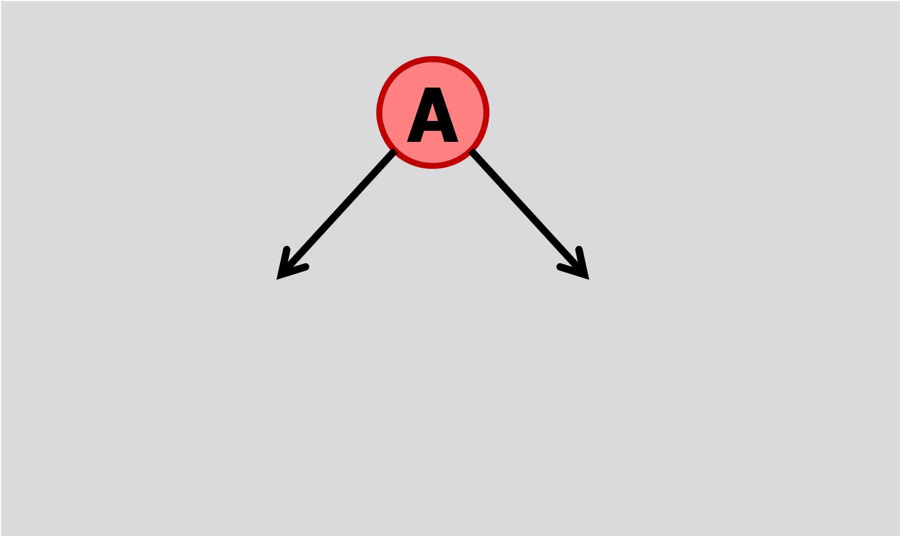
insert(37)

No longer balanced after insertion!

Need to rebalance!



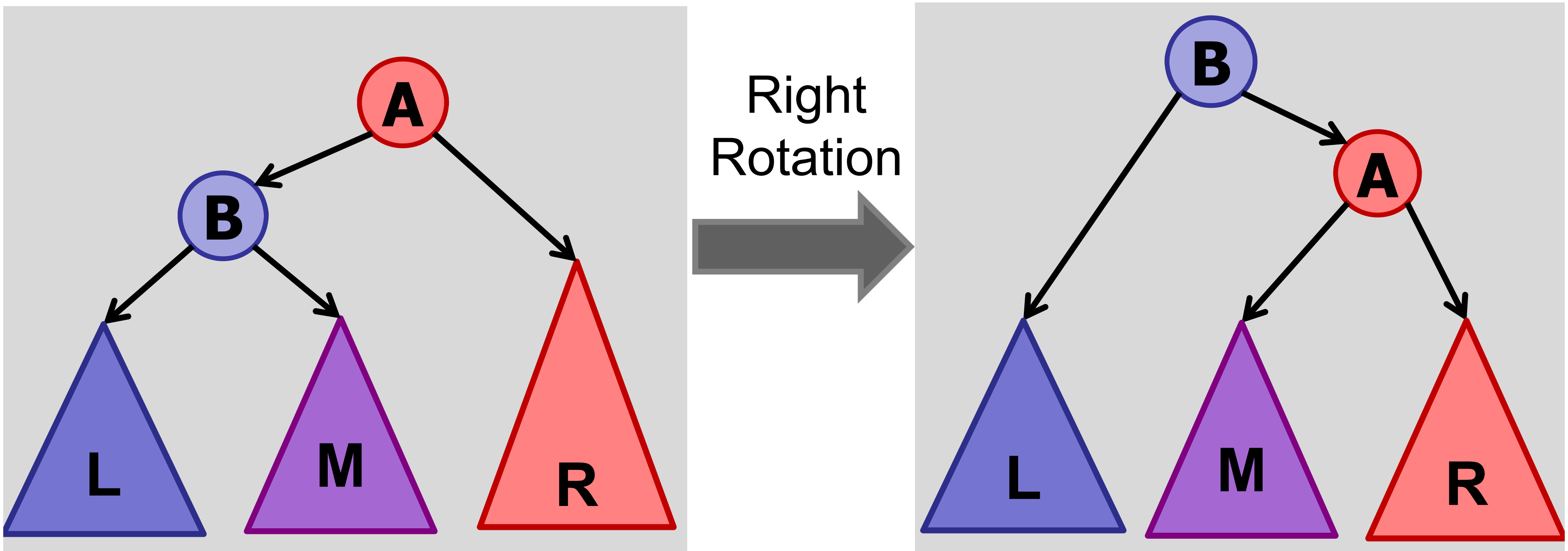
Tree Rotations



A is **LEFT-heavy** if left sub-tree has larger height than right sub-tree.

A is **RIGHT-heavy** if right sub-tree has larger height than left sub-tree.

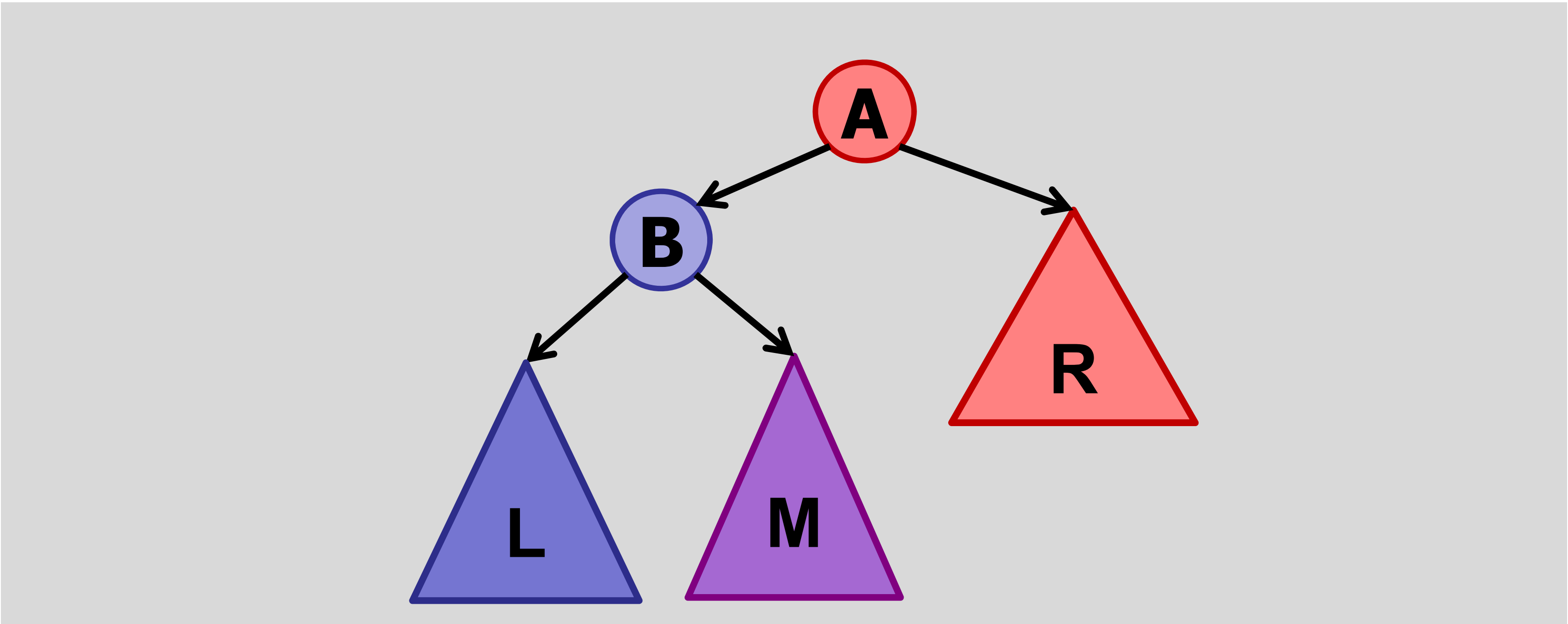
Tree Rotations



Use tree rotations to restore balance.

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

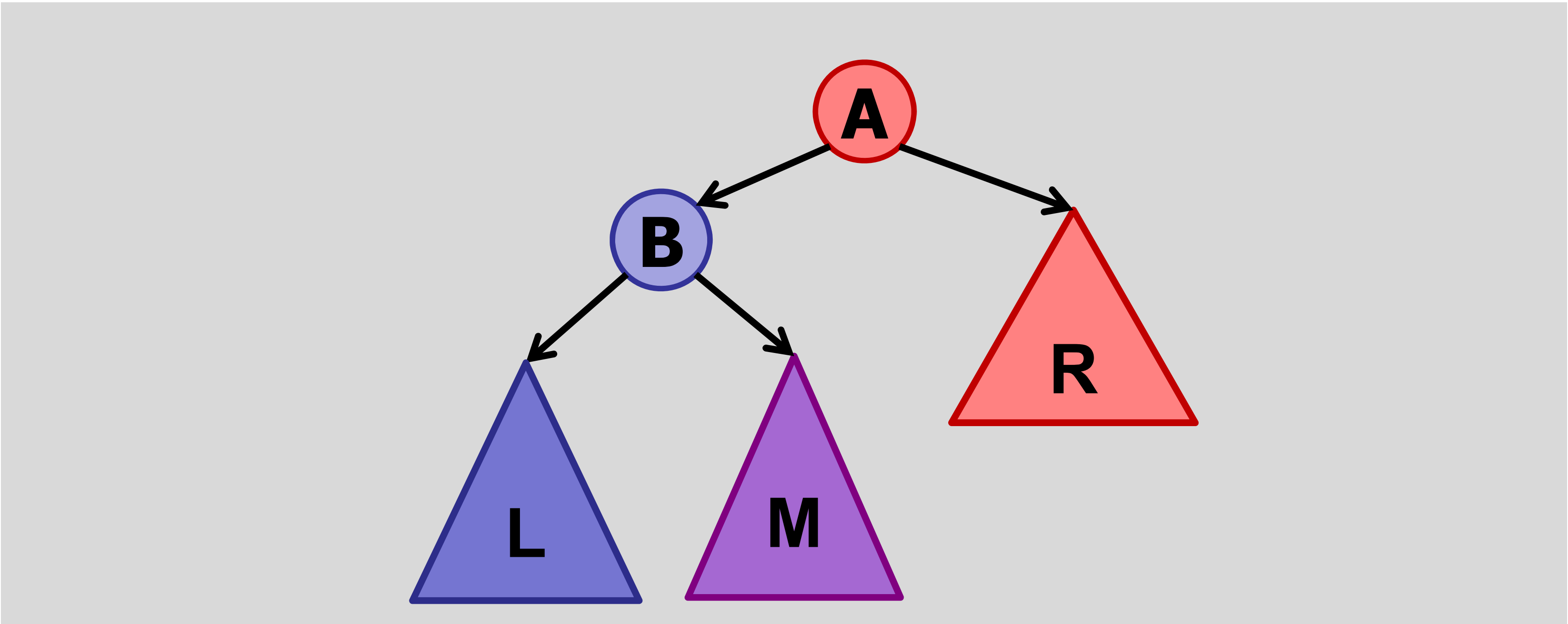
Tree Rotations



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

Assume A is **LEFT-heavy**.

Tree Rotations (Left Heavy)

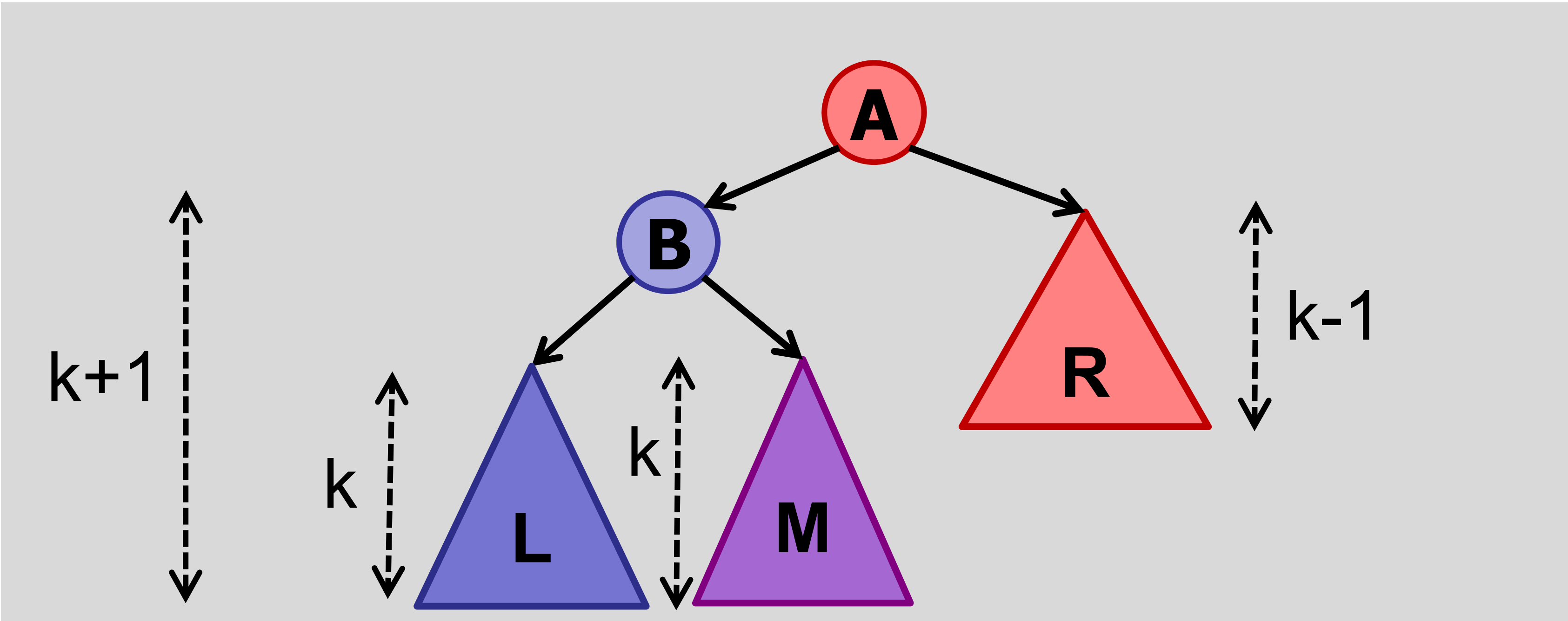


Assume **A** is the lowest node in the tree violating balance property.

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

h(**R**) = h(**B**) – 2

Tree Rotations (Left Heavy)

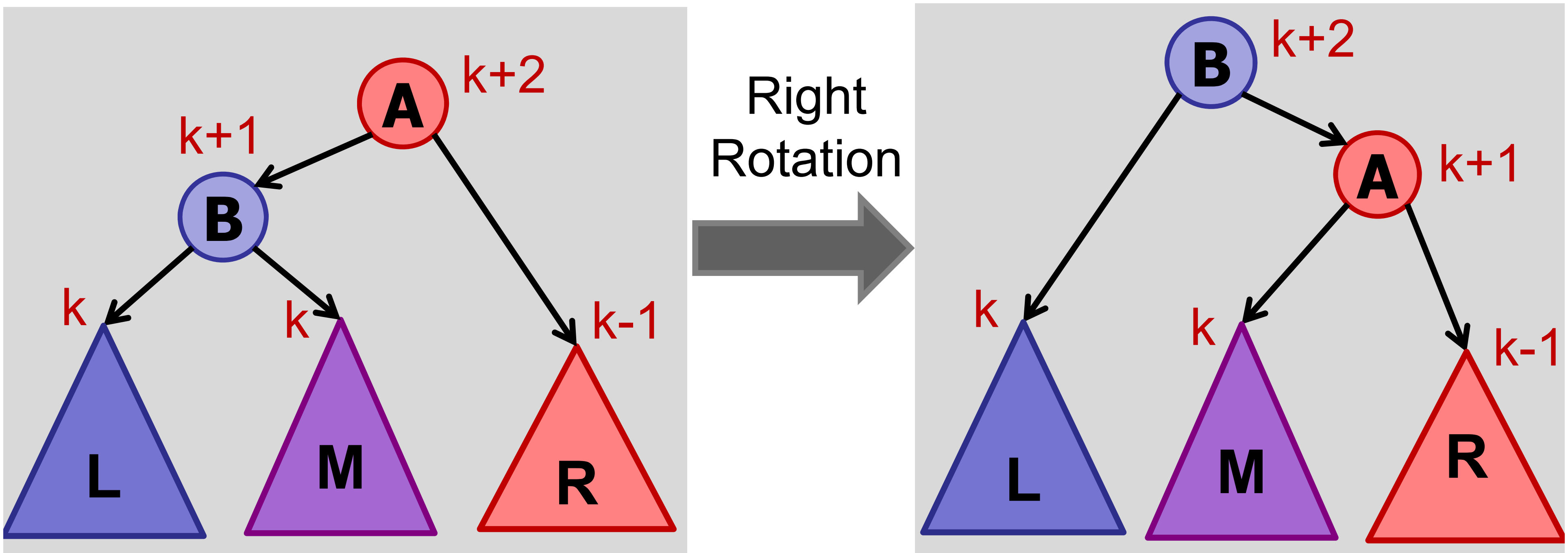


Assume **A** is the lowest node in the tree violating balance property.

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

h(**R**) = h(**M**) – 1

Tree Rotations

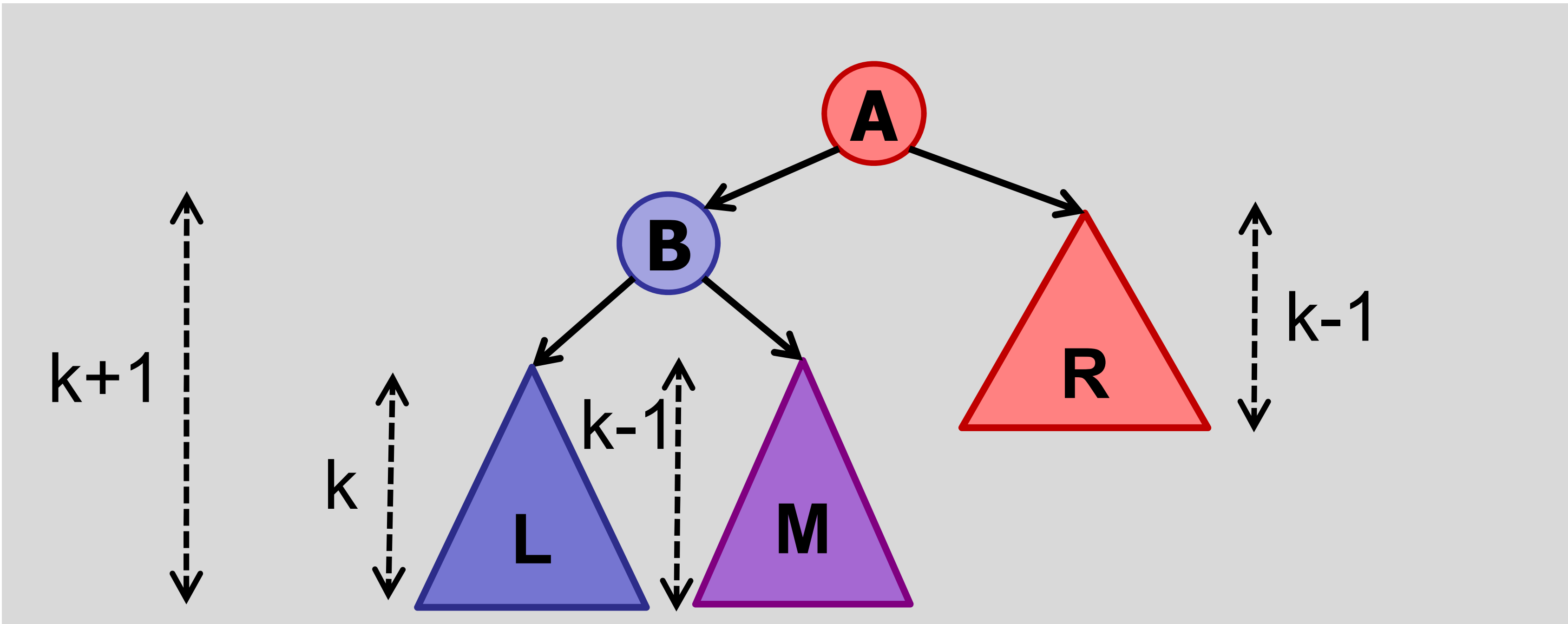


right-rotate:

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

h(**R**) = h(**M**) – 1

Tree Rotations (Left Heavy)

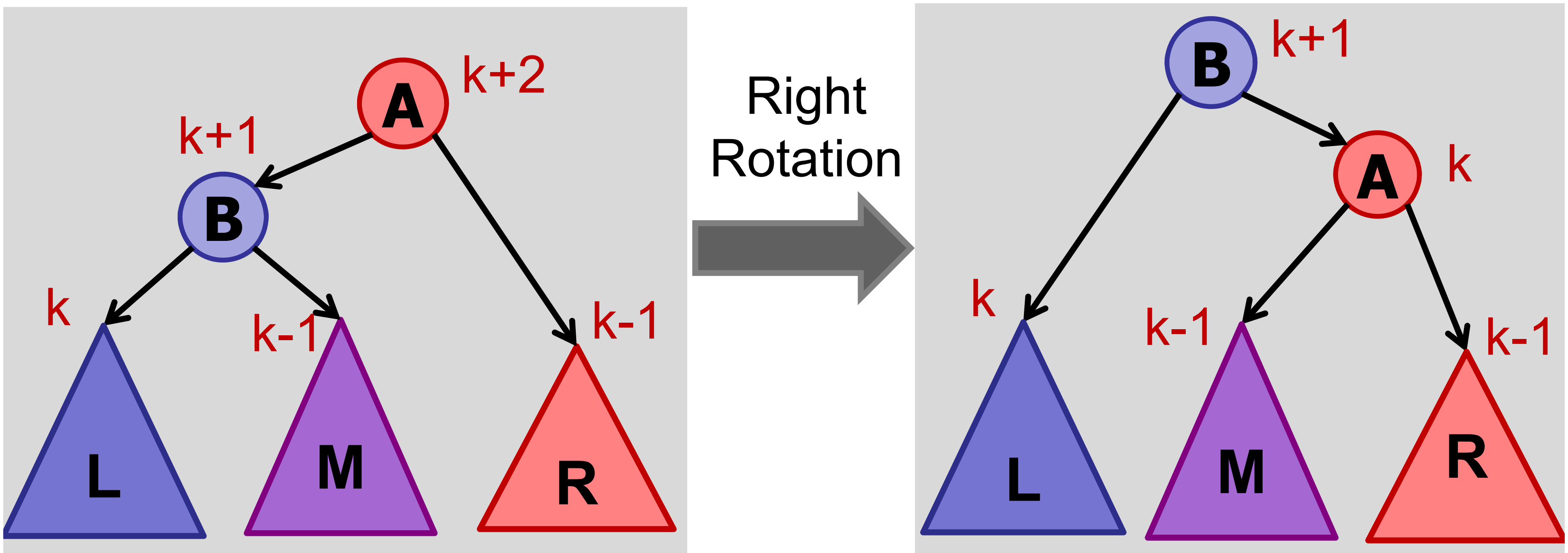


Assume **A** is the lowest node in the tree violating balance property.

