BINARY SEARCH TREES

cs2420 | Introduction to Algorithms and Data Structures | Spring 2015

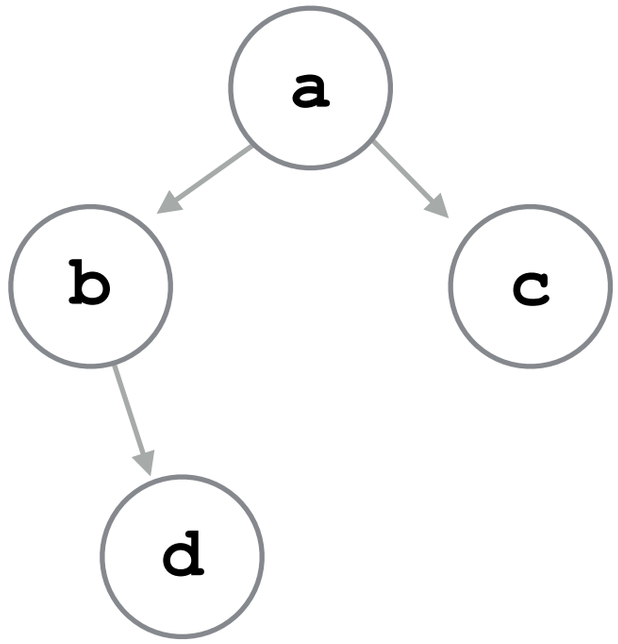
administrivia…

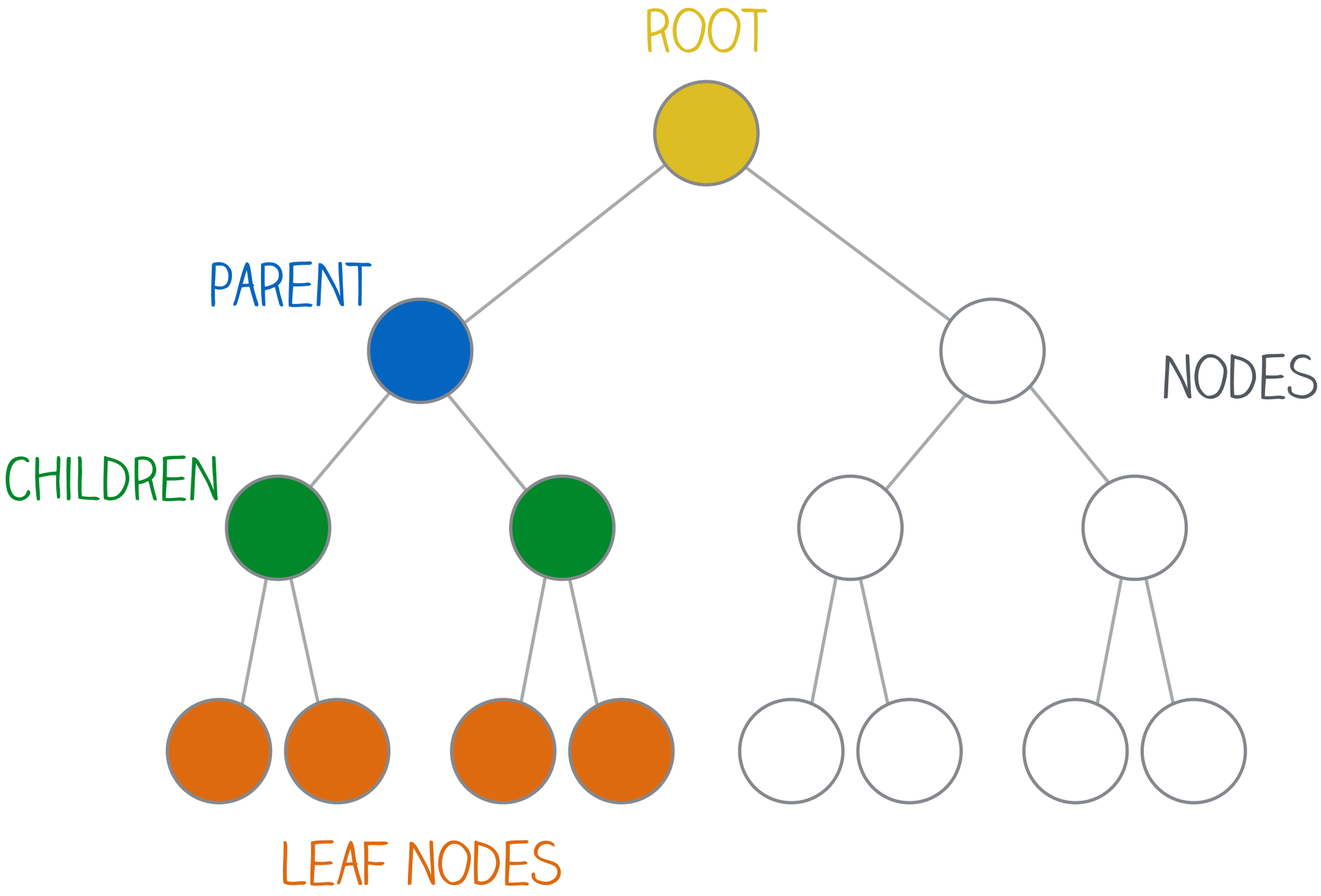
-assignment 7 due tonight at midnight

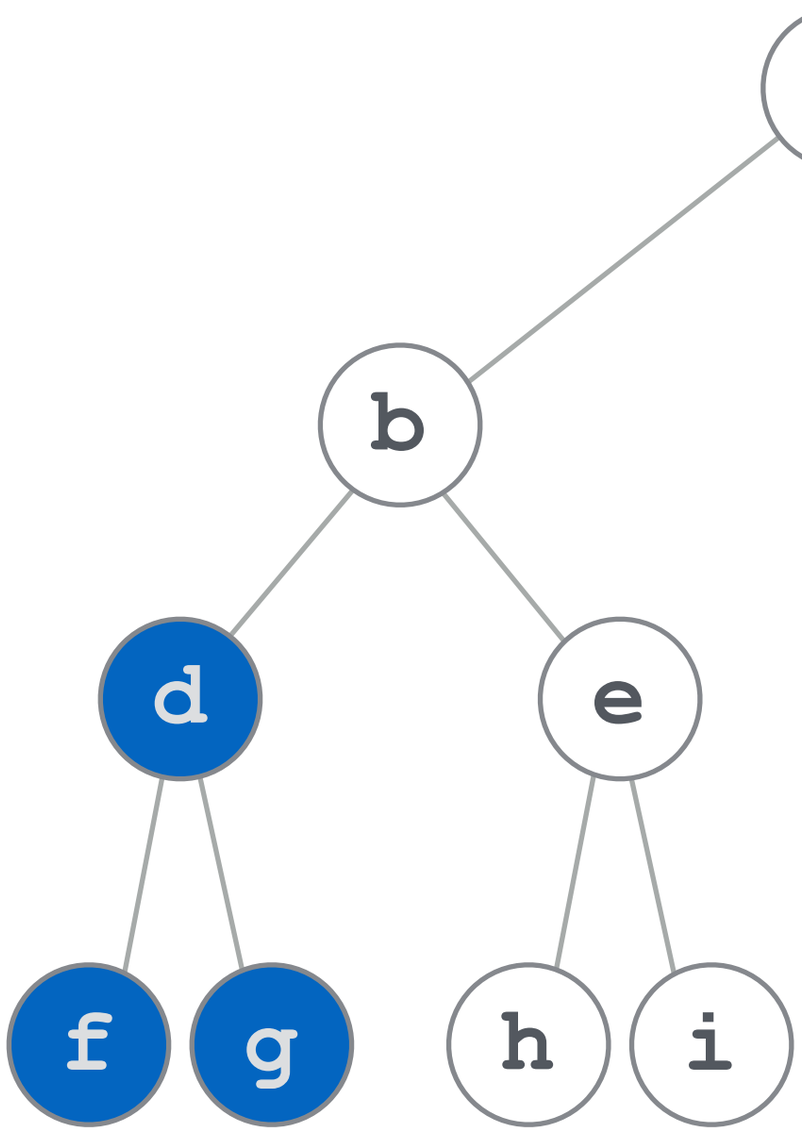
-asking for regrades through assignment 5 and midterm must be complete by **Friday**

last time…

- **trees** are a linked data structure with a hierarchical formation
  - any node is a subtree of some larger tree
    - *except the very top node!*
- there is a strict parent-to-child relationship among nodes
  - links *only* go from parent to child
- there is exactly one path from the root to any other node




ROOT
PARENT
NODES
CHILDREN
LEAF NODES



SUBTREE ROOTED AT NODE **c**
(LEAF NODES ARE TREES TOO!)

