CS2040S

Data Structures and Algorithms

On the importance of being balanced

(Act 2)

Puzzle of the Week:

100 prisoners. Every so often, one is chosen at random to enter a room with a light bulb. You can turn the light bulb on or off.

- WIN if one prisoner announces correctly that all have visited the room.
- LOSE if announcement is incorrect.

What if, initially, the state of the light is unknown, either on or off?

Where are we?

Trees

- Terminology
- Traversals
- Operations

Balanced Trees

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

Tutorial Plans

Topics

- Review of tree basics
- Interesting tree application (order maintenance)
- Interesting QuickSort applications

Questions

- How to choose which data structure to use to solve a problem?
- Can you use existing solutions as a black box to simplify?
- Can similar techniques be used to solve different problems?
- How to measure the goodness of a solution?

Recitation Plans

Main topic: B-trees

- Another example of a balanced search tree.
- The most important balanced search tree in the world today (maybe) ➔ used in (almost) every database in existence.

Questions

- How do you invent a new data structure?
- Rule-based design
- A process:
  - Choose invariants/rules.
  - Show they provide good outcomes.
  - Show how to maintain those rules.

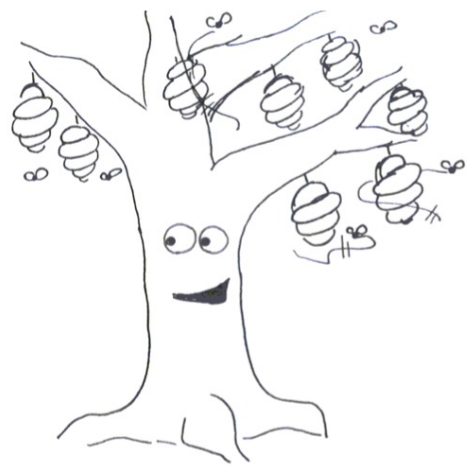


Image credit: Jeremy Fineman

Problem Set

Scapegoat Trees!

- Simple type of balanced tree.
- Fast in practice (but amortized!).
- Complete the implementation form the previous problem set.

Autocomplete and Pattern Matching

- Design a new data structure to solve autocomplete.
- Simple pattern matching.

- Optional: think about how to implement regular expressions!

(It is not easy!)

Recess Week

Announcements

Midterm : Thursday March 10, 6:30pm

Location: ~14 different venues (MPSH).

Note: In person, face-to-face

Safe distancing: 48 students / room, spaced

About < 5 people have e-mailed me with conflicts (i.e., other midterms, national service, etc.). So I assume this is a good time for the remaining 645+ of you!

AVL Trees

On the importance of being balanced

Todays Plan

On the importance of being balanced

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

Tries

- How to handle text?

Data structure design

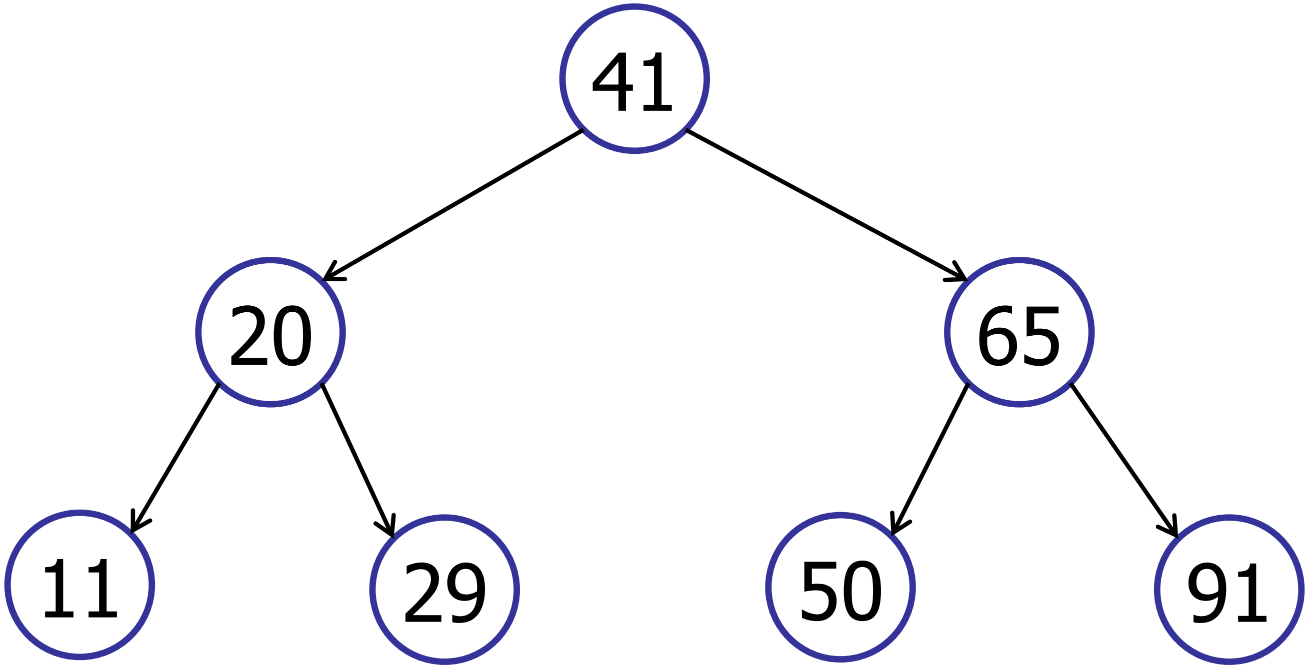
- How to build new structures on existing ideas?

Recap: 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

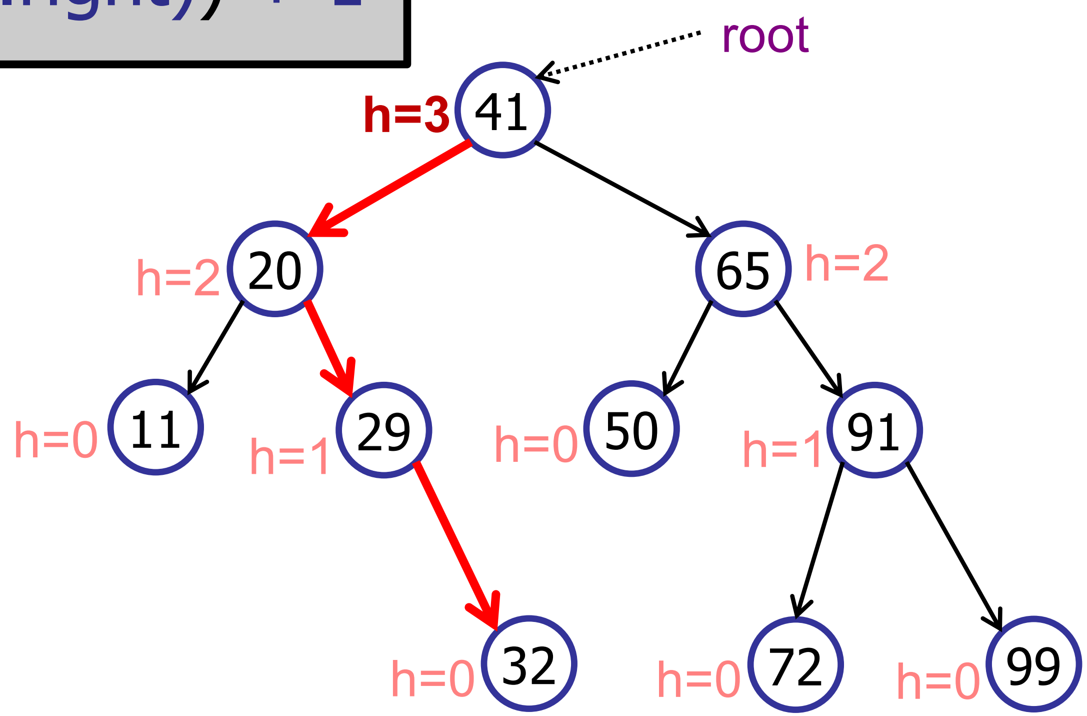
Recap: Heights

Height:

Number of edges on longest path from root to leaf.

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

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



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

Binary Search Tree

Modifying Operations: O(h)

- insert
- delete

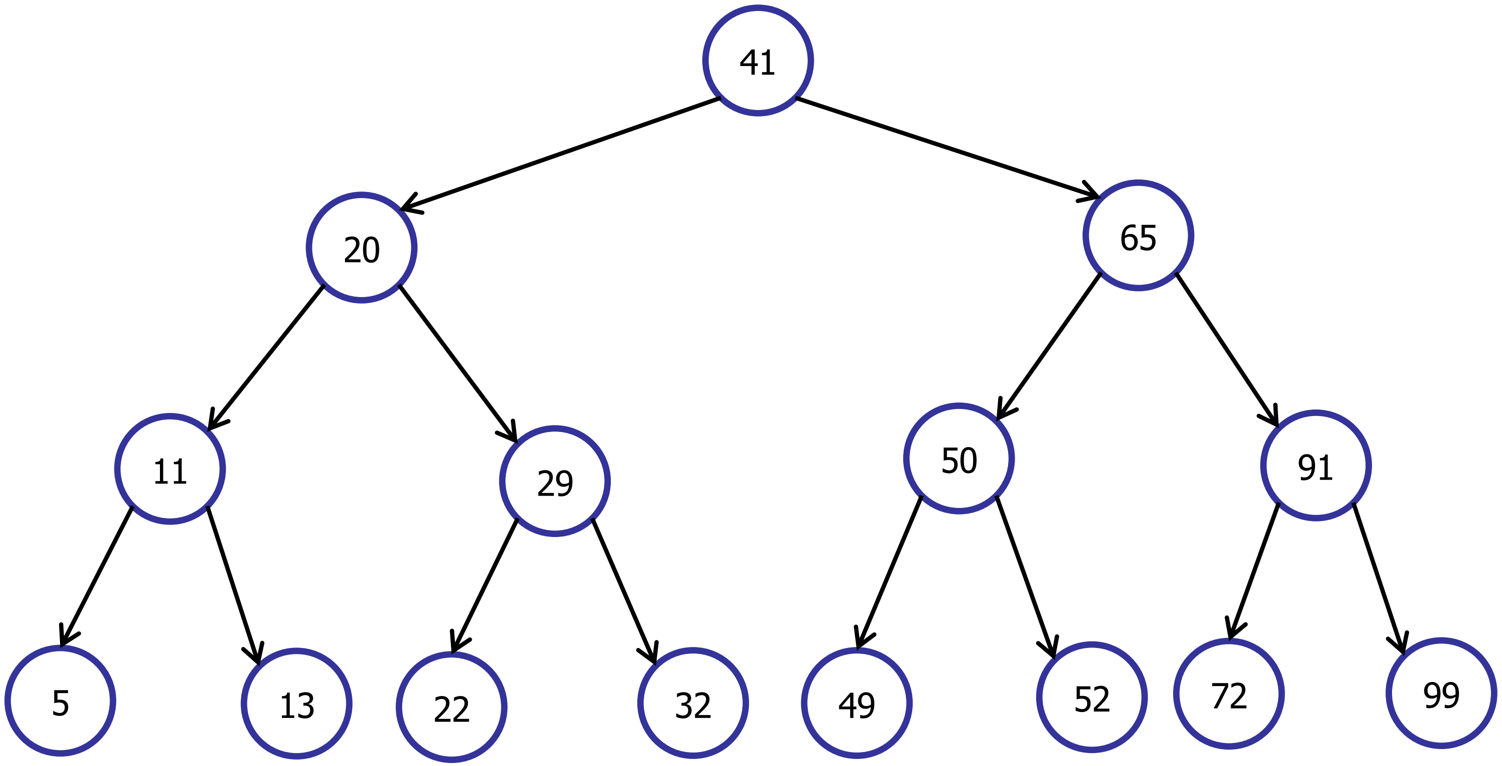
Query Operations: O(h)

- search
- predecessor, successor
- findMax, findMin

Traversals: O(n)

The Importance of Being Balanced

Operations take O(h) time



What is the largest possible height h?

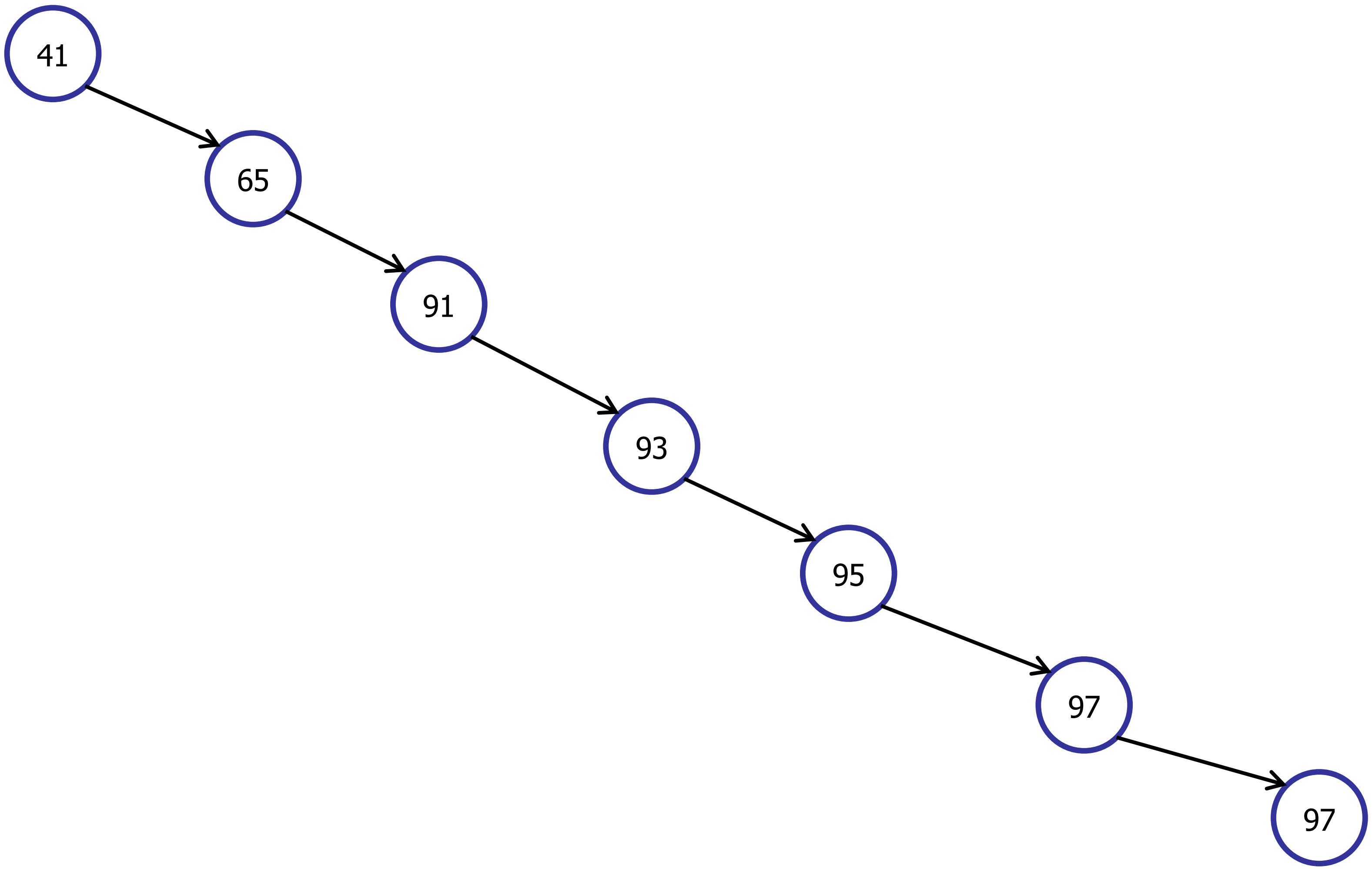
Last time…

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?

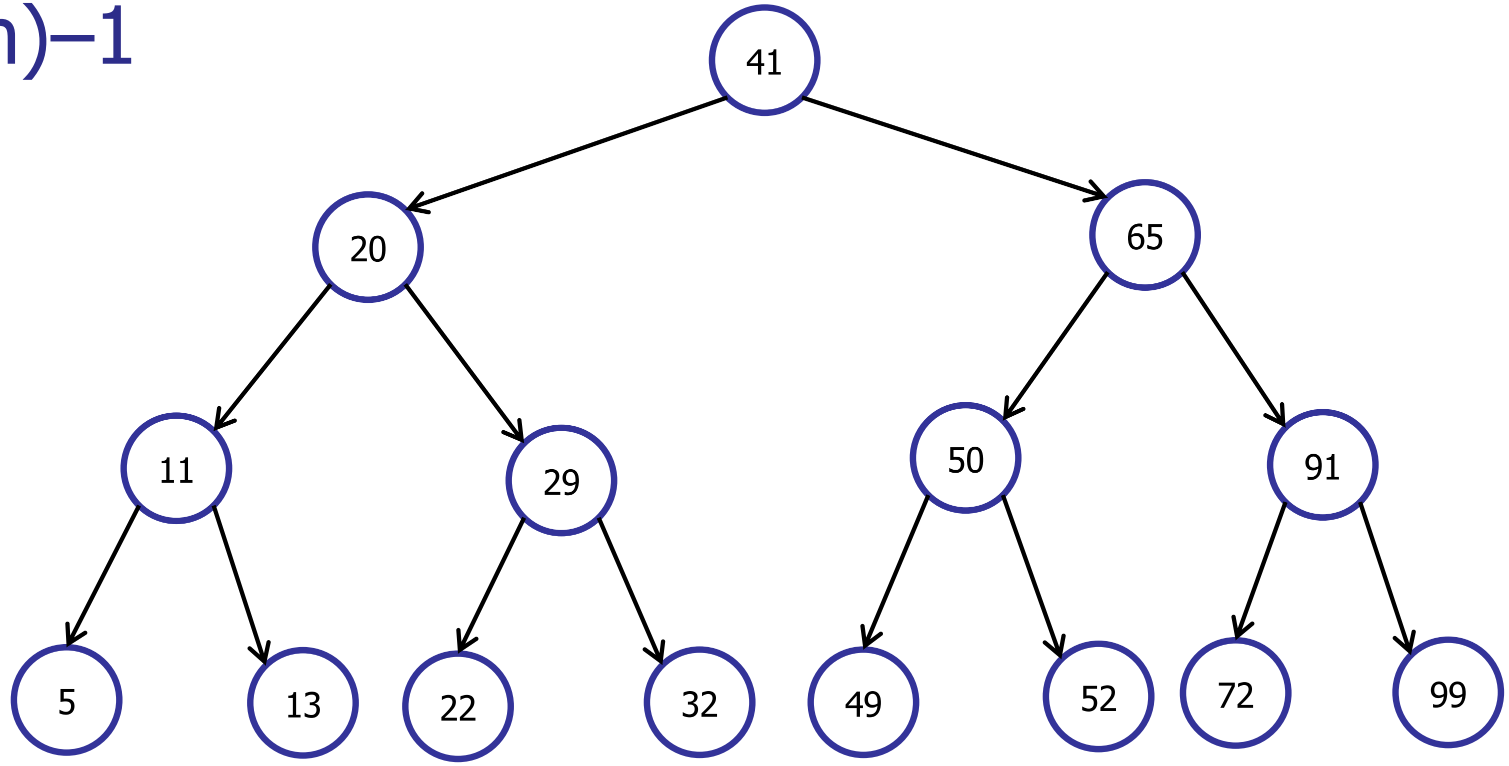
Last time...

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

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

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

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

- Store the height in each node
- Maintain the height on insertion and deletion.

Step 1: Define Balance Condition

- Tree is height balanced if siblings height differ by at most 1.

Step 2: Maintain Balance

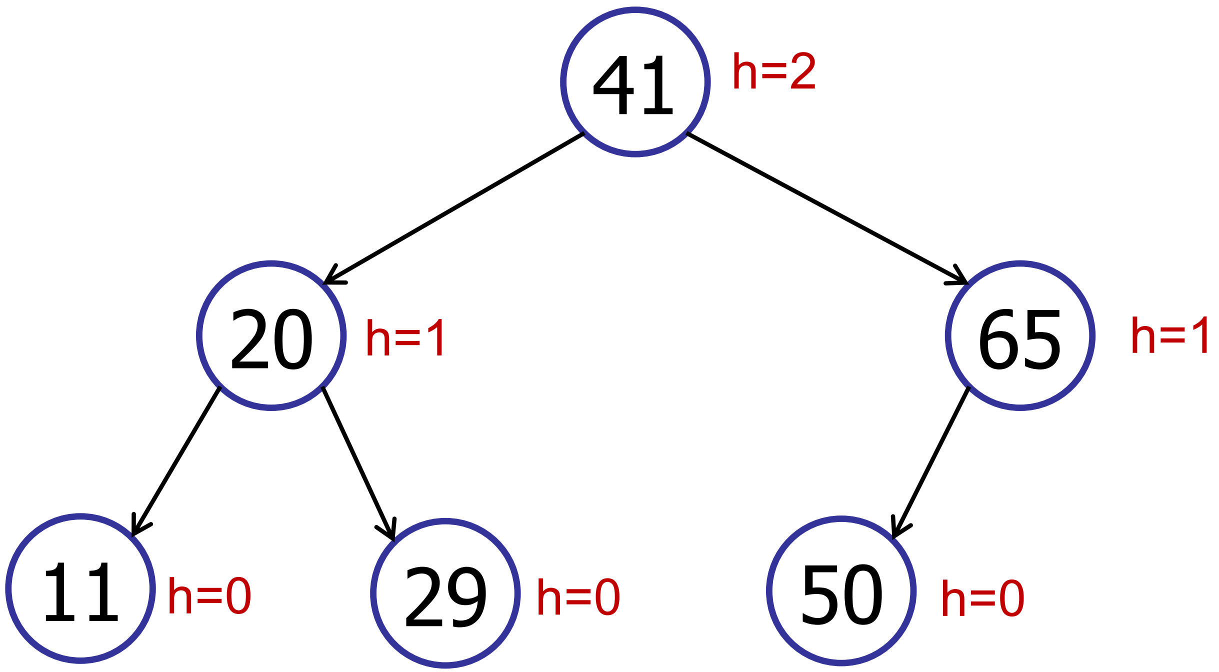
- Use rotations to maintain the balance of the tree

AVL Trees [Adelson-Velskii & Landis 1962]

Step 0: Augment

In every node v, store height and update on insert/delete operations:

v.height = h(v)



Quick review: a tree is height-balanced iff:

1. It is perfectly balanced.
2. It has height O(log n).
3. For every node u, the number of nodes in its left and right subtrees is within 1 of each other.
4. For every node u, the number of nodes in its left and right subtrees is within a factor of 2 of each other.
5. For every node u, the height of its children is within one of each other.
6. For every node u, the height of its children is within a factor of 2 of each other.

ARCHIPELAGO

is open

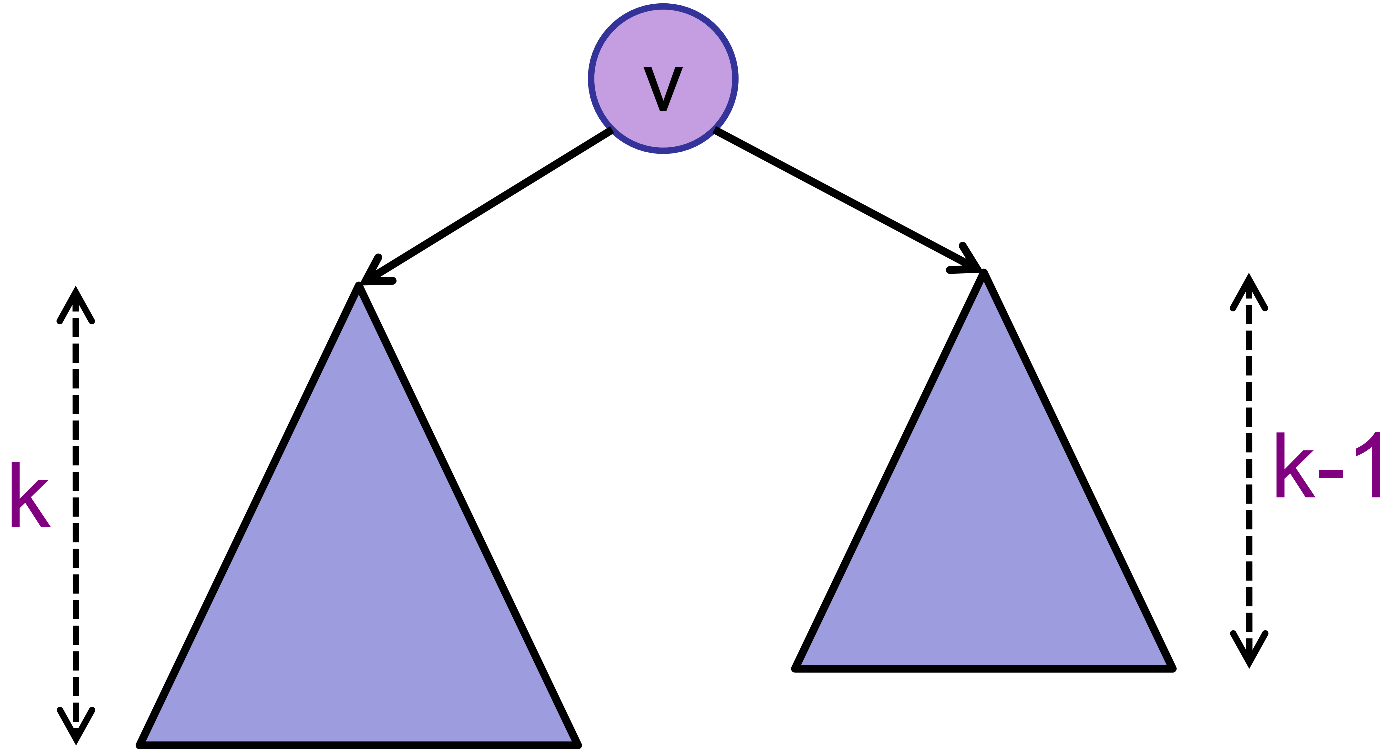
AVL Trees [Adelson-Velskii & Landis 1962]

Key definition

Step 1: Define Invariant

- A node v is **height-balanced** if:

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



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.

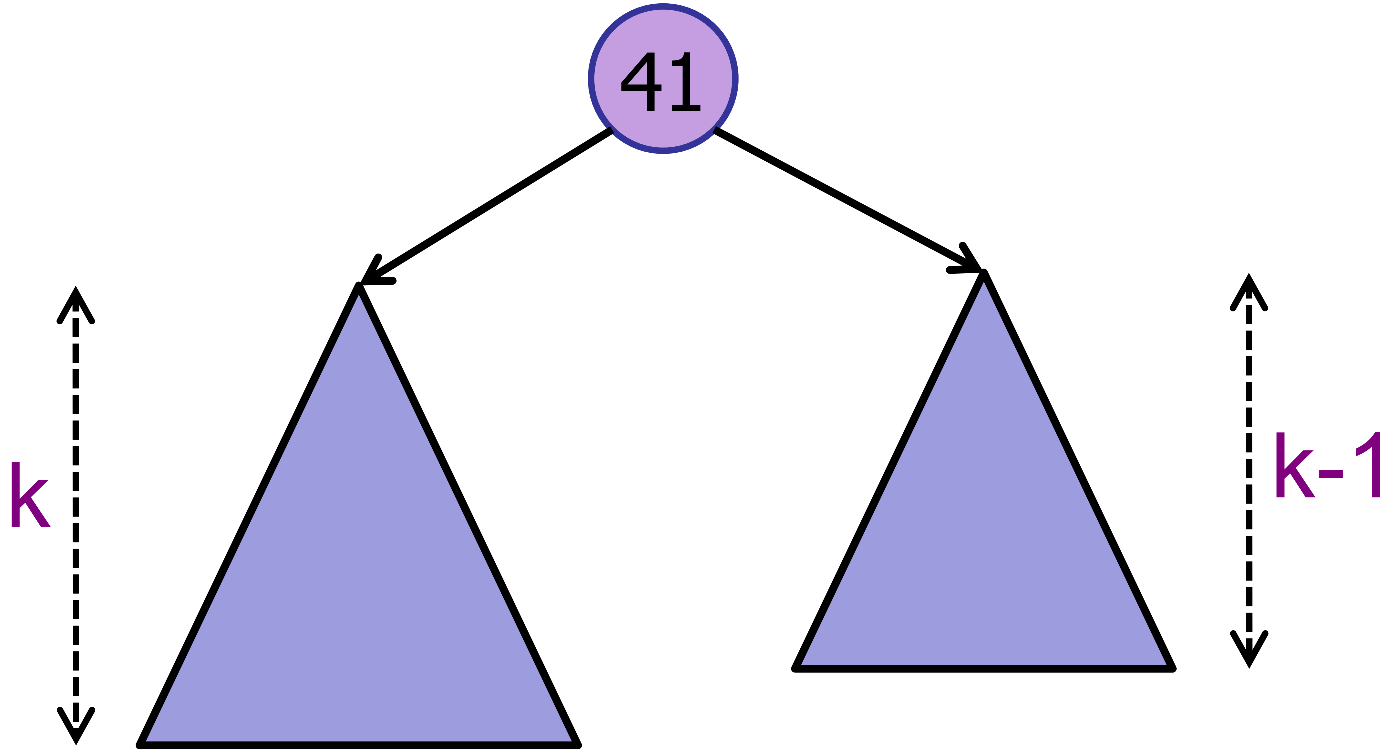
Height-Balanced Trees

Claim:

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

AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Show how to maintain height-balance



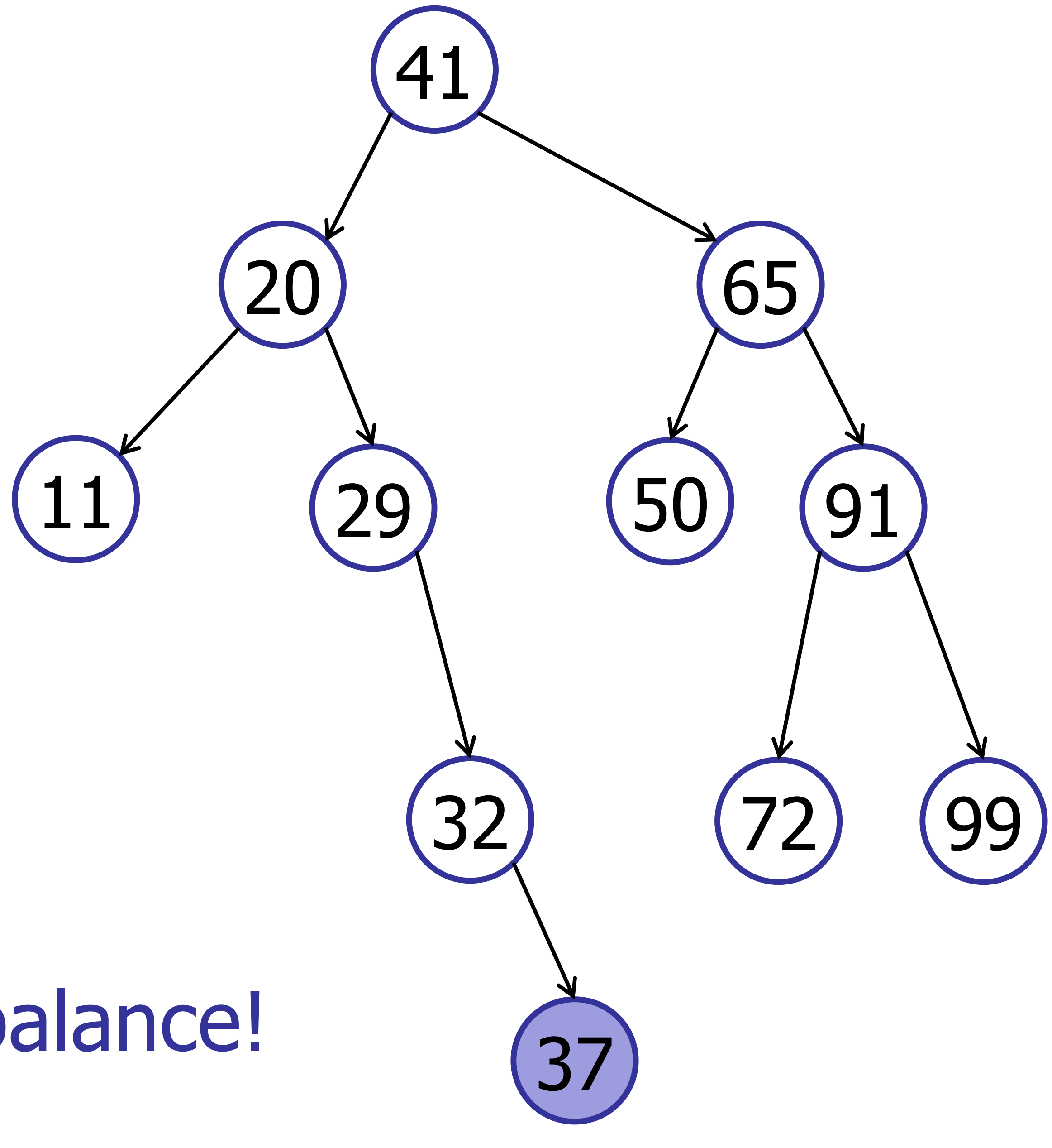
Inserting in an AVL Tree

Initially balanced

insert(37)

No longer balanced after insertion!