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

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

Tree Rotations

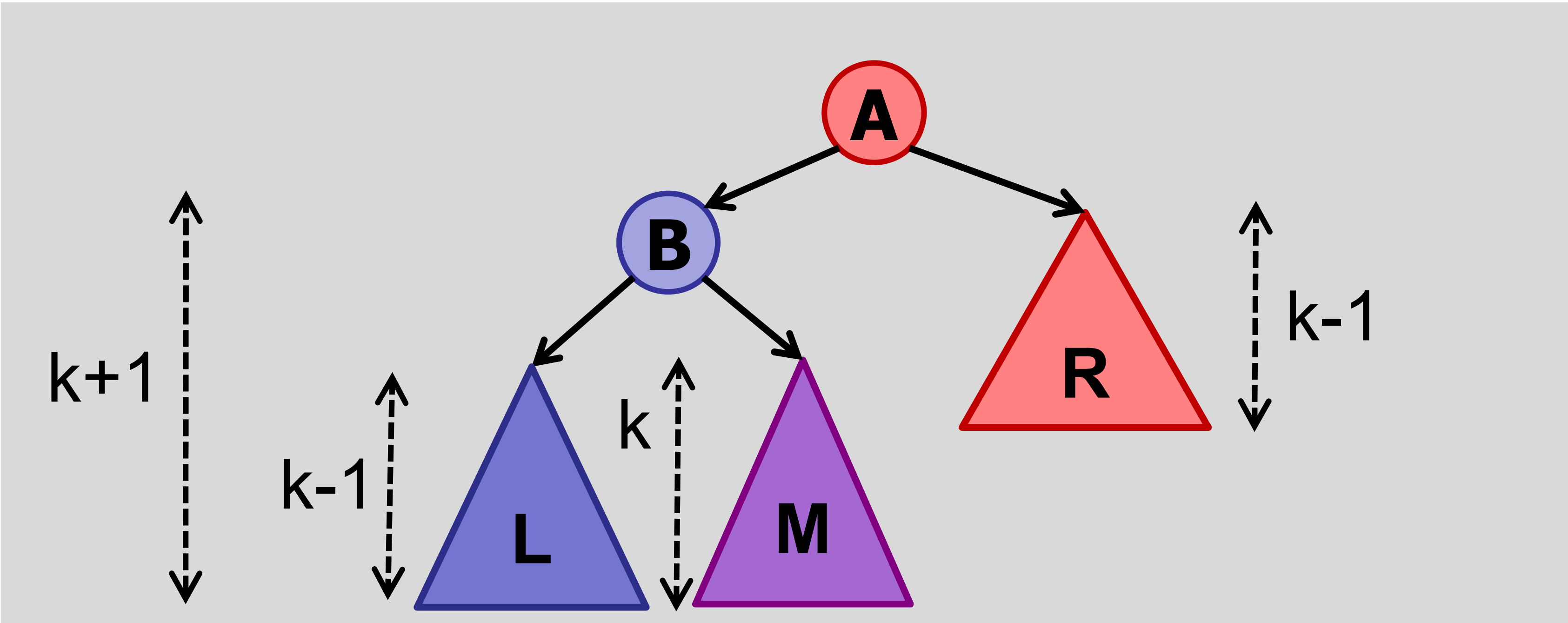


right-rotate:

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

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

Tree Rotations (Left Heavy)

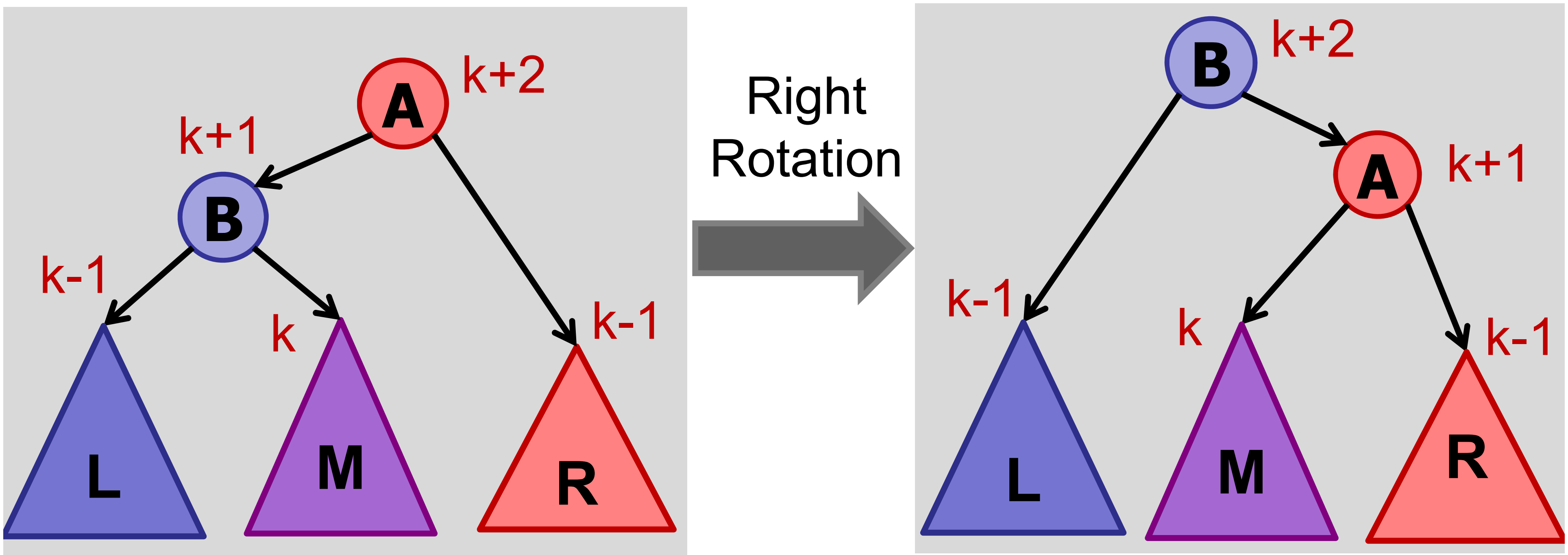


Assume **A** is the lowest node in the tree violating balance property.

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

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

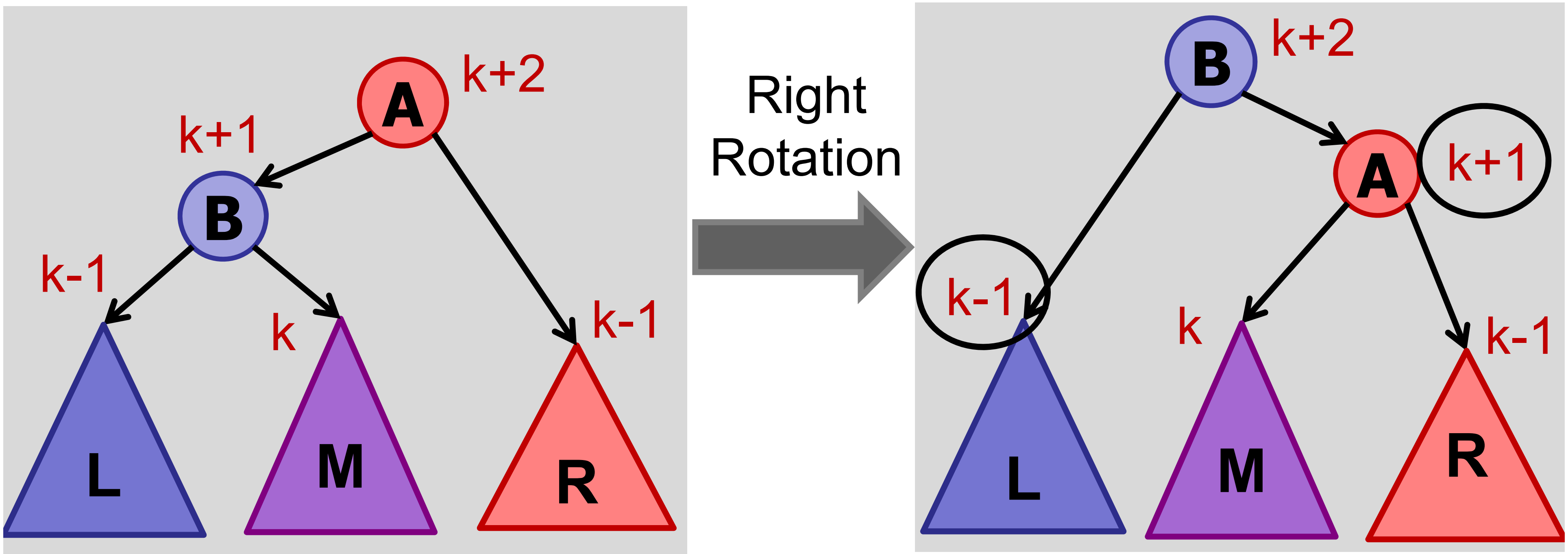
Tree Rotations



right-rotate:

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

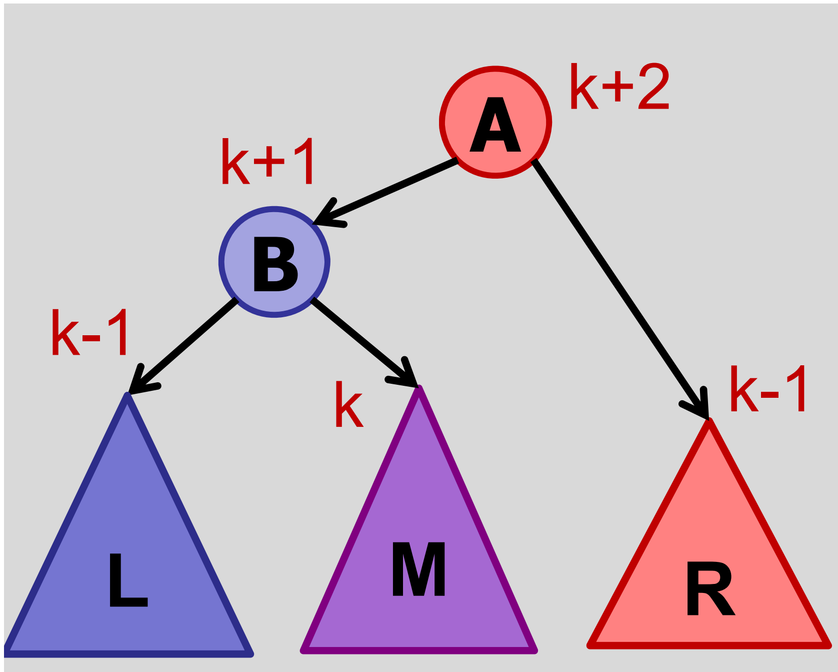
h(**R**) = h(**L**)



Are we done?

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

Tree Rotations



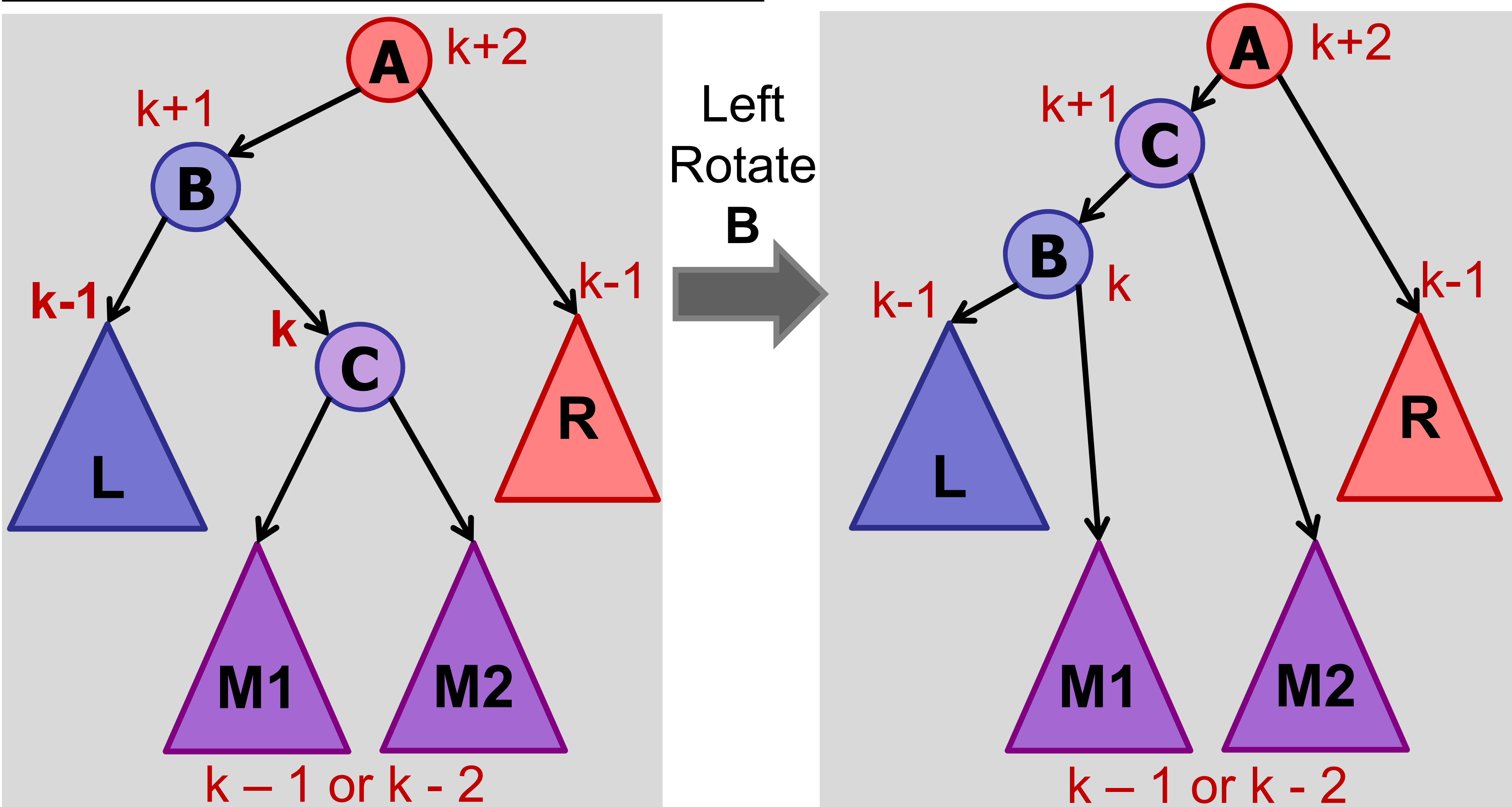
Let's do something first before we right-rotate(A)

right-rotate:

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

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

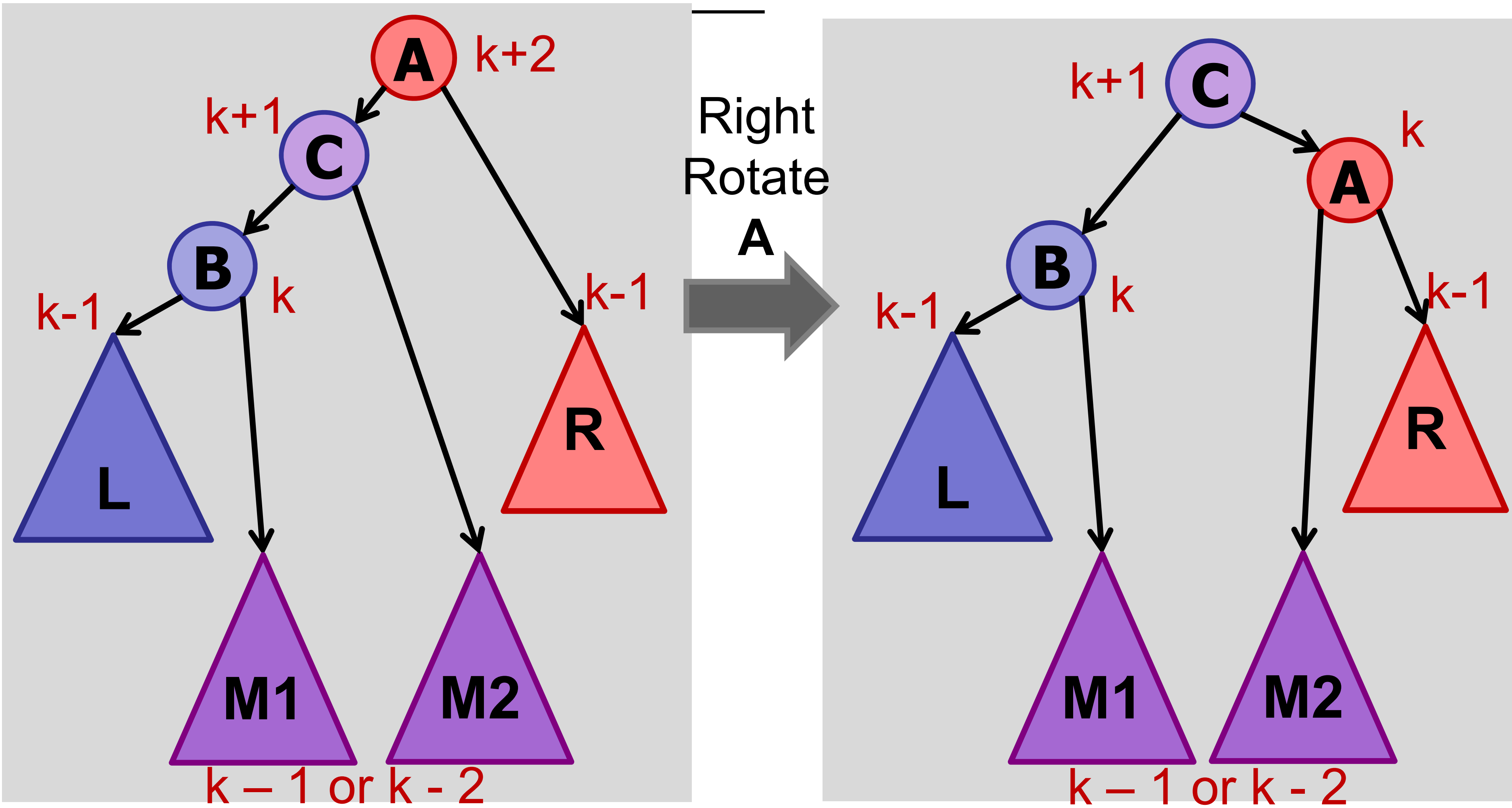
Tree Rotations



Left-rotate B

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

Tree Rotations



After right-rotate A: 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….

How many rotations do you need after an insertion (in the worst case)?

1. 1
2. 2
3. 4
4. log(n)
5. 2log(n)
6. n

How many rotations do you need after an insertion (in the worst case)?

1. 1
2. 2
3. 4
4. log(n)
5. 2log(n)
6. n

Question:

Why isn't it 2log(n)?

Insert in AVL Tree

Summary:

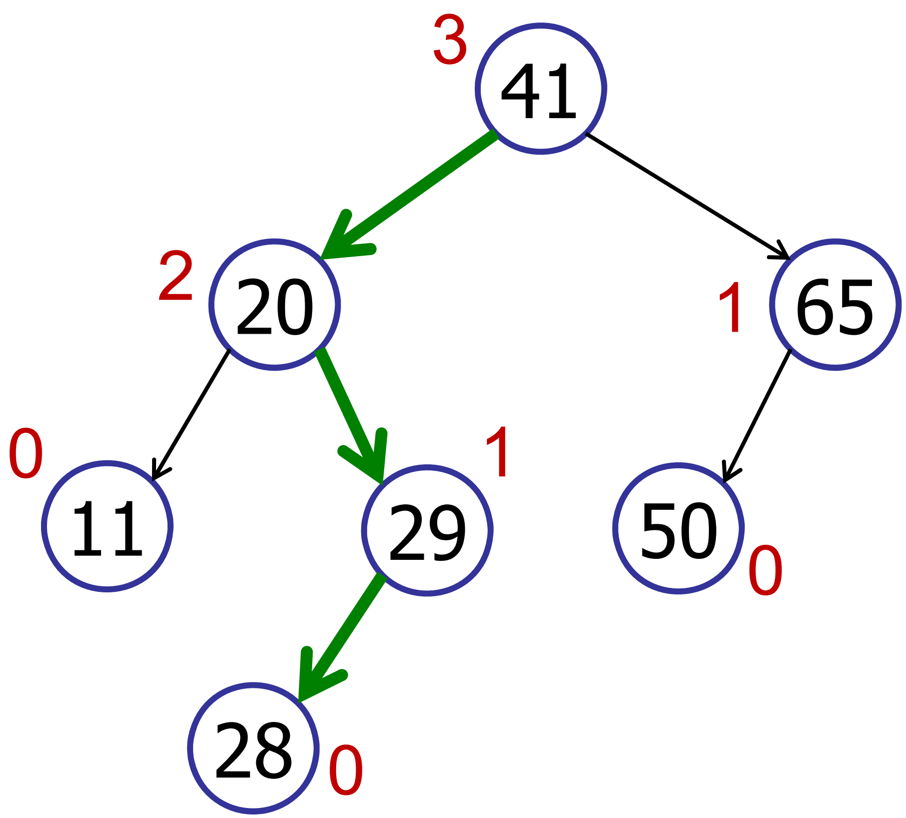
- Insert key in BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance.

Note: only need to perform two rotations

- Why?
- In each case, reduce height of sub-tree by 1
- What about Case 1, above?