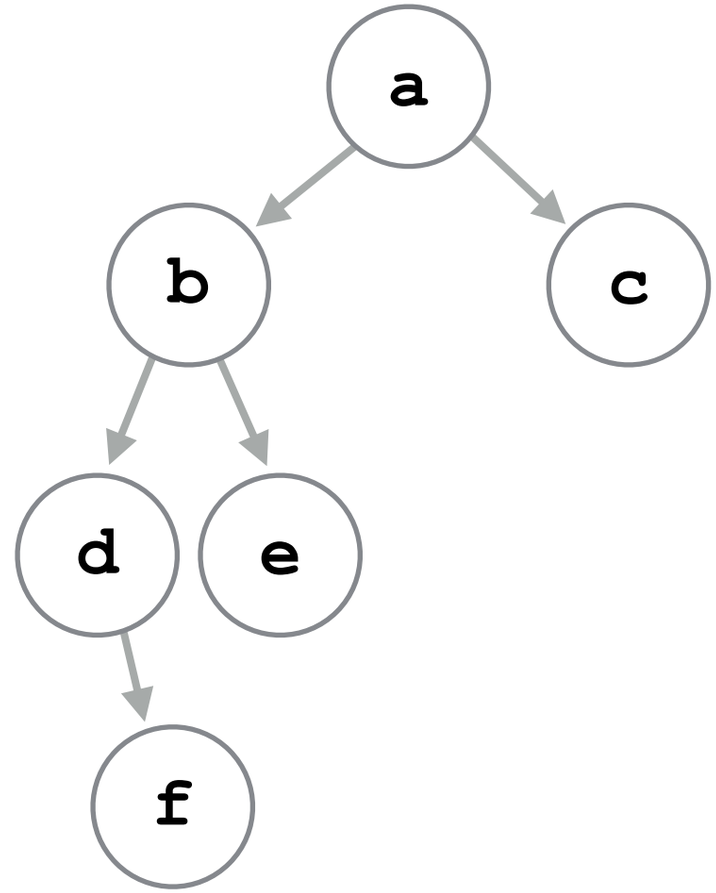
SUBTREE ROOTED AT NODE **d**

```
public static void DFT(BinaryNode N)
{
  if(N == null)
    return;

  System.out.println(N.data);
  DFT(N.left);
  DFT(N.right);
}
```

WHAT DOES THIS PRINT OUT?

a b d f e c



- **pre-order:**

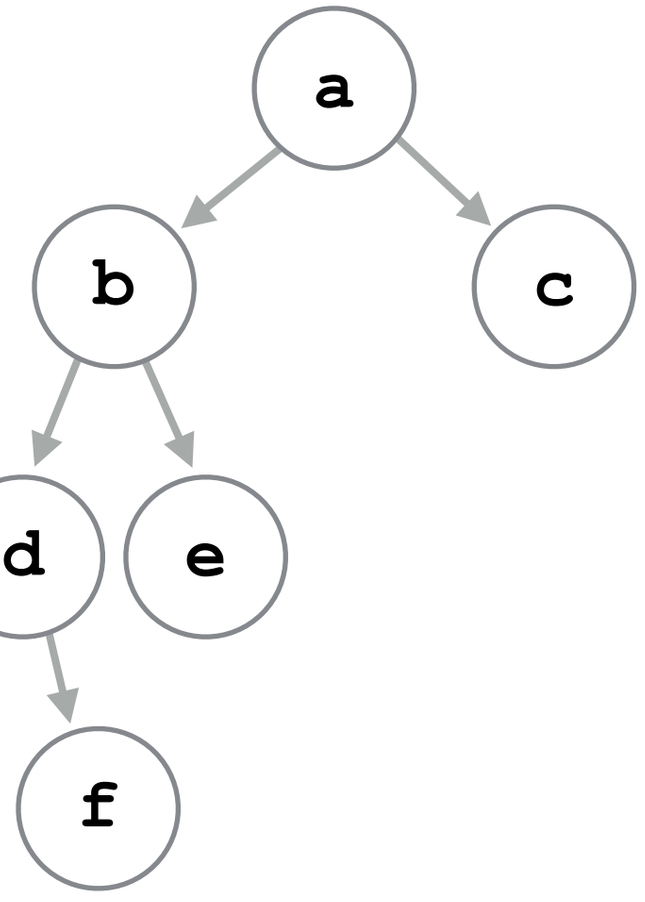
```
use N    // eg. print N
DFT(N.left);
DFT(N.right);
```