Use rotations to rebalance!



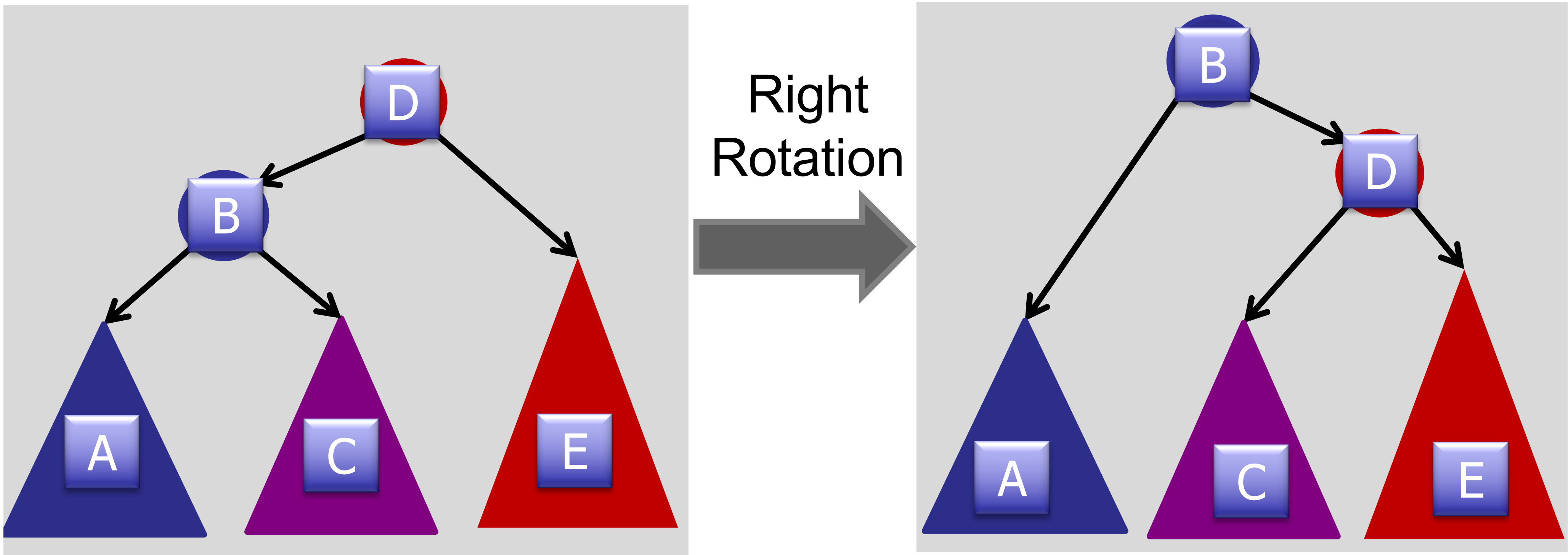
Quick review: a rotation costs:

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

ARCHIPELAGO

is open

Tree Rotations

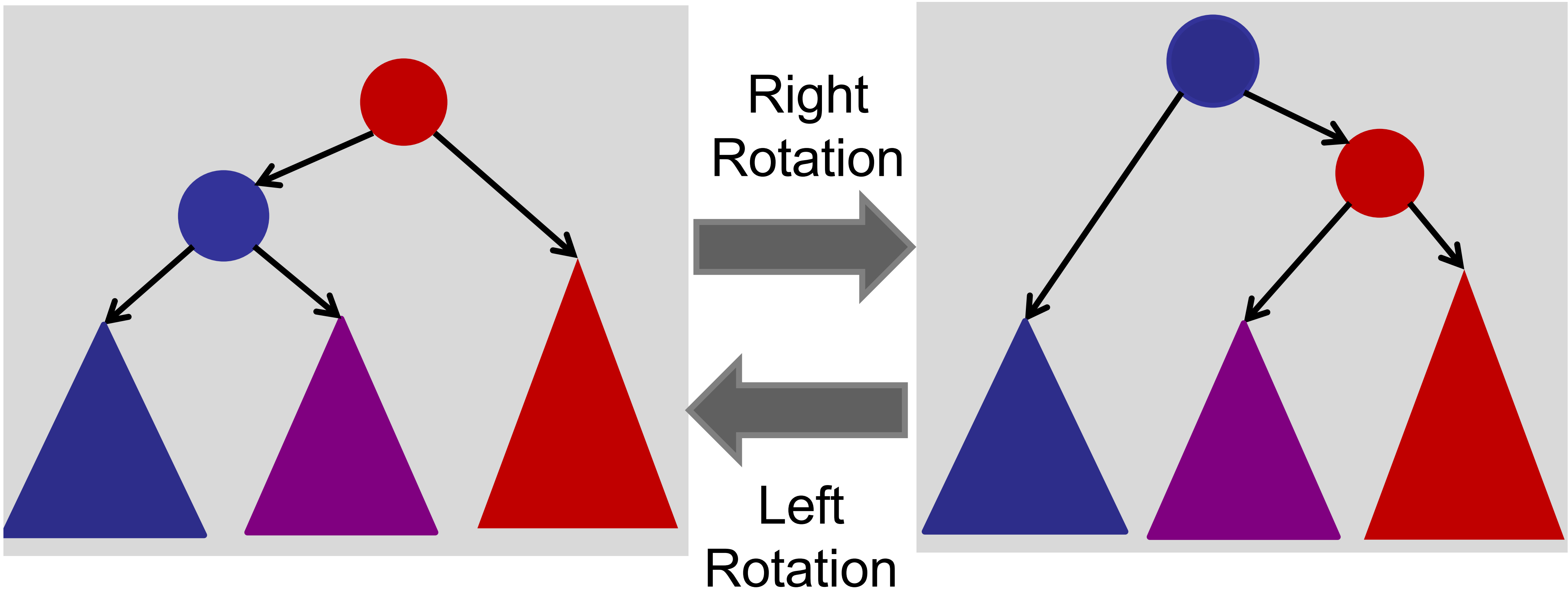


A < B < C < D < E

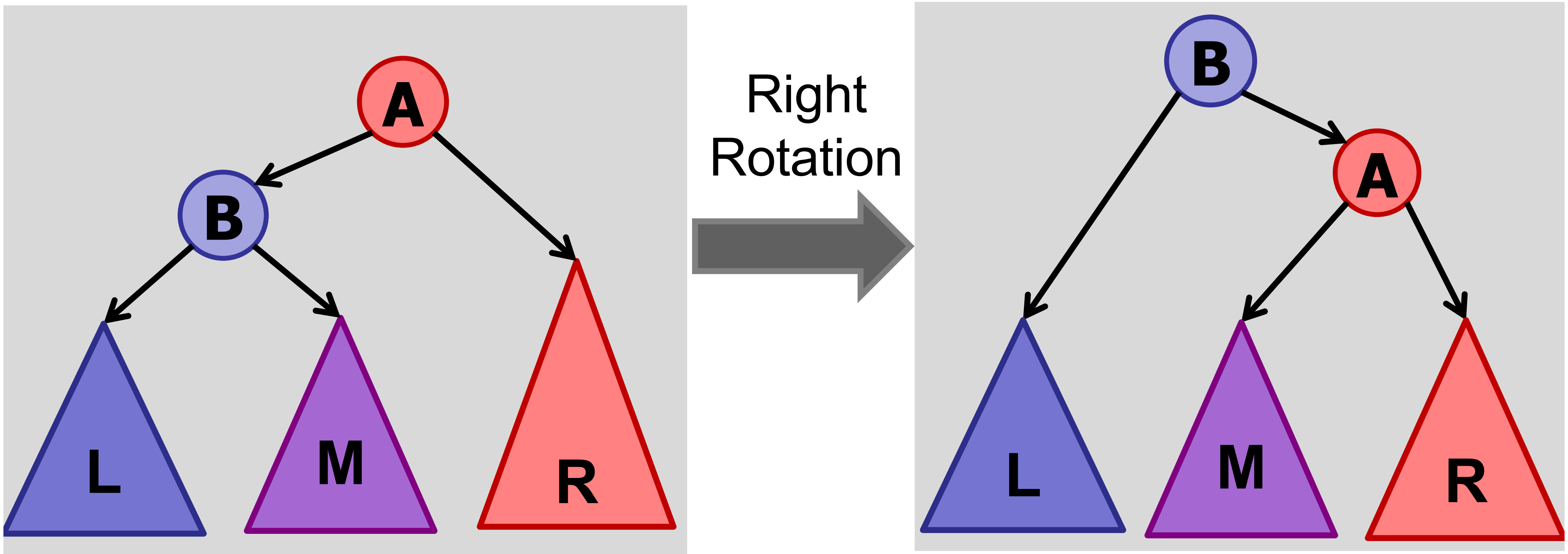
Rotations maintain ordering of keys.

⇒ Maintains BST property.

Tree Rotations



Tree Rotations

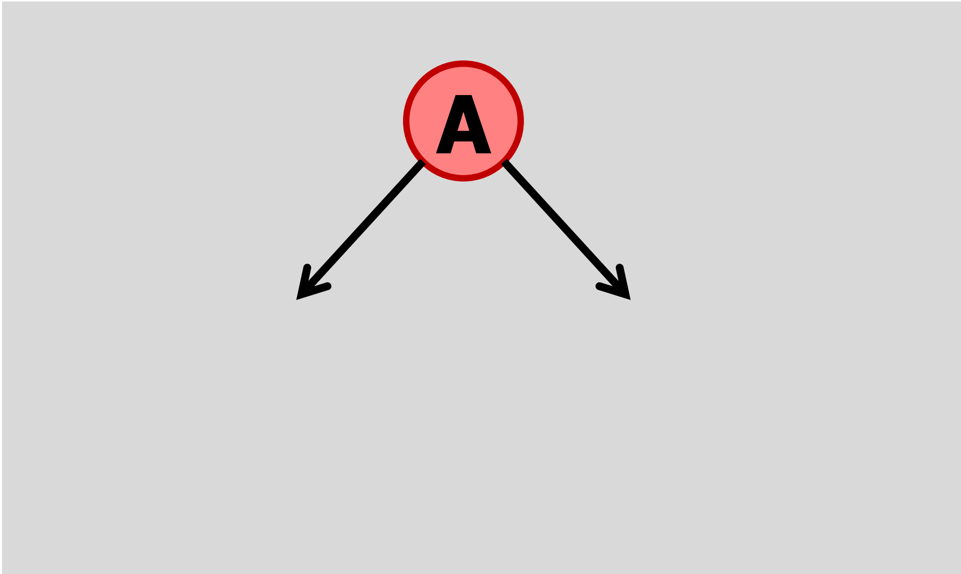


After insert:

Use tree rotations to restore balance.

Height is out-of-balance by 1

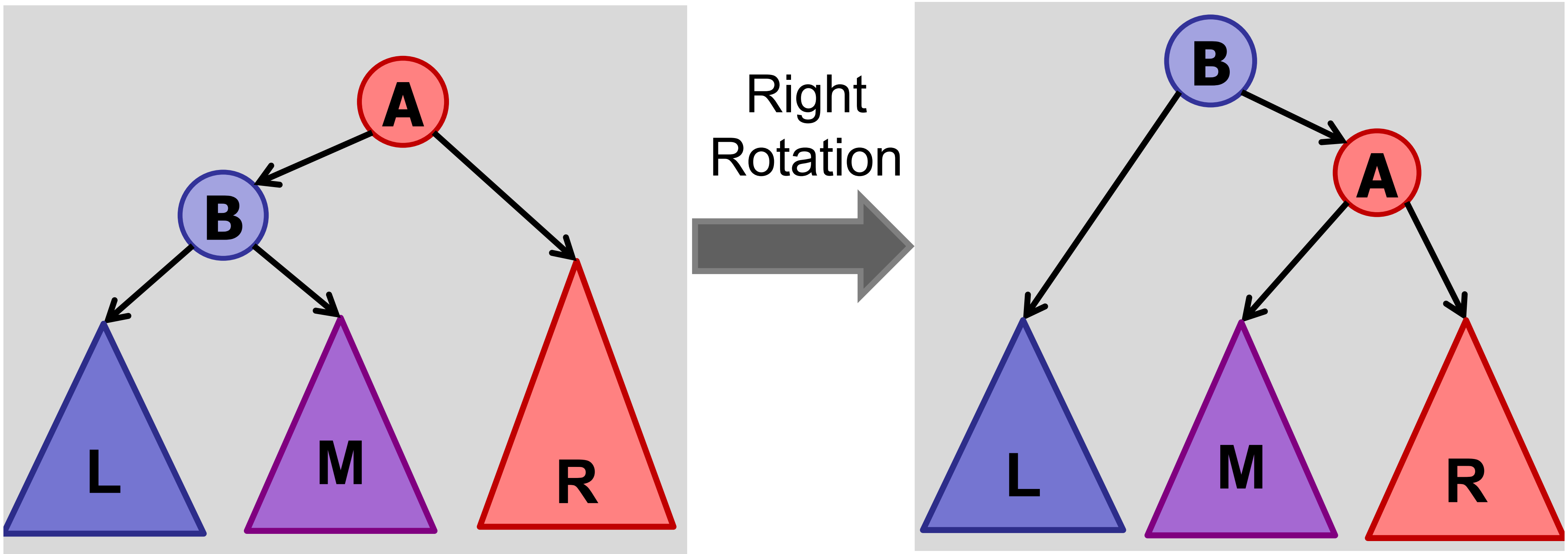
Tree Rotations



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.

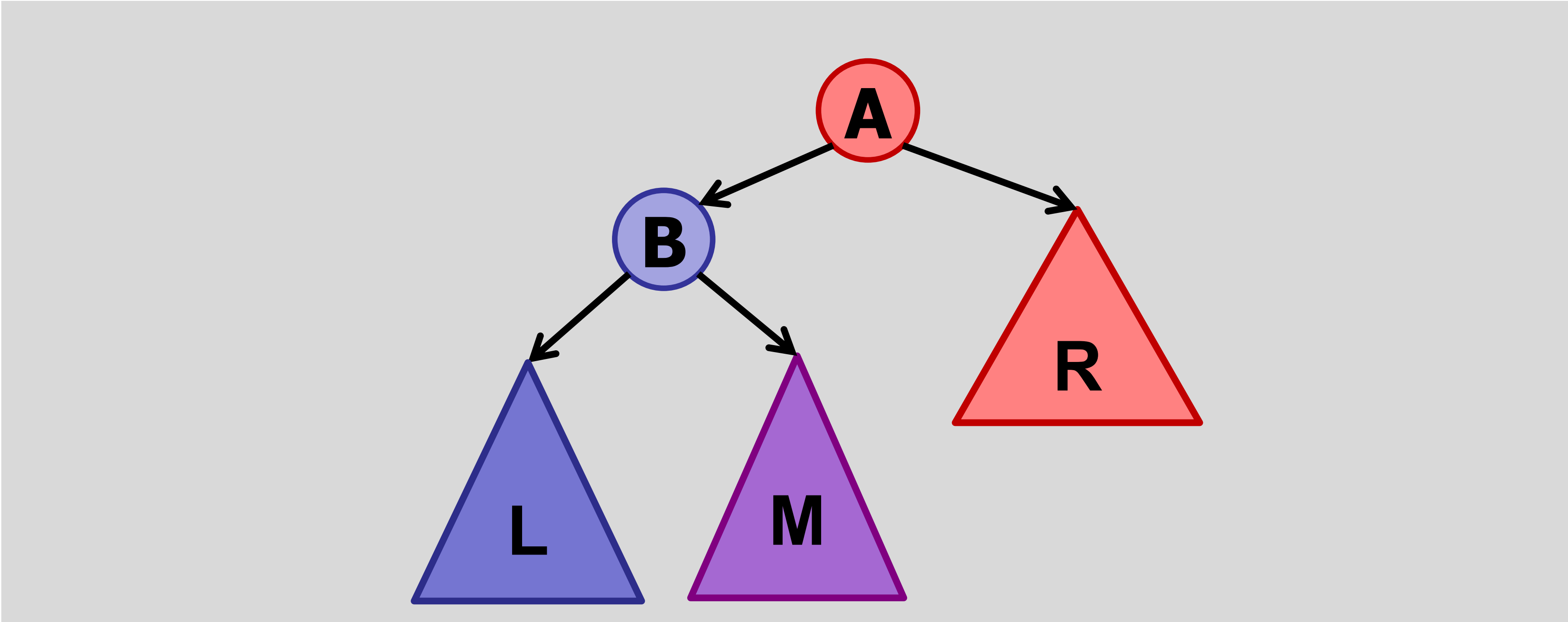
Tree Rotations



Use tree rotations to restore balance.

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

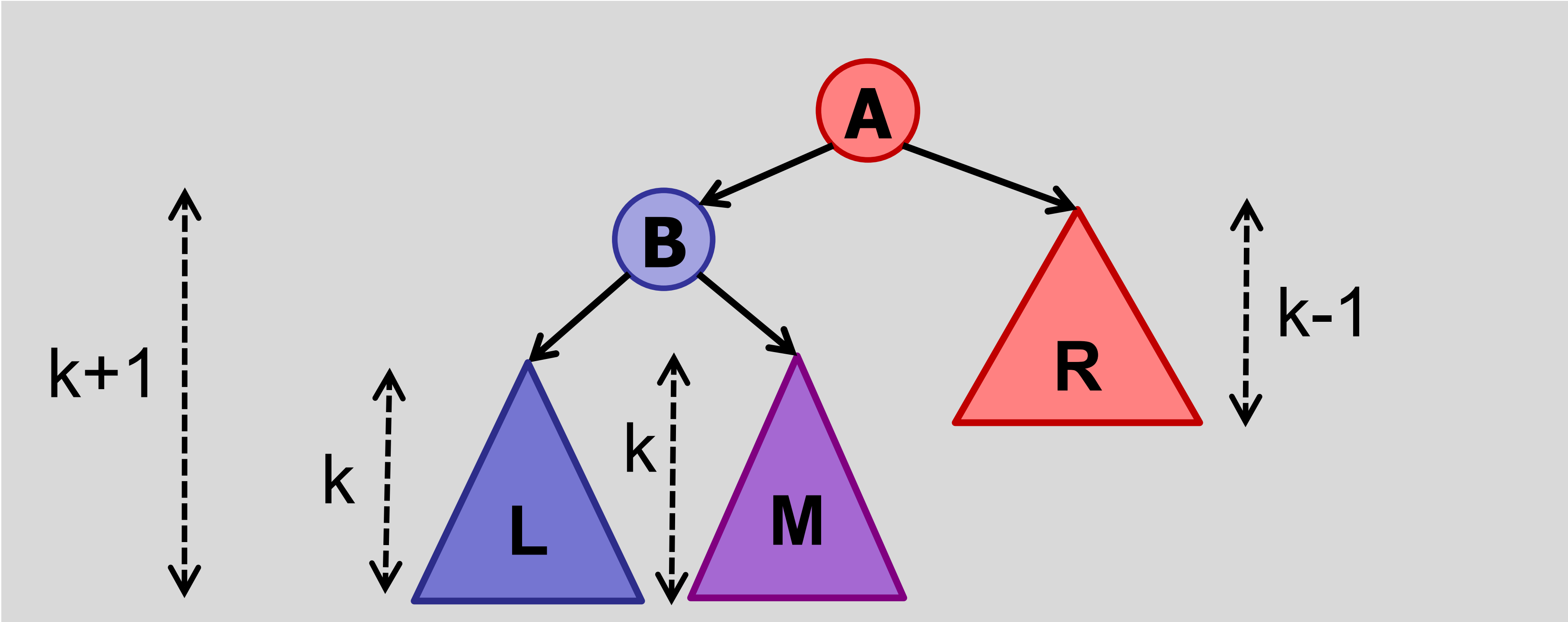
Tree Rotations



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

Assume A is **LEFT-heavy**.

Tree Rotations (Left Heavy)

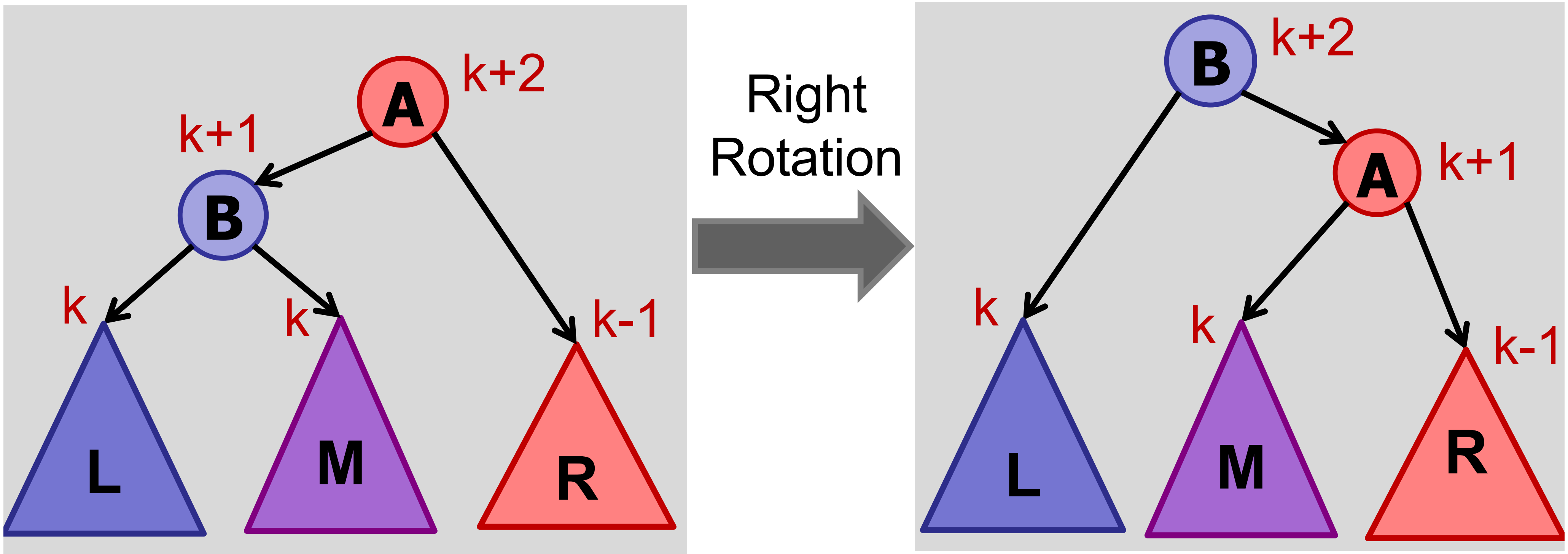


Assume **A** is the lowest node in the tree violating balance property.

Case 1: **B** is equi-height : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations

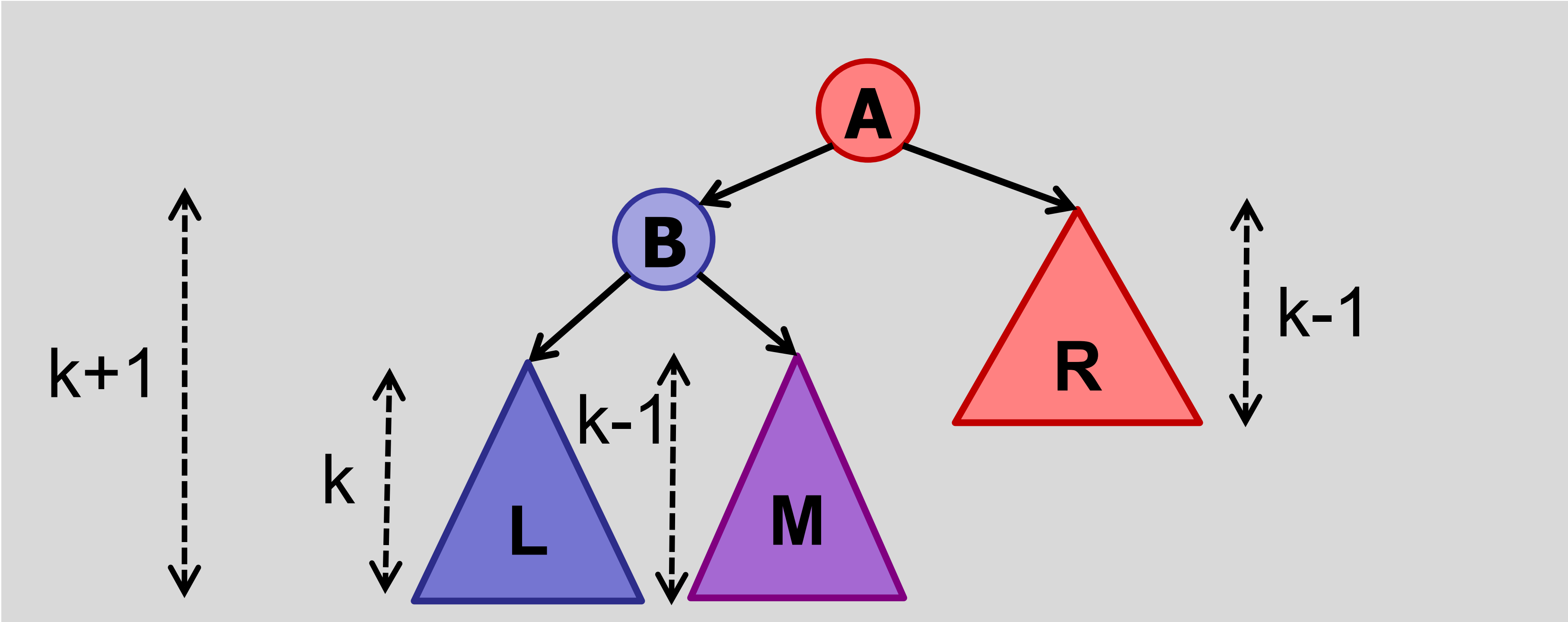


right-rotate:

Case 1: **B** is equi-height : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations (Left Heavy)

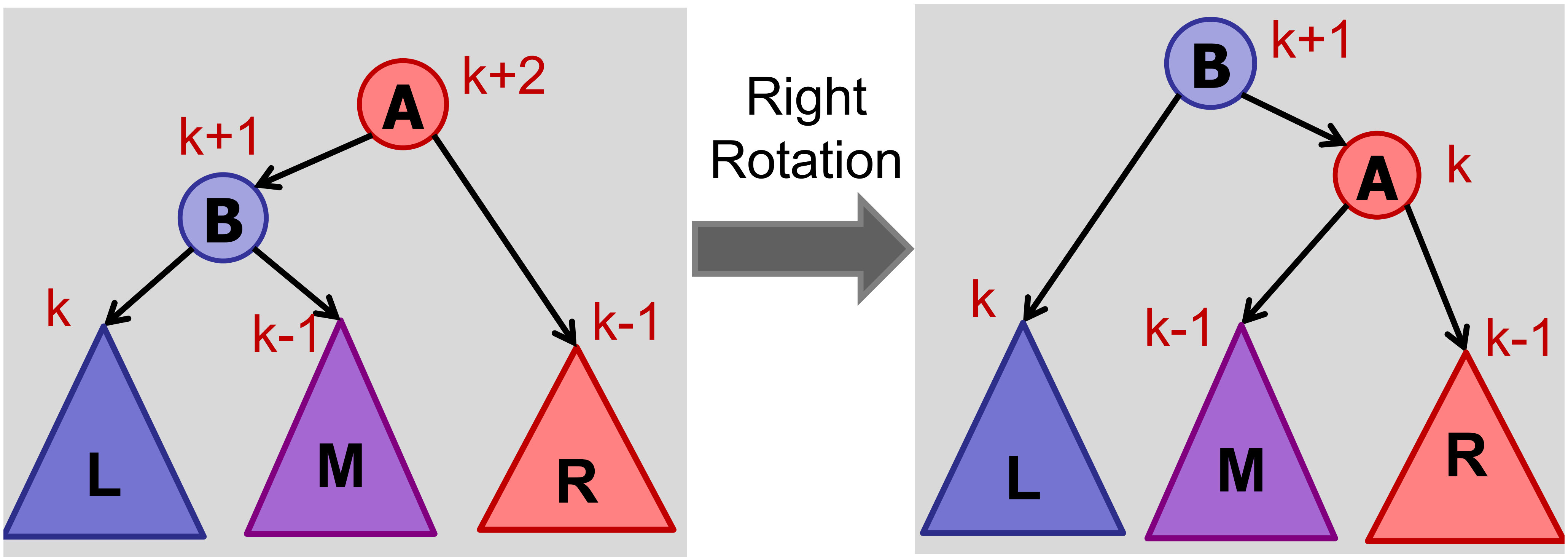


Assume **A** is the lowest node in the tree violating balance property.