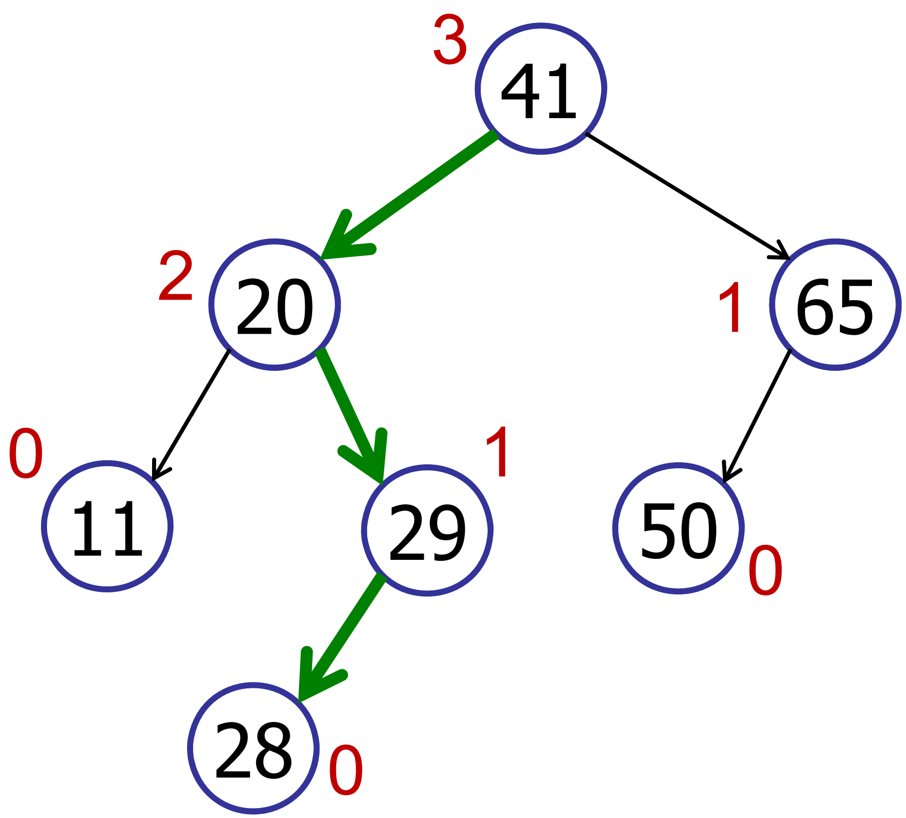
Example

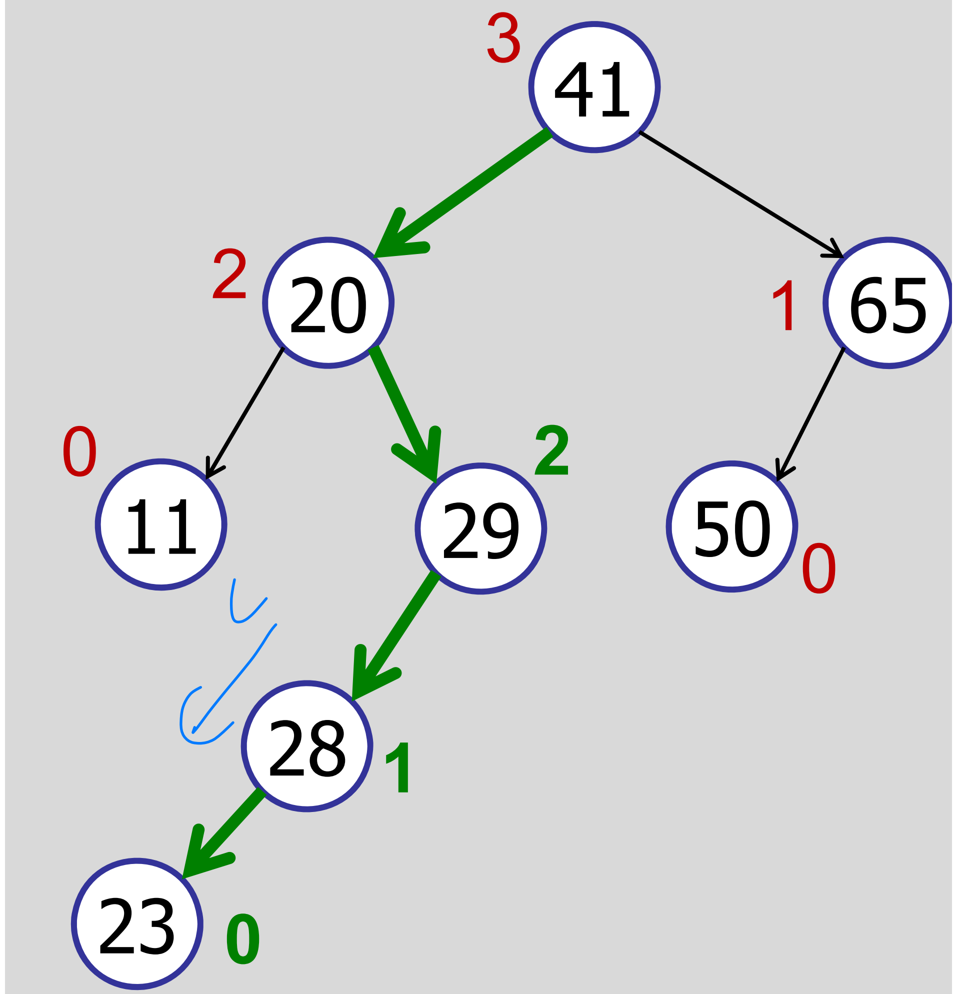
insert(23)



Example

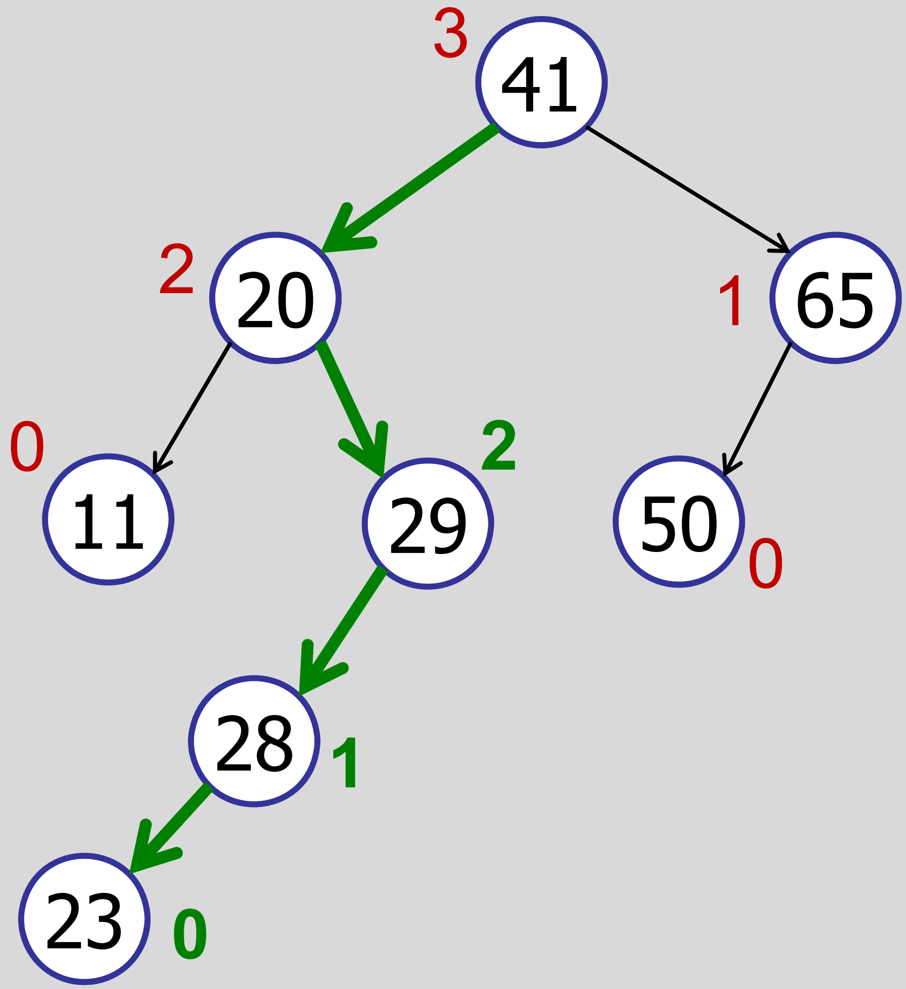
insert(23)

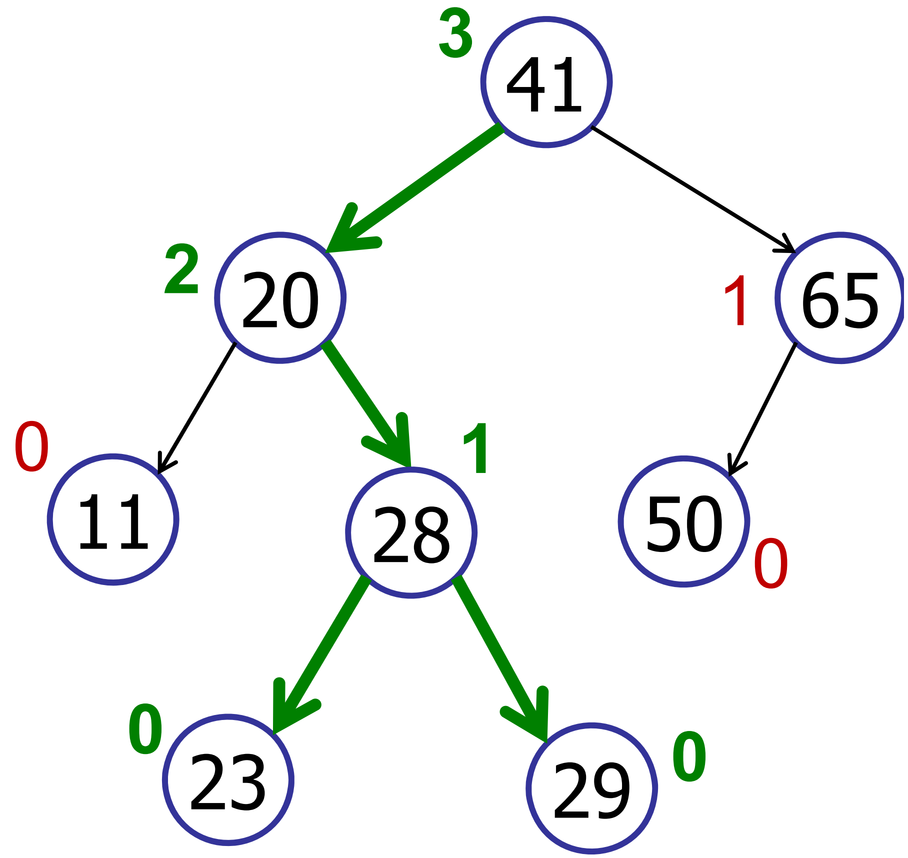




Example

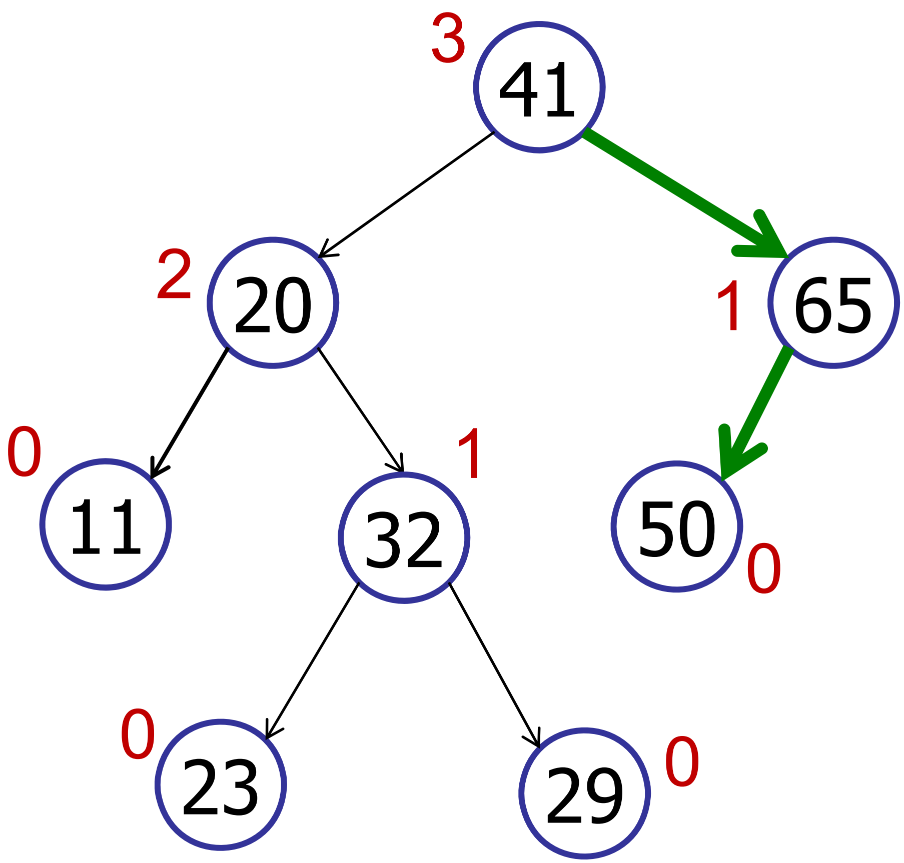
right-rotate(29)





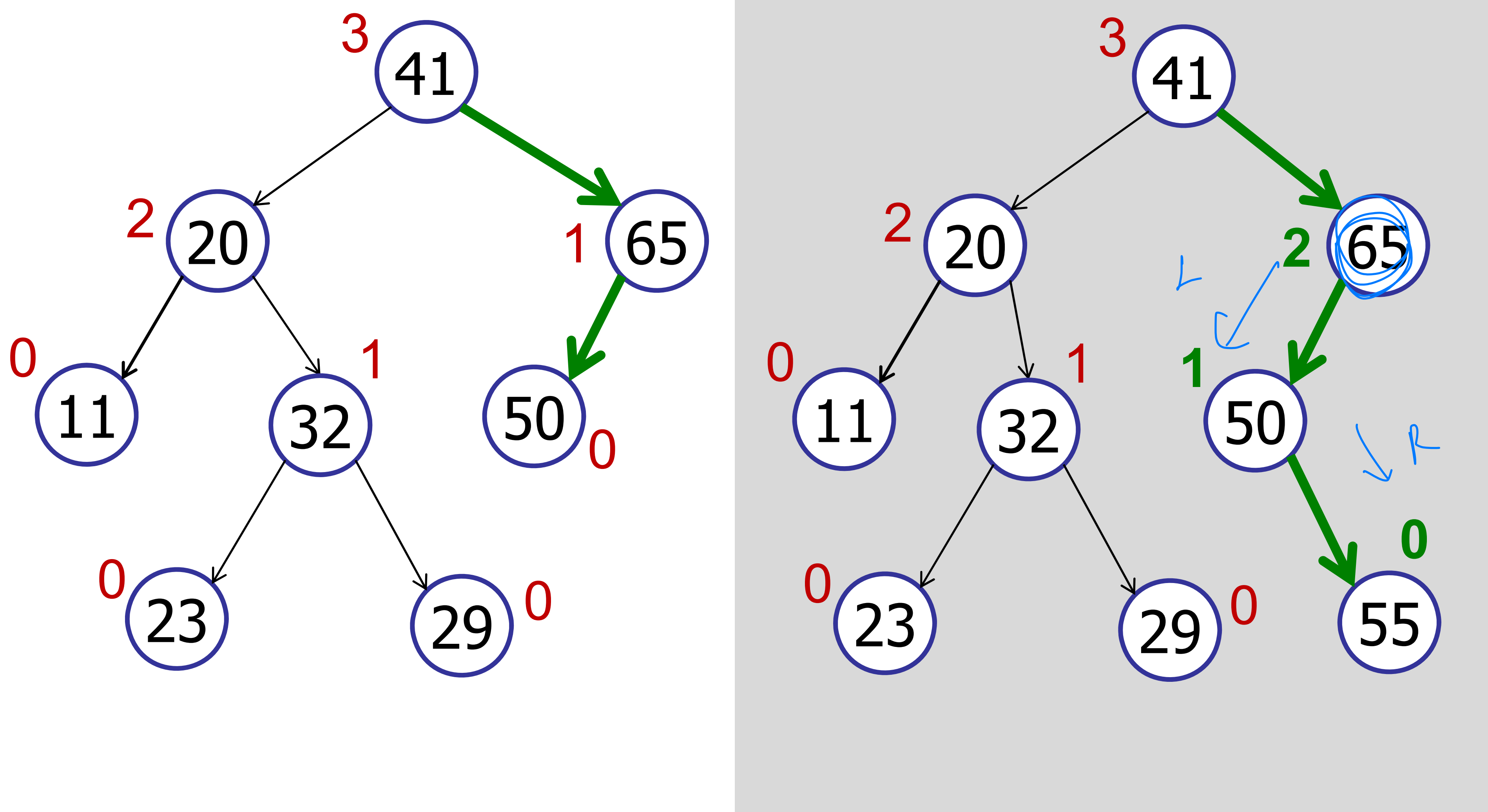
Example

insert(55)



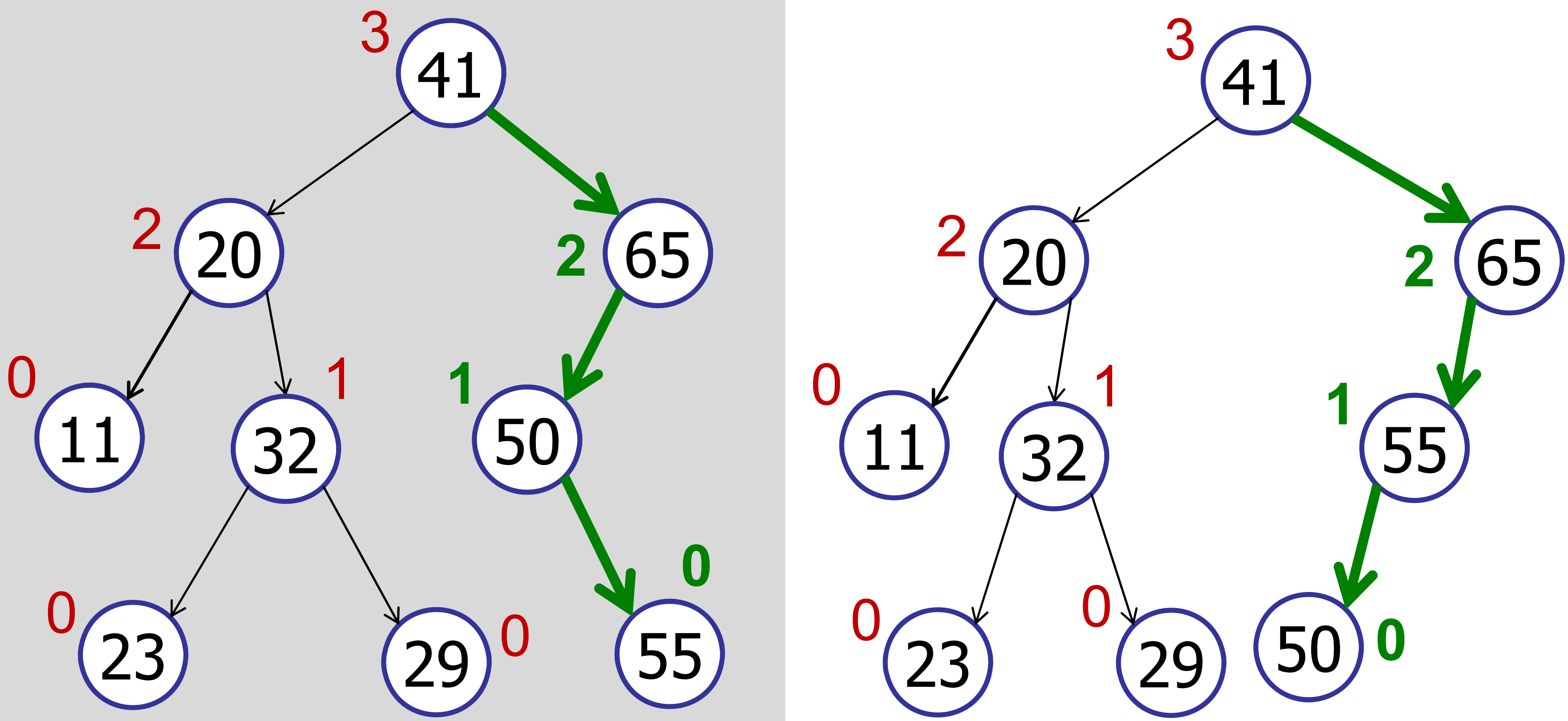
Example

insert(55)



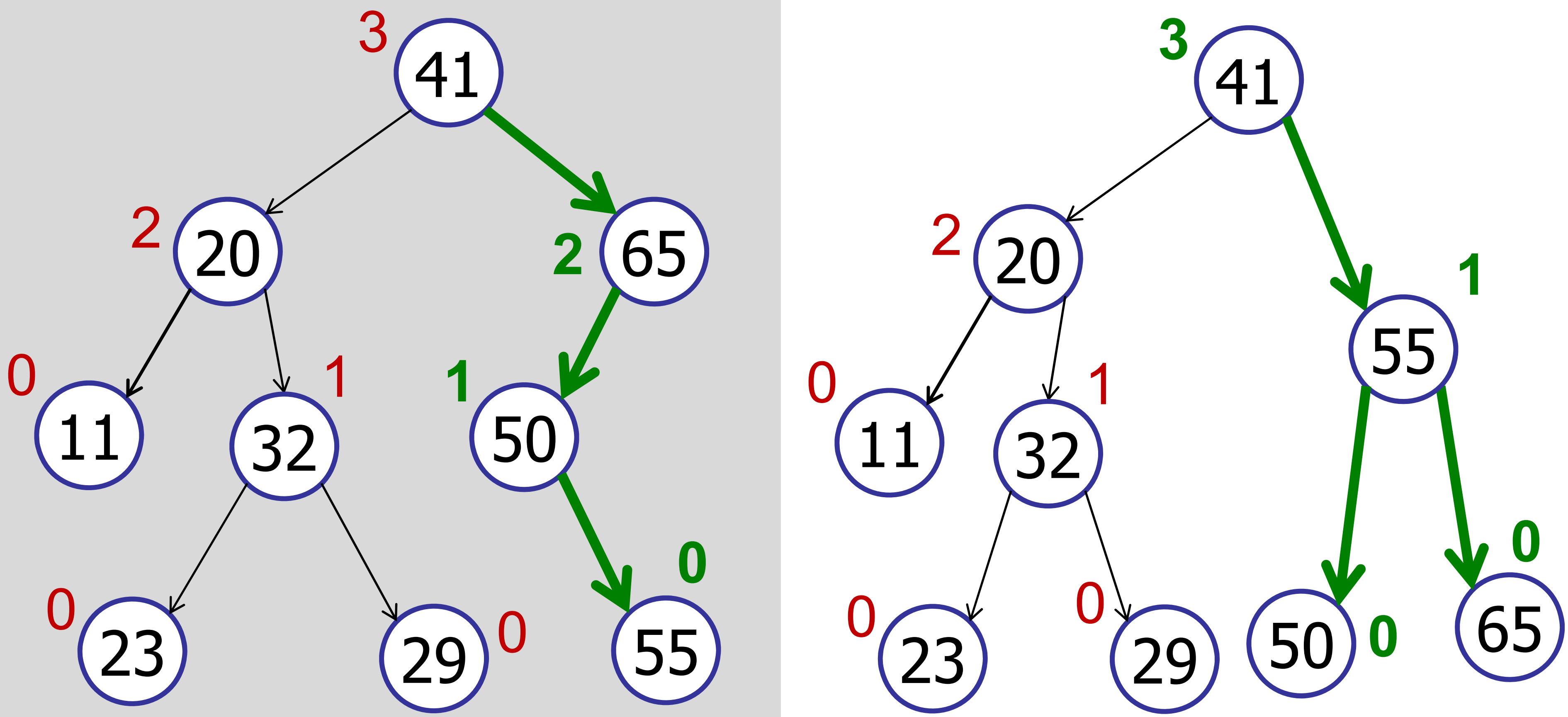
Example

left-rotate(50)