- **in-order:**

```
DFT(N.left);
use N    // eg. print N
DFT(N.right);
```

- **post-order:**

```
DFT(N.left);
DFT(N.right);
use N    // eg. print N
```



NOTE: NODES ARE STILL TRAVERSED IN THE SAME ORDER, BUT "USED" (PRINTED) IN A DIFFERENT ORDER

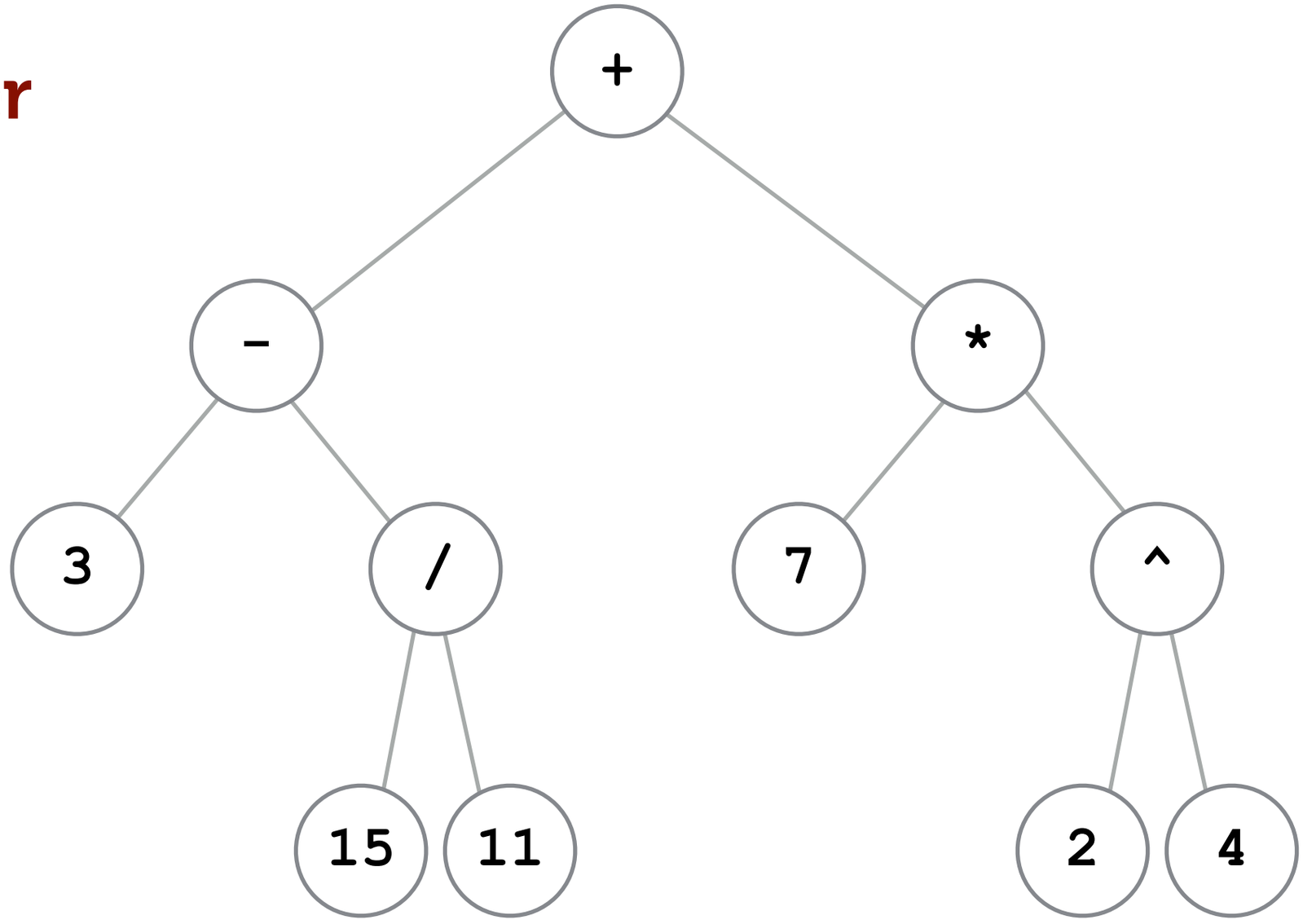
HOW TO COMPUTE THE HEIGHT OF A BINARY TREE?

```
int height(Node n)
{
  if(n == null)
    return -1;
  else
    return max(height(n.left),
               height(n.right)) + 1;
}
```

COMPLEXITY?