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

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

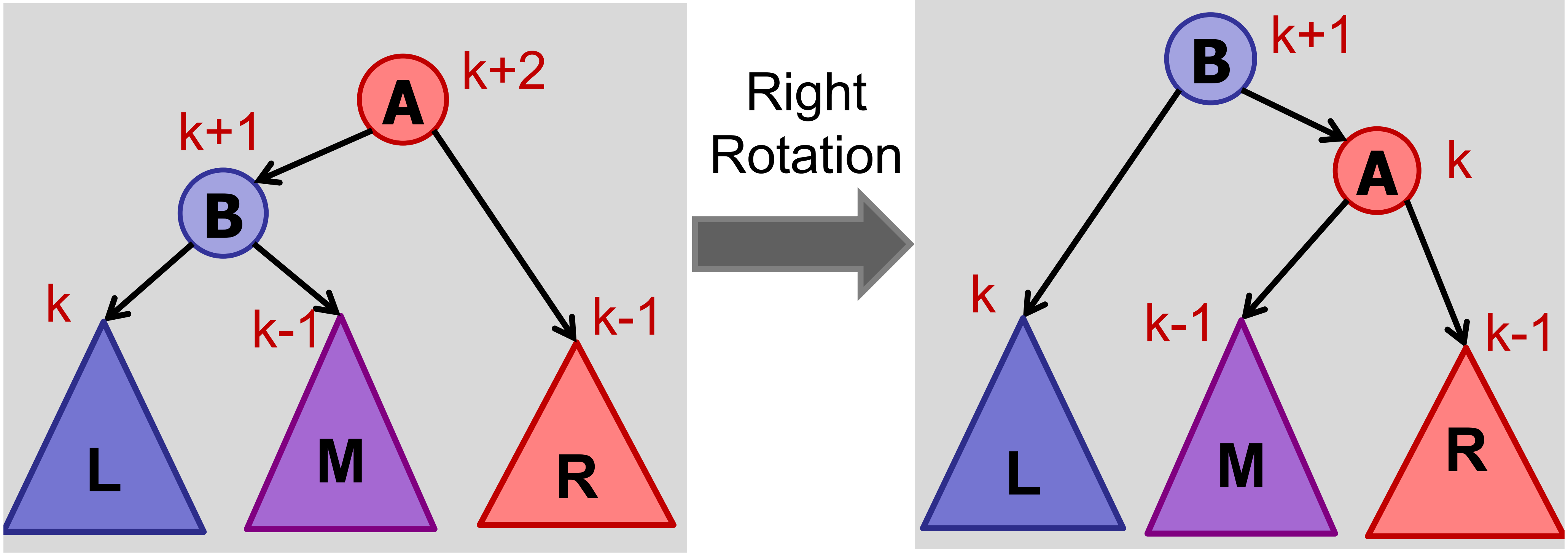
Tree Rotations



right-rotate:

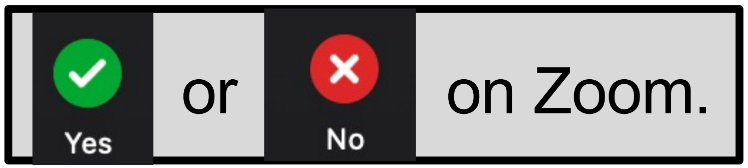
Case 2: **B** is left-heavy: h(**L**) = h(**M**) +1

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

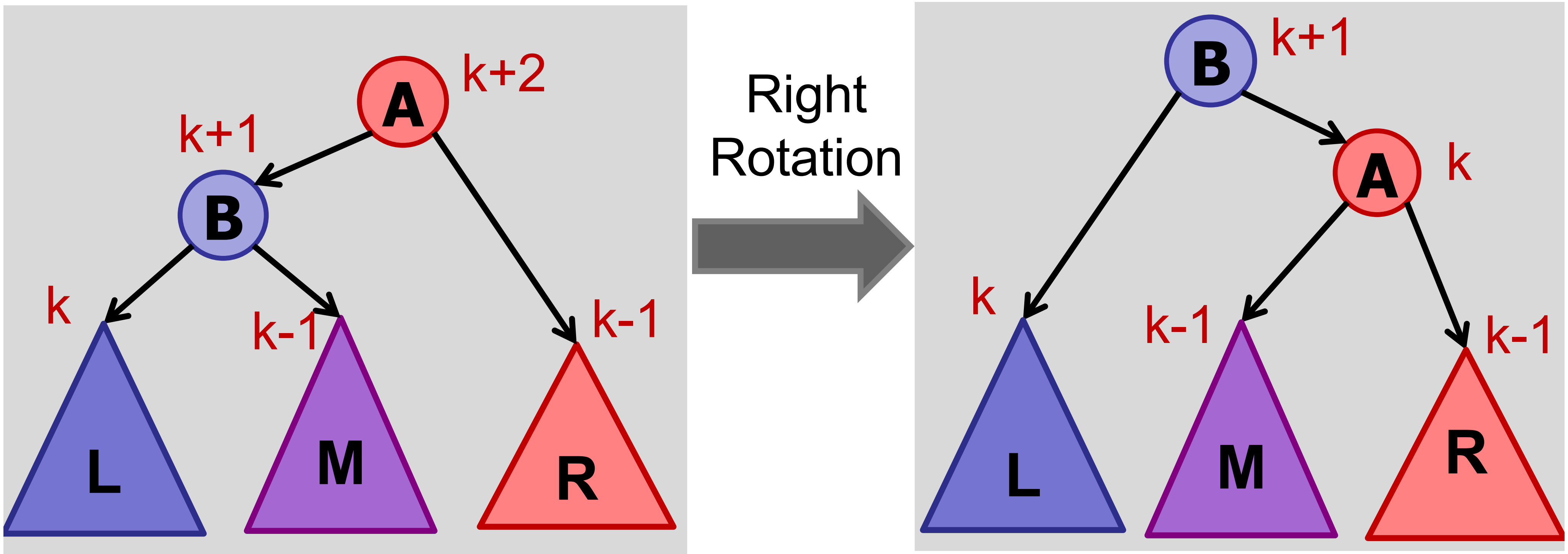


Is it balanced?

✓ 1. Yes.
2. No.
3. Maybe.



Tree Rotations

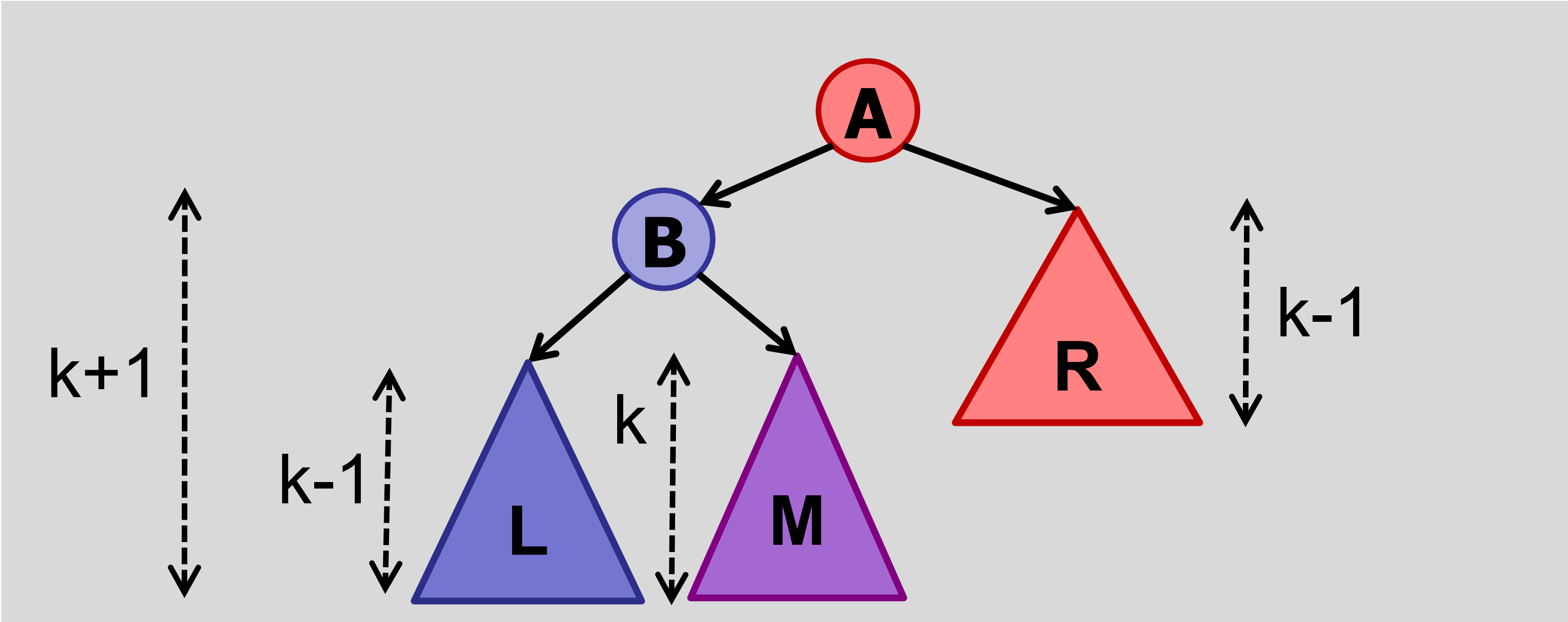


right-rotate:

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

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

Tree Rotations (Left Heavy)

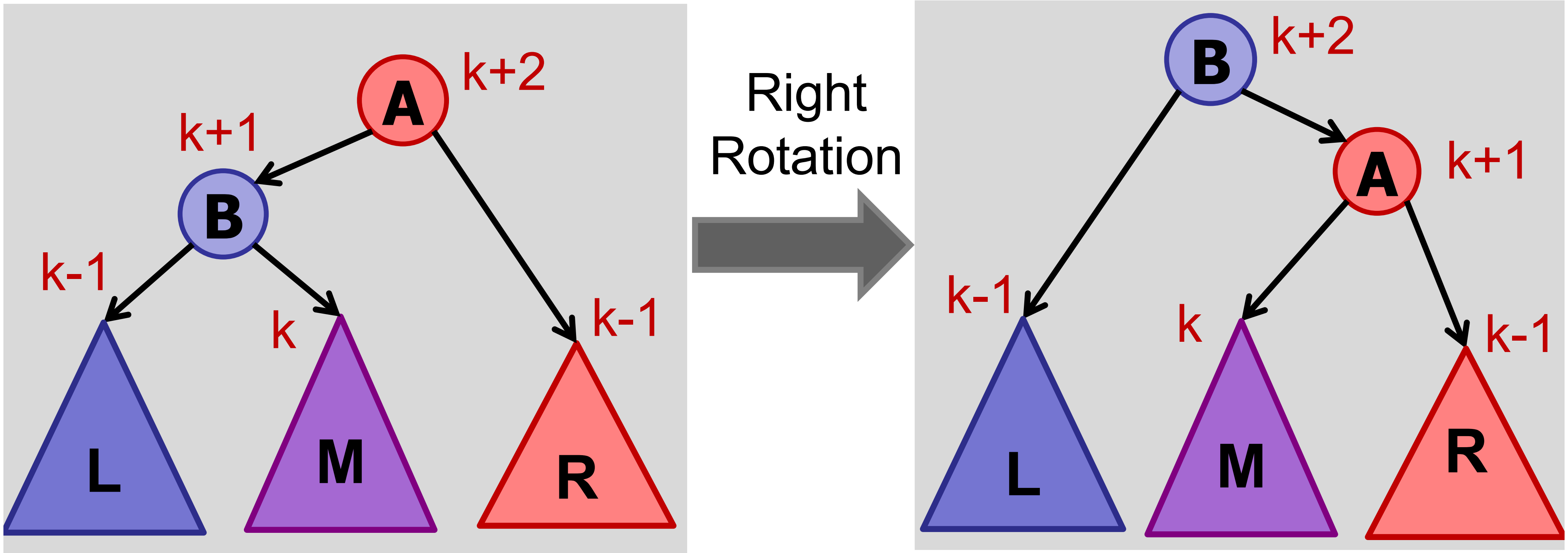


Assume **A** is the lowest node in the tree violating balance property.

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

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

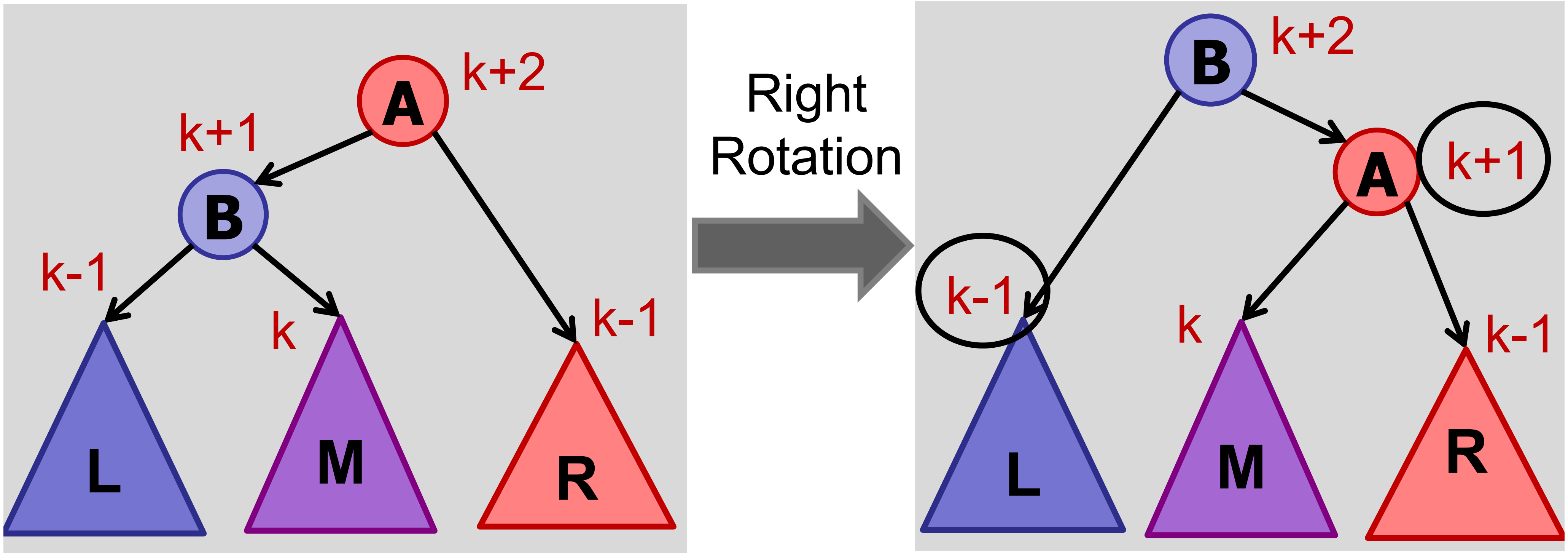
Tree Rotations



right-rotate:

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

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

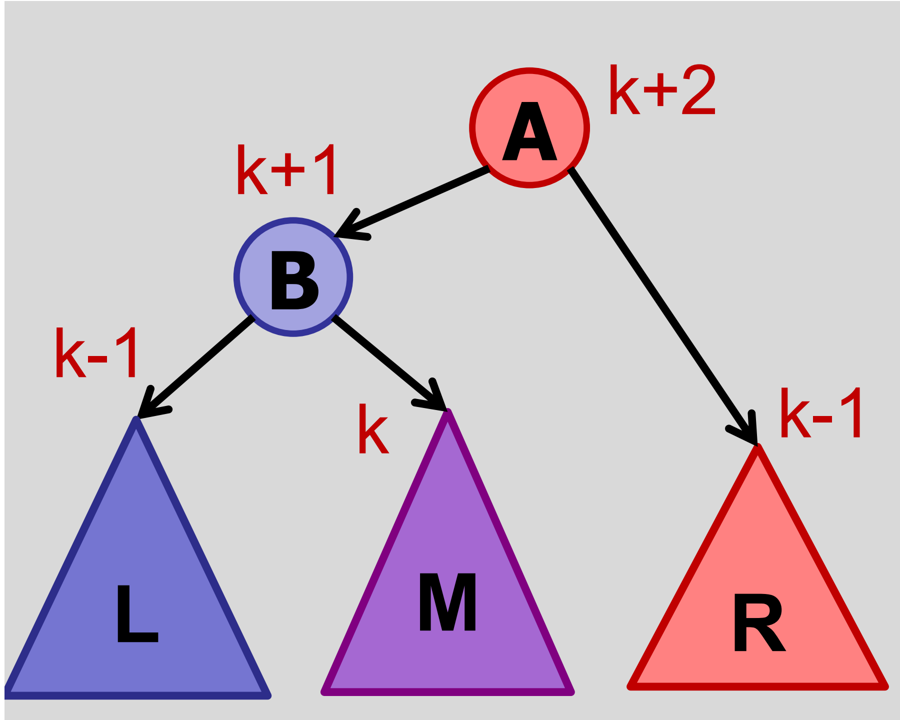


Is it balanced?

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

Yes or No on Zoom.

Tree Rotations



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

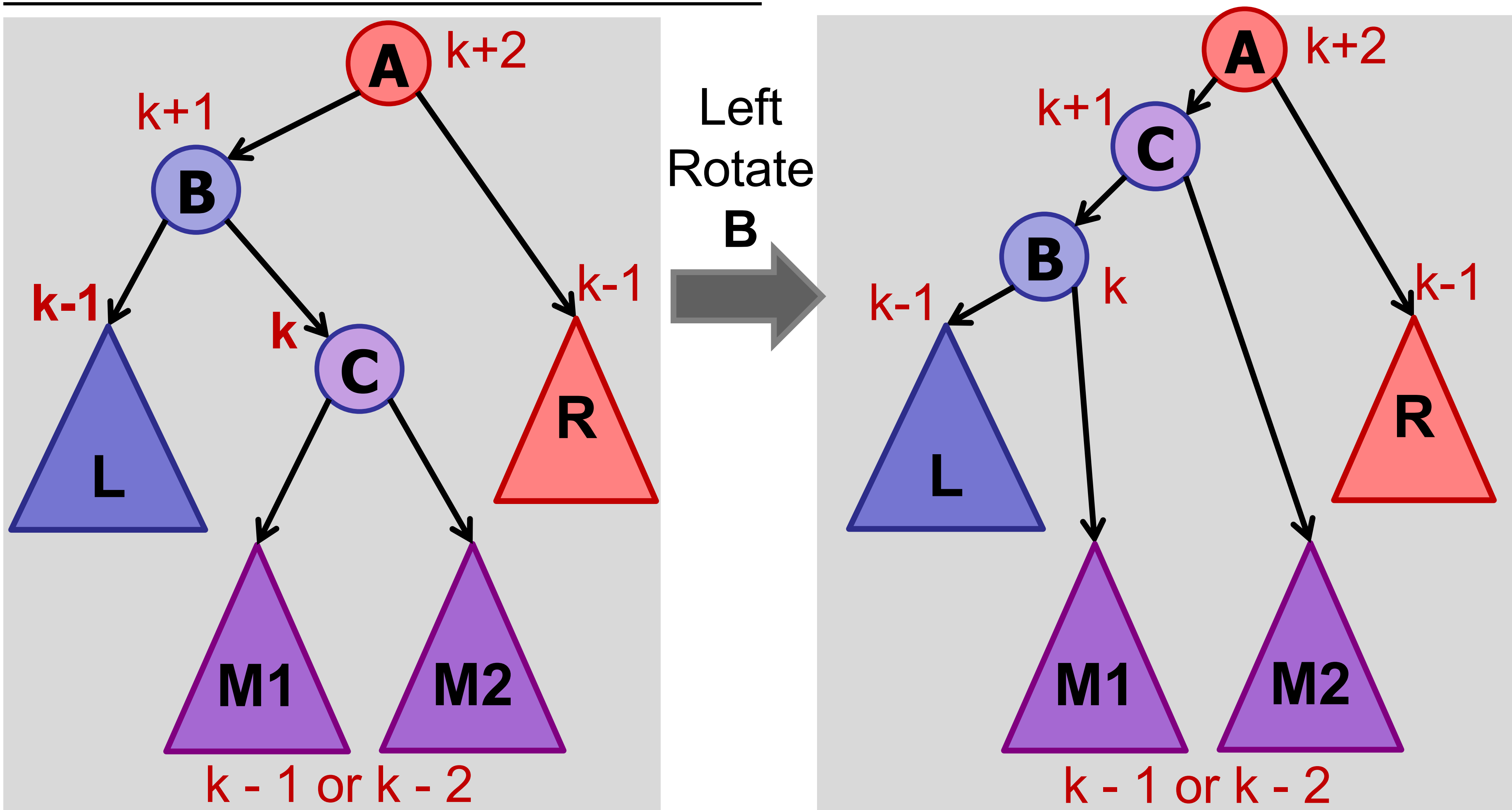
(Reduce it to a problem we have already solved!)

right-rotate:

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

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

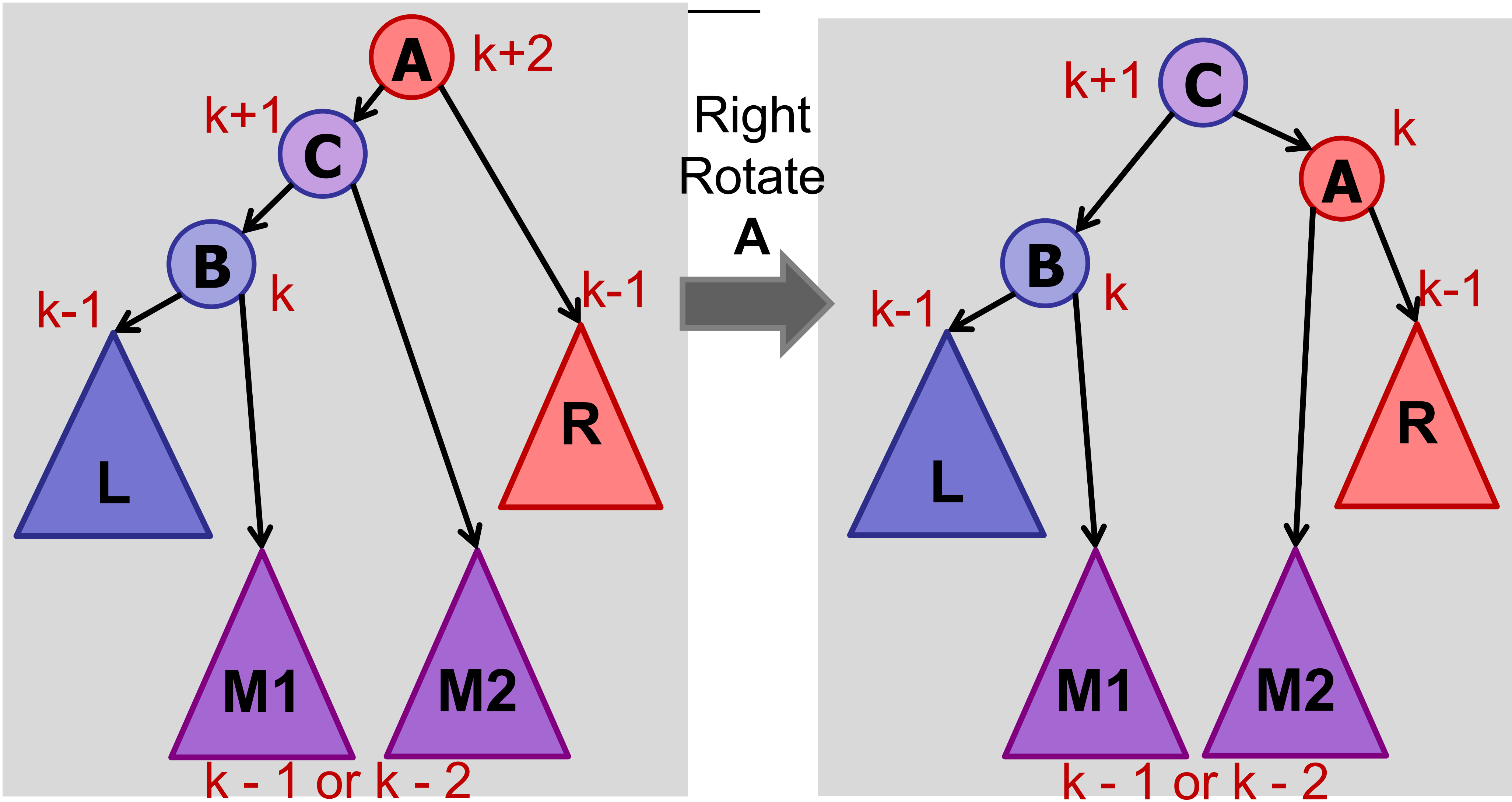
Tree Rotations

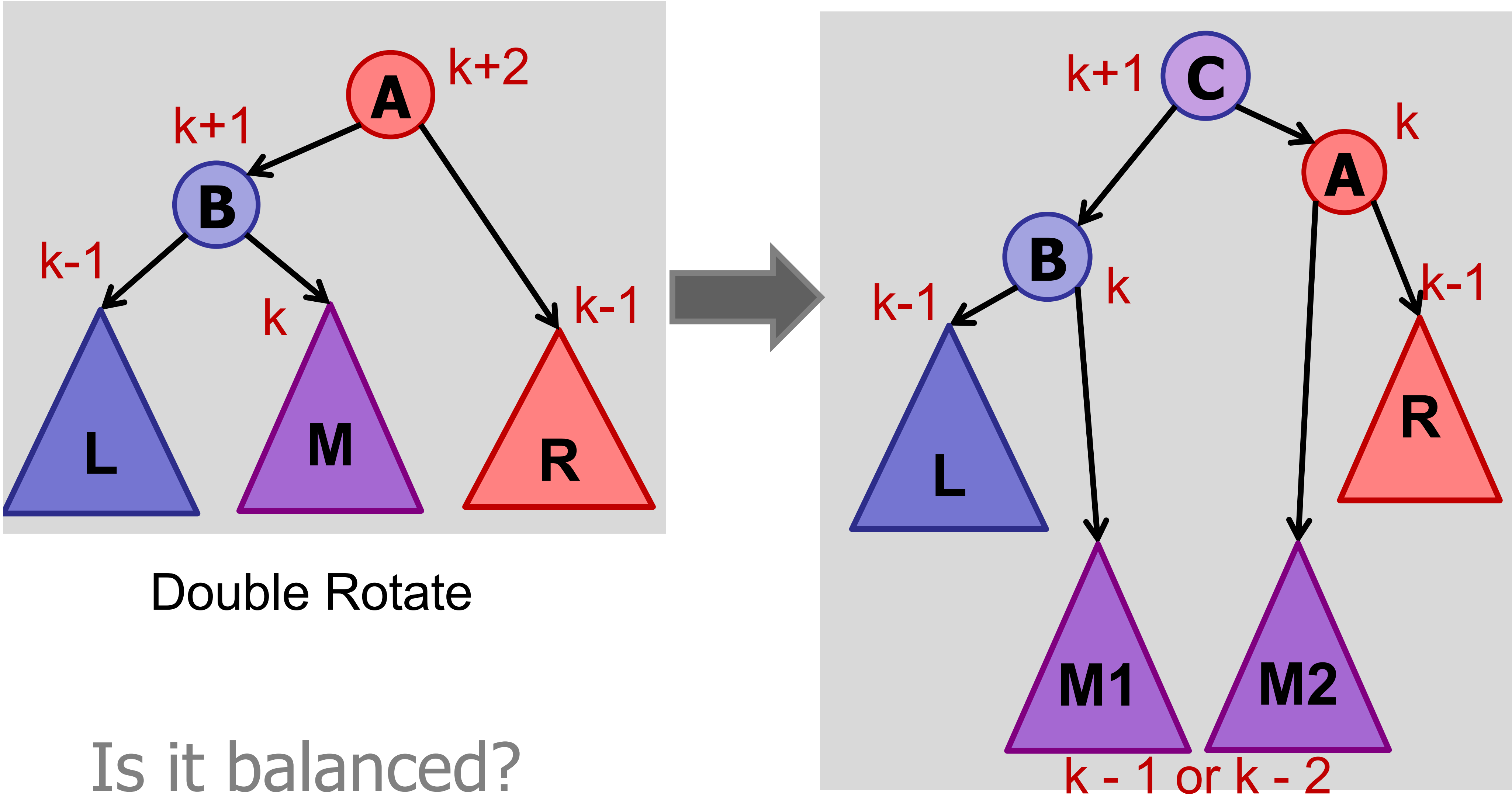


Left-rotate B

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

Tree Rotations

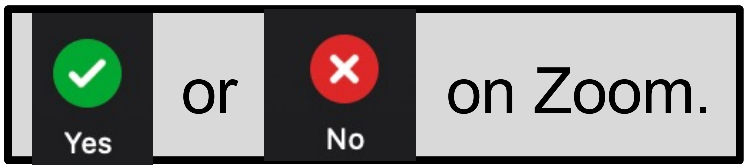




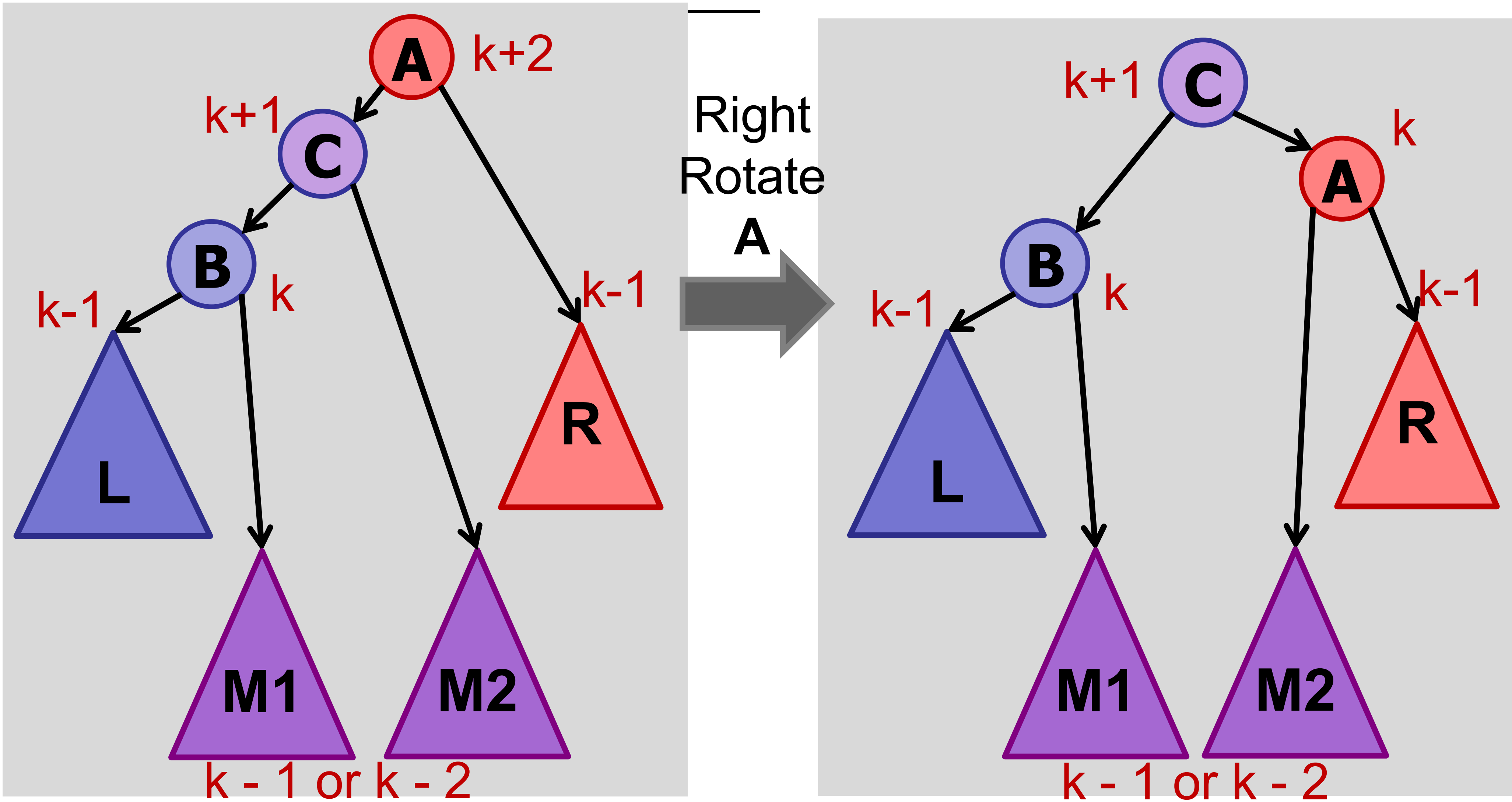
Double Rotate

Is it balanced?