Example

right-rotate(65)

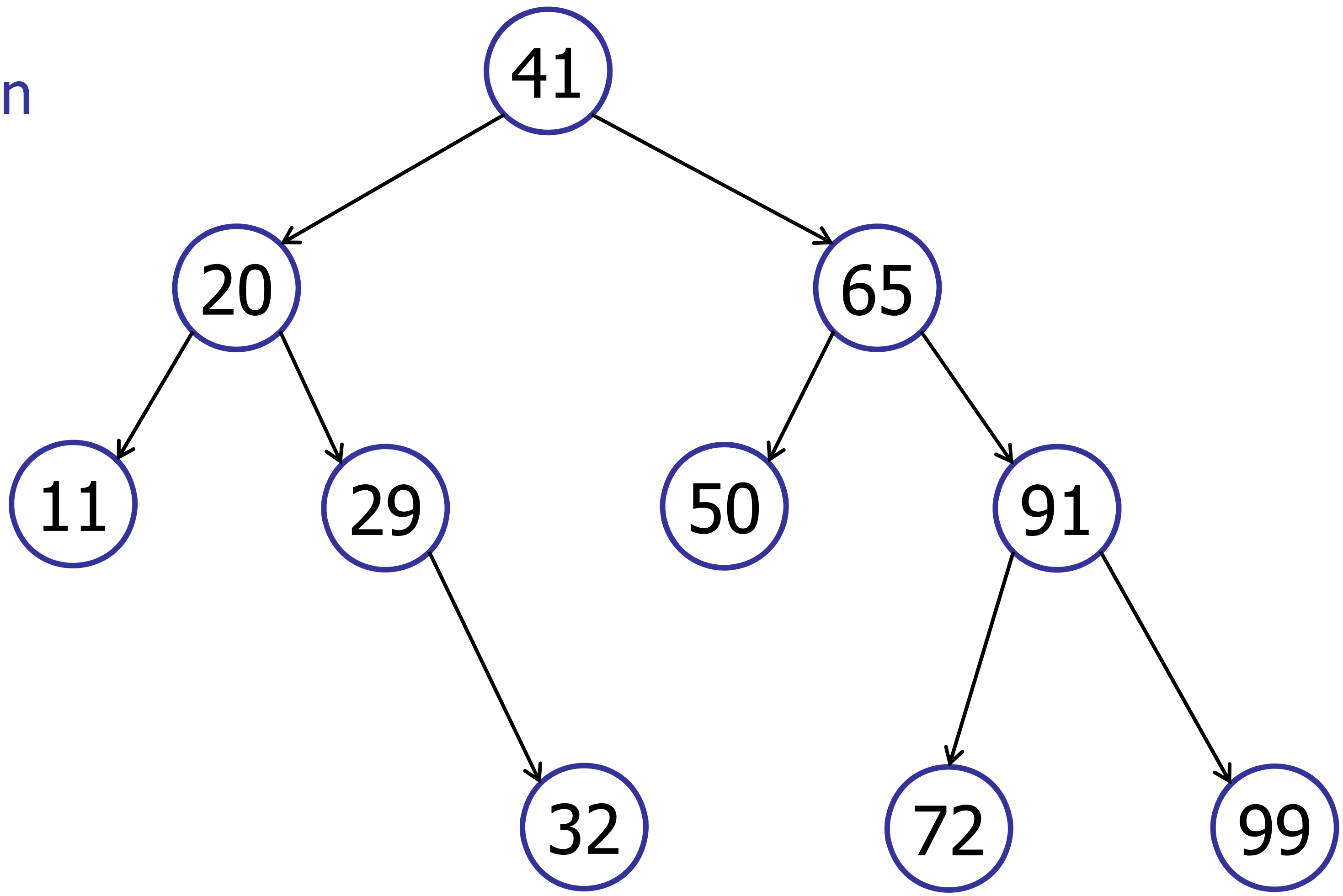


Binary Search Tree

delete(v)

Three cases:

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



Binary Search Tree

delete(v)

1. If v has two children, swap it with its successor.

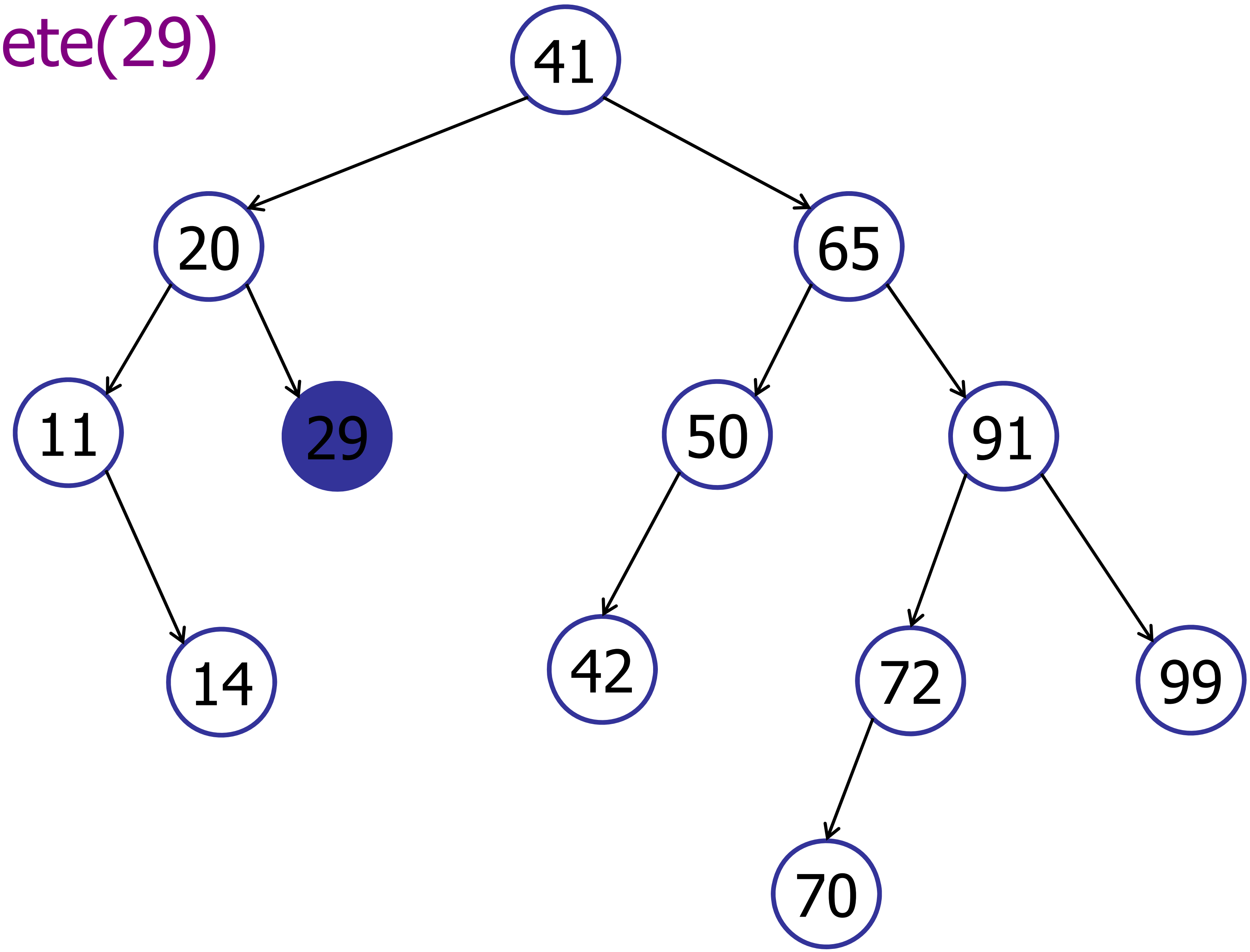
2. Delete node v from binary tree (and reconnect children).

3. For every ancestor of the deleted node:
   - Check if it is height-balanced.
   - If not, perform a rotation.
   - Continue to the root.

Deletion may take up to O(log(n)) rotations.

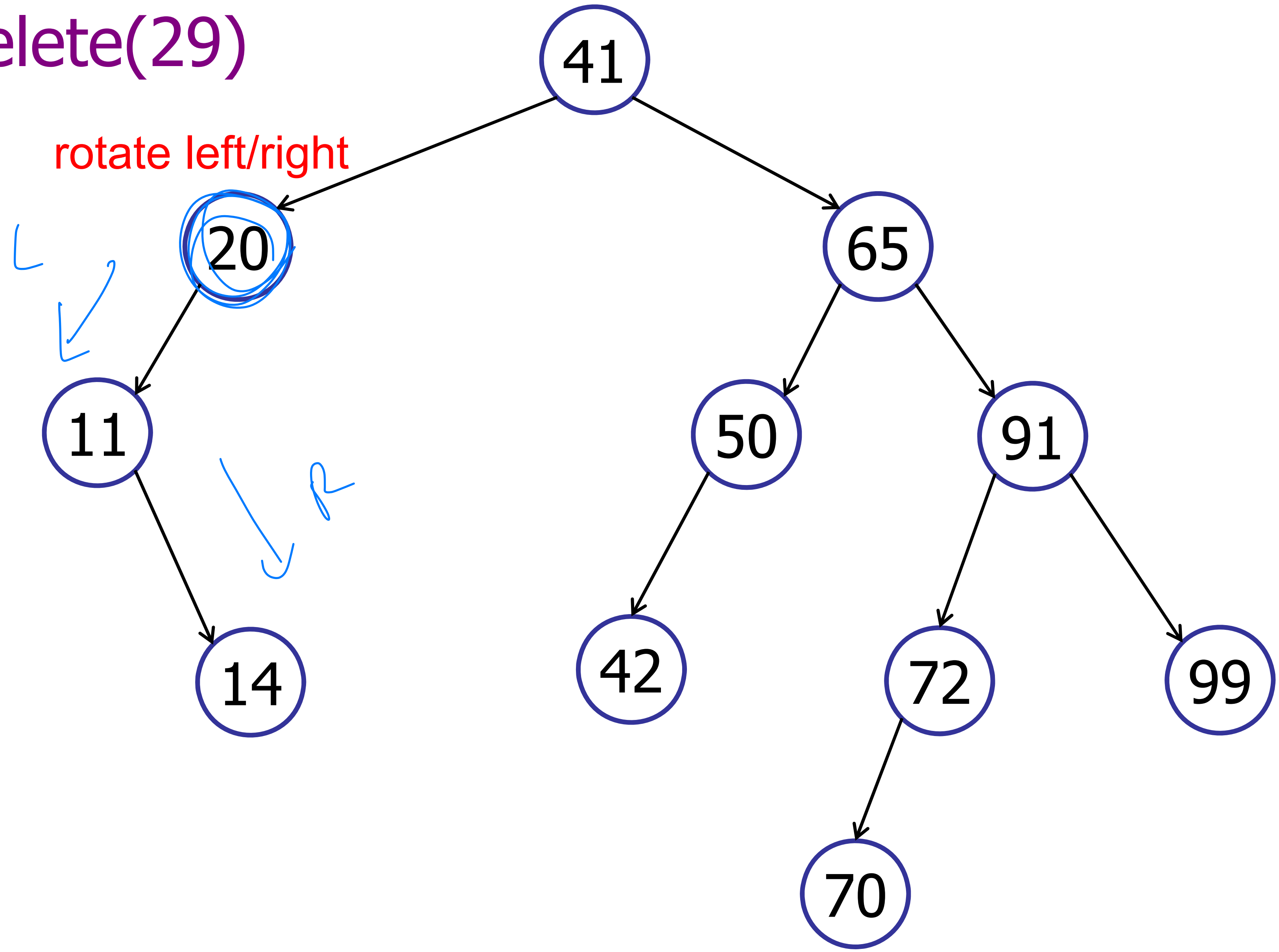
Binary Search Tree

delete(29)



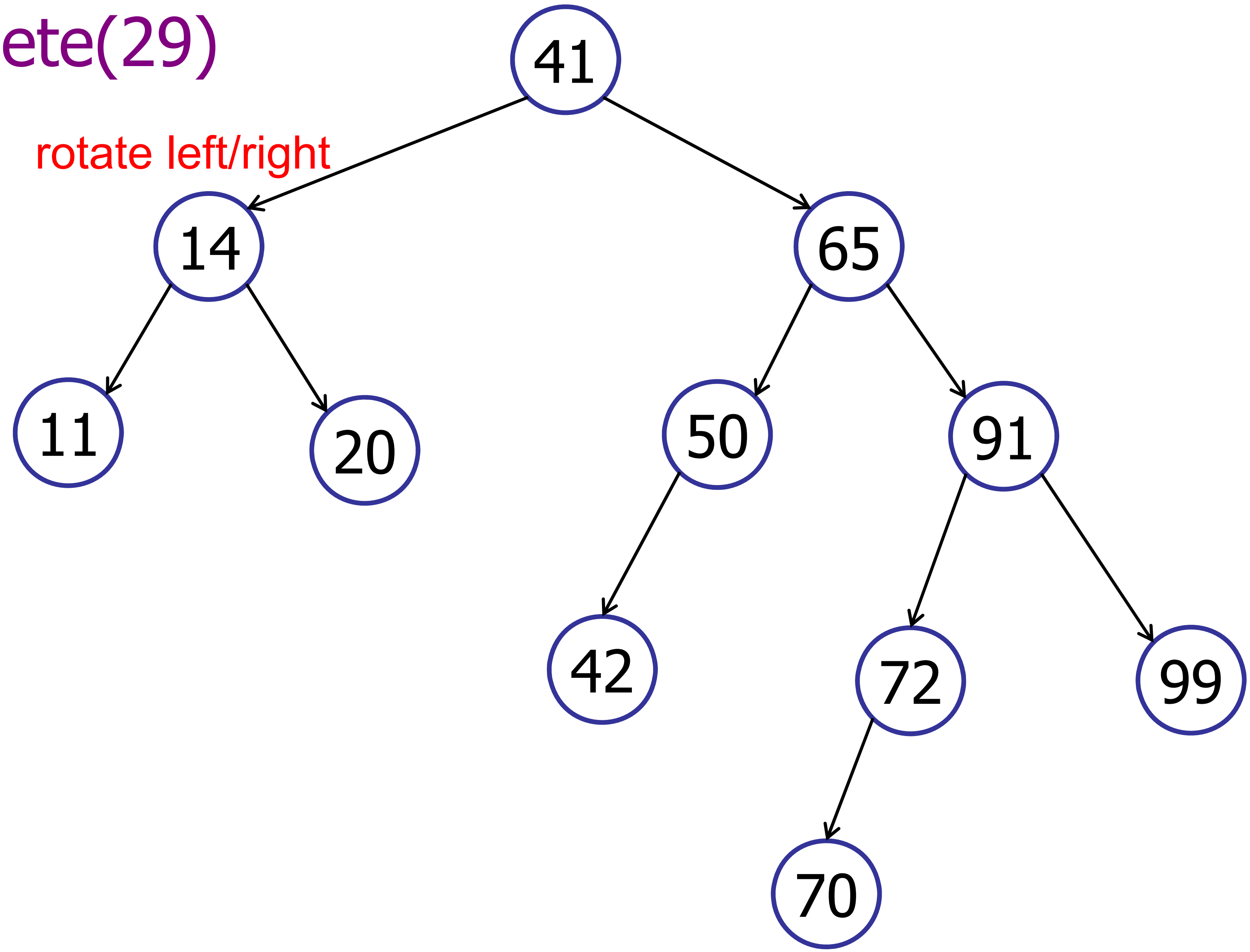
Binary Search Tree

delete(29)



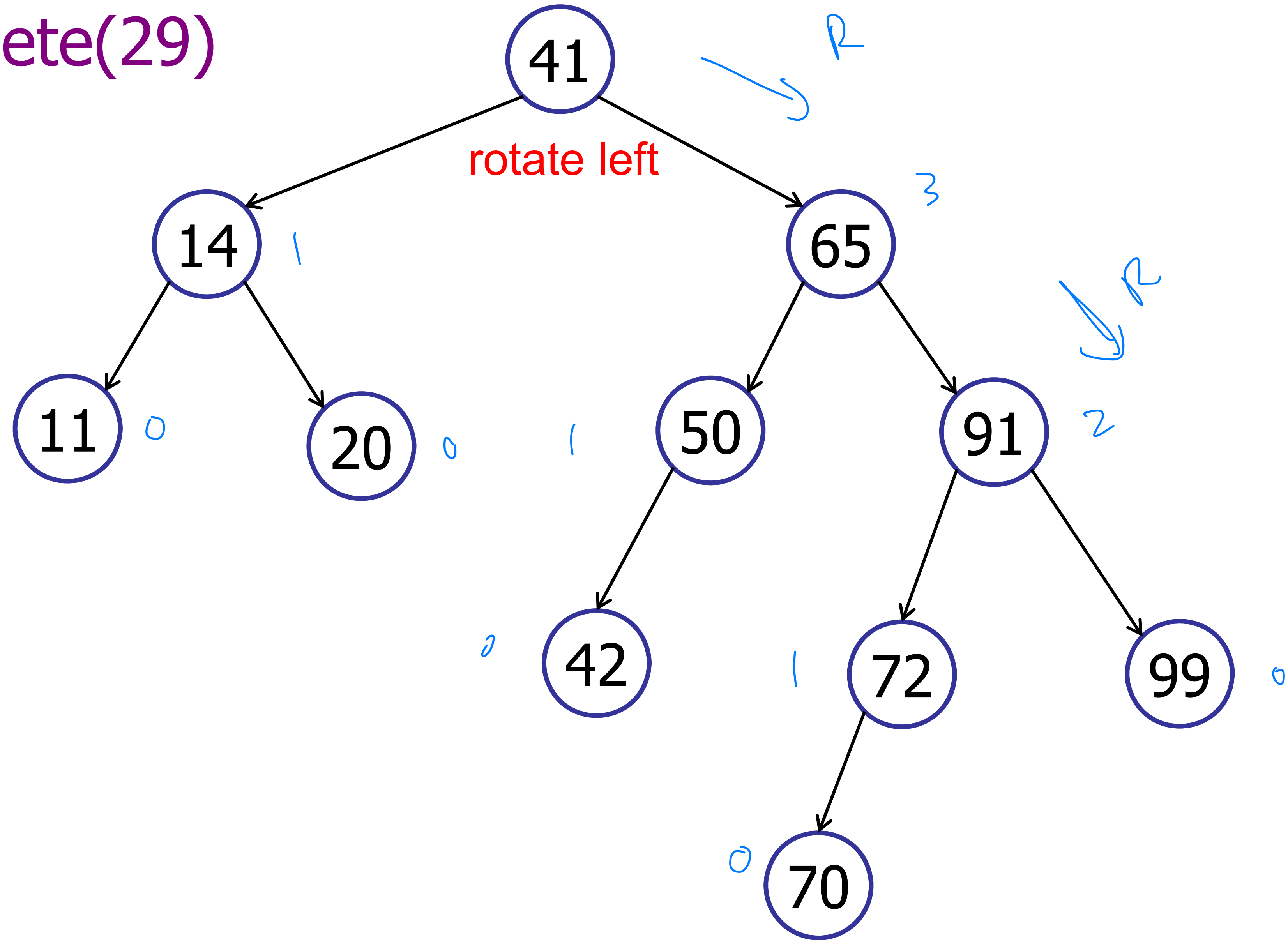
Binary Search Tree

delete(29)



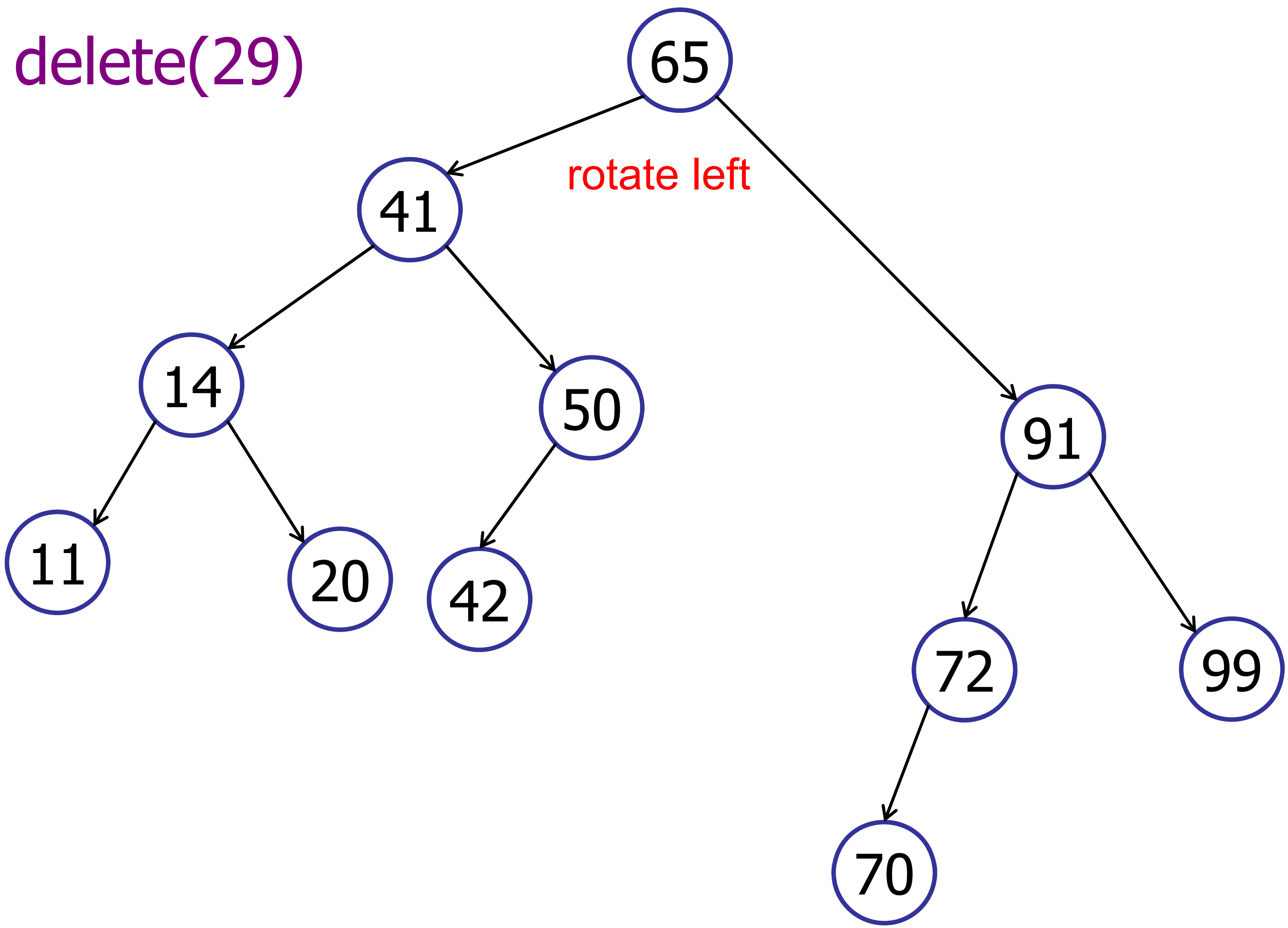
Binary Search Tree

delete(29)



Binary Search Tree

delete(29)



Quick review: a rotation costs:

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

Every insertion requires 1 or 2 rotations?