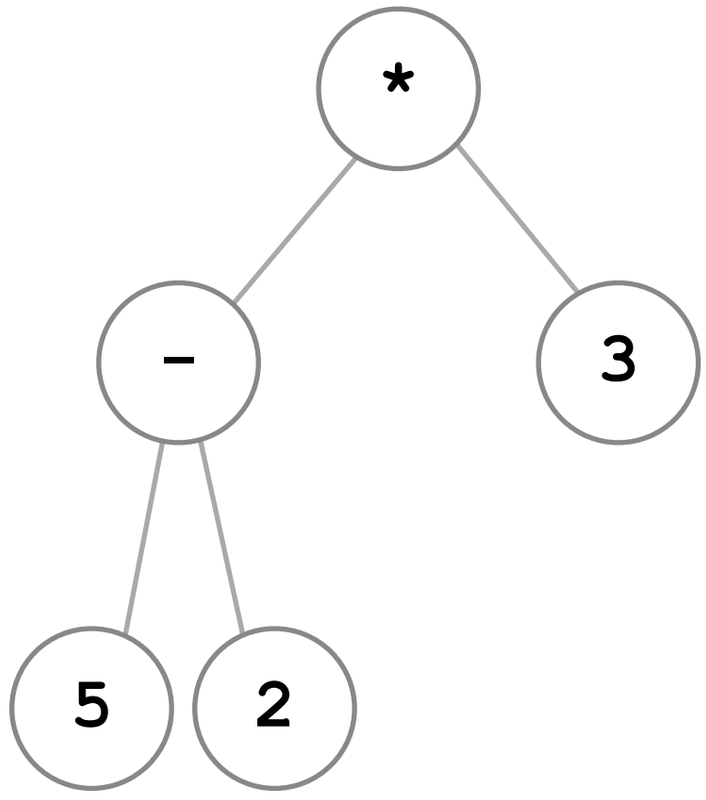
EVALUATING AN EXPRESSION TREE USES WHICH OF THE FOLLOWING TRAVERSAL ORDERS?

A) **pre-order**
B) **in-order**
C) **post-order**



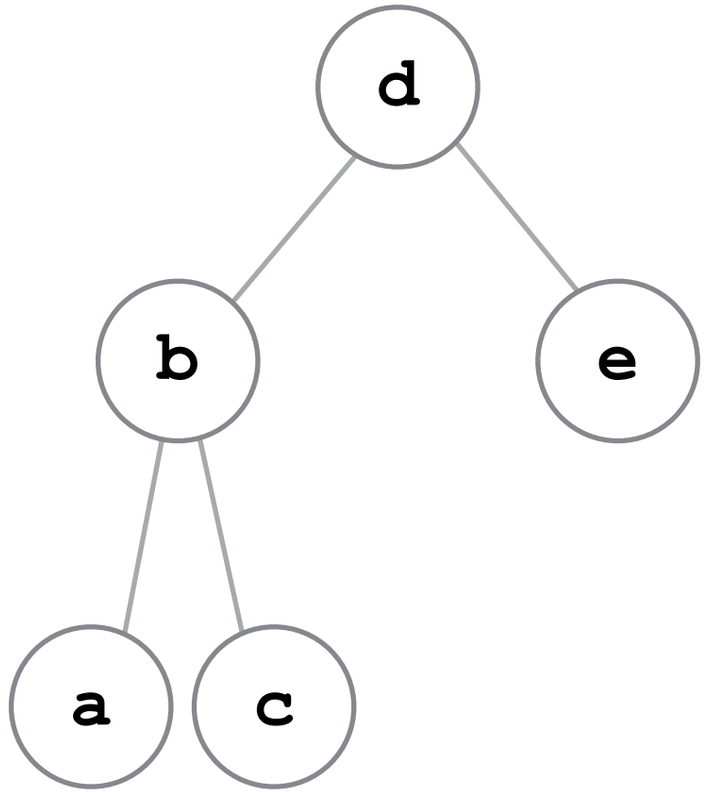
WHAT IS THE VALUE OF THIS EXPRESSION TREE?