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



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

ARCHIPELAGO

is open

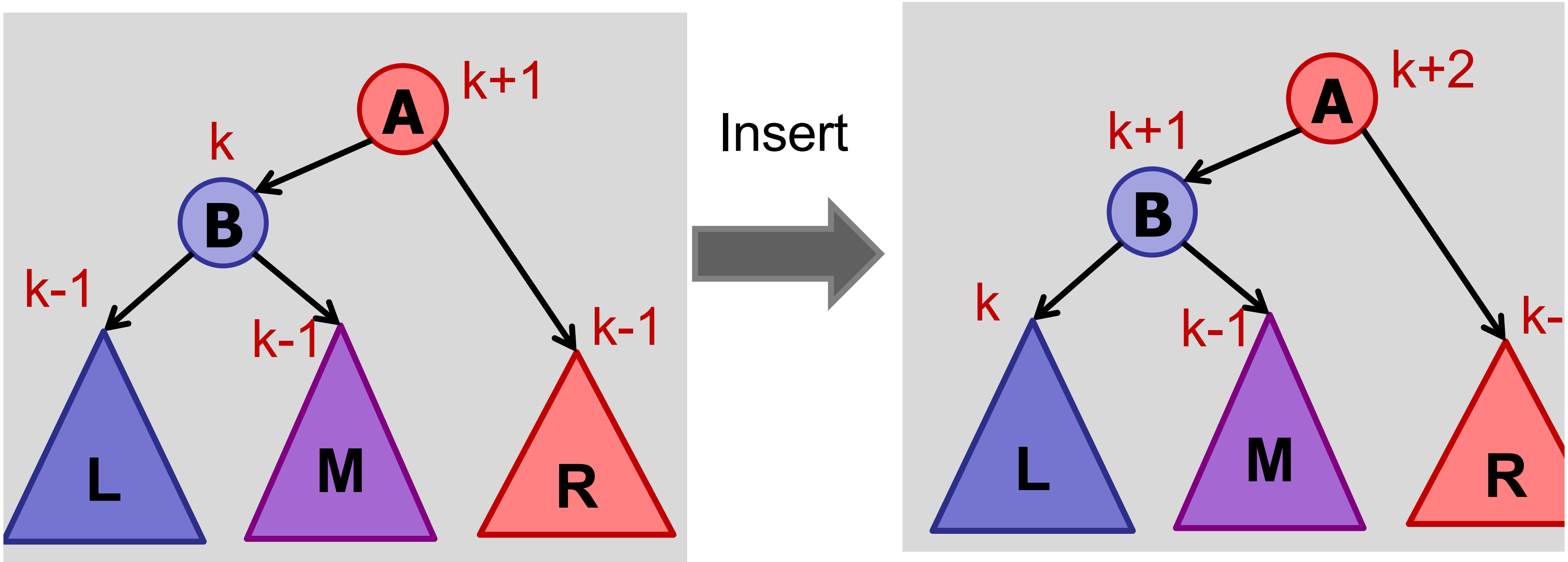
How many rotations do you need after an insertion (in the worst case)?

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

Question:

Why isn't it 2log(n)?

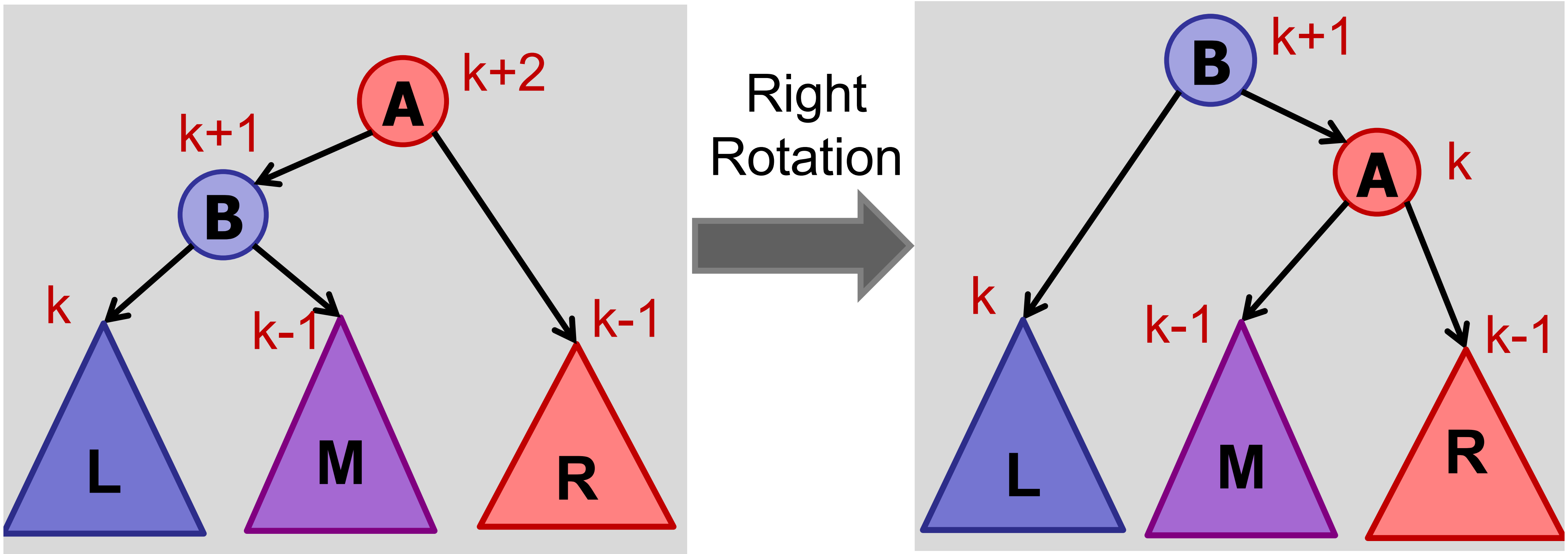
How many rotations?



Case 2: **B** is left-heavy

Insert increased heights by 1.

How many rotations?

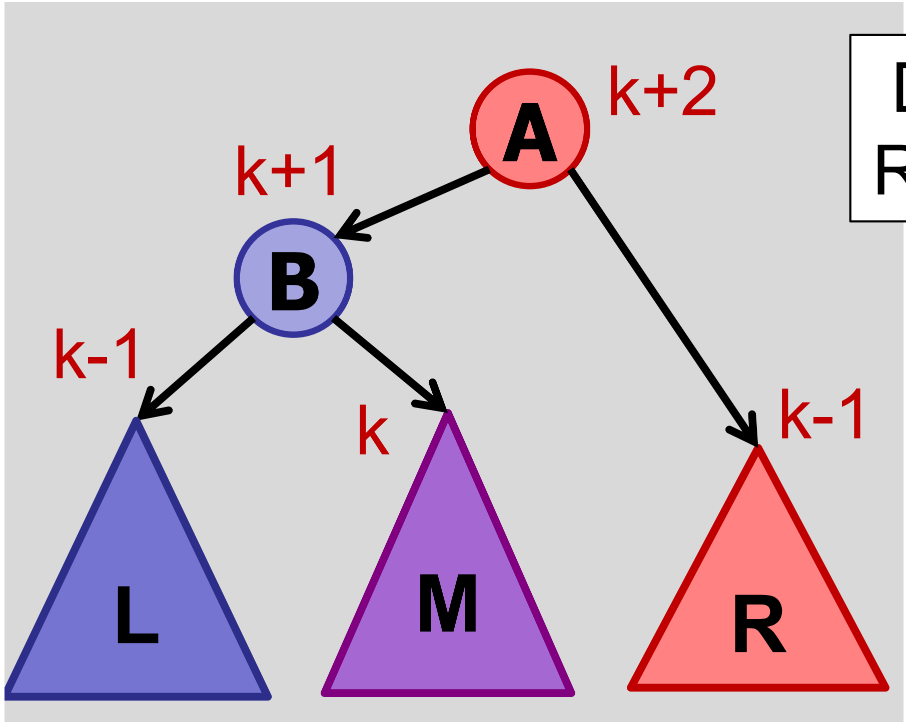


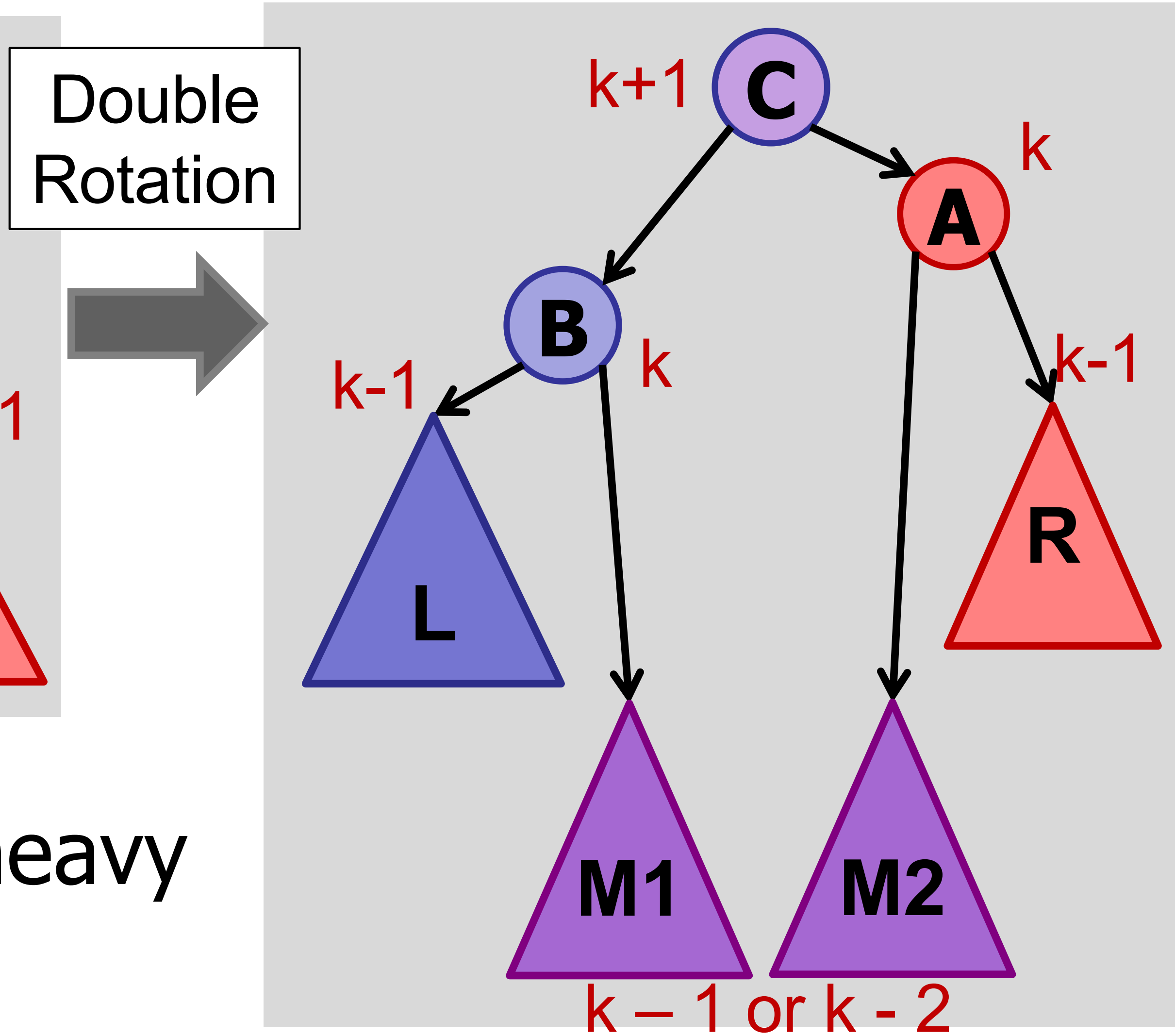
Case 2: **B** is left-heavy

Rotation reduces root height by 1.

(Everything higher in tree is unchanged!)

How many rotations?

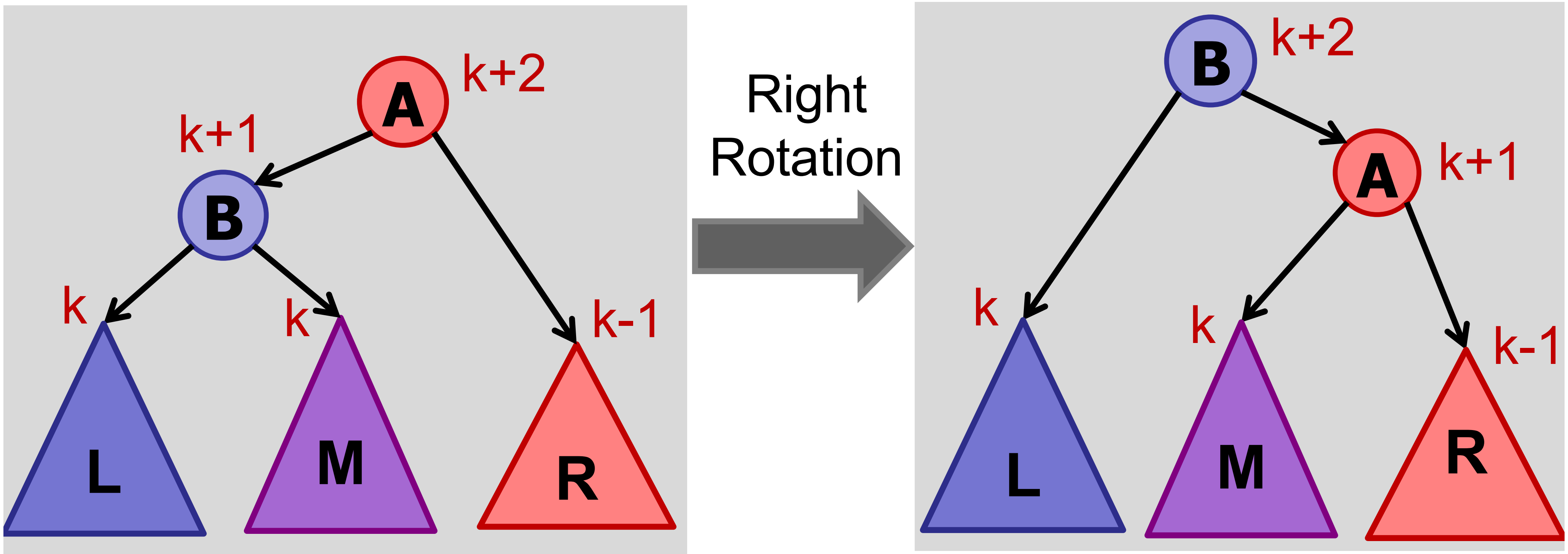




Case 3: **B** is right-heavy

Rotation reduces root height by 1.

How many rotations?



Case 1: **B** is balanced

Rotation does *not* reduce height by 1.

Challenge: figure out why this is okay!

Insert in AVL Tree

Summary:

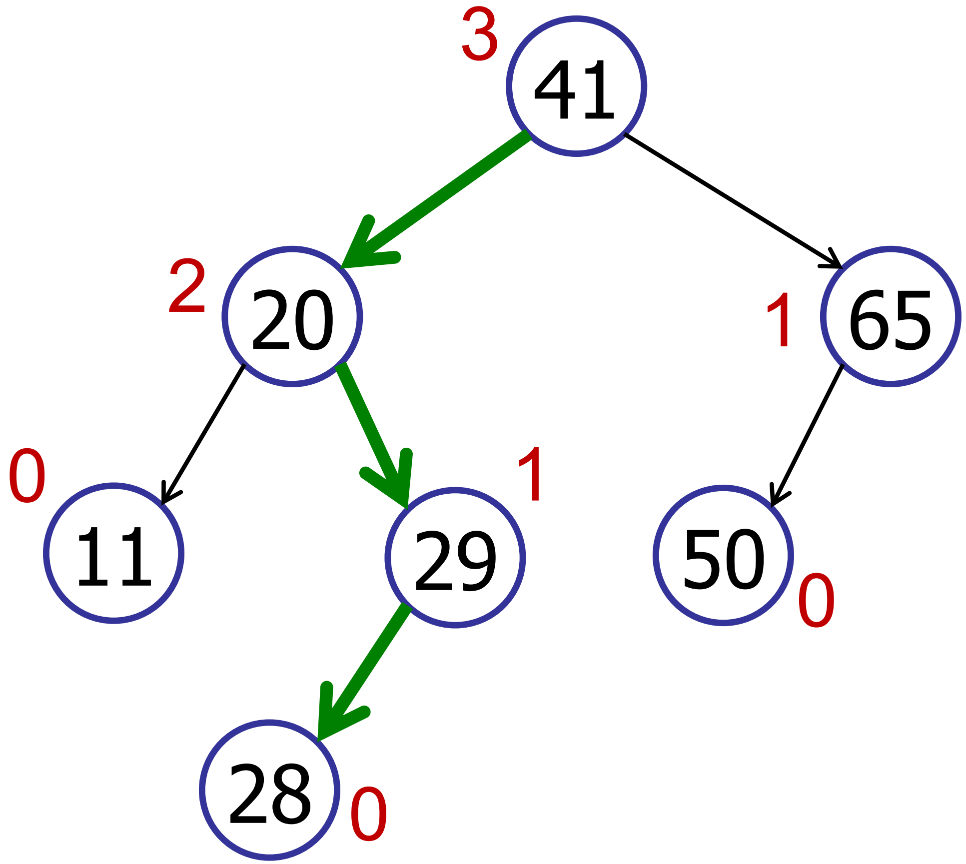
- Insert key in BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance and return.

Key observation:

- Only need to fix *lowest* out-of-balance node.
- Only need at most two rotations to fix.

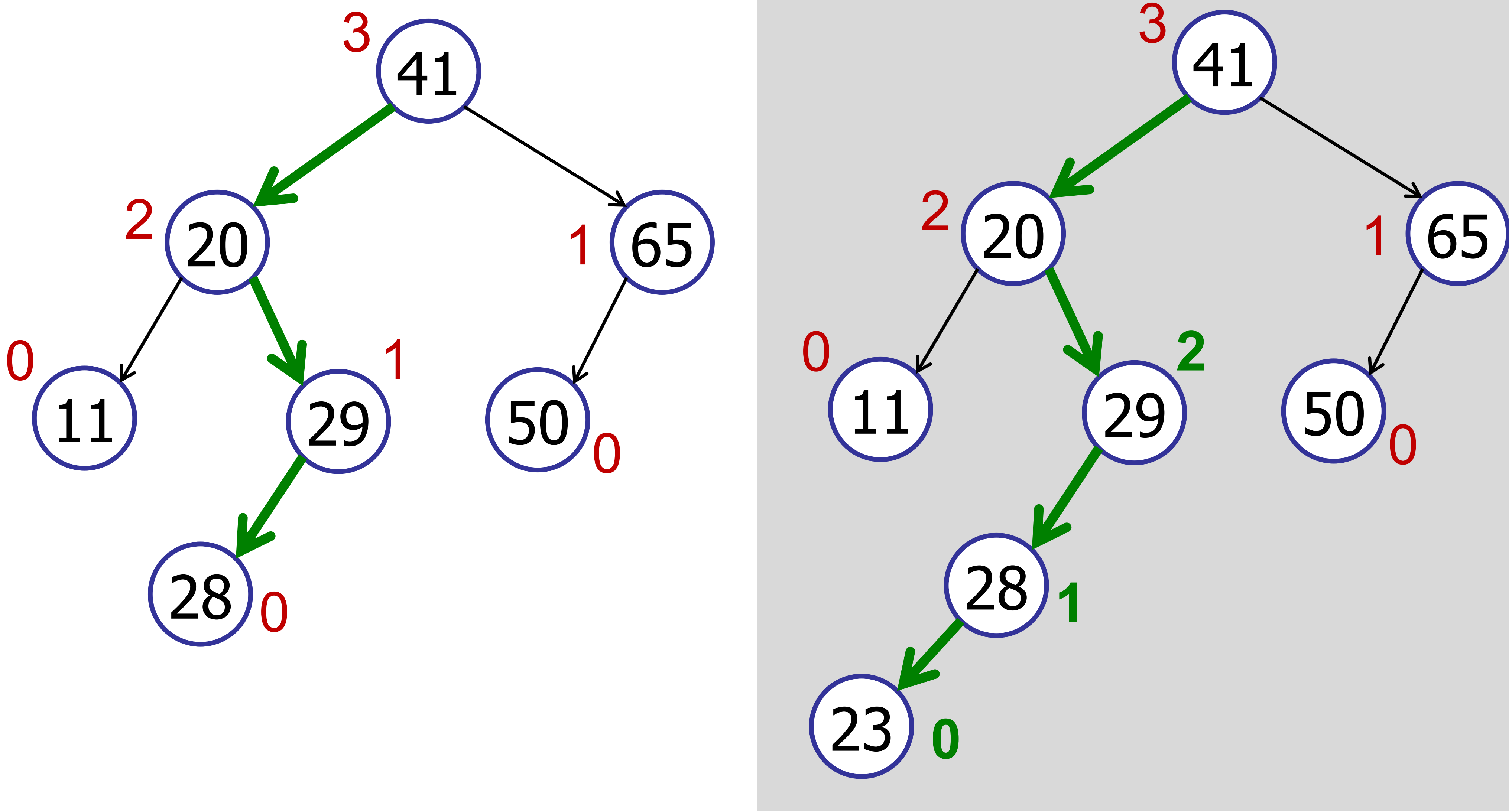
Example

insert(23)



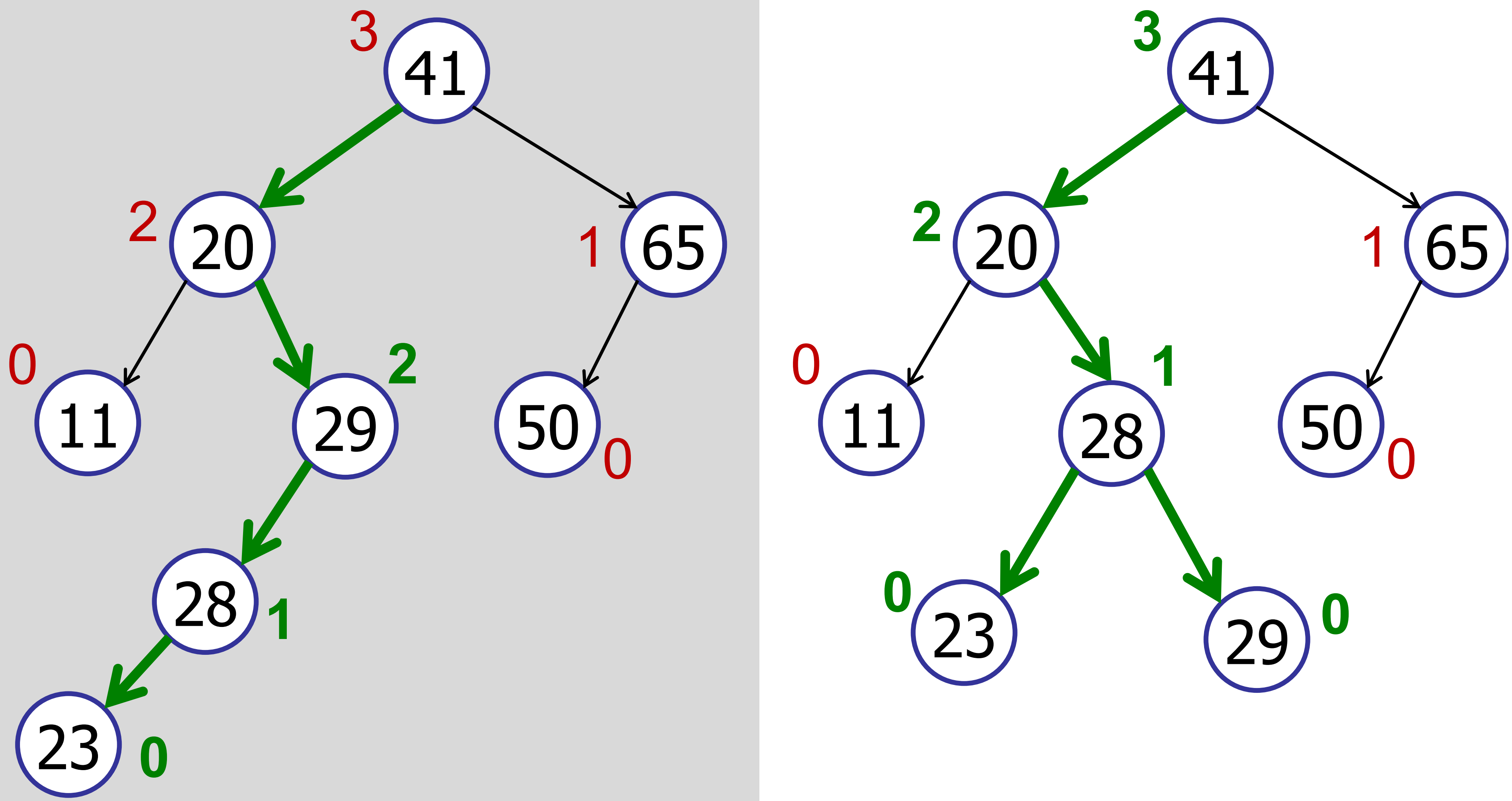
Example

insert(23)



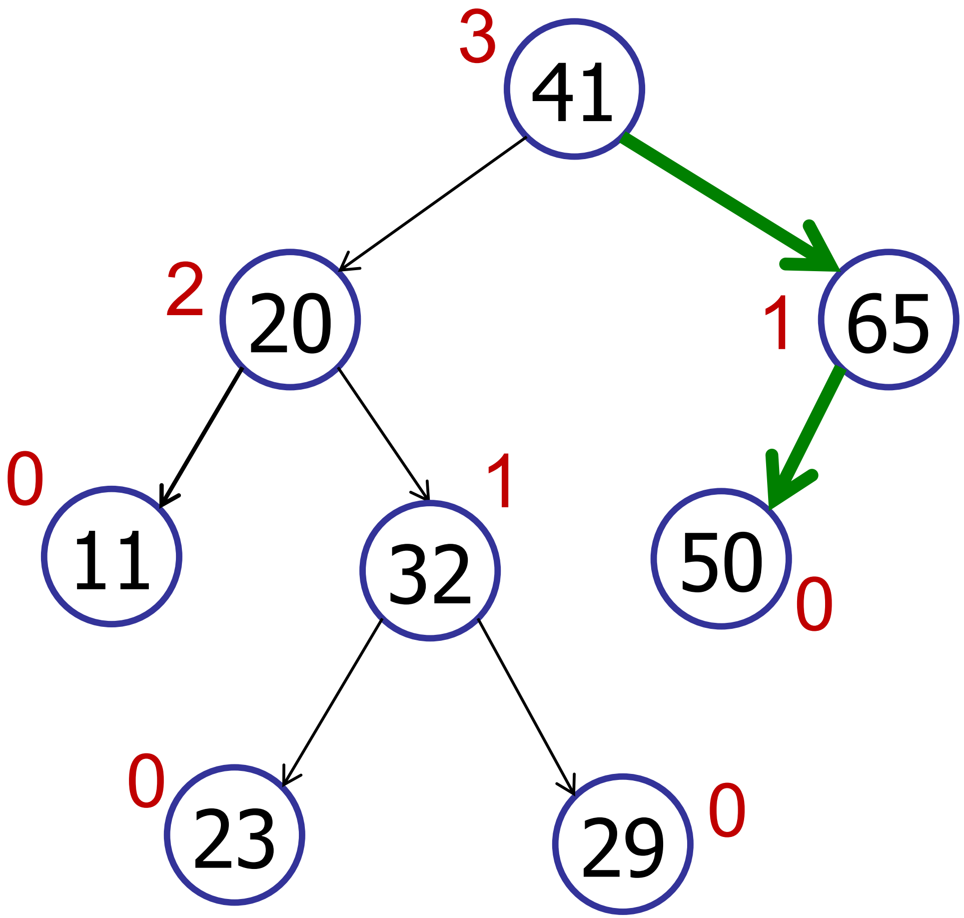
Example

right-rotate(29)



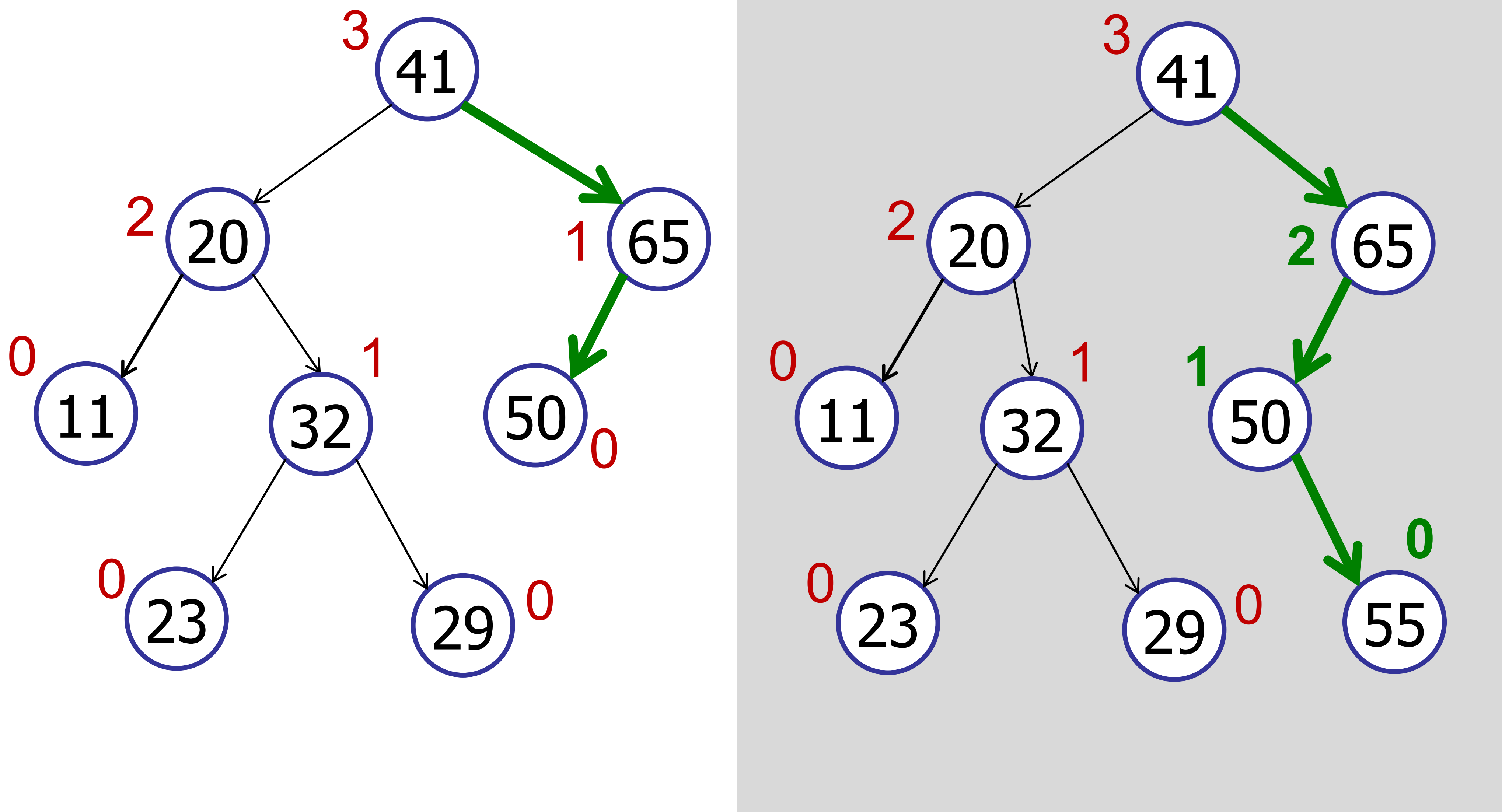
Example

insert(55)