1. Yes
2. No
3. I don’t know

Using rotations, you can create every possible "tree shape."

✓ 1. True
2. False
3. I don't know

AVL Trees

What if you do not remove deleted nodes?

- Mark a node "deleted" and leave it in the tree.

Logical deletes:

- Performance degrades over time.
- Clean up later? (Amortized performance…)

AVL Trees

What if you do not want to store the height in every node?

– Only store difference in height from parent.

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)

Balanced Search Trees

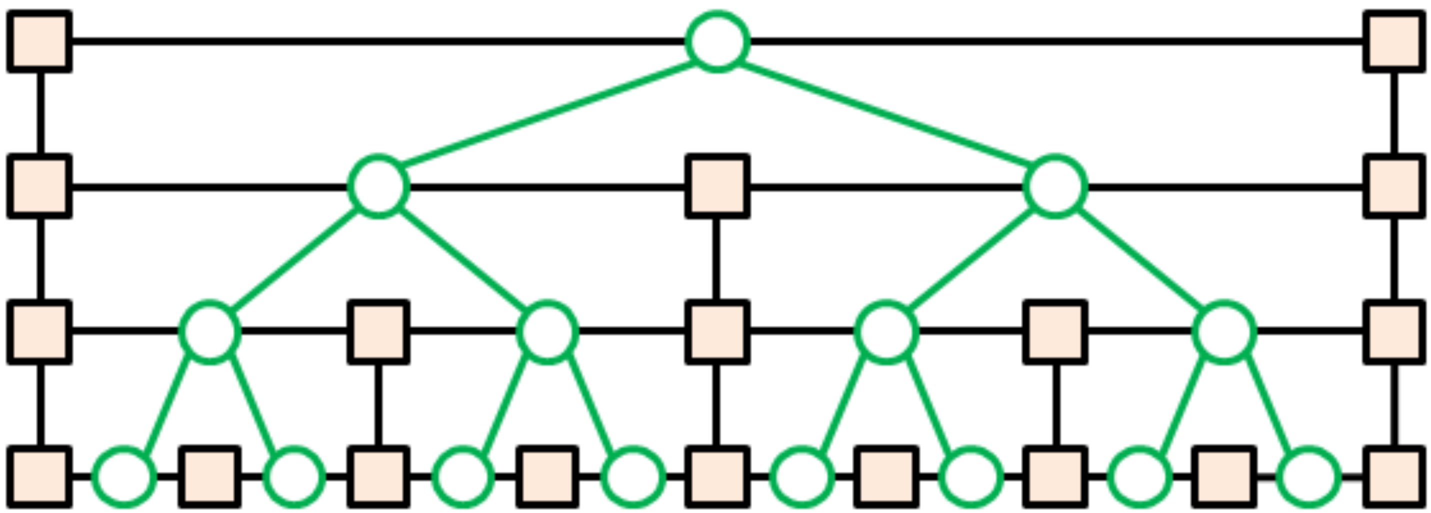
Red-Black trees

- More loosely balanced
- Rebalance using rotations on insert/delete
- O(1) rotations for all operations.
- Java TreeSet implementation
- Faster (than AVL) for insert/delete
- Slower (than AVL) for search

Balanced Search Trees

Skip Lists and Treaps

- Randomized data structures
- Random insertions => balanced tree
- Use randomness on insertion to maintain balance



Plan of the Day

Trees

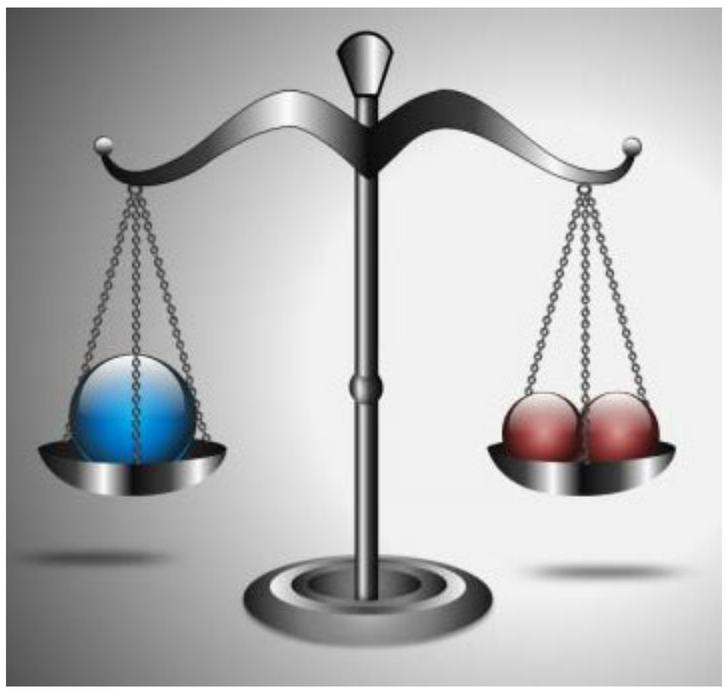
- Terminology
- Traversals
- Operations

Balanced Trees

- Height-balanced binary search trees
- AVL trees
- Rotations

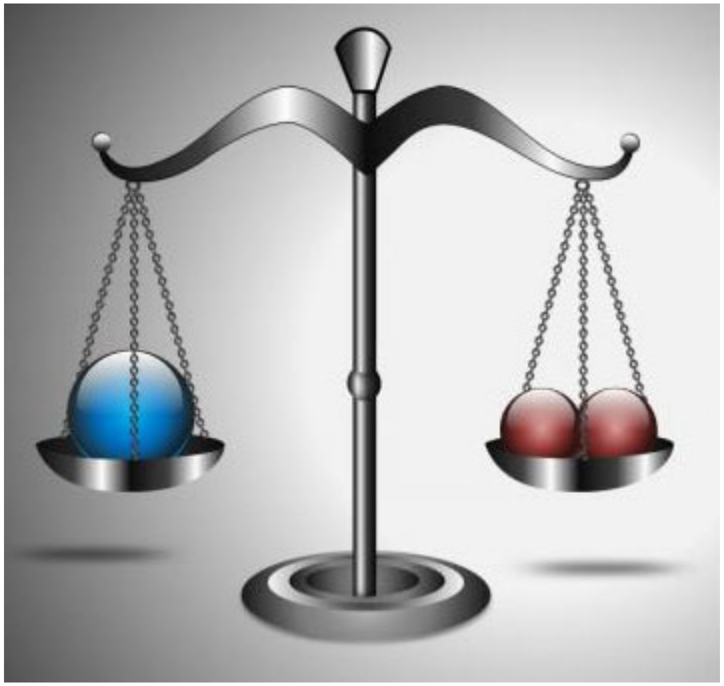
Puzzle Break

If you are given 8 balls, they all look identical and one of them is heavier. Can you tell me which one is different by using the scale balance only three times?



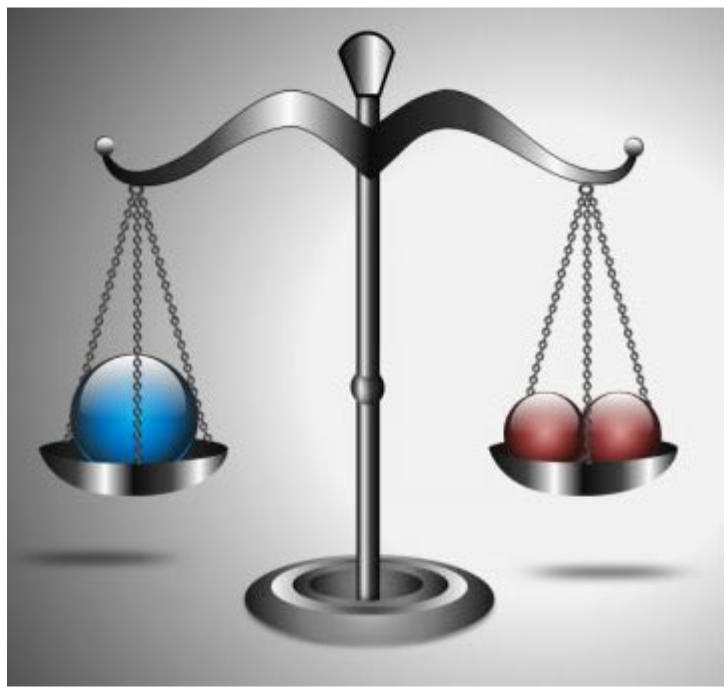
Puzzle Break

If you are given 8 balls, they all look identical and one of them is heavier. Can you tell me which one is different by using the scale balance only two times?

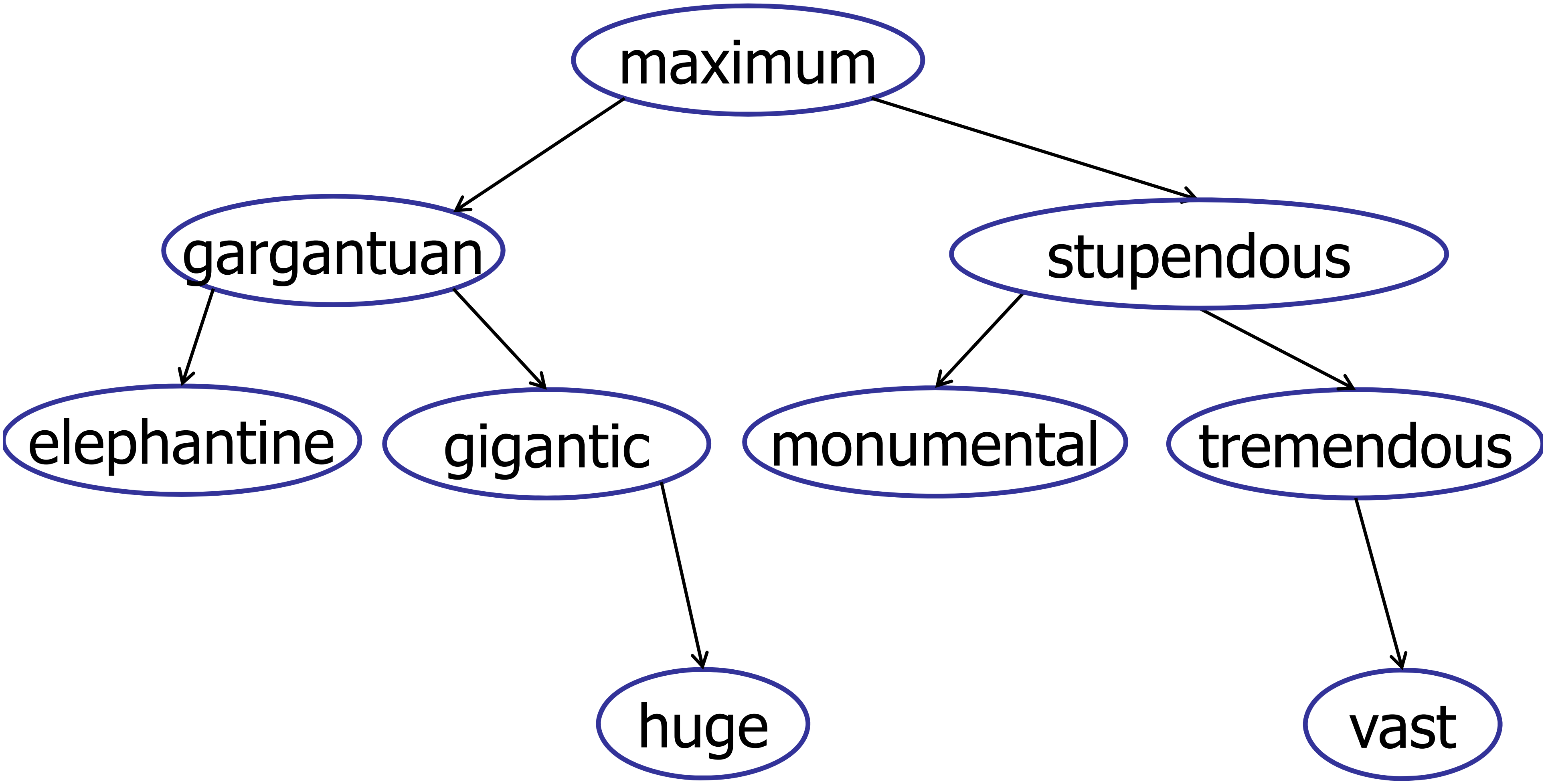


Puzzle Break

If you are given 12 balls, they all look identical and one of them has a different weight. Can you tell me which one is different by using the scale balance only three times?



What about text strings?



Implement a searchable dictionary!

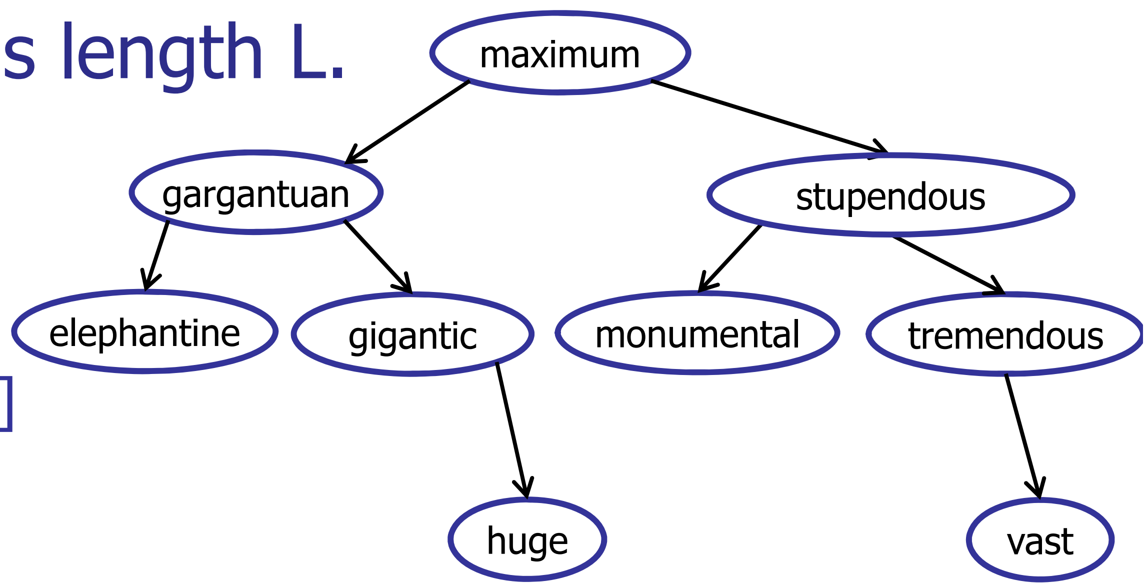
What about text strings?

Cost of comparing two strings:

- Cost[A ?? B] = min(A.length, B.length)
- Compare strings letter by letter (?)

Cost of tree operation:

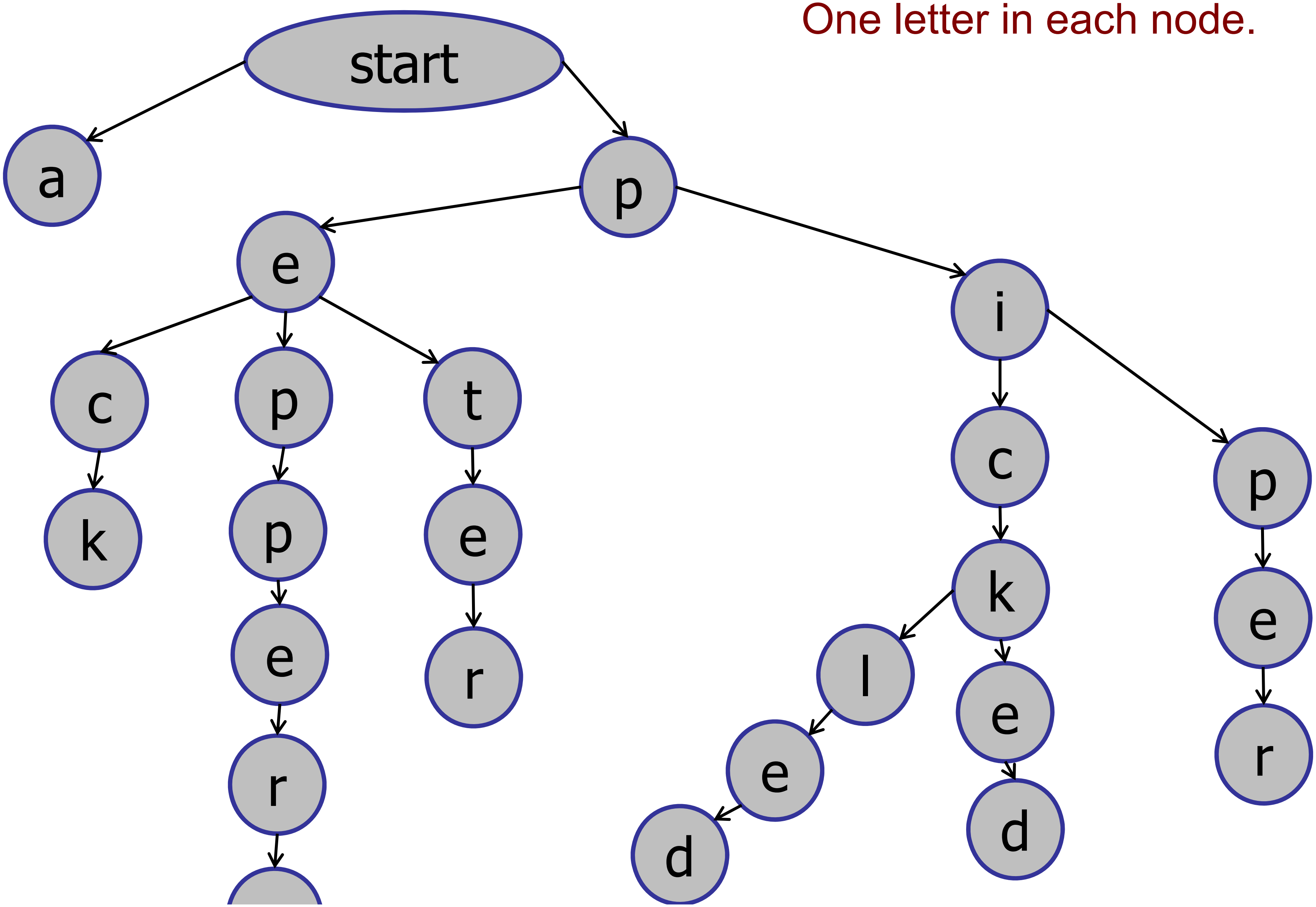
- Assume string has length L.
- Cost: O(hL)



[Optimizations are possible.]