A) **9**
B) **-1**
C) **-5**



WHAT ORDER ARE THE NODES PRINTED USING PRE-ORDER TRAVERSAL?

A) **d b a c e**
B) **a b c d e**
C) **a c b e d**



vcg.informatik.uni-rostock.de

How to cite this site?

Check out other surveys!

treevis.net - A Visual Bibliography of Tree Visualization 2.0 by Hans-Jörg Schulz

v.16-FEB-2015

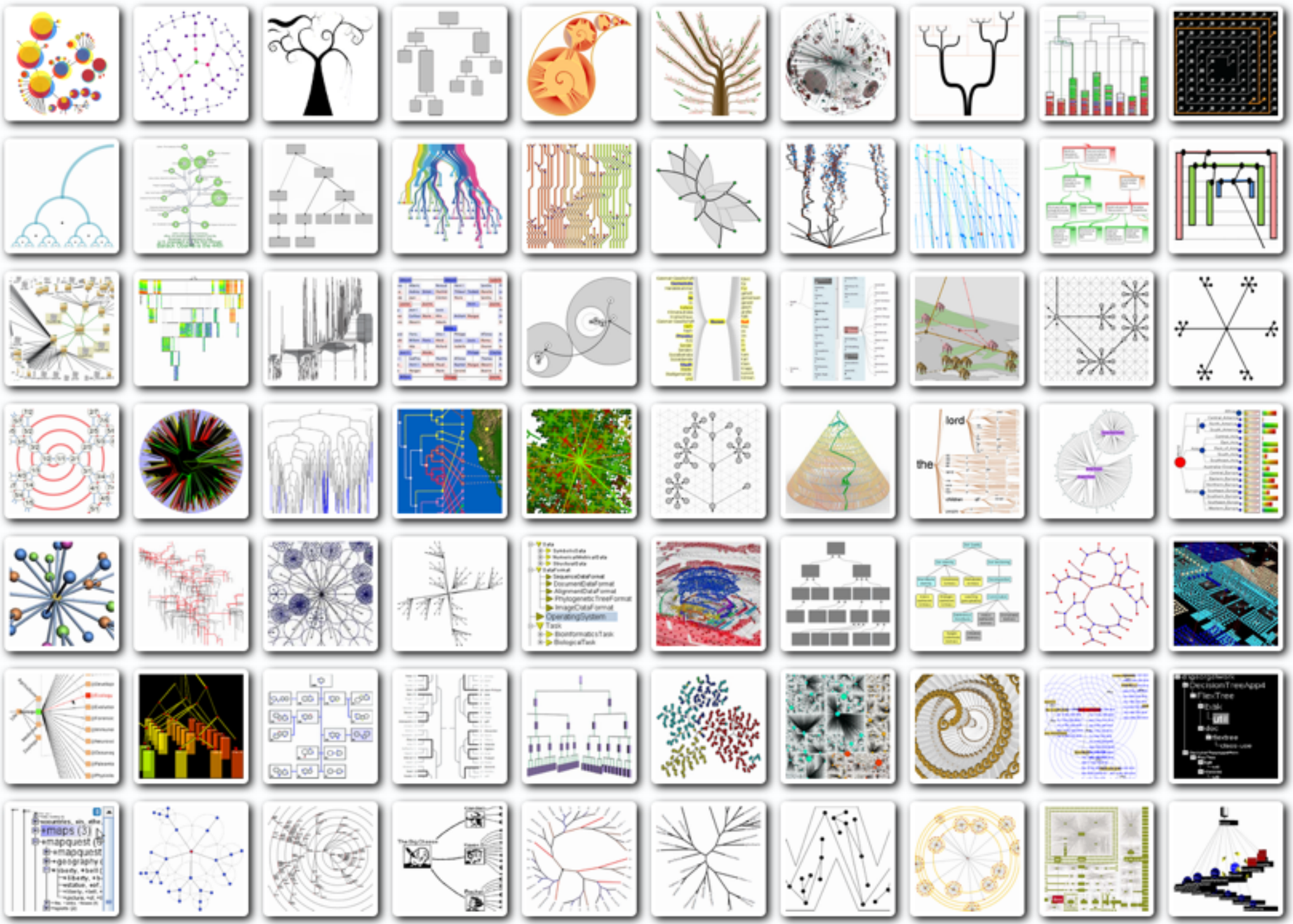
Dimensionality: All

Representation: All

Alignment: All

Fulltext Search

Techniques Shown: 128



today…

-binary search trees

-insertion

-performance

-deletion

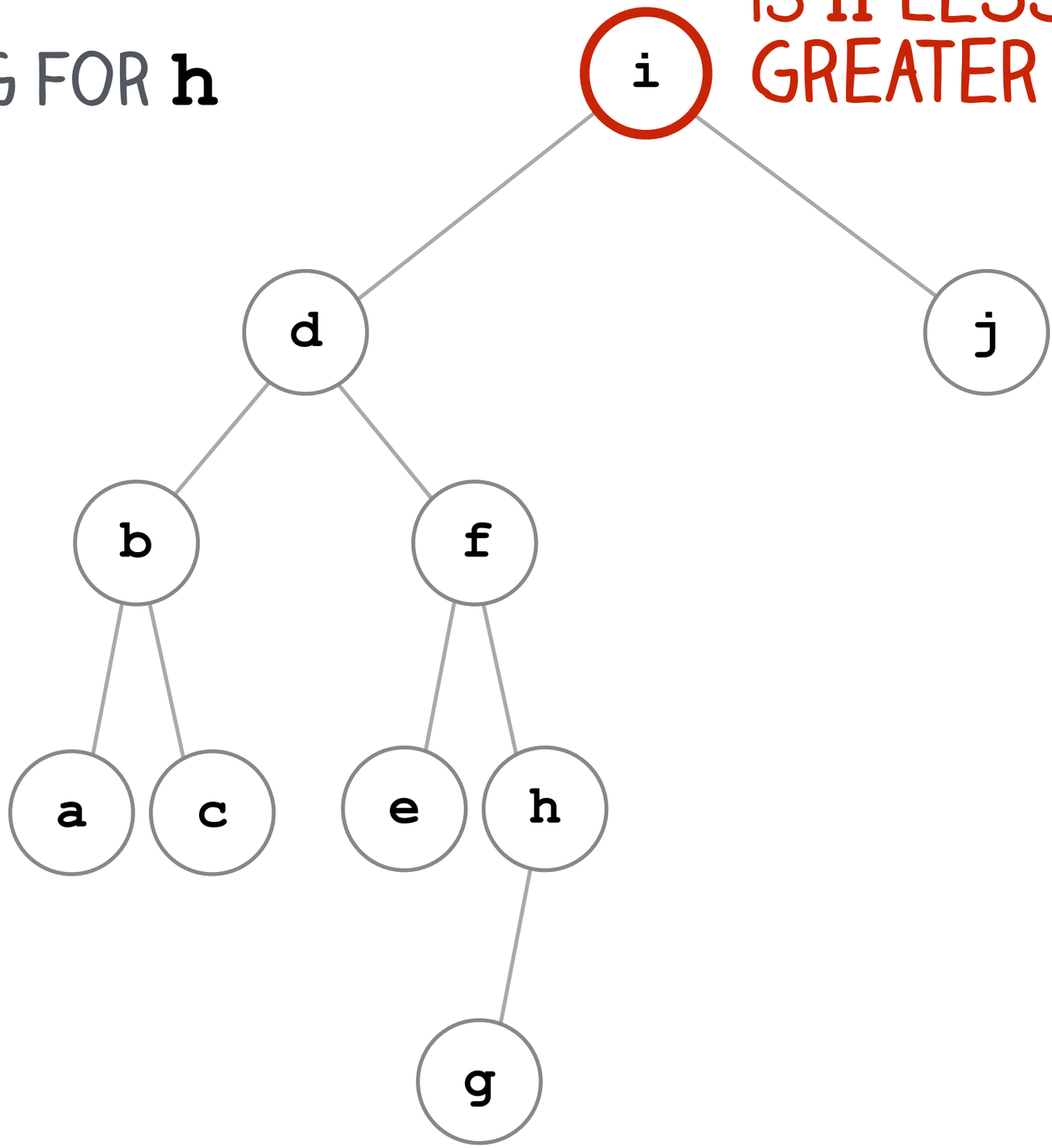
binary search trees (BSTs)