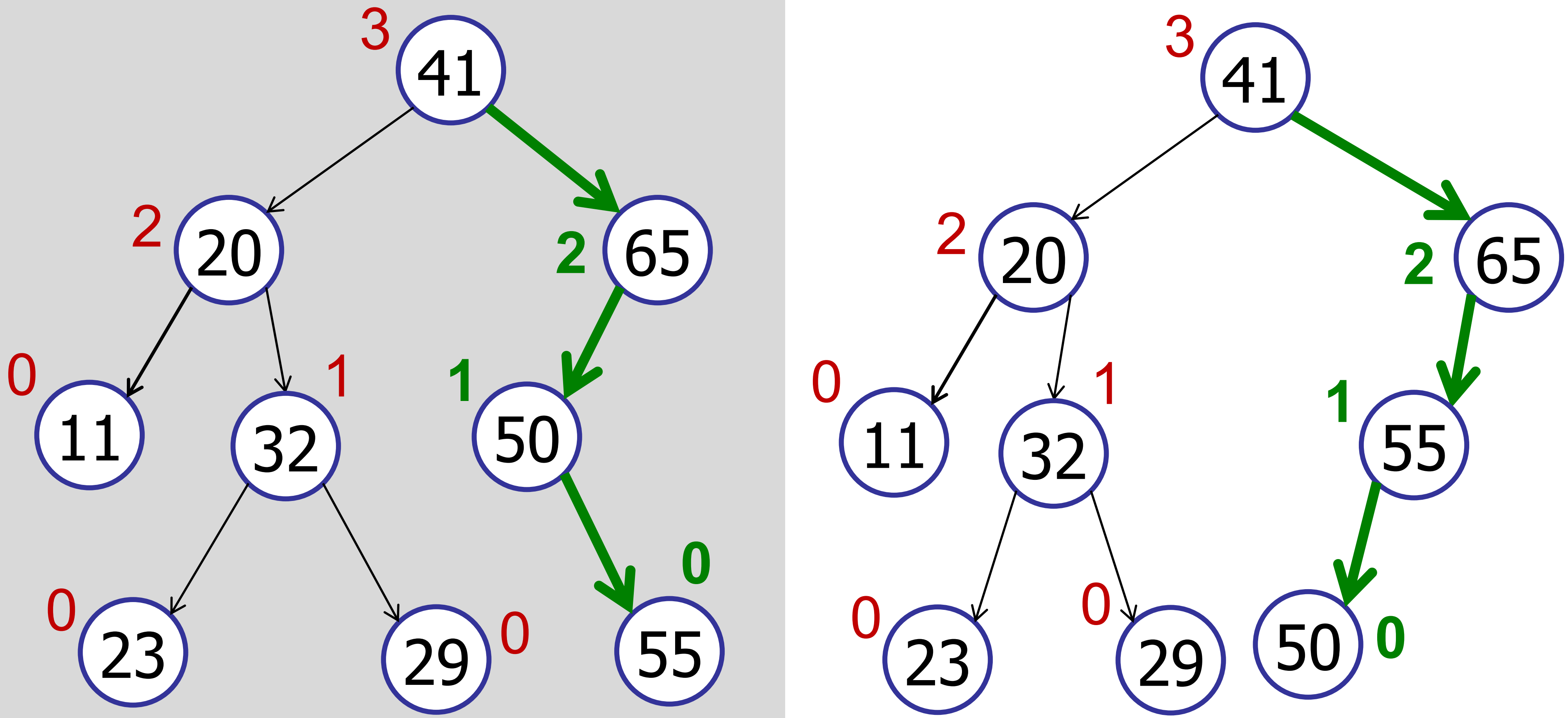
Example

insert(55)



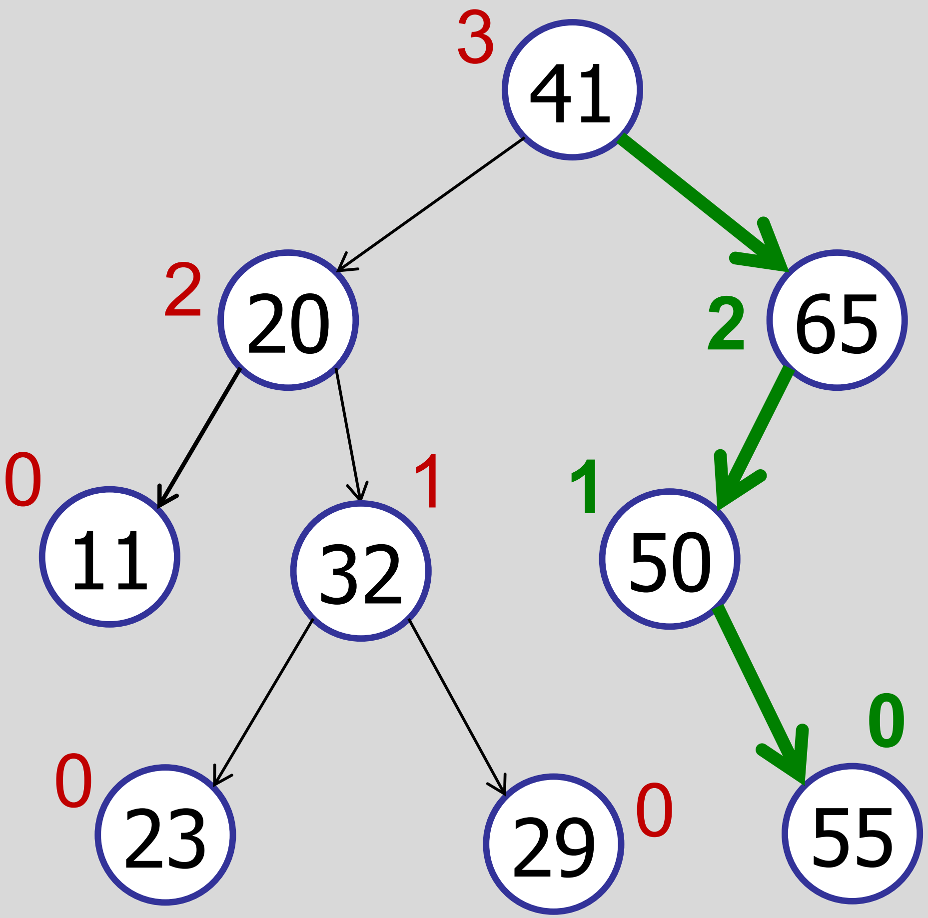
Example

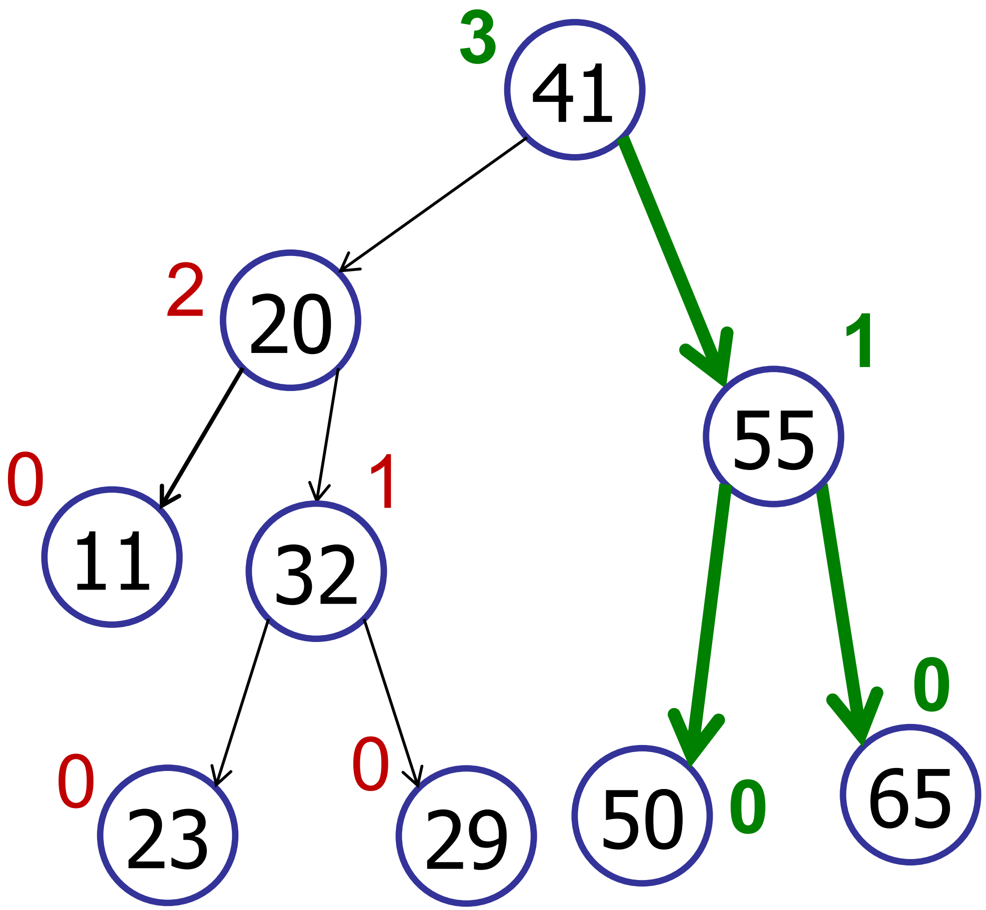
left-rotate(50)



Example

right-rotate(65)



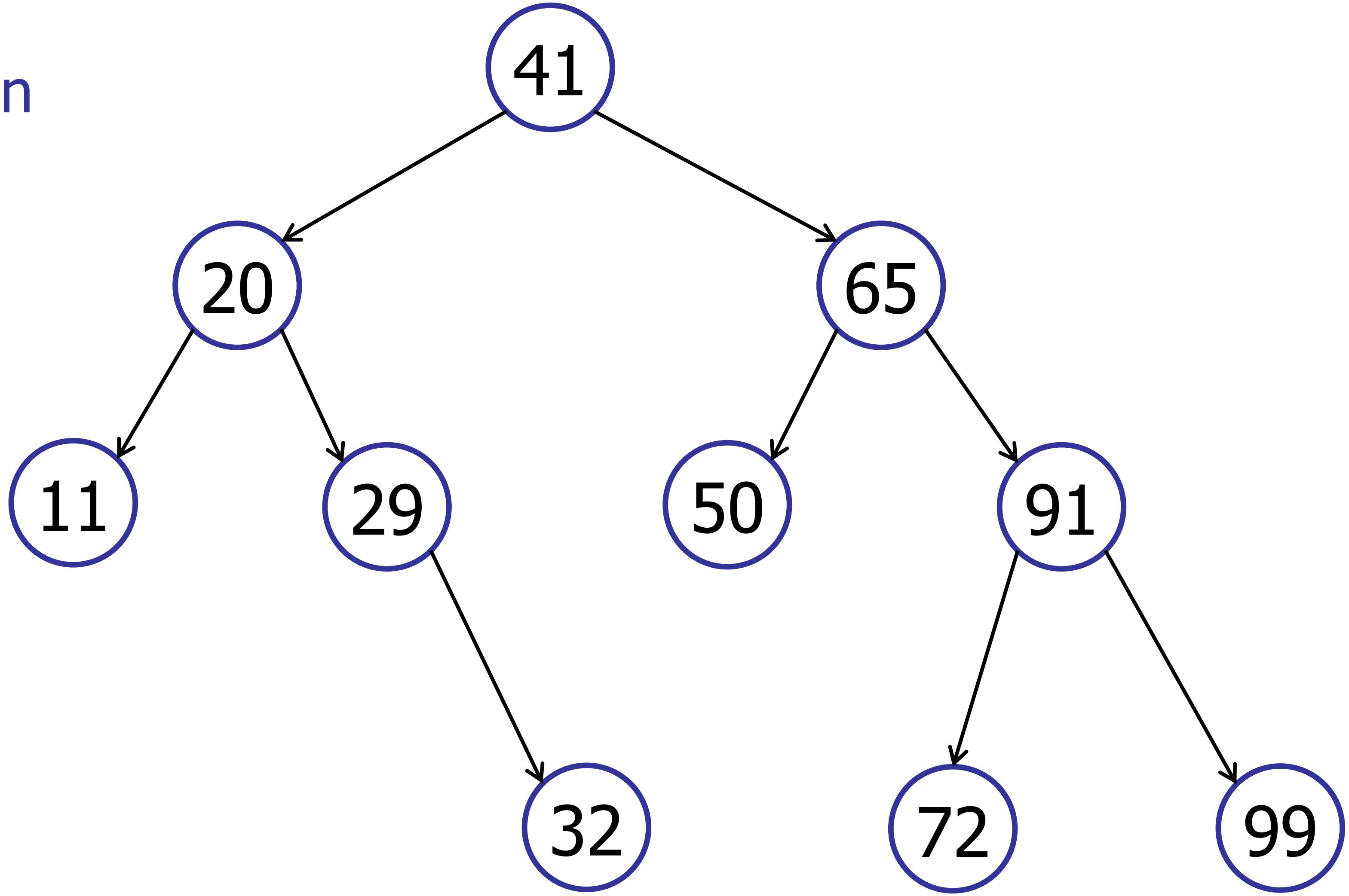


Binary Search Tree

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.

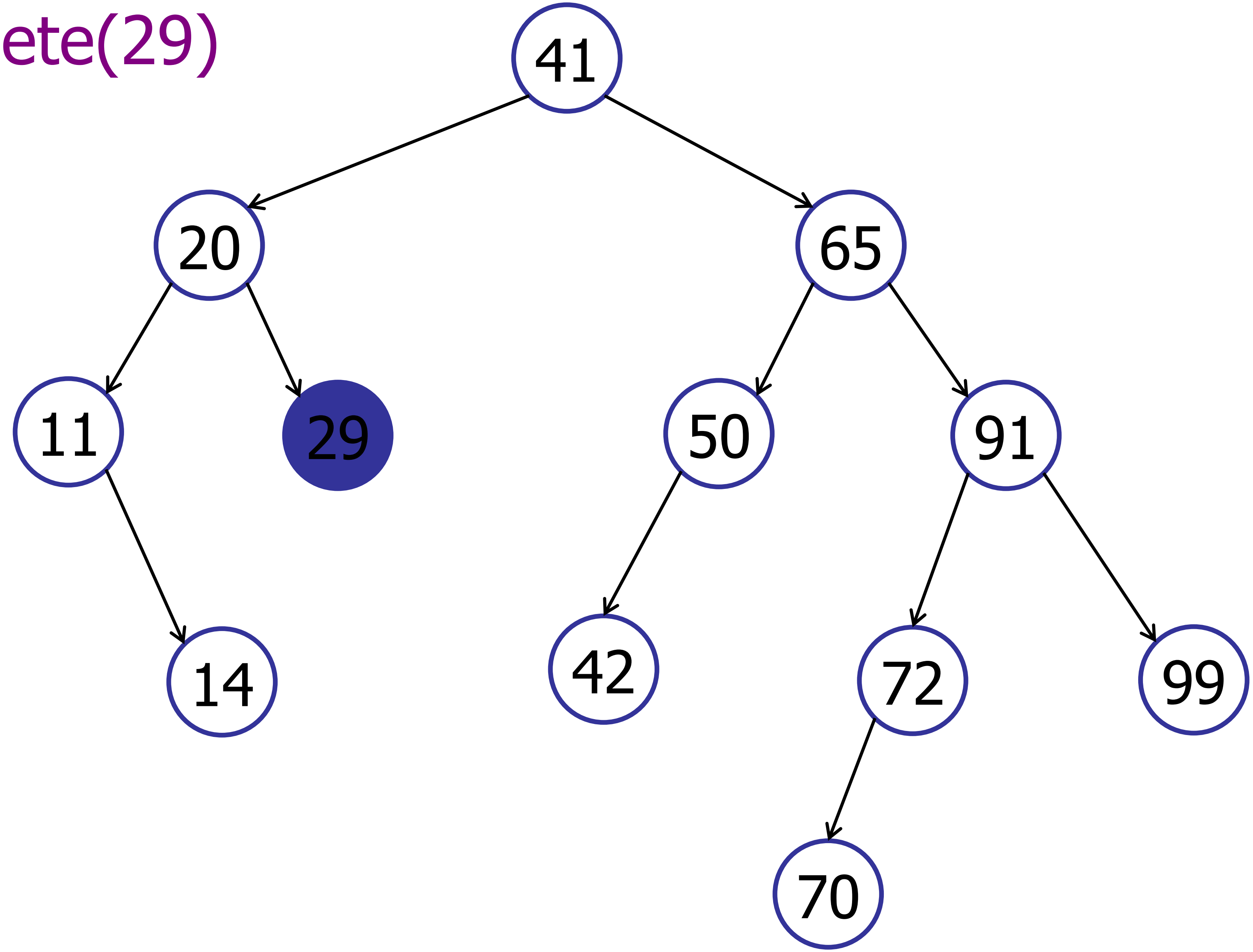
2. Delete node v from binary tree (and reconnect children).

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.

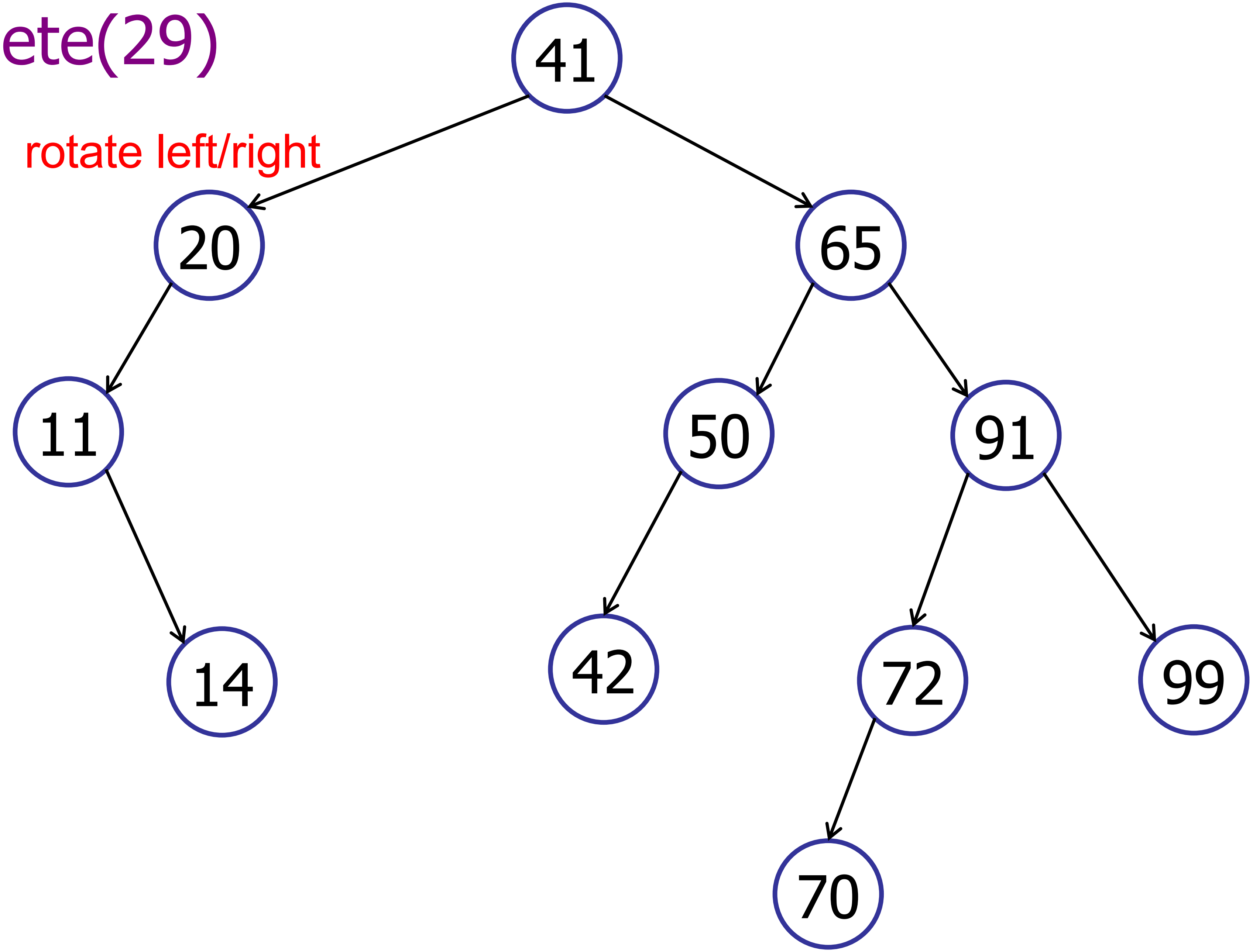
Binary Search Tree

delete(29)



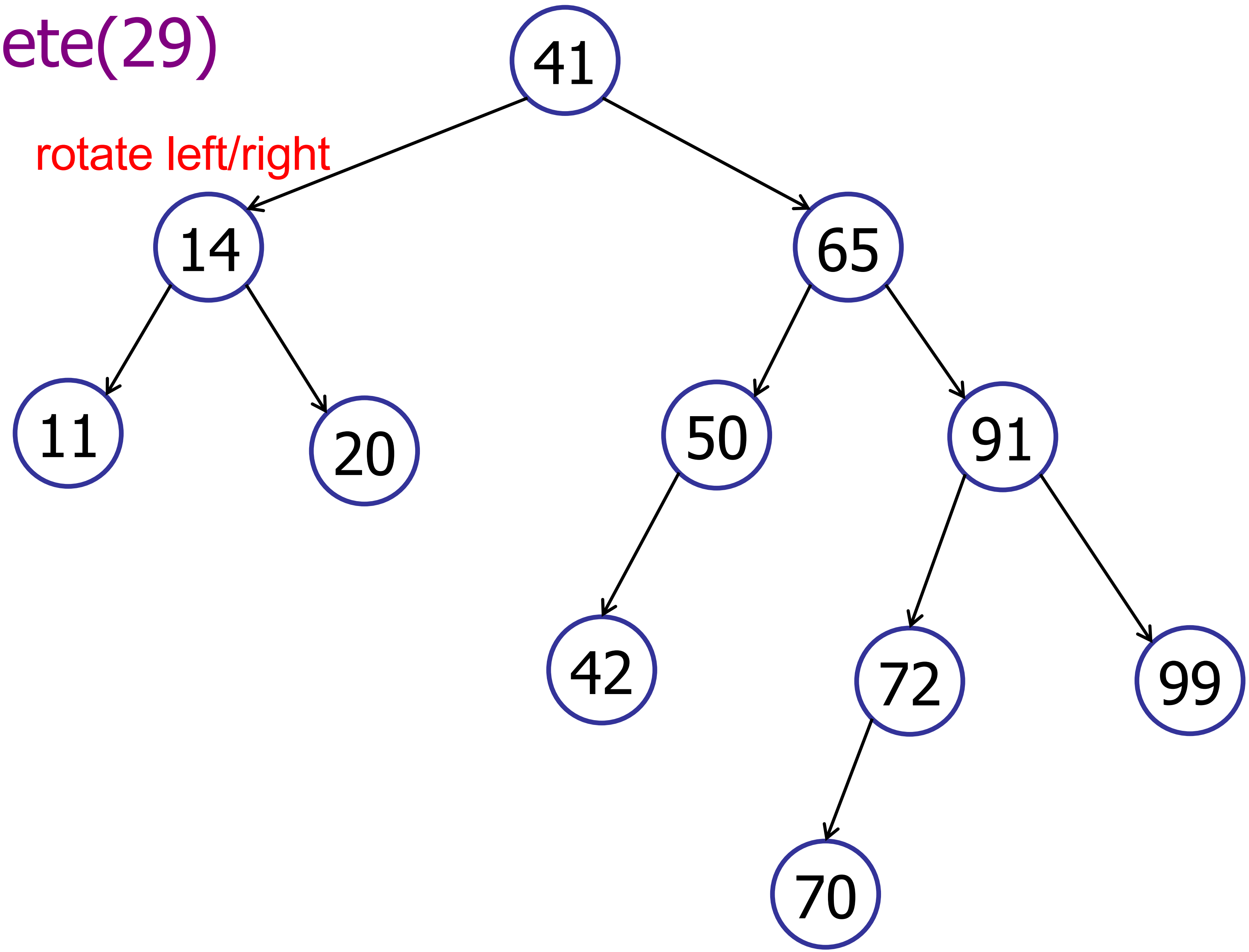
Binary Search Tree

delete(29)



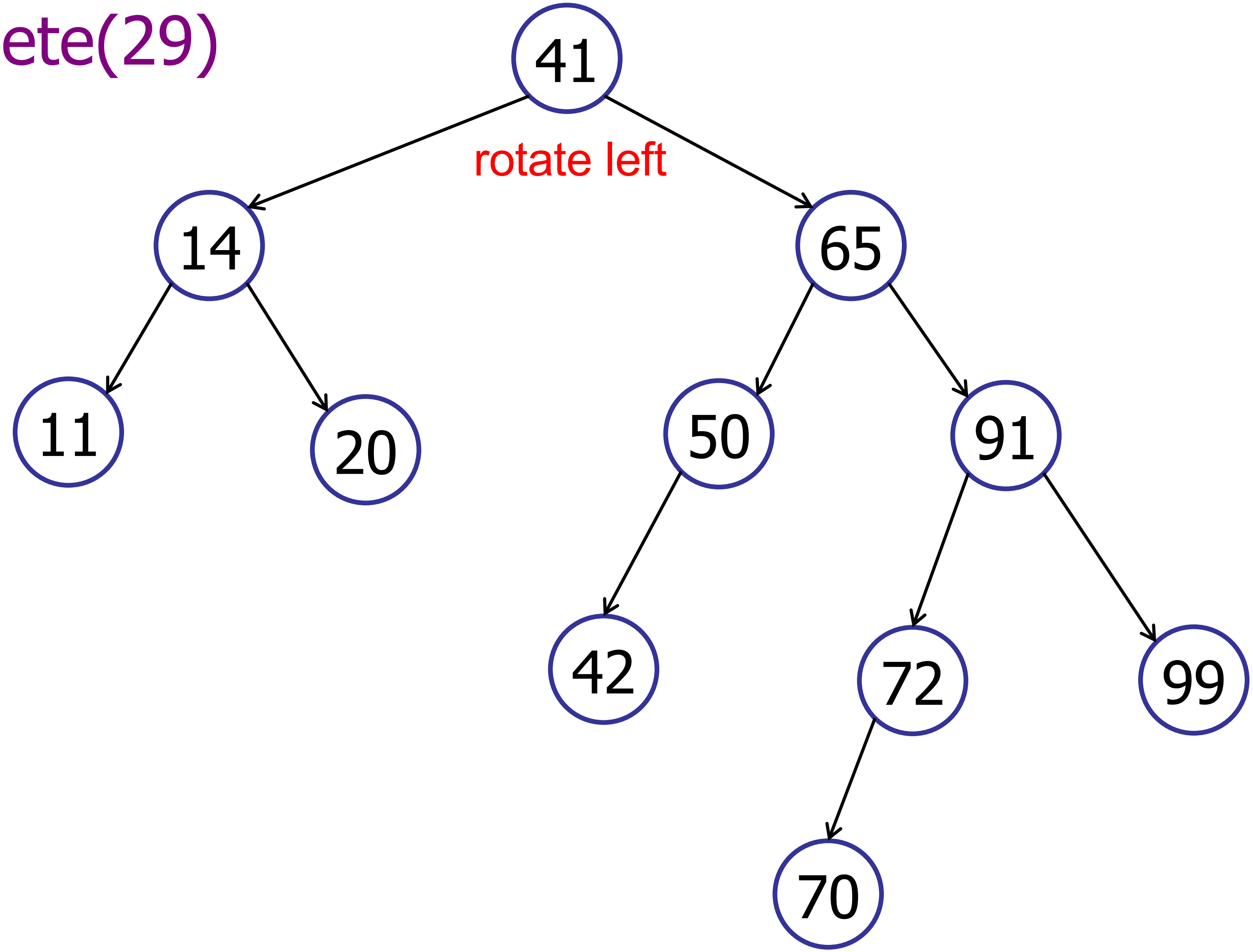
Binary Search Tree

delete(29)



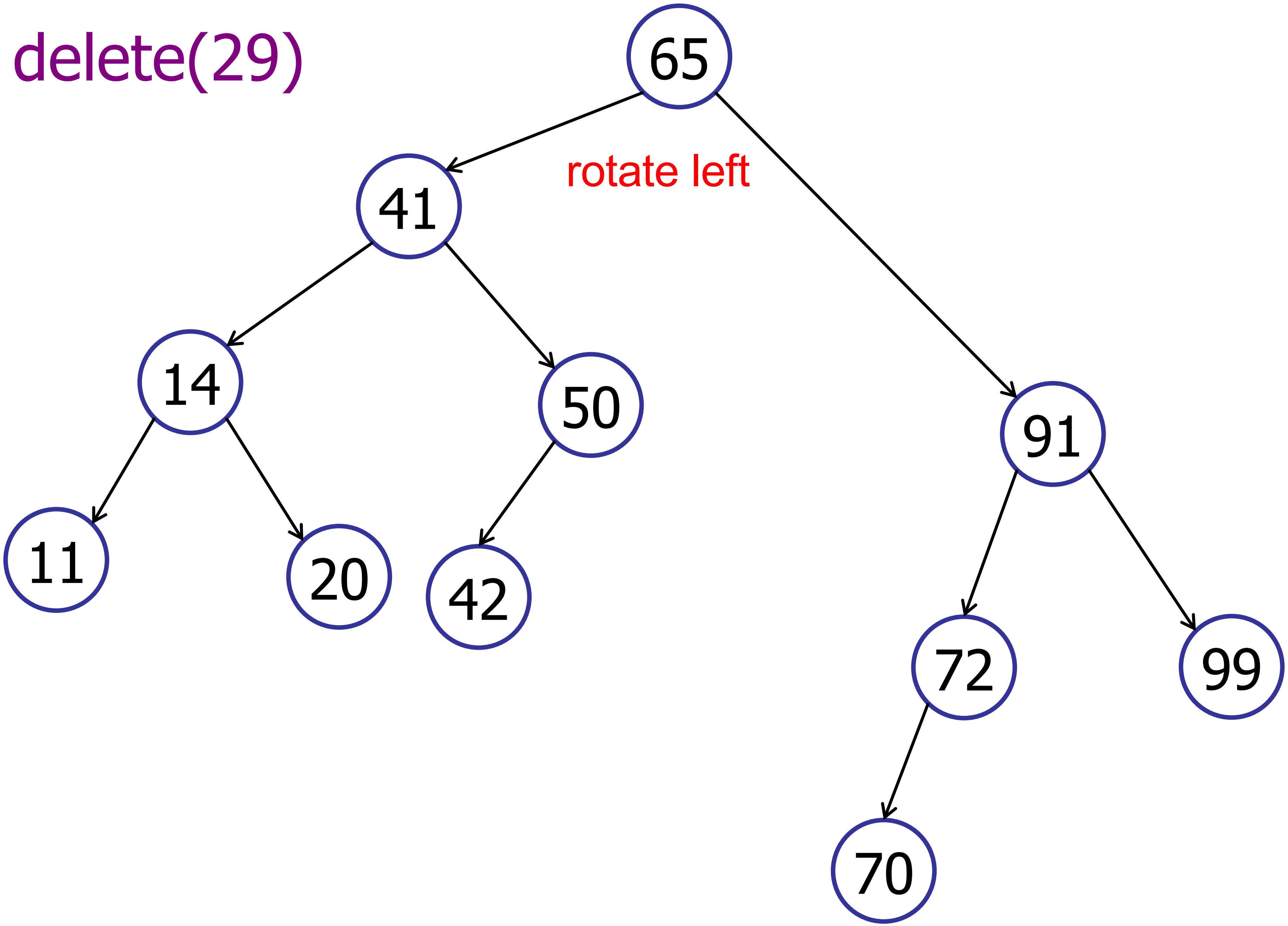
Binary Search Tree

delete(29)



Binary Search Tree

delete(29)



How many rebalances?