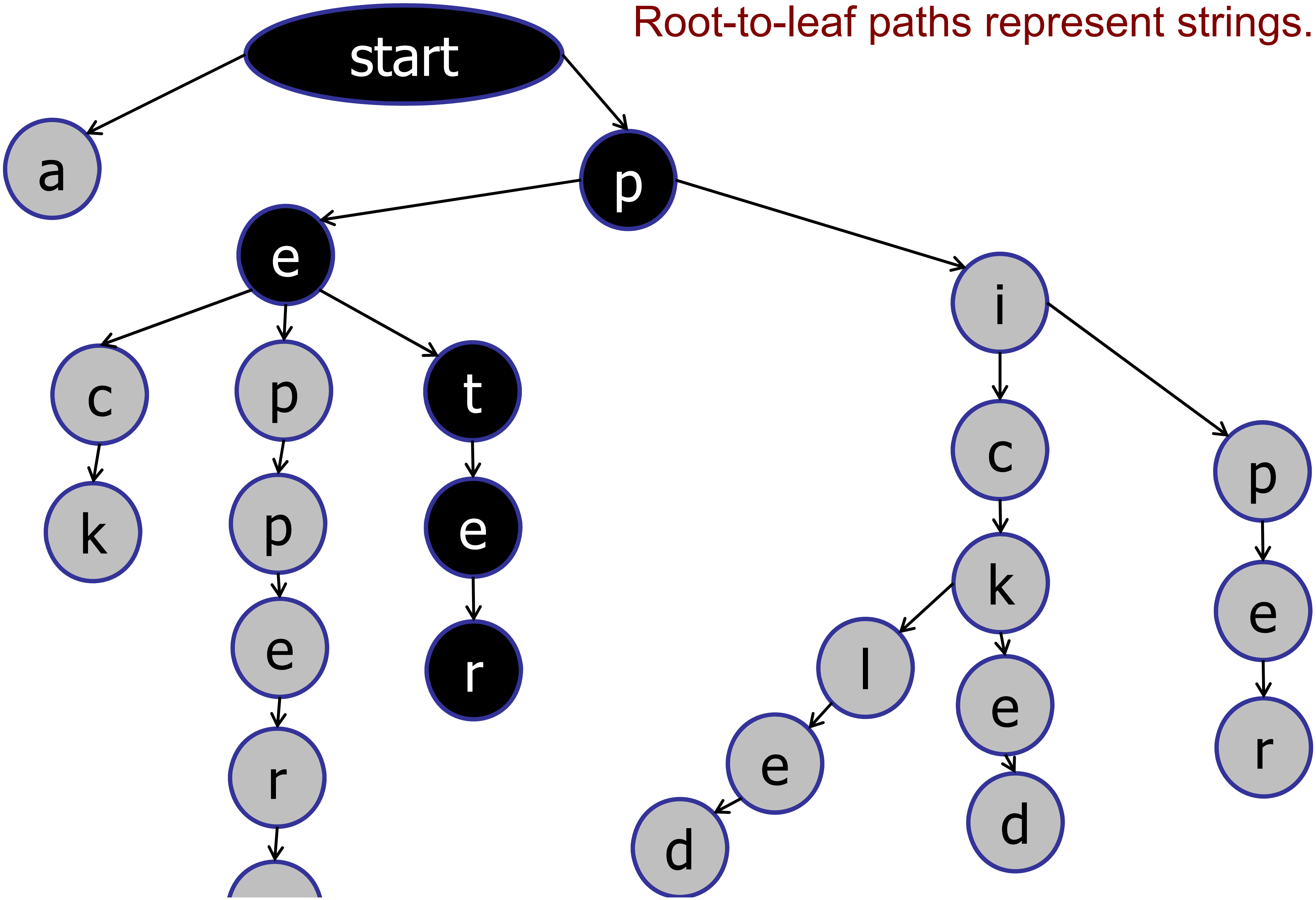
Trie [prounounced: try]

One letter in each node.



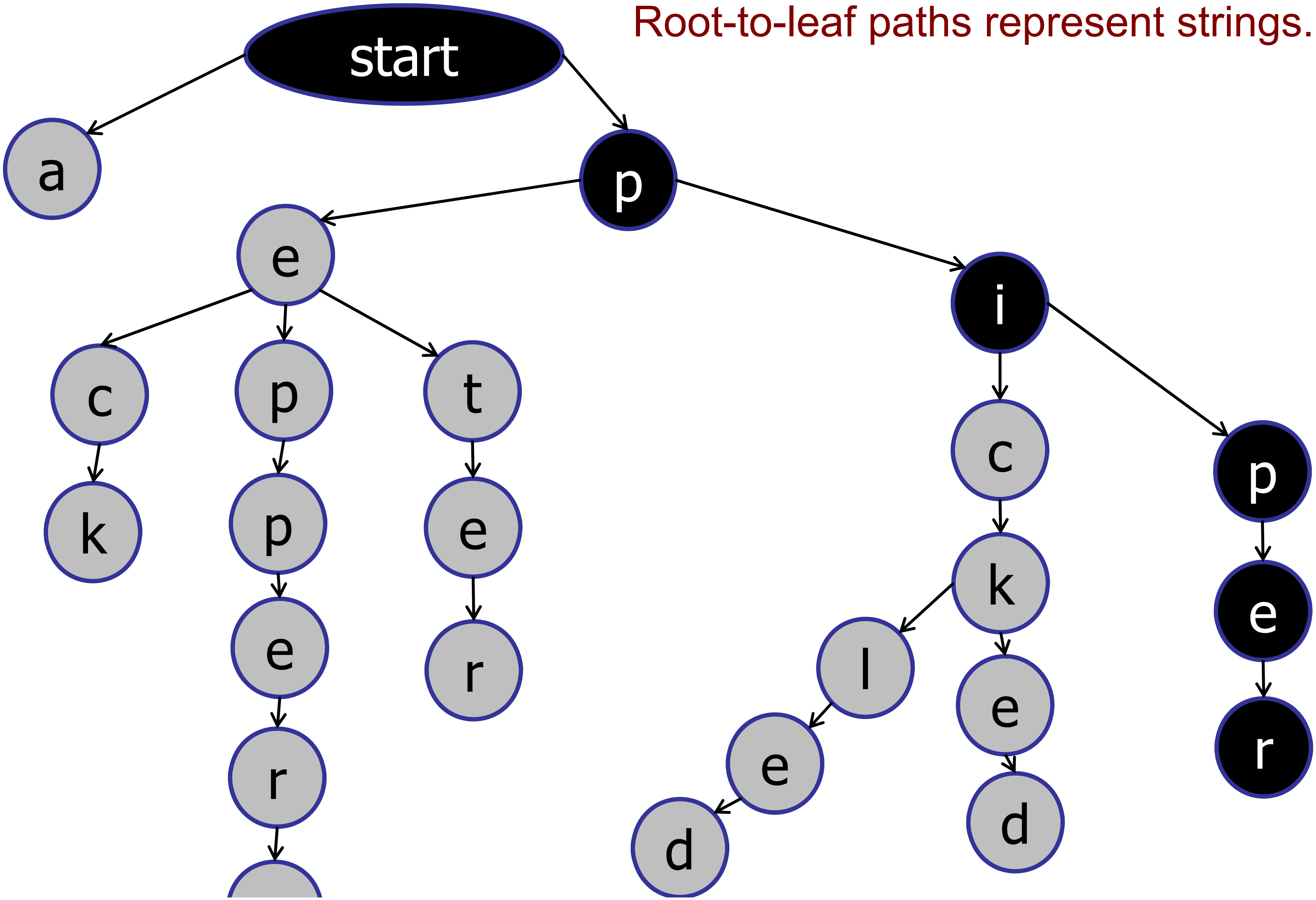
Trie [prounounced: try]

Root-to-leaf paths represent strings.



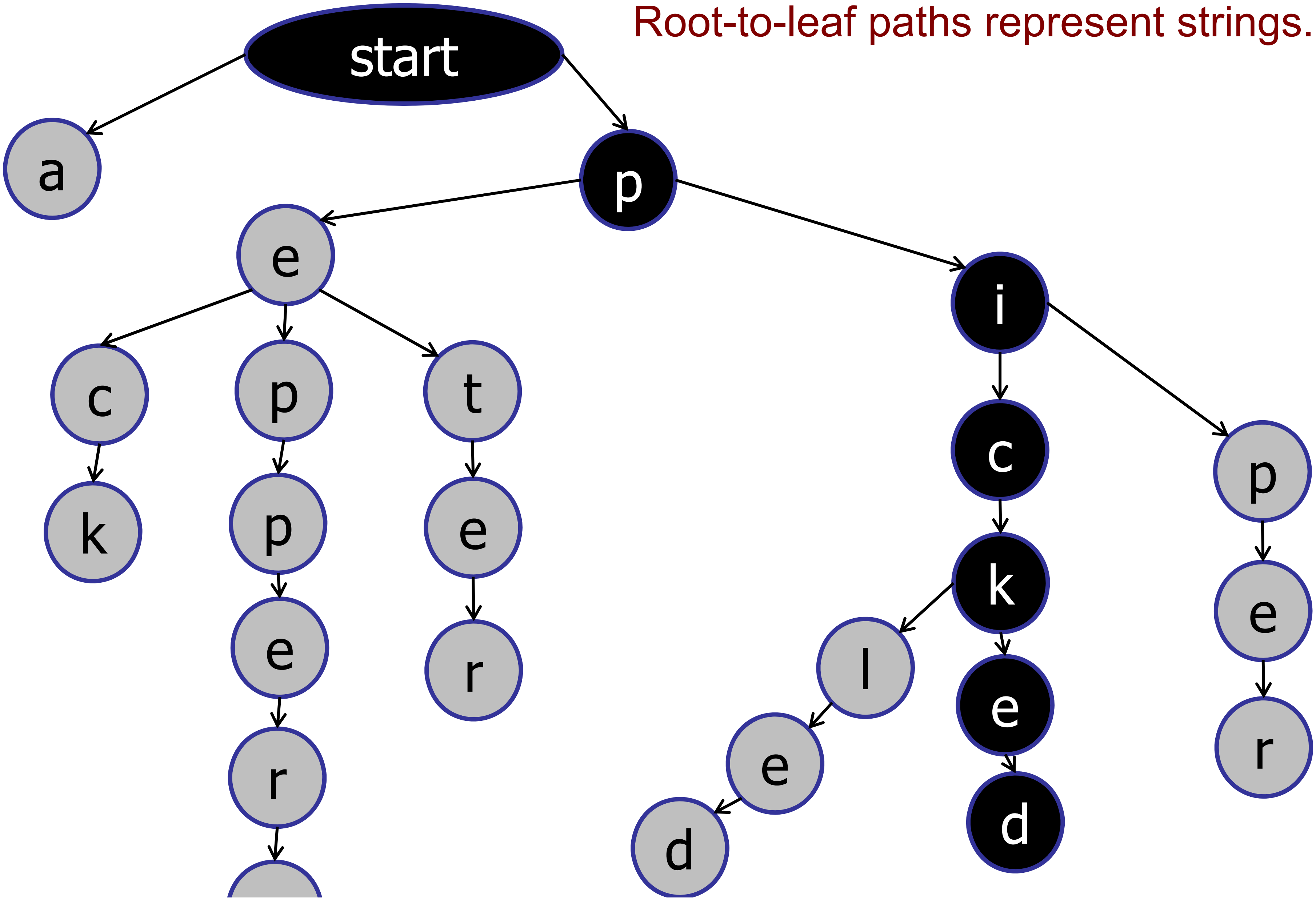
Trie [prounounced: try]

Root-to-leaf paths represent strings.



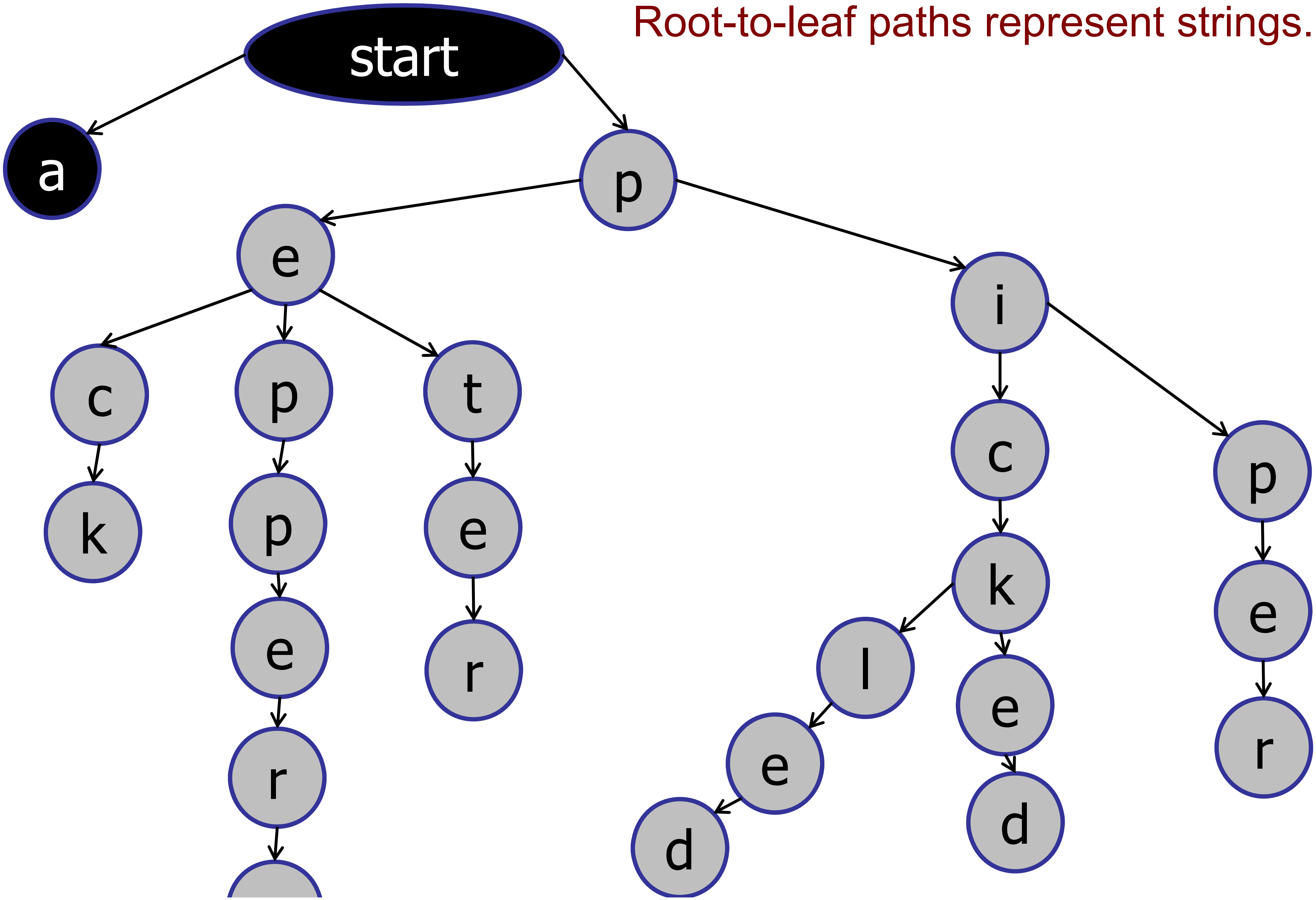
Trie [prounounced: try]

Root-to-leaf paths represent strings.



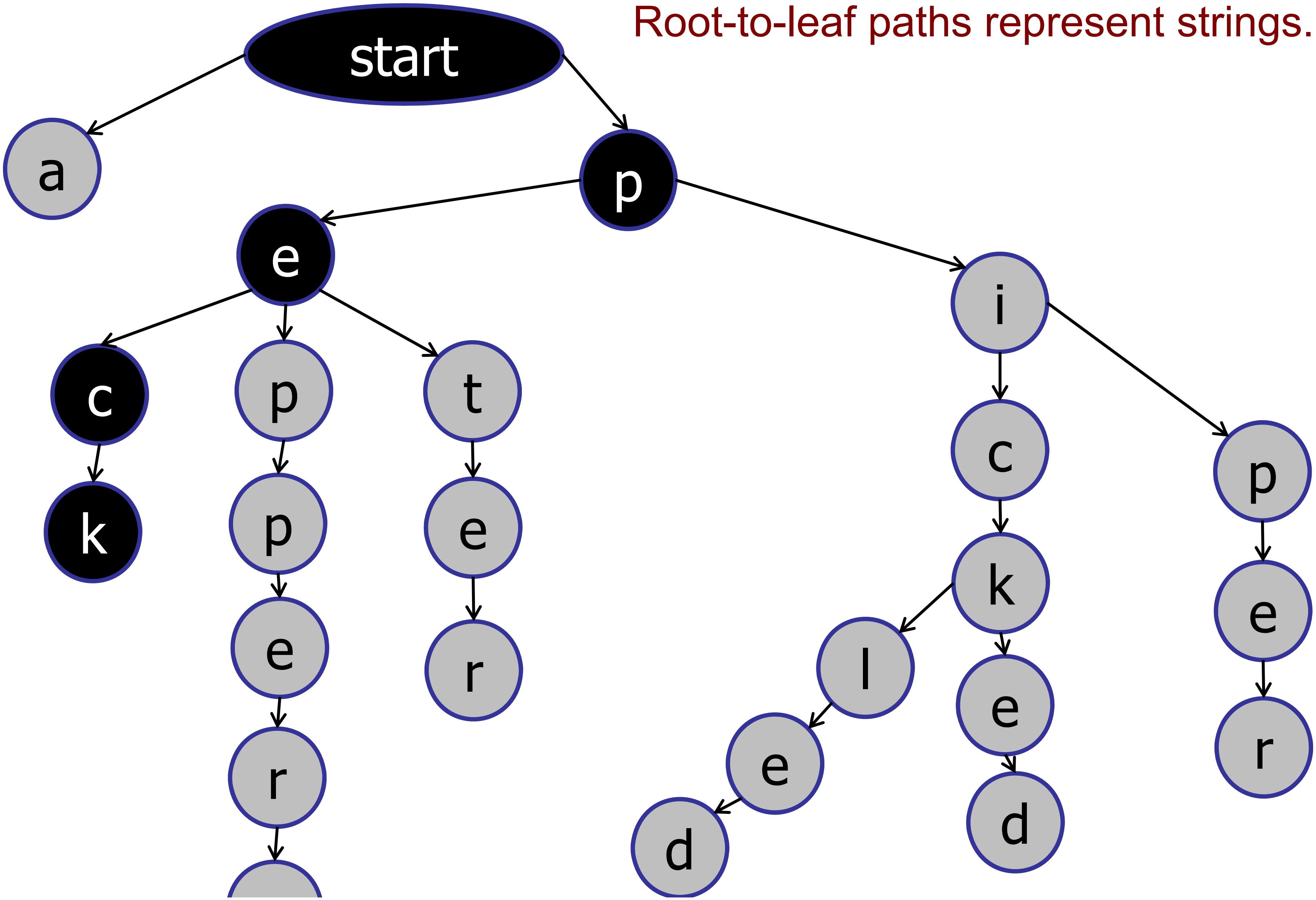
Trie [prounounced: try]

Root-to-leaf paths represent strings.



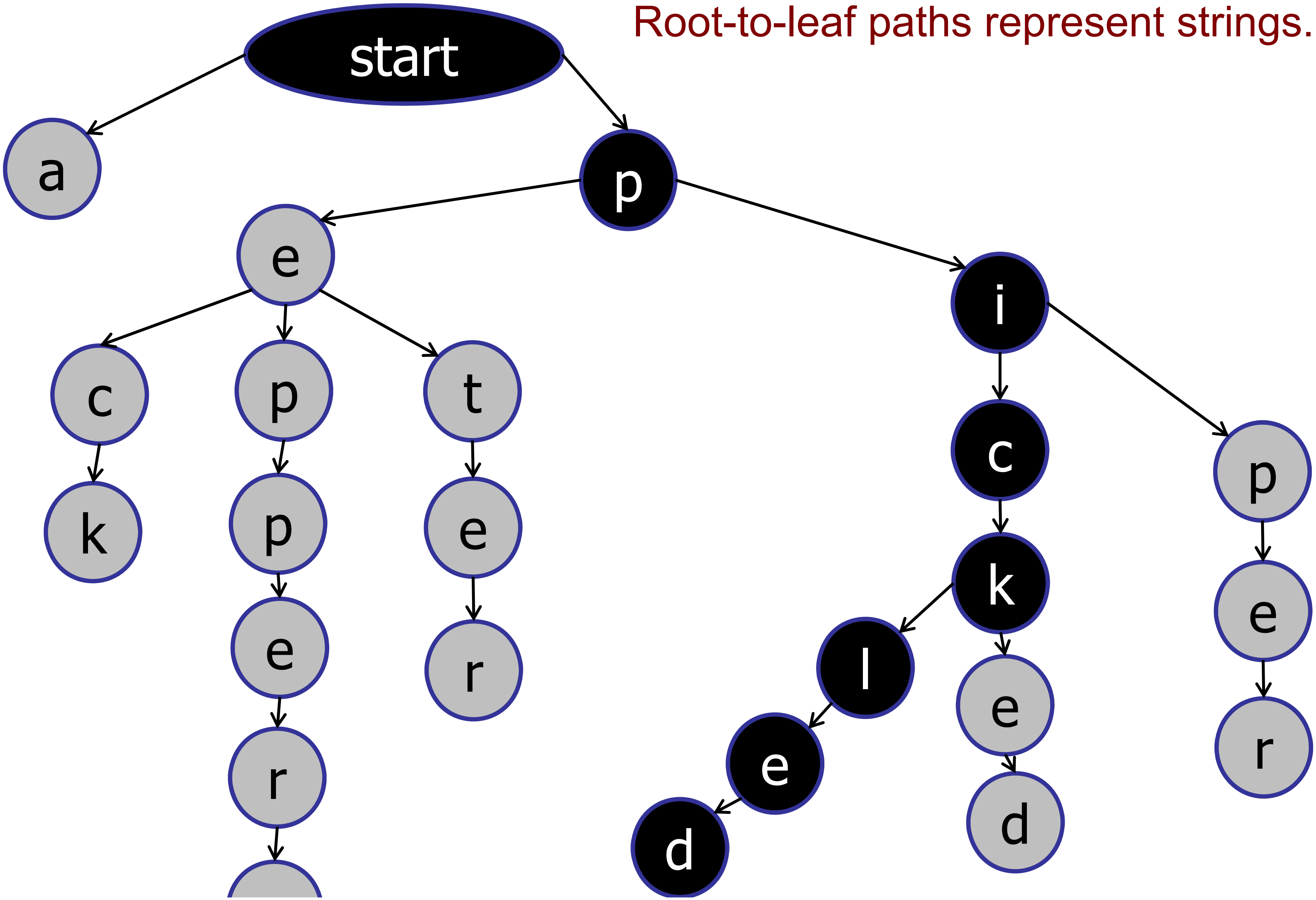
Trie [prounounced: try]

Root-to-leaf paths represent strings.