- a **binary search tree** is a binary tree with a restriction on the ordering of nodes
  - all items in the **left** subtree of a node are *less than* the item in the node
  - all items in the **right** subtree of a node are *greater than or equal to* the item in the node

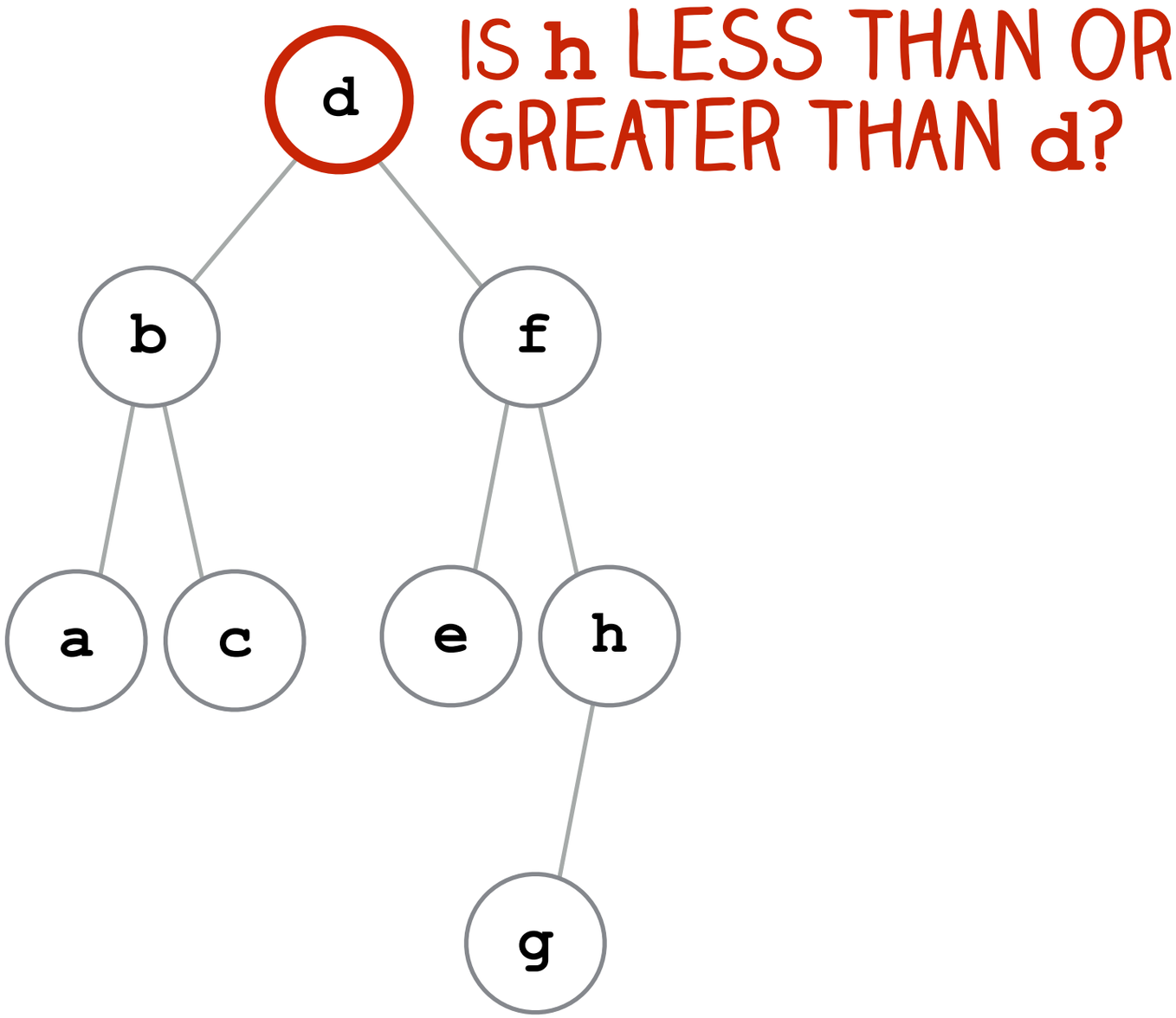