Why are two rotations not enough?

- Delete reduced height.
- Rotations (to rebalance) reduce height!

Key observation:

- Rebalancing does not “undo” the change in height caused by insertion.

Delete in AVL Tree

Summary:

- Delete key from BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance.
  - Continue to root.

Key observation:

- It is *not* sufficient to only fix lowest out-of-balance node in tree.

Every insertion requires 1 or 2 rotations?

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

ARCHIPELAGO

is open

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

ARCHIPELAGO

is open

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

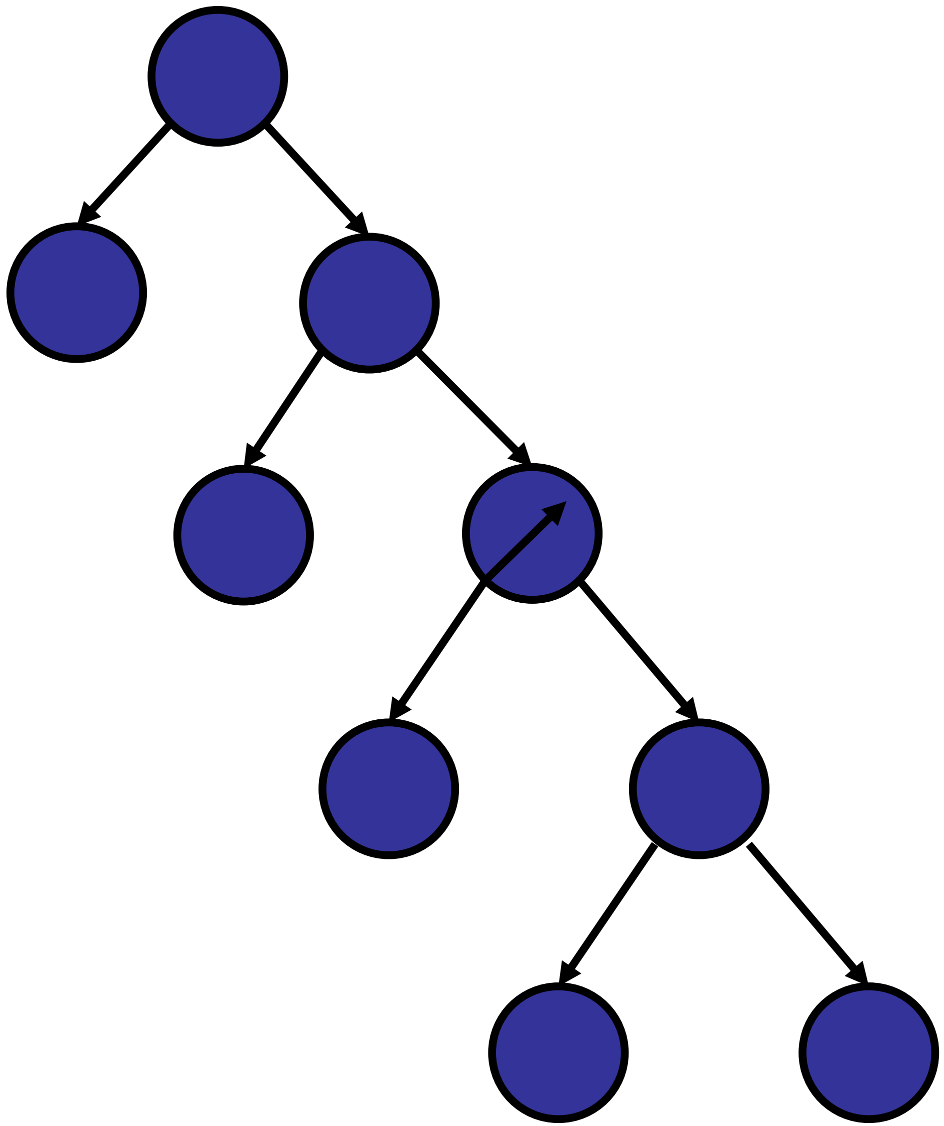
1. Yes
2. No
3. I don't know

ARCHIPELAGO

is open

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

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

ARCHIPELAGO

is open

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

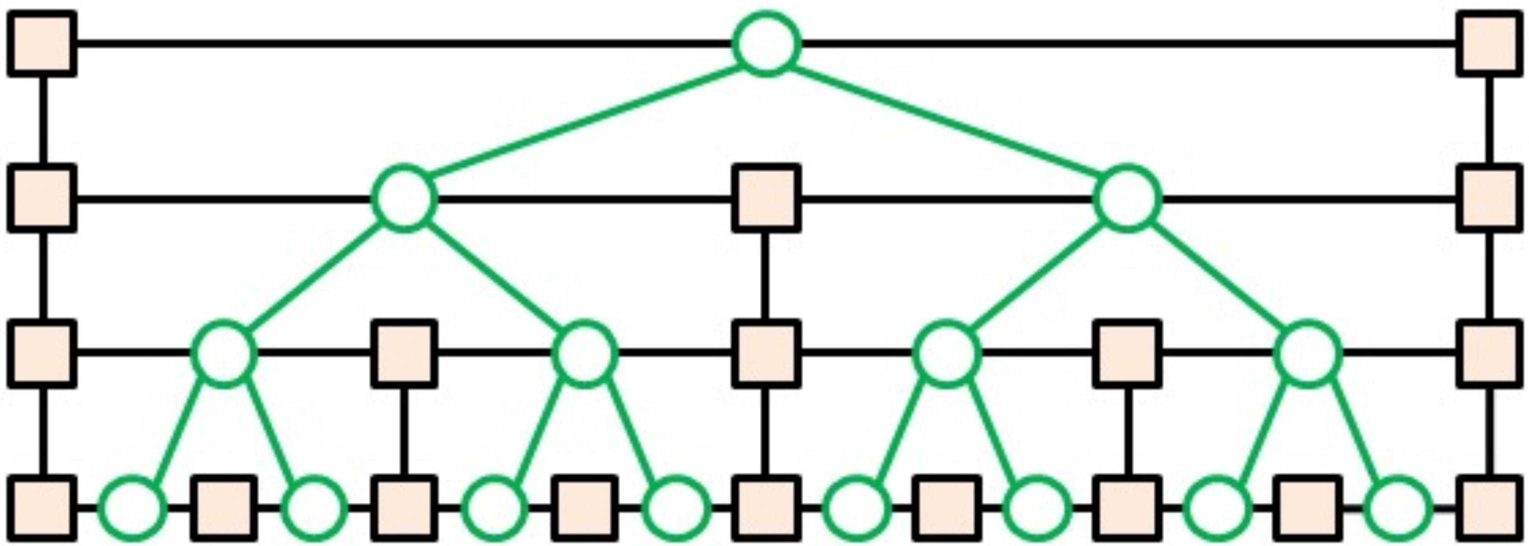
Red-Black trees

- 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



CS2040S
Data Structures and Algorithms

On the importance of being balanced
(Act 2)

Puzzle of the Week:

100 prisoners. Every so often, one is chosen at random to enter a room with a light bulb. You can turn the light bulb on or off.

- WIN if one prisoner announces correctly that all have visited the room.
- LOSE if announcement is incorrect.

What if, initially, the state of the light is unknown, either on or off?

Todays Plan

On the importance of being balanced

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

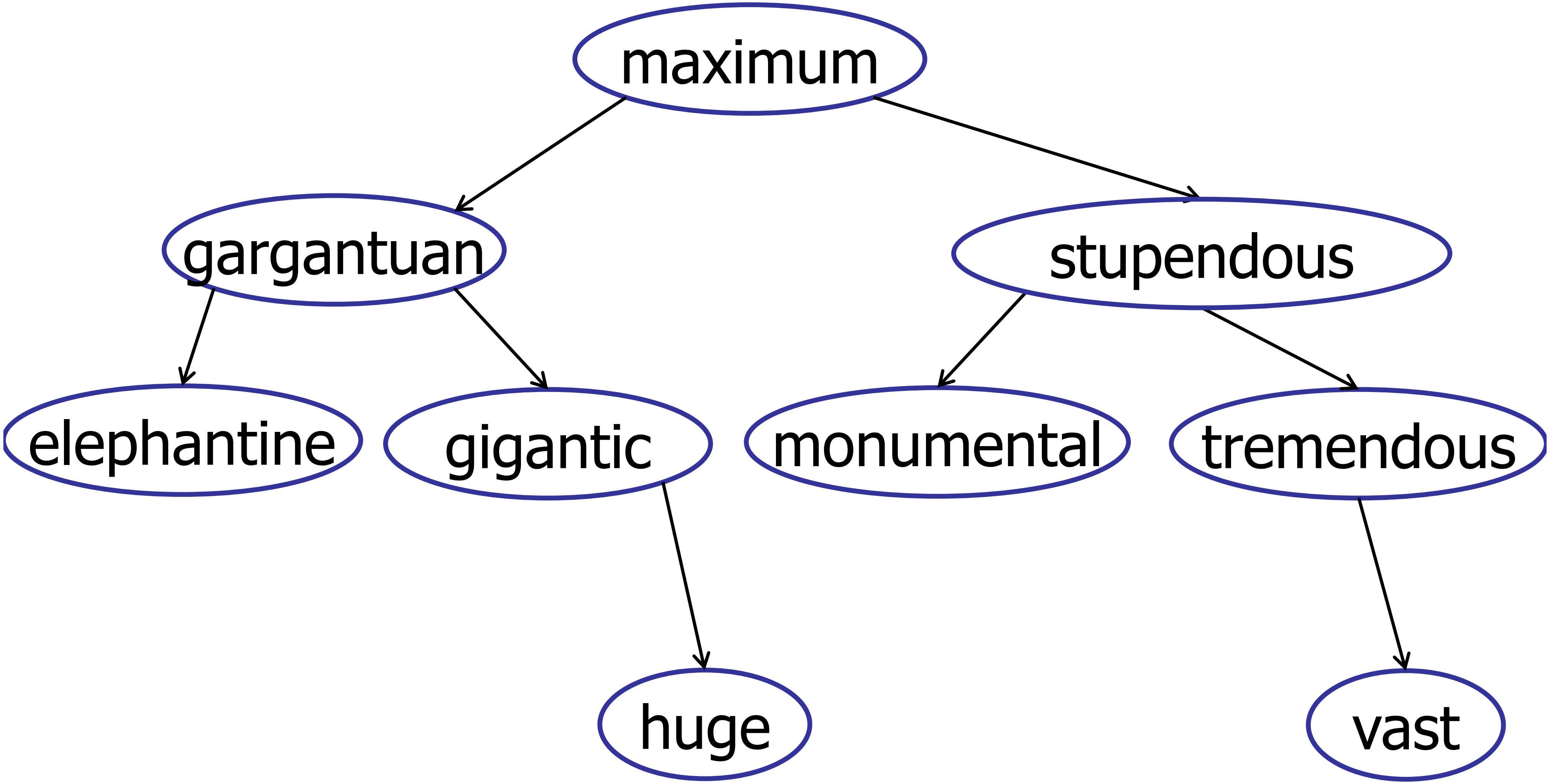
Tries

- How to handle text?

Data structure design

- How to build new structures on existing ideas?

What about text strings?



Implement a searchable dictionary!

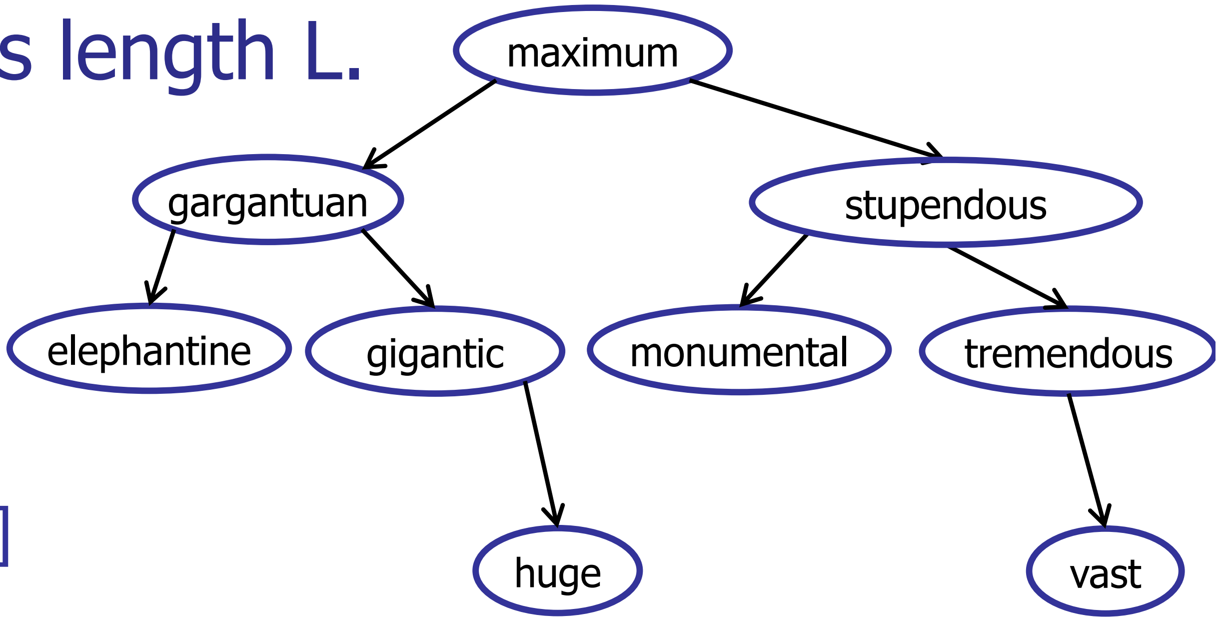
What about text strings?

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)

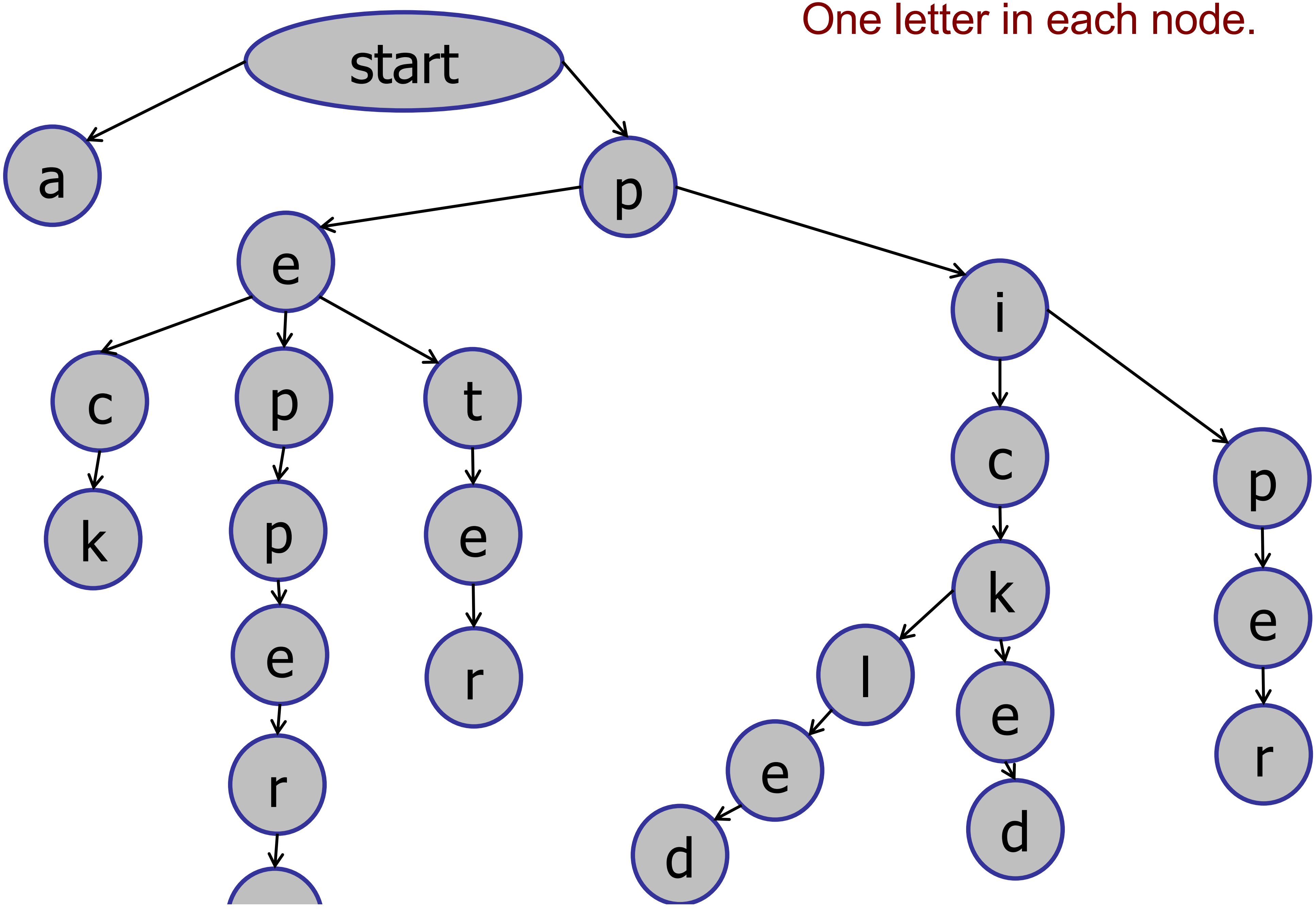


[In the worst case.]

[Optimizations are possible.]

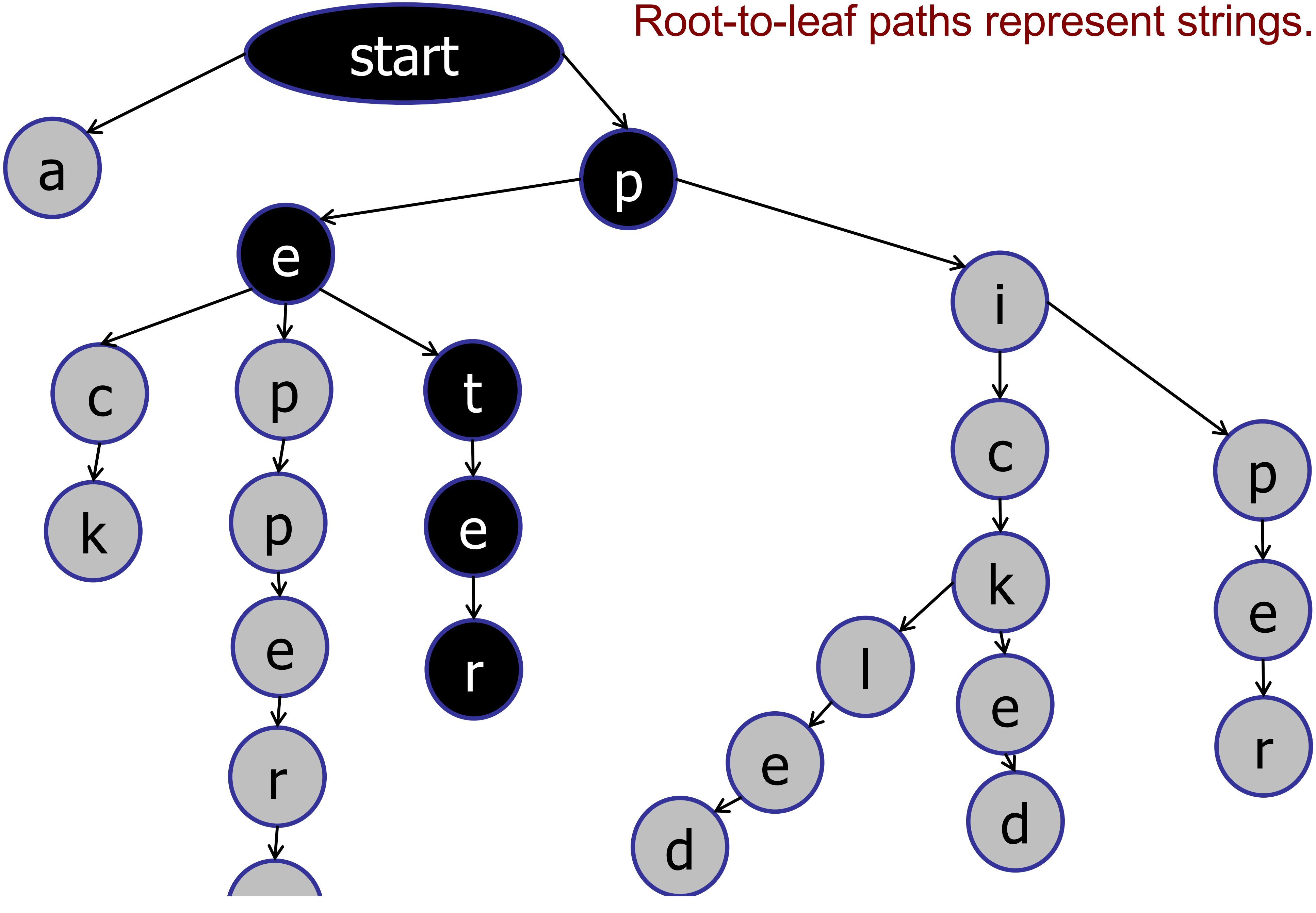
Trie [prounounced: try]

One letter in each node.



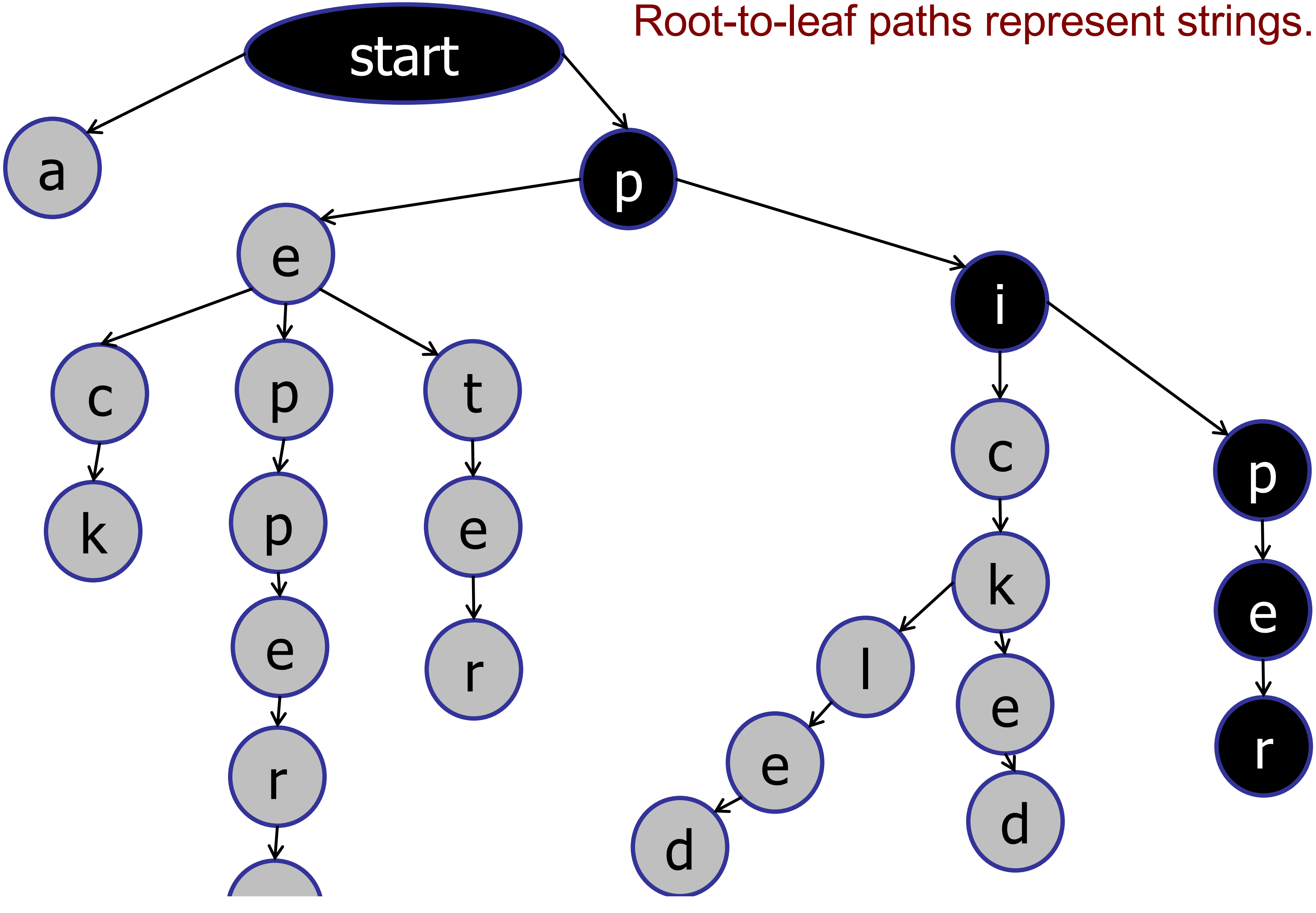
Trie [prounounced: try]

Root-to-leaf paths represent strings.



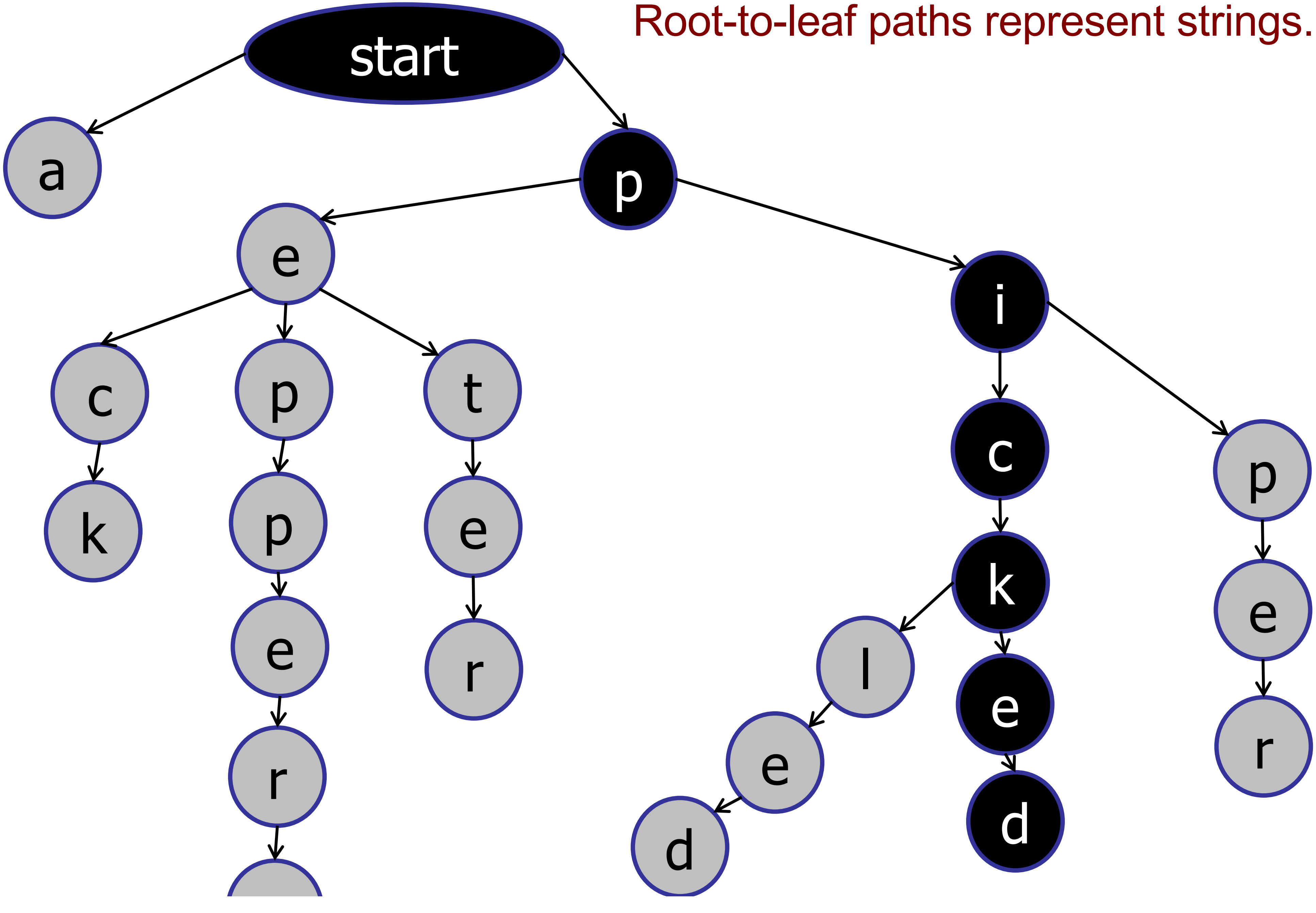
Trie [prounounced: try]

Root-to-leaf paths represent strings.