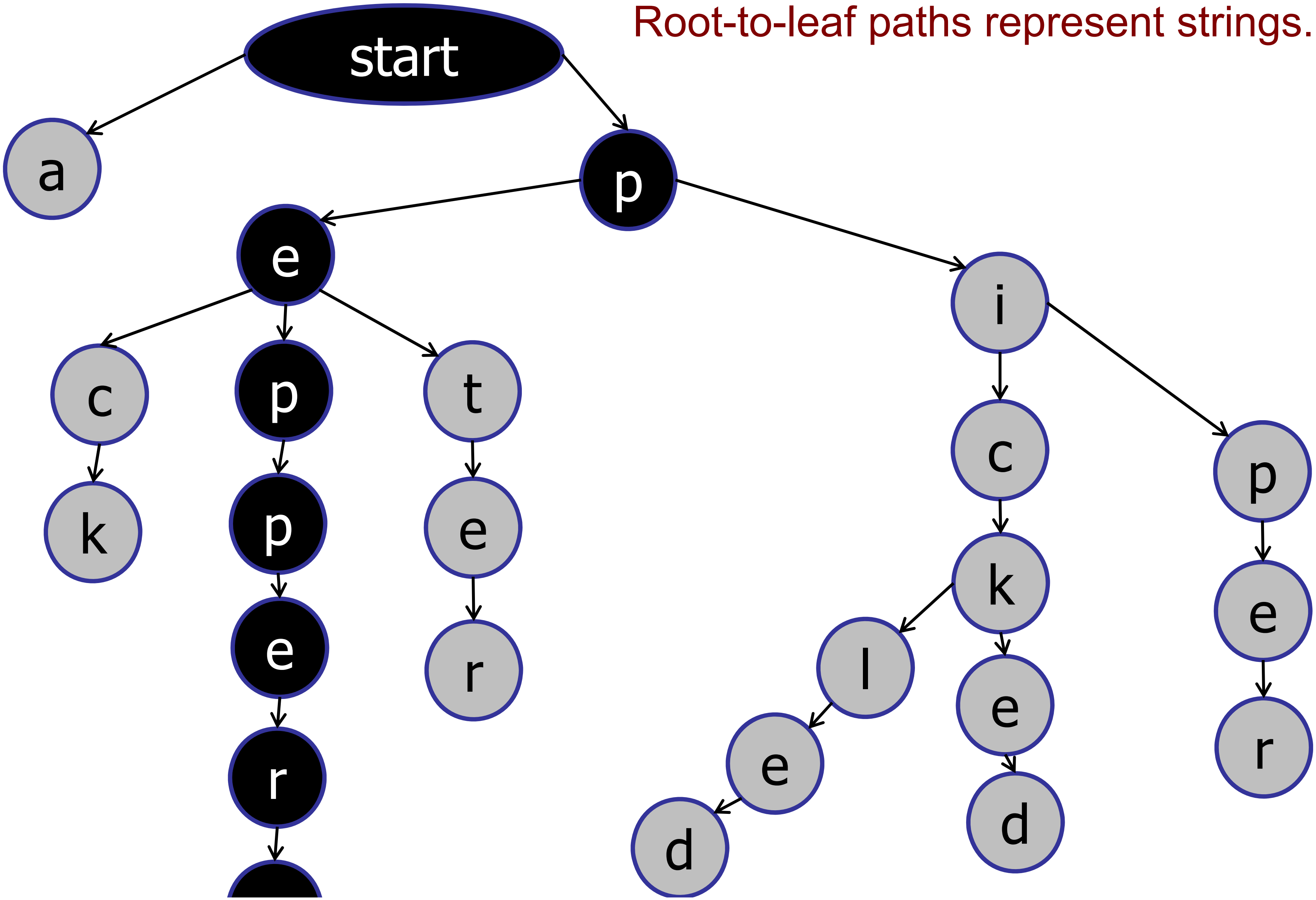
Trie [prounounced: try]

Root-to-leaf paths represent strings.



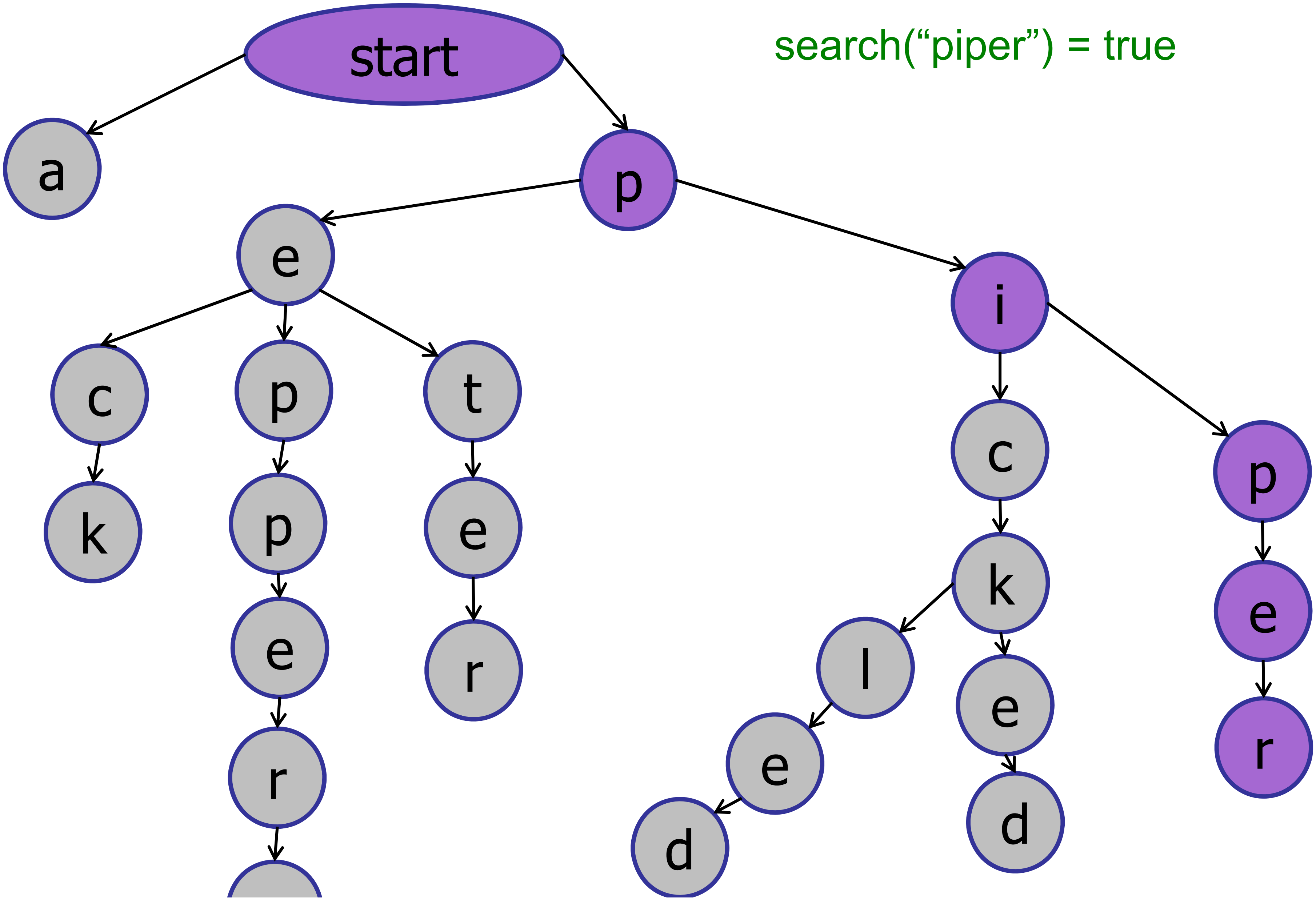
Trie [prounounced: try]

Root-to-leaf paths represent strings.

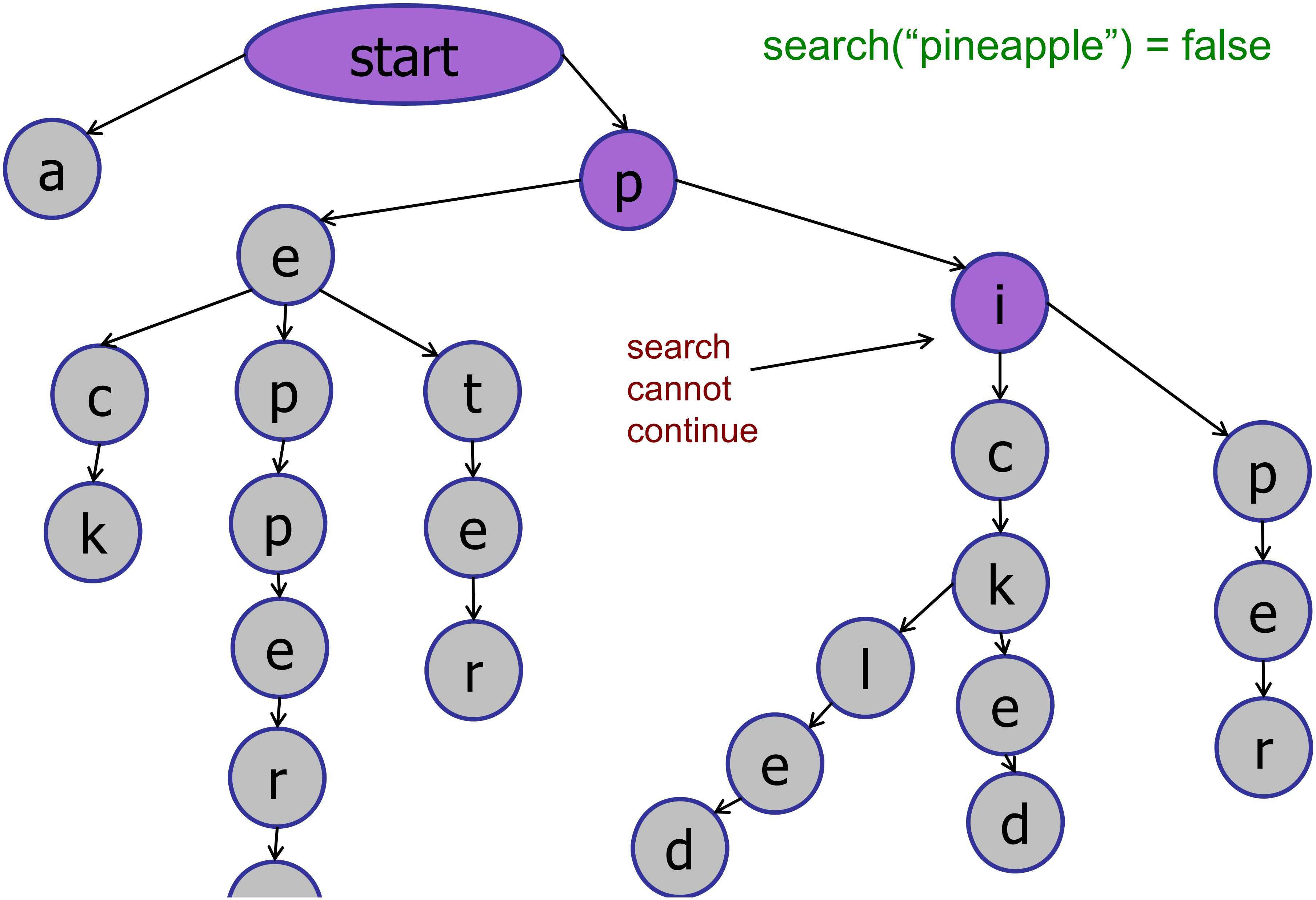


Searching a Trie

search("piper") = true

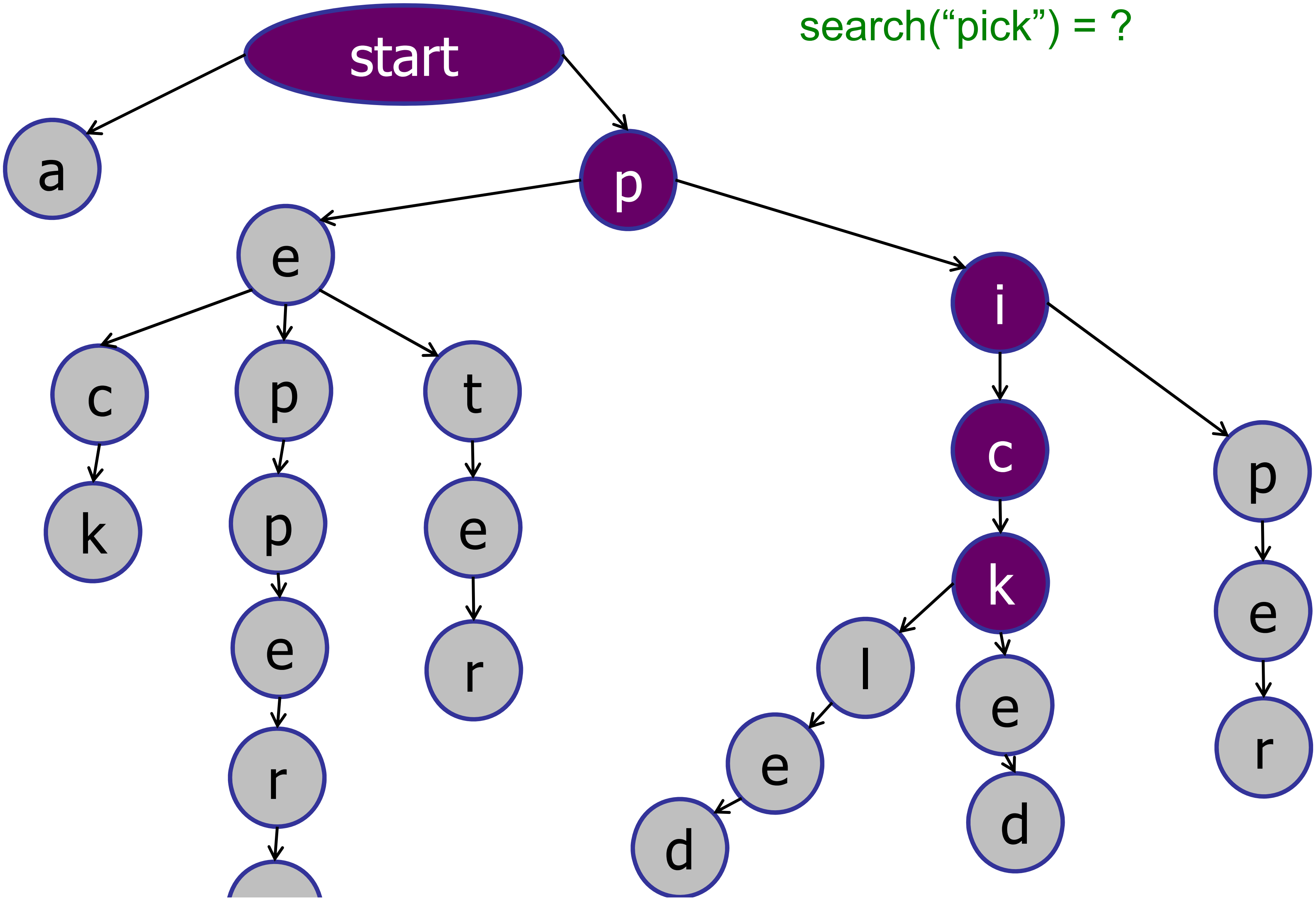


Searching a Trie



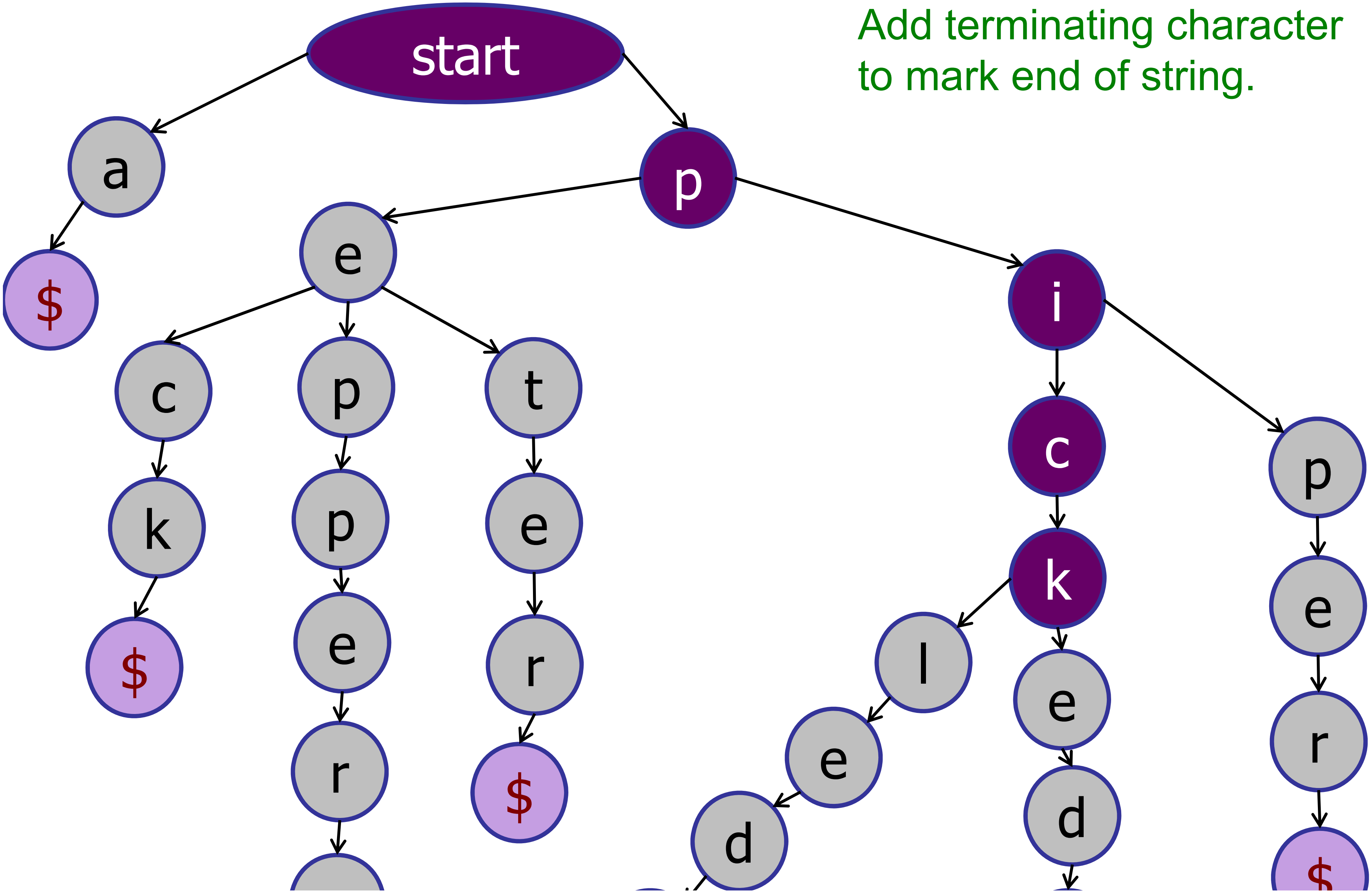
Trie Details

search("pick") = ?



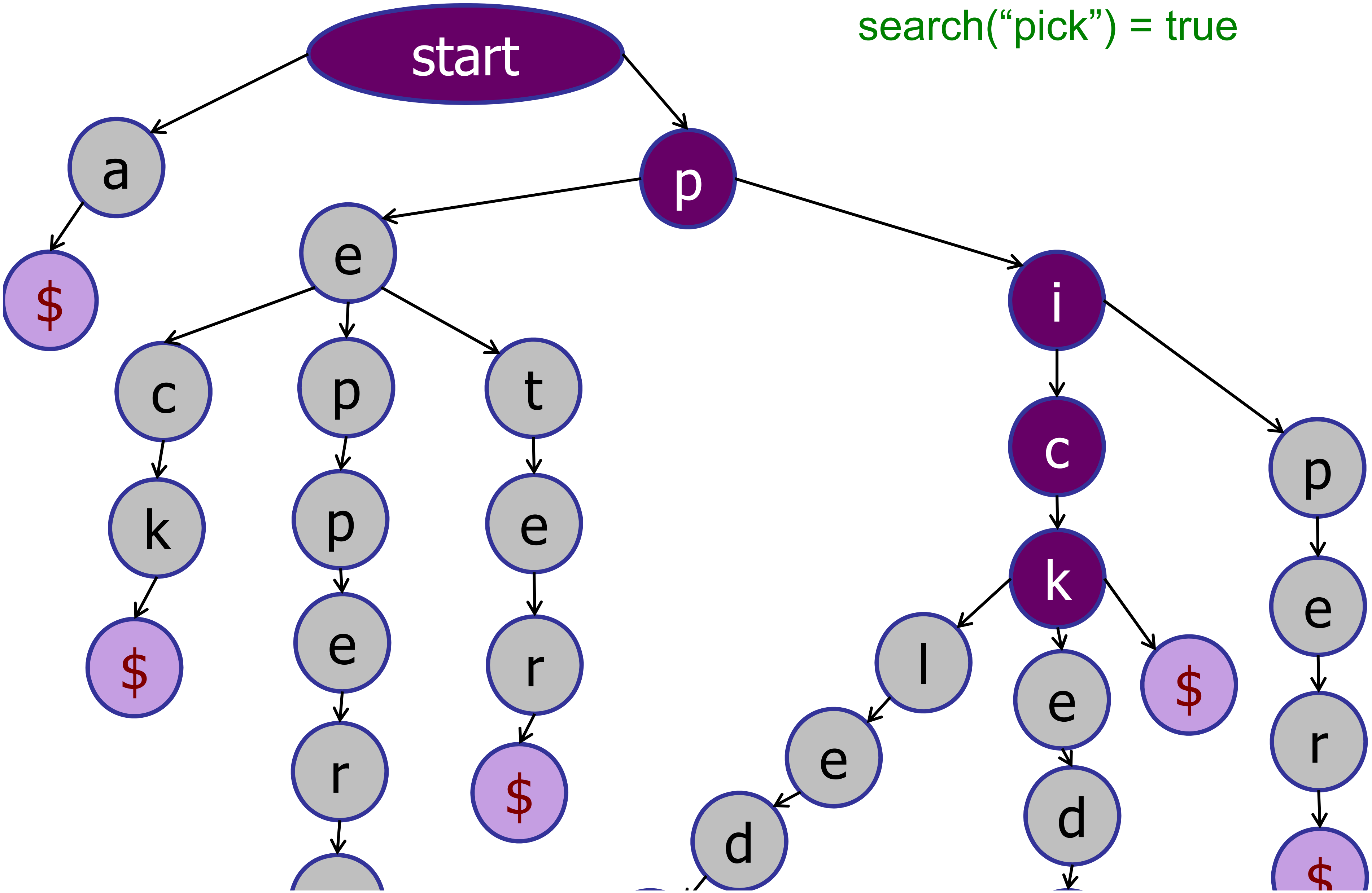
Trie Details

Add terminating character to mark end of string.

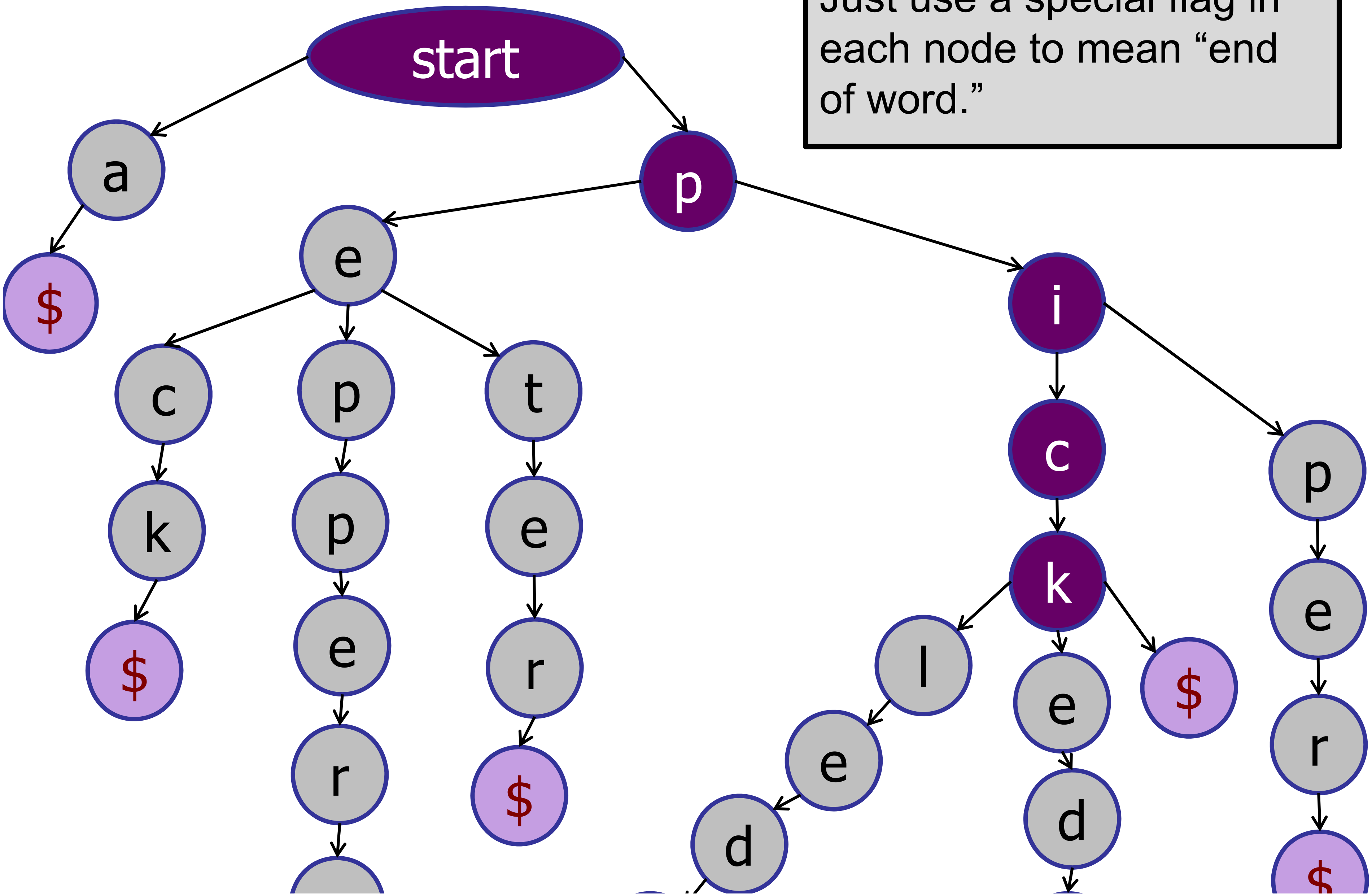


Trie Details

search("pick") = true



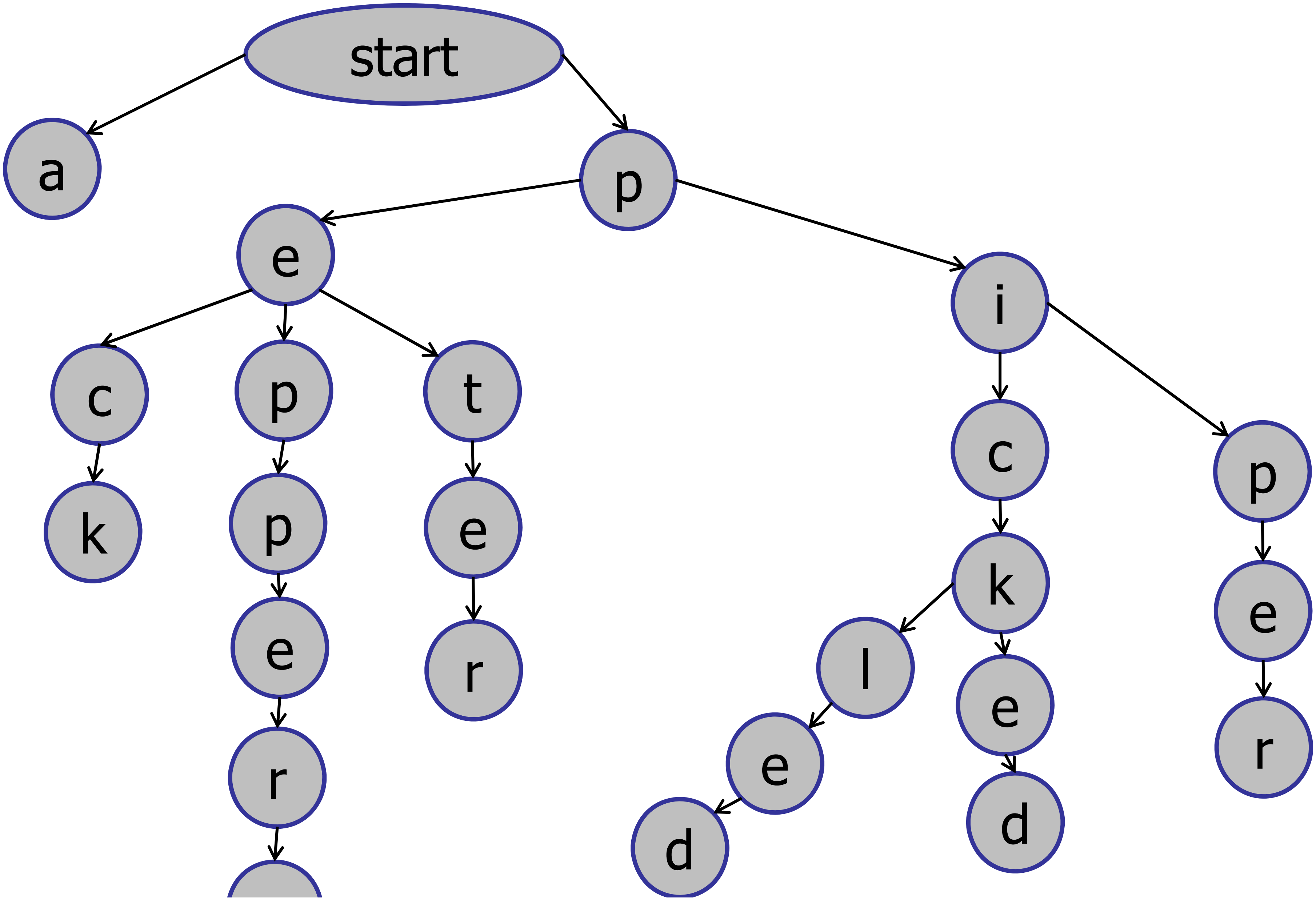
Trie Details



<u>Or:</u>
Just use a special flag in each node to mean "end of word."

Trie

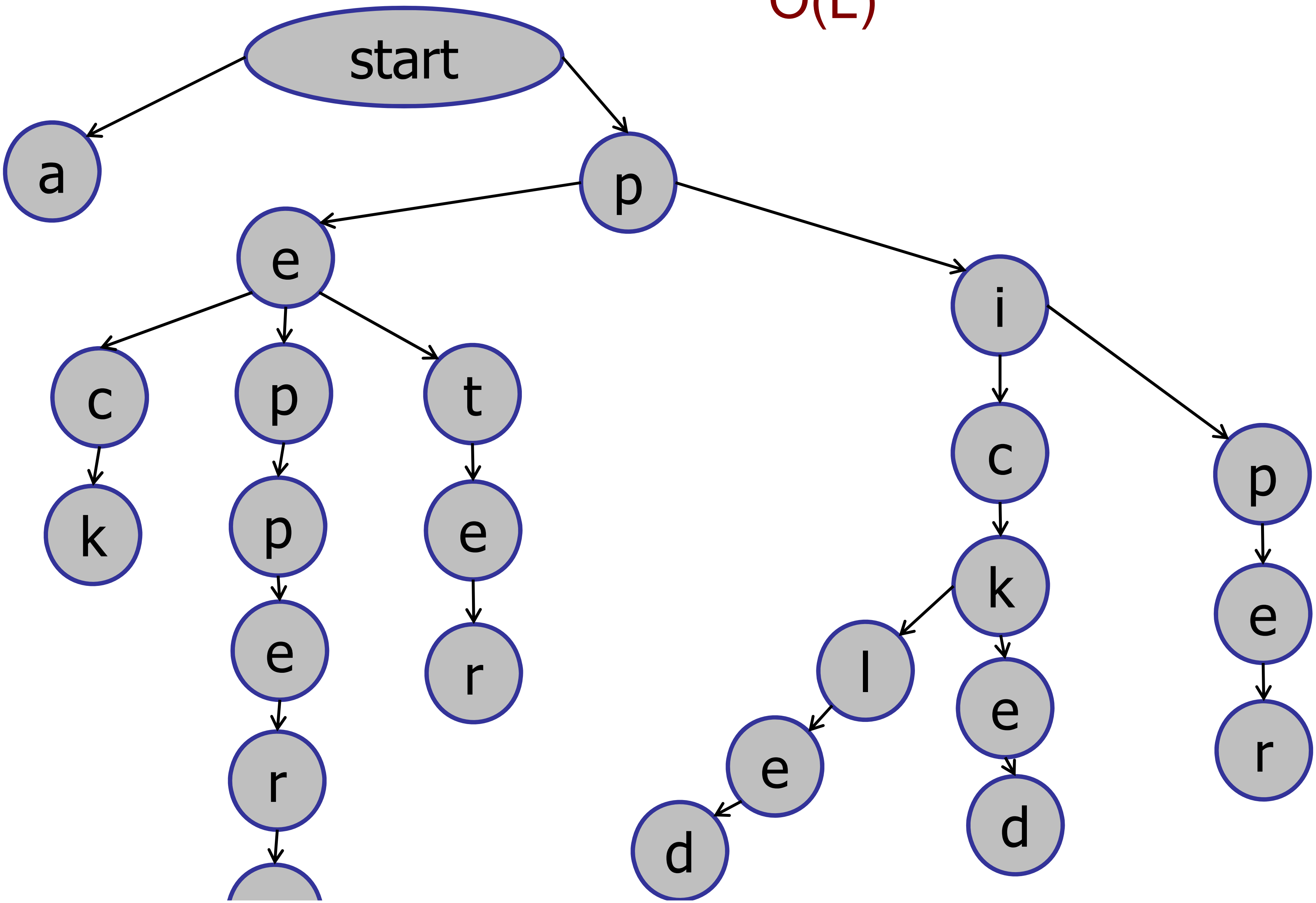
Cost to search for a string of length L?



Trie

Cost to search for a string of length L?

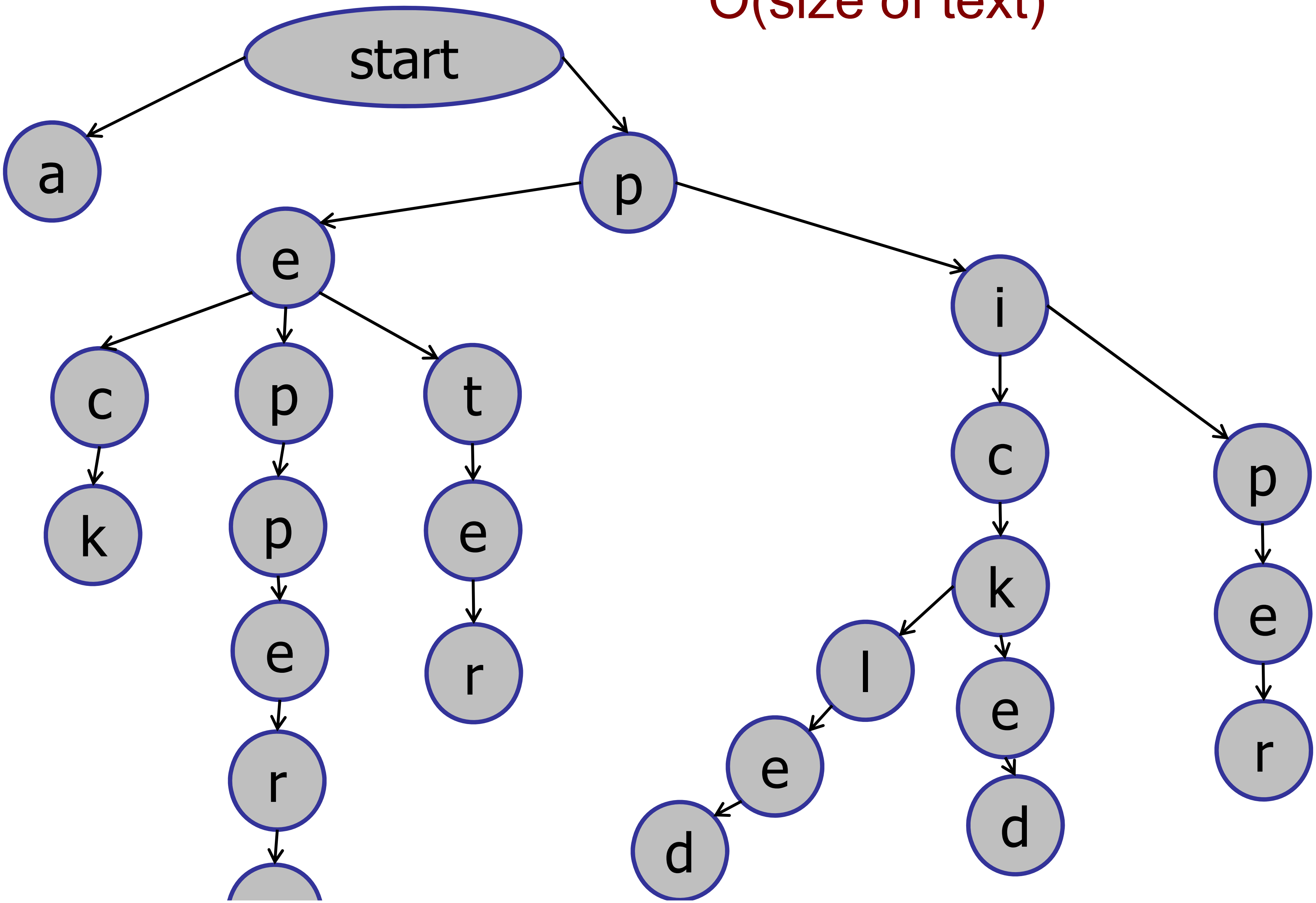
O(L)



Trie

Space for storing a try?

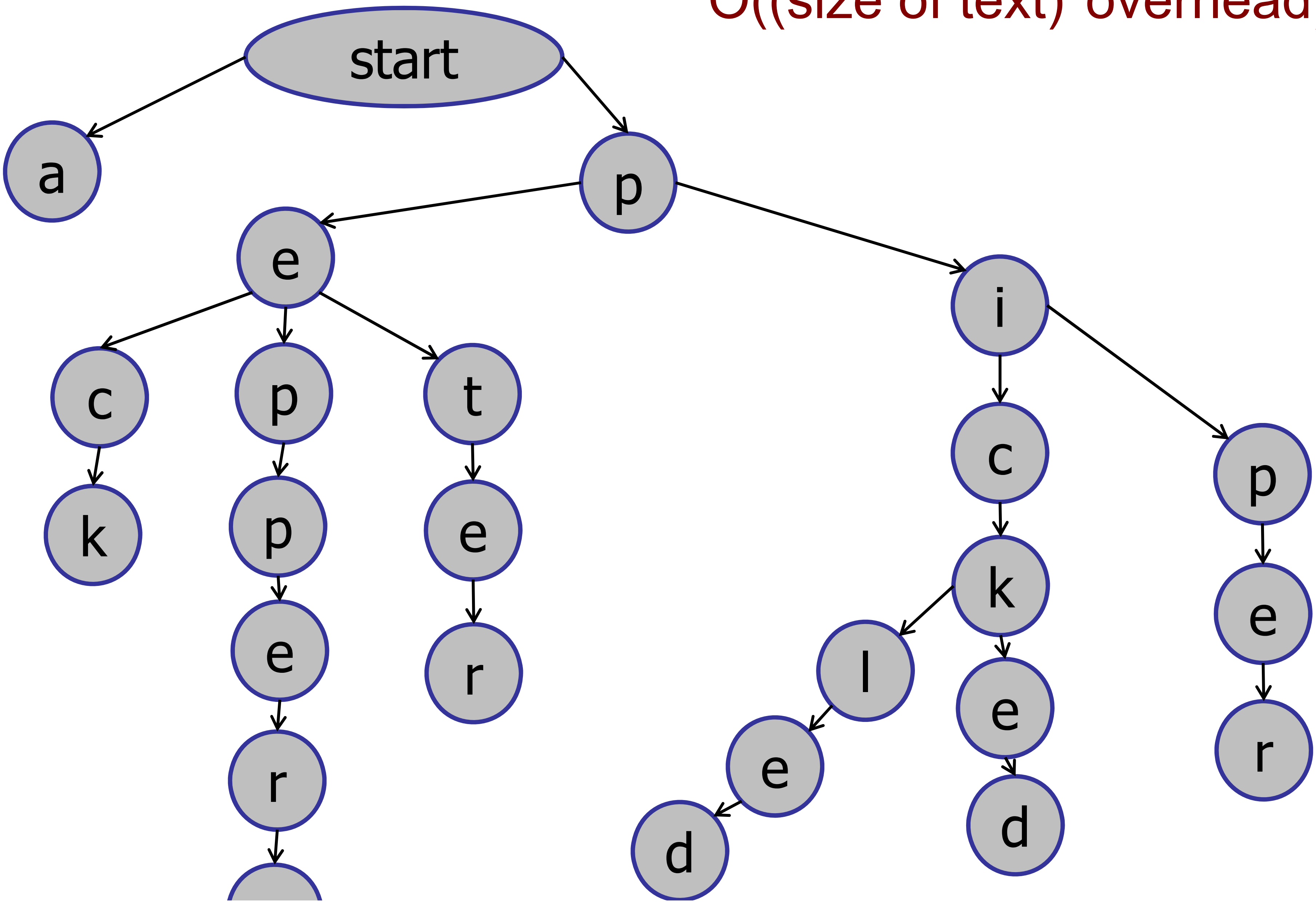
O(size of text)



Trie

Space for storing a try?

O((size of text)*overhead)



Trie Tradeoffs

Time:

- Trie tends to be faster: O(L).
- Does not depend on size of total text.
- Does not depend on number of strings.

Even faster if string is not in trie!

Trie Tradeoffs

Time:

- Trie tends to be faster: O(L).
- Does not depend on size of total text.
- Does not depend on number of strings.

Space:

- Trie tends to use more space.
- BST and Trie use O(text size) space.
- But Trie has more nodes and more overhead.

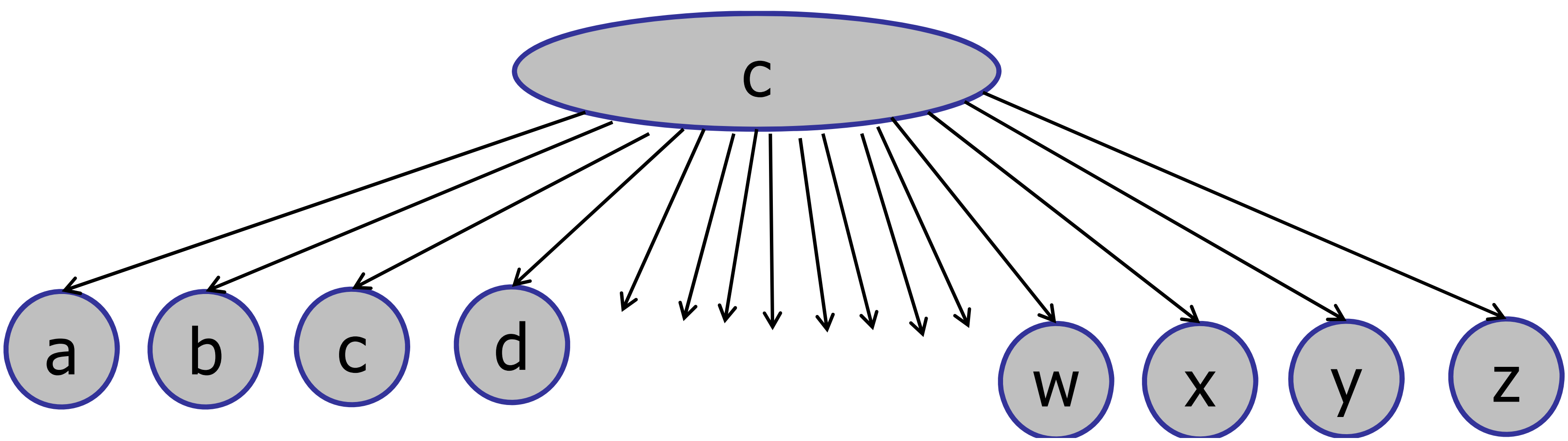
Trie Space

Trie node:

- Has many children.
- For strings: fixed degree.
- Ascii character set: 256

wasted space?

```
TrieNode children[] = new TrieNode[256];
```



Trie Applications

String dictionaries

- Searching
- Sorting / enumerating strings

Partial string operations:

- Prefix queries: find all the strings that start with pi.
- Long prefix: what is the longest prefix of “pickling” in the trie?
- Wildcards: find a string of the form “pi??le” in the trie.

Balanced Search Trees

Summary:

- The Importance of Being Balanced
- Height Balanced Trees
- Rotations
- AVL trees
- Tries