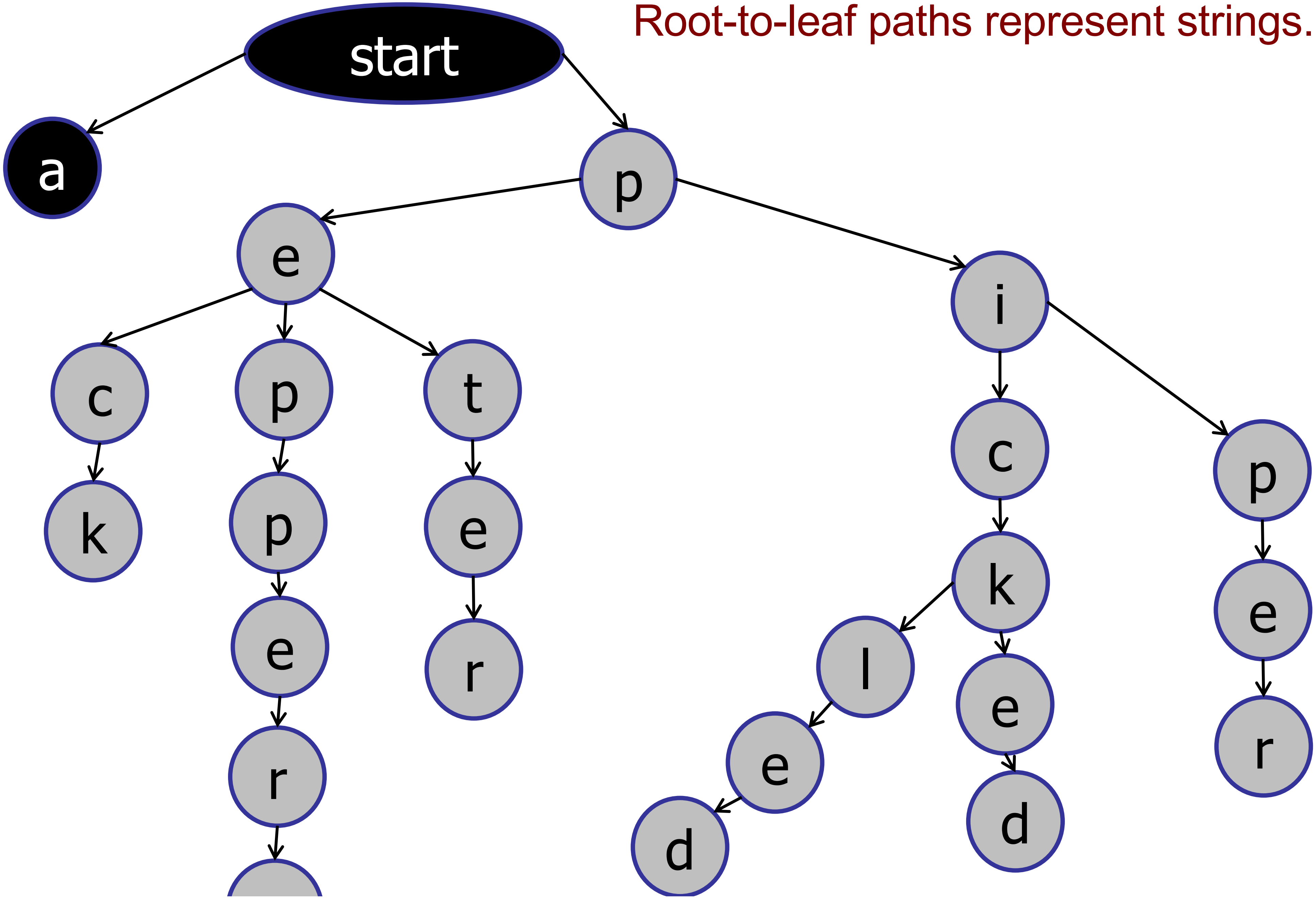
Trie [prounounced: try]

Root-to-leaf paths represent strings.



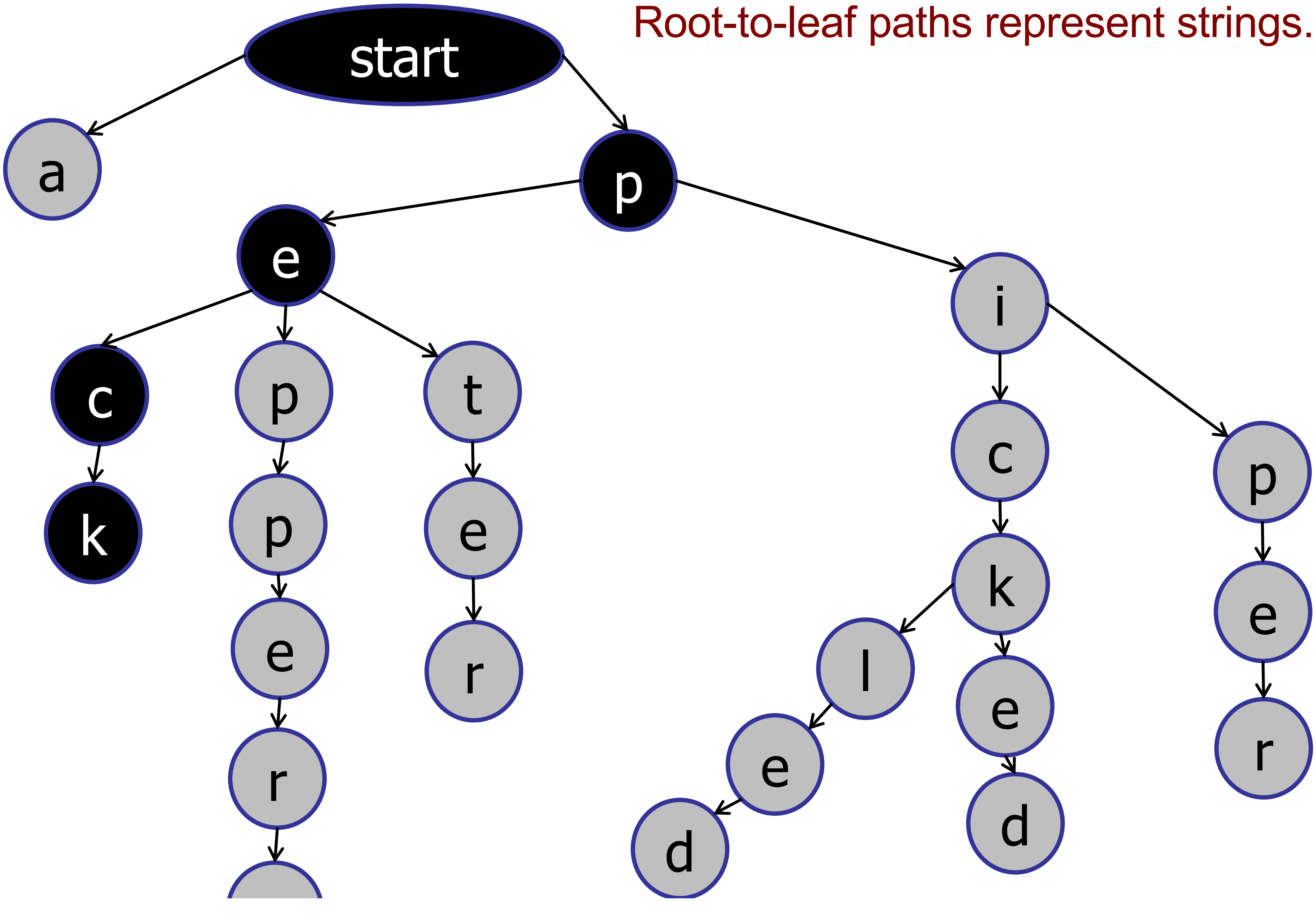
Trie [prounounced: try]

Root-to-leaf paths represent strings.



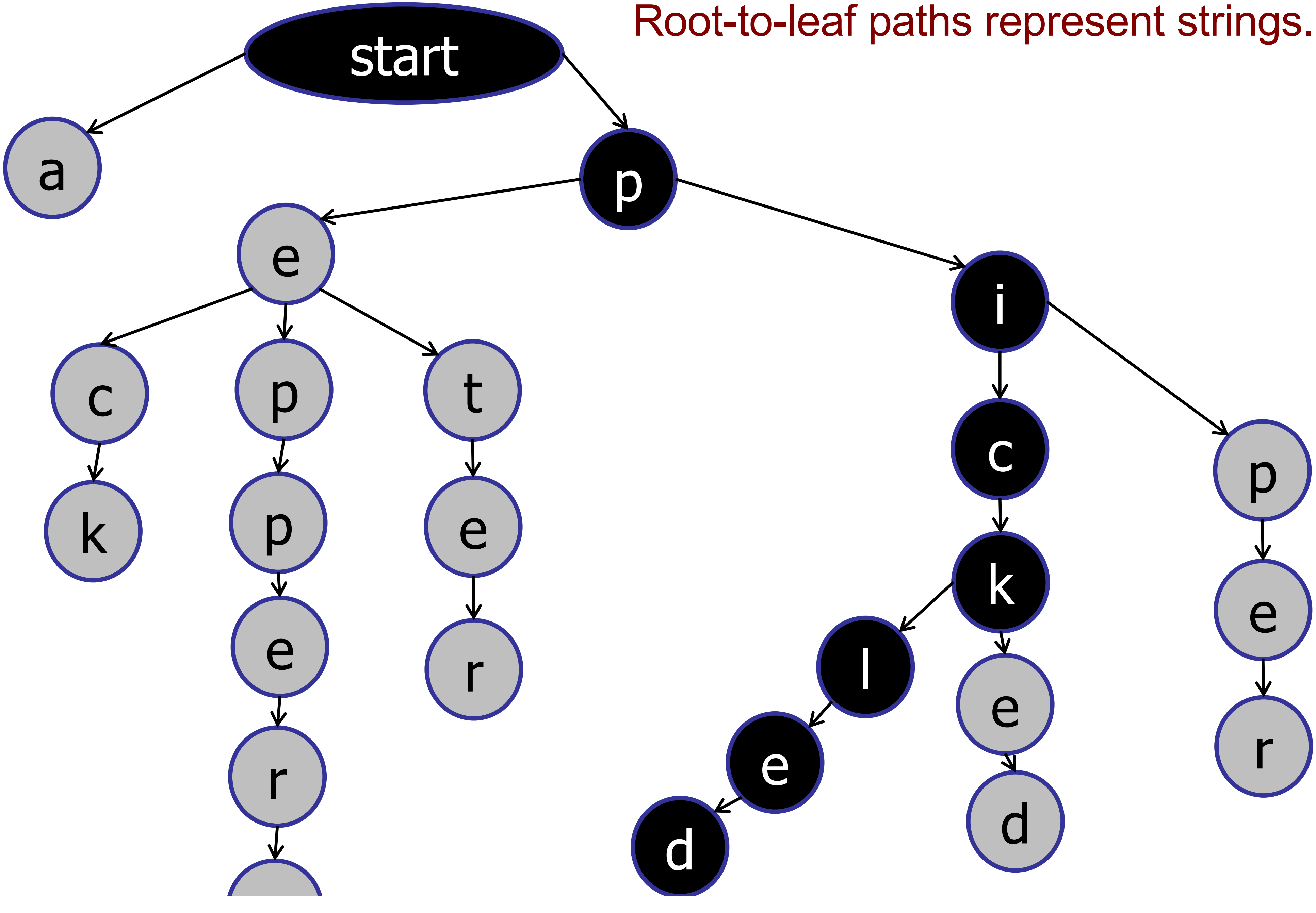
Trie [prounounced: try]

Root-to-leaf paths represent strings.



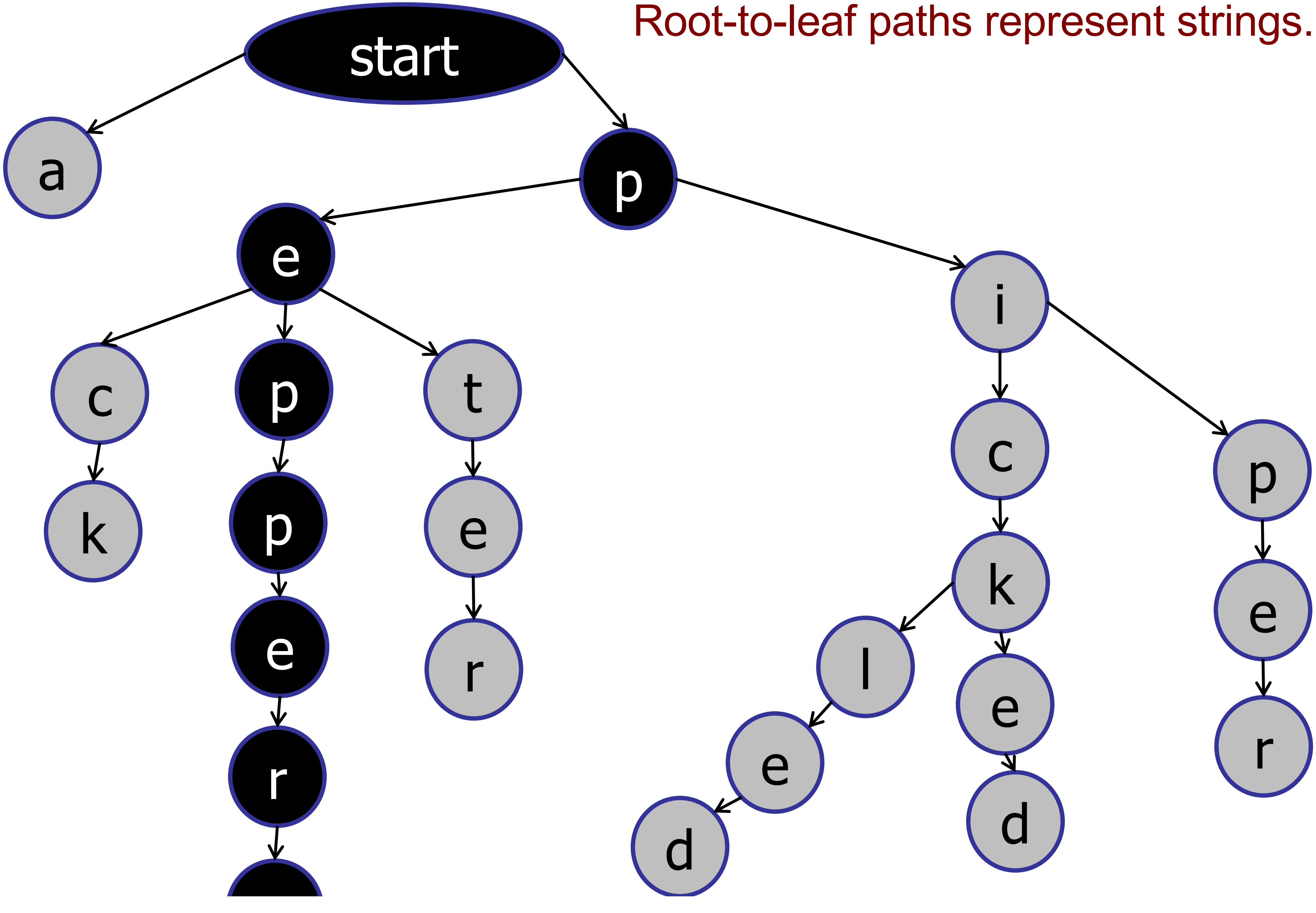
Trie [prounounced: try]

Root-to-leaf paths represent strings.

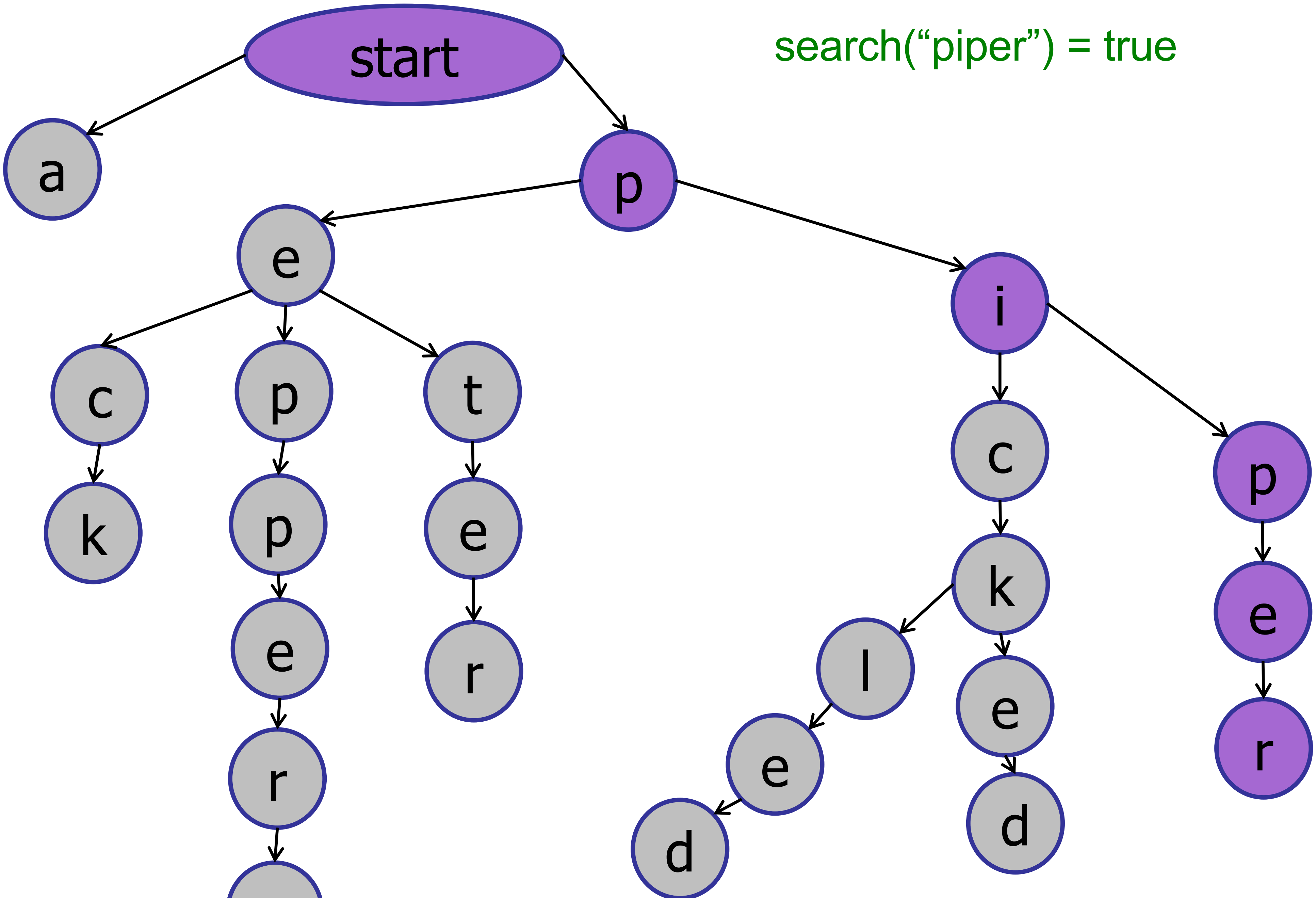


Trie [prounounced: try]

Root-to-leaf paths represent strings.

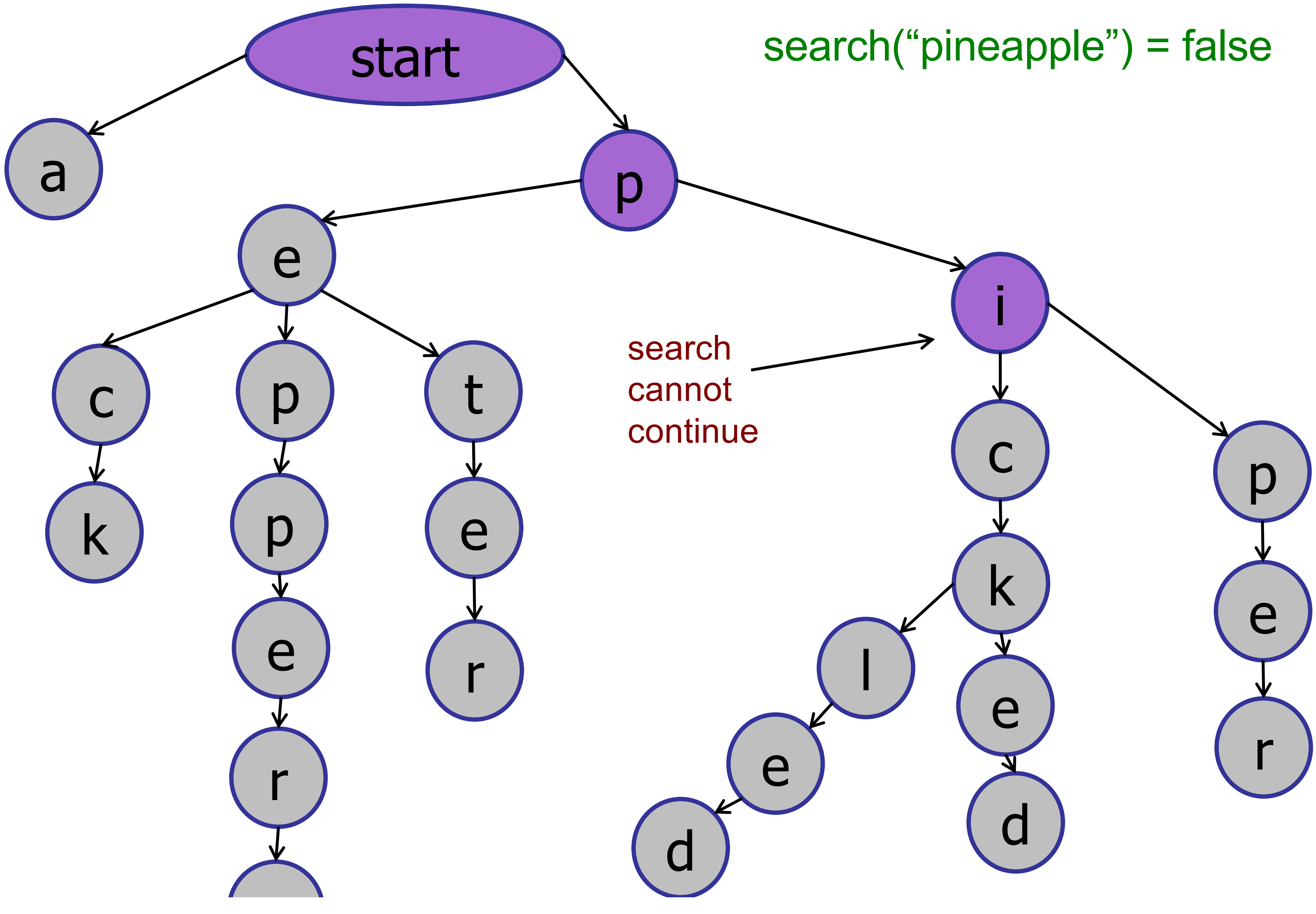


Searching a Trie



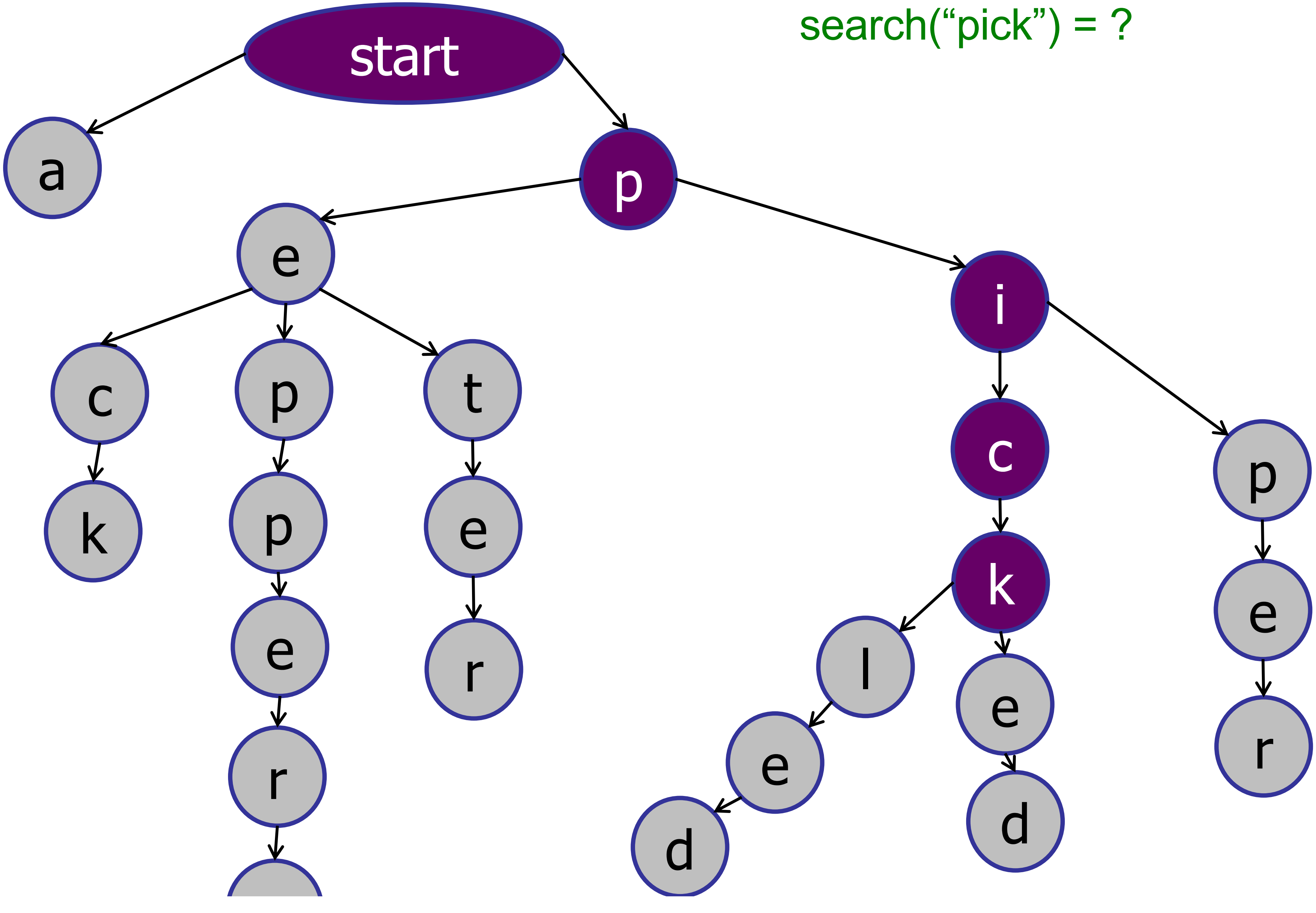
Searching a Trie

search(“pineapple”) = false

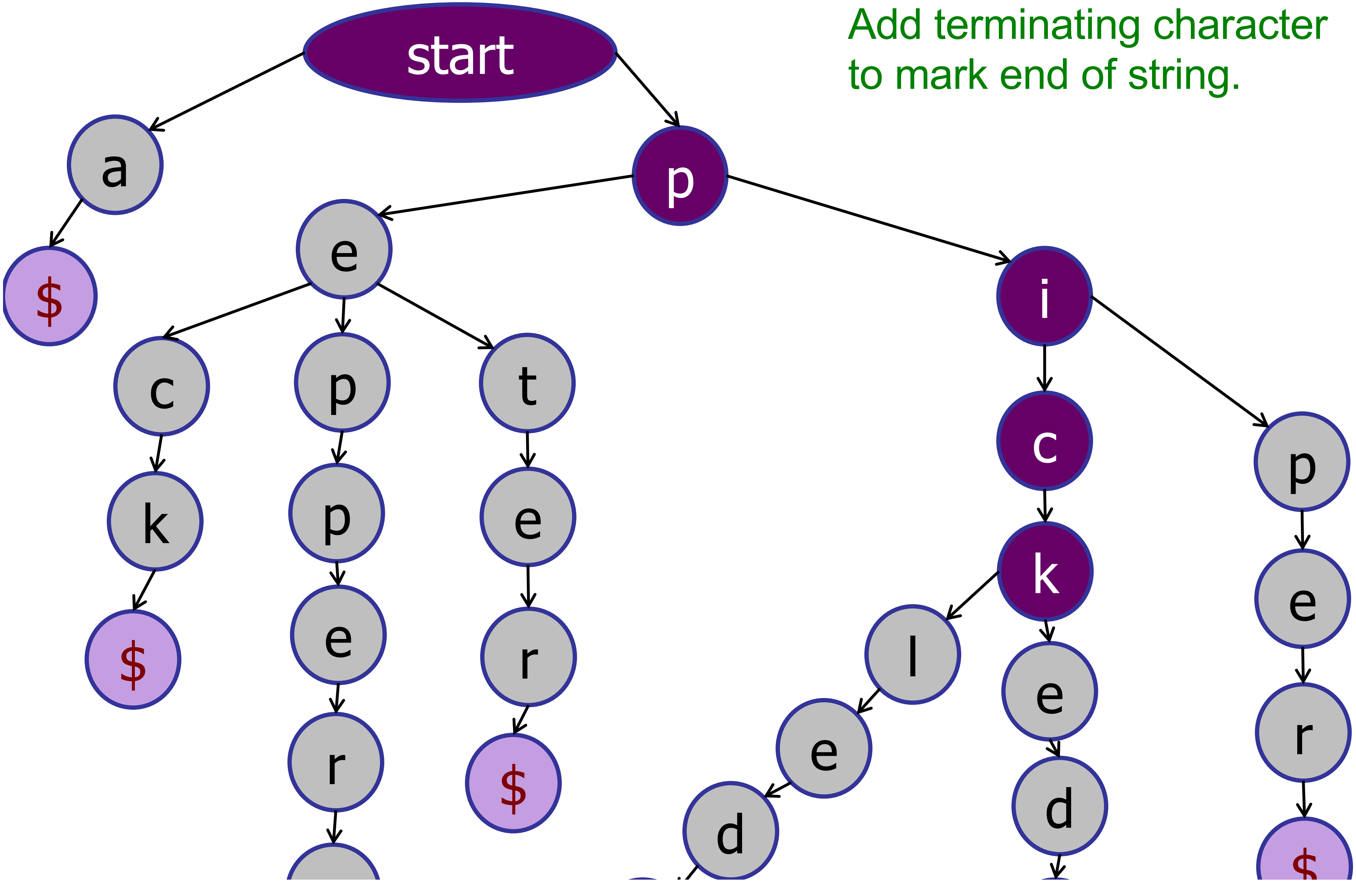


Trie Details

search("pick") = ?



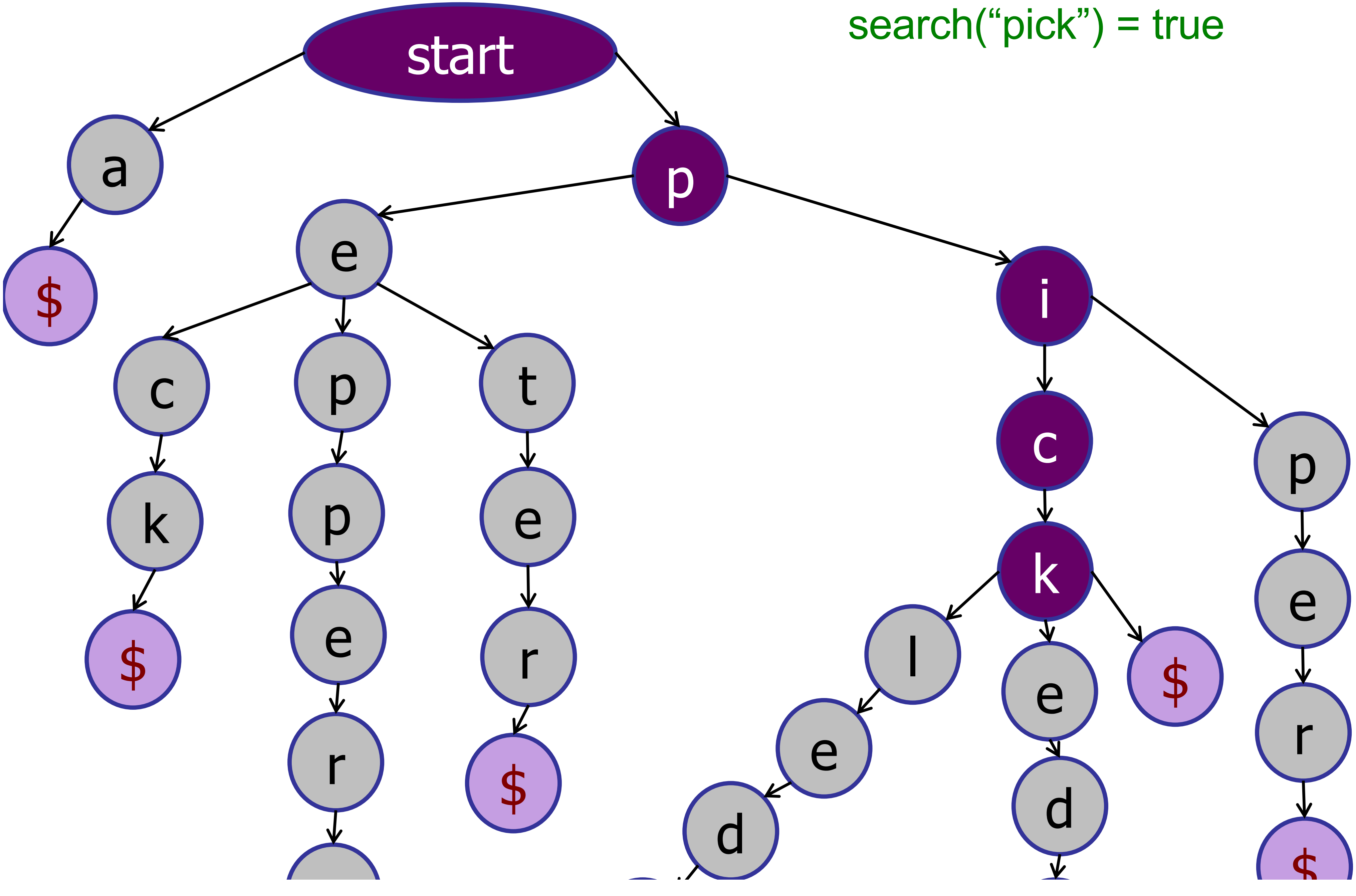
Trie Details



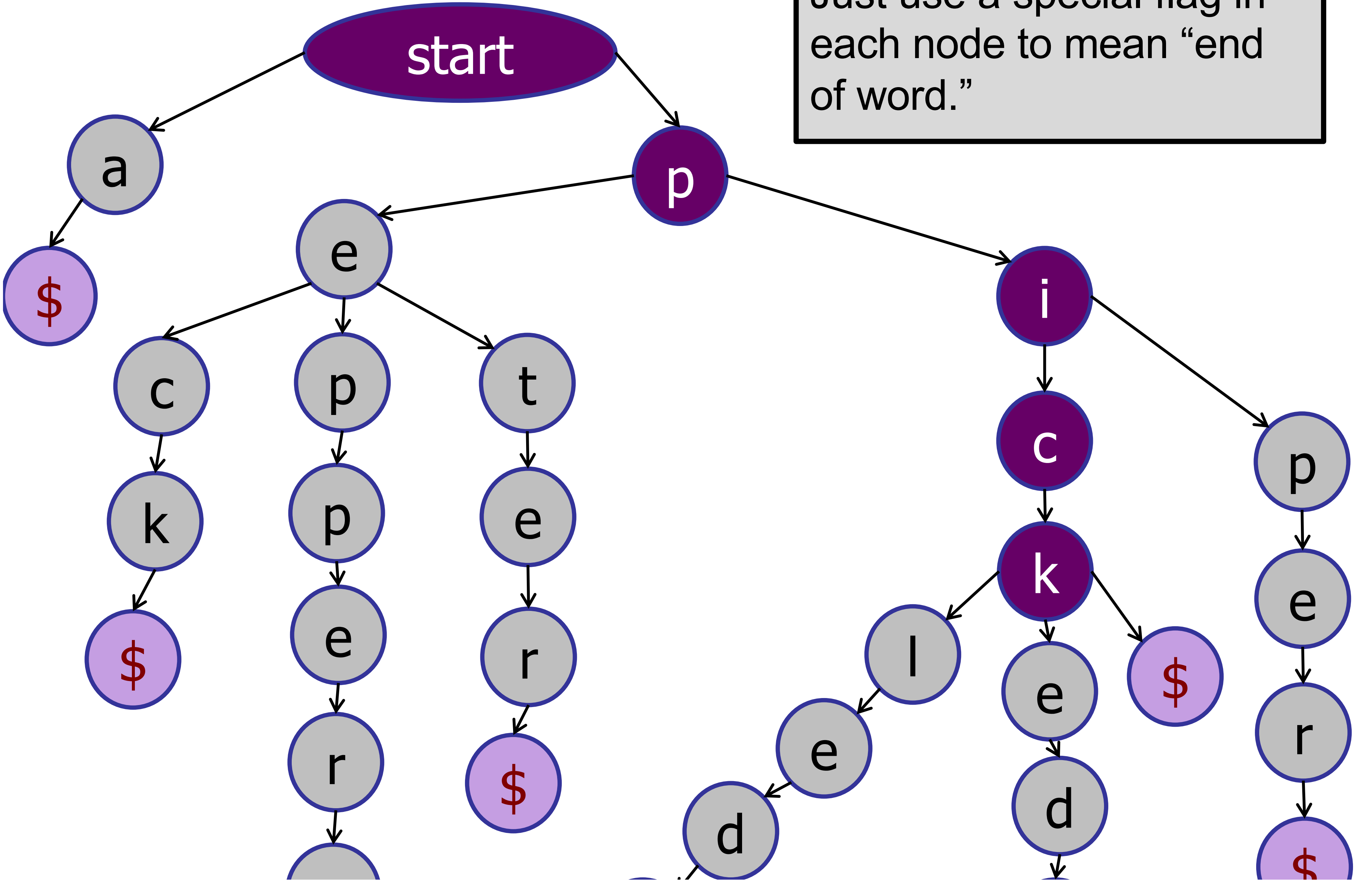
Add terminating character to mark end of string.

Trie Details

search(“pick”) = true



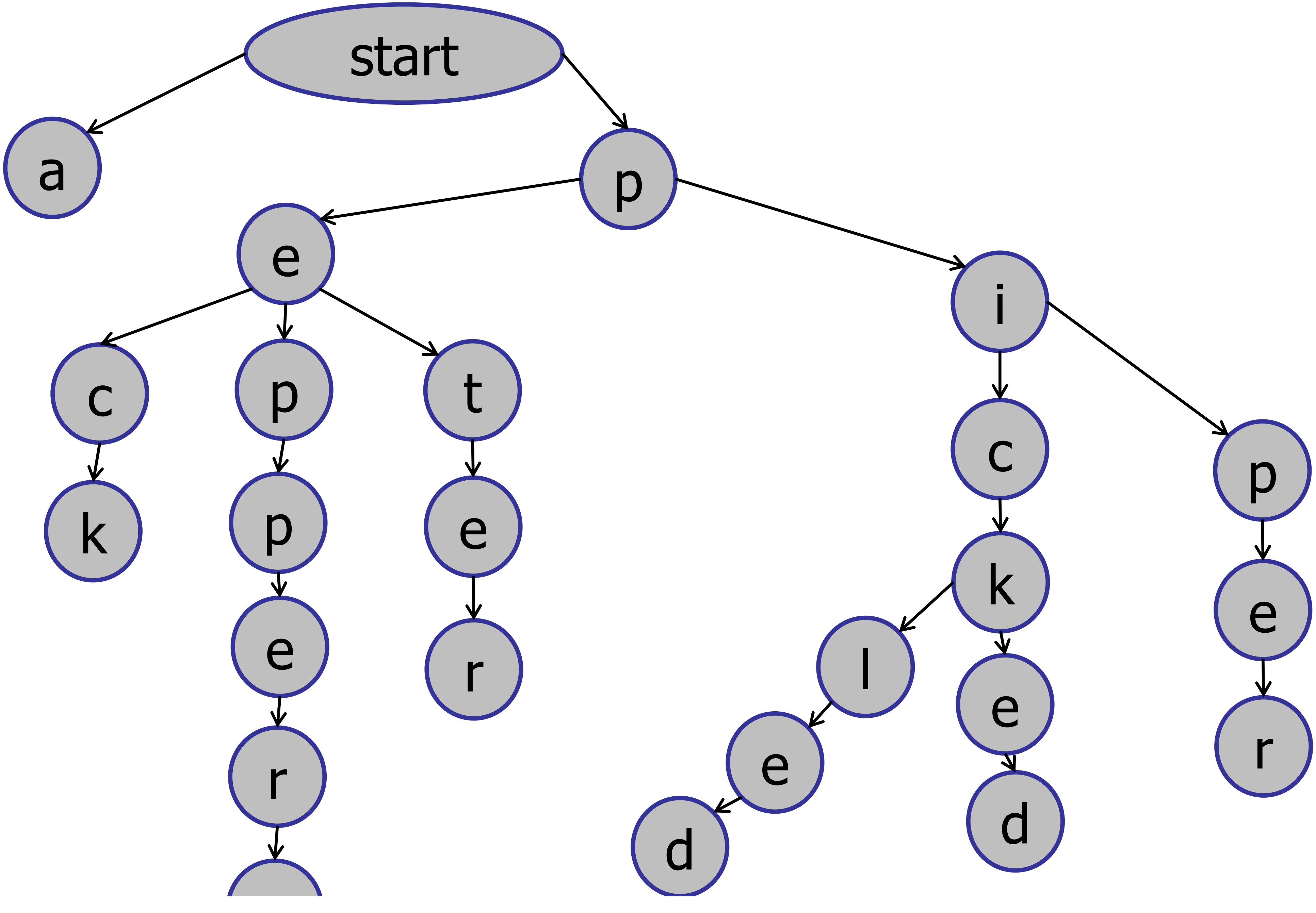
Trie Details



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

Trie

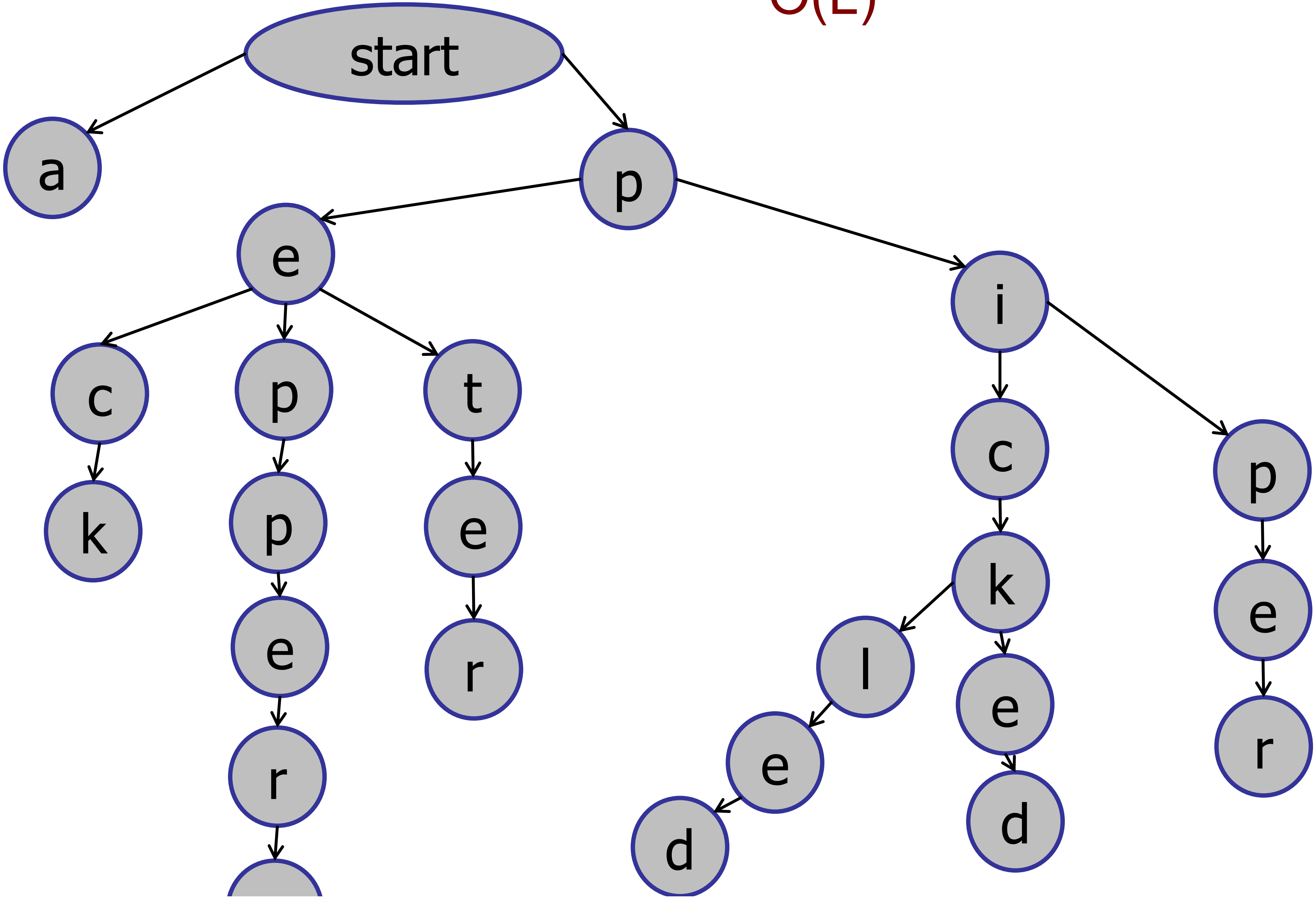
Cost to search for a string of length L?



Trie

Cost to search for a string of length L?

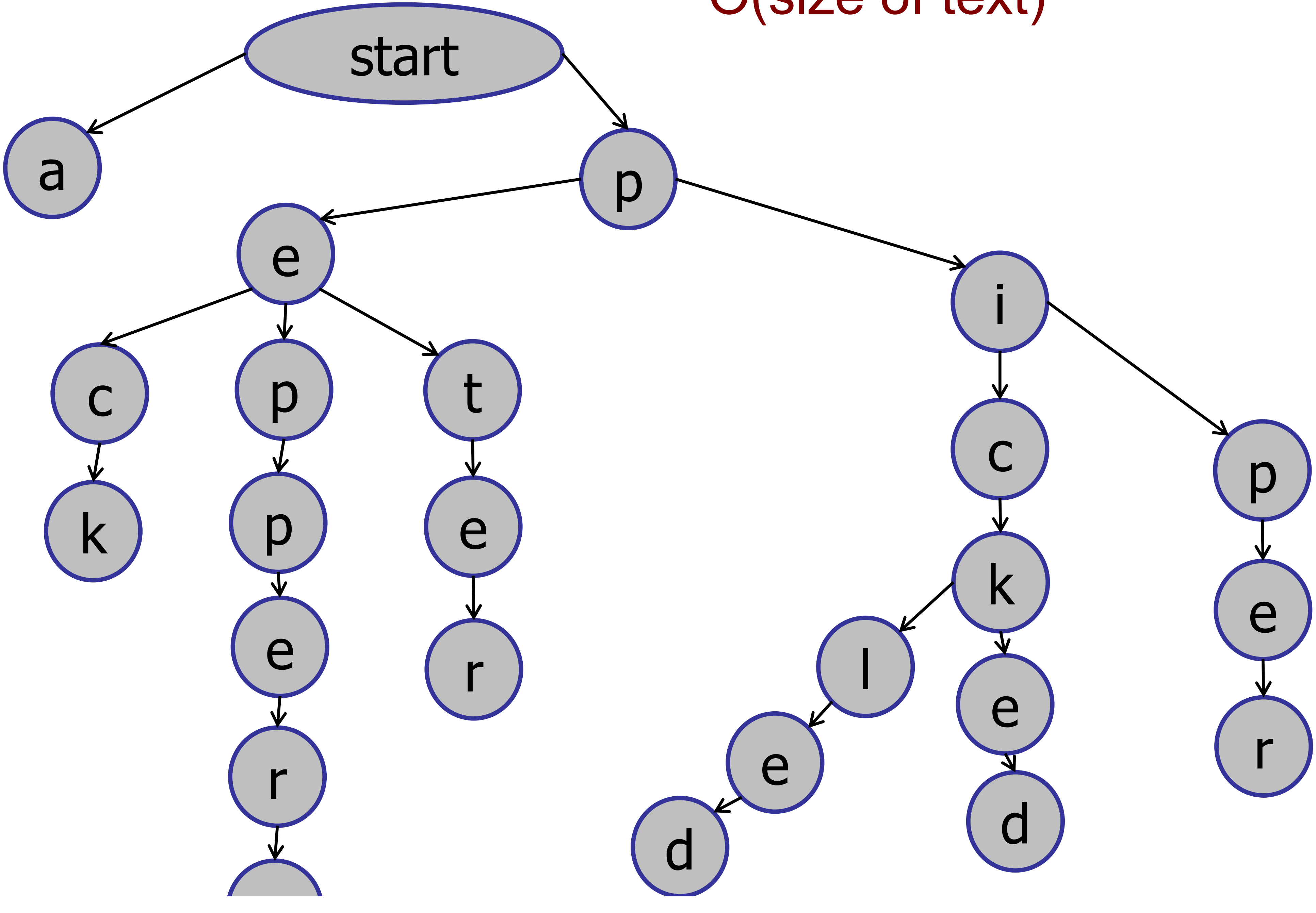
O(L)



Trie

Space for storing a try?

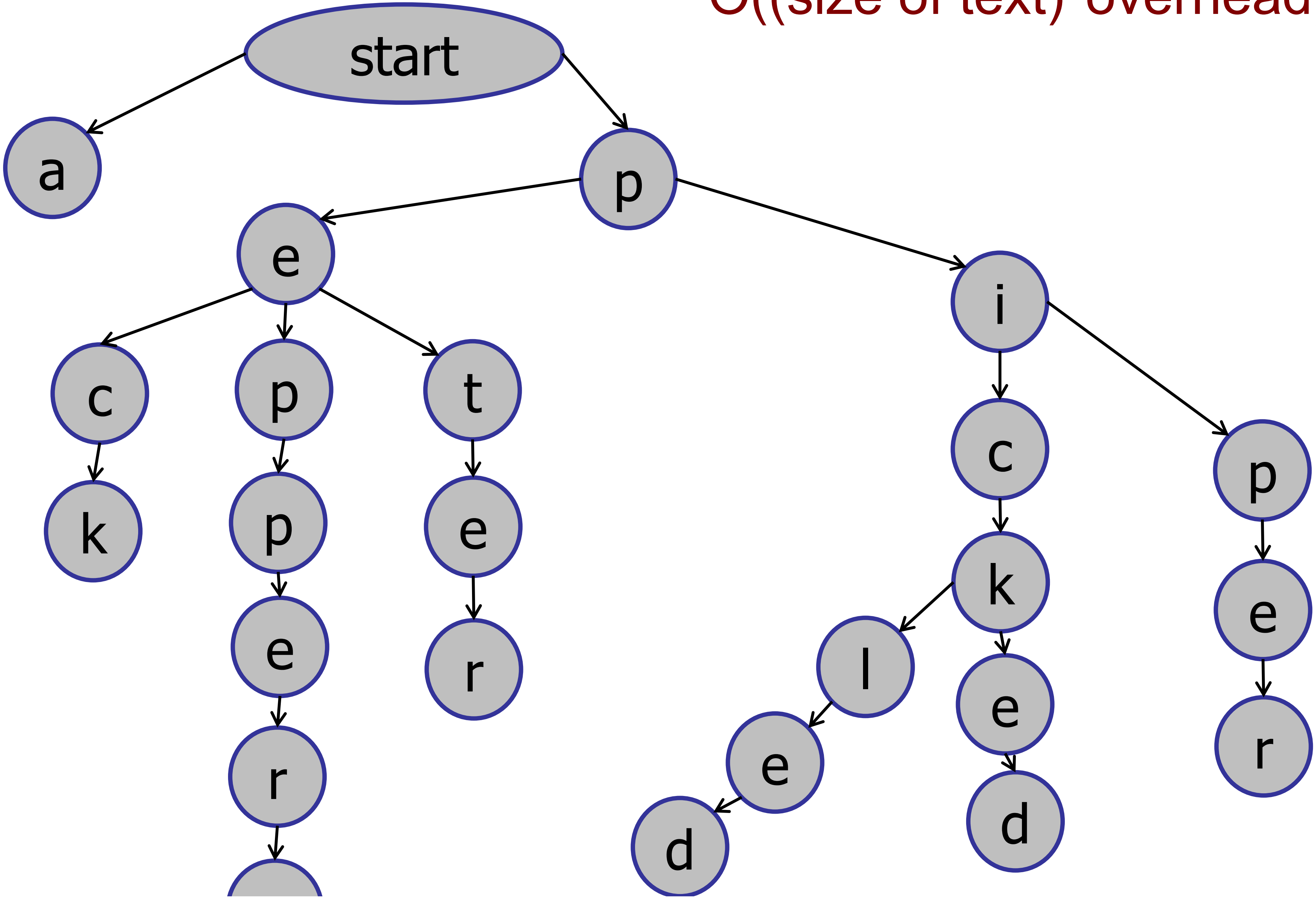
O(size of text)



Trie

Space for storing a try?

O((size of text)*overhead)



Trie Tradeoffs

Time:

- Trie tends to be faster: O(L) vs. O(Lh).
- 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.

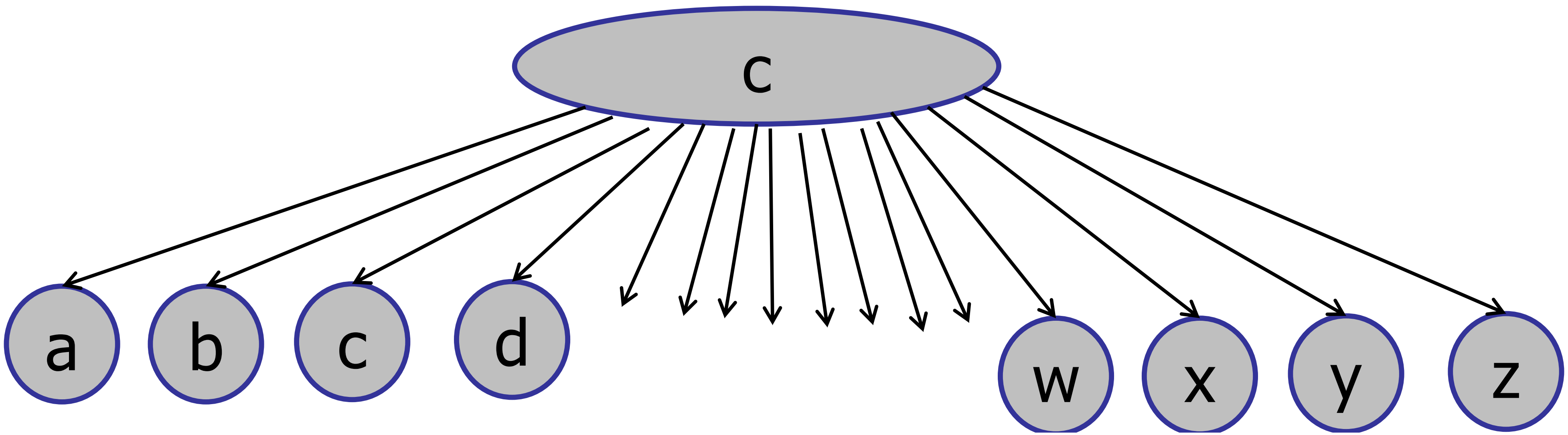
Trie Space

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.

Todays Plan

On the importance of being balanced

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

Tries

- How to handle text?

Data structure design

- How to build new structures on existing ideas?

Dynamic Data Structures

1. Maintain a set of items

2. Modify the set of items

3. Answer queries.

Dynamic Data Structures

1. Maintain a set of items

2. Modify the set of items

3. Answer queries.

B-trees are at the heart of *every* database!

Big picture idea:

Trees are a good way to store, summarize, and search dynamic data.

Dynamic Data Structures

- Operations that create a data structure
  - build (preprocess)
- Operations that modify the structure
  - insert
  - delete
- Query operations
  - search, select, etc.

"Why do we need to learn how an AVL tree works?"

Just use a Java TreeMap, right?

“Why do we need to learn how an AVL tree works?”

1. Learn how to think like a computer scientist.

“Why do we need to learn how an AVL tree works?”

1. Learn how to think like a computer scientist.
2. Learn to modify existing data structures to solve new problems.

Augmented Data Structures

Many problems require storing additional data in a standard data structure.

Augment more frequently than invent…

Next few lectures…

Three examples of augmenting balanced BSTs

1. Order Statistics

2. Interval Queries

3. Orthogonal Range Searching

Augmenting data structures

Basic methodology:

1. Choose underlying data structure

   (tree, hash table, linked list, stack, etc.)

Augmenting data structures

Basic methodology:

1. Choose underlying data structure

   (tree, hash table, linked list, stack, etc.)

2. Determine additional info needed.

Augmenting data structures

Basic methodology:

1. Choose underlying data structure

   (tree, hash table, linked list, stack, etc.)

2. Determine additional info needed.

3. Modify data structure to *maintain* additional info when the structure changes.

   (subject to insert/delete/etc.)

Augmenting data structures

Basic methodology:

1. Choose underlying data structure

   (tree, hash table, linked list, stack, etc.)

2. Determine additional info needed.

3. Modify data structure to *maintain* additional info when the structure changes.

   (subject to insert/delete/etc.)

4. Develop new operations.

Order Statistics

Input

A set of integers.

Output: select(k)

The k^{th} item in the set.

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

select(4)

select(2) returns:

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

1. 52
2. 9
3. 13
4. 43
5. 25

ARCHIPELAGO

is open

Order Statistics

Input

A set of integers.

Output: select(k)

The k^{th} item in the set.

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

select(4)

Order Statistics

Input

A set of integers.

Output: select(k)

The k^{th} item in the set.

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

select(4)

Order Statistics

Input

A set of integers.

Output: select(k) → Sort: O(n log n)

The k^{th} item in the set.

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

select(4)

Order Statistics

Input

A set of integers.

Output: select(k) → QuickSelect: O(n)

The k^{th} item in the set.

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

↑

select(4)

Order Statistics

Solution 1:

Sort: O(n log n)

Solution 2:

QuickSelect: O(n)

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

select(4)

Order Statistics

Solution 1:

Preprocess: sort --- O(n log n)

Select: O(1)

Solution 2:

Preprocess: nothing --- O(1)

QuickSelect: O(n)

Order Statistics

Solution 1:

Preprocess: sort --- O(n log n)

Select: O(1)

Solution 2:

Preprocess: nothing --- O(1)

QuickSelect: O(n)

Trade-off: how many items to select?

Dynamic Order Statistics

Implement a data structure that supports:

- insert(int key)
- delete(int key)

and also:

- select(int k)

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

select(4)

Dynamic Order Statistics

Solution 1:

Basic structure: sorted array A.

insert(int item): add item to sorted array A.

select(int k): return A[k]

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

Dynamic Order Statistics

Solution 2:

Basic structure: unsorted array A.

insert(int item): add item to end of array A.

select(int k): run QuickSelect(k)

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

When is it more efficient to maintain a sorted array (Solution 1)?

A. Always

B. When there are more inserts than selects.

C. When there are more selects than inserts.

D. Never

E. I’m confused.

ARCHIPELAGO

is open

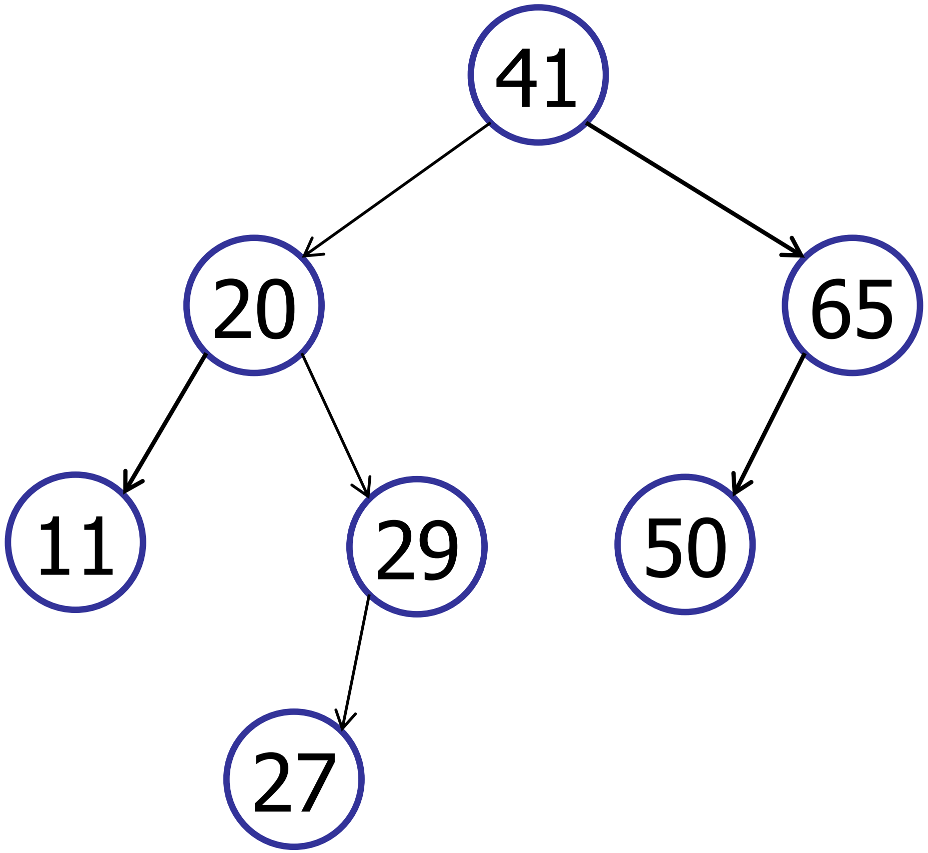
Dynamic Order Statistics

	Insert	Select
Solution 1: Sorted Array	O(n)	O(1)
Solution 2: Unsorted Array	O(1)	O(n)

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

Dynamic Order Statistics

Today: use a (balanced) tree



11	**20**	**27**	**29**	**41**	**50**	**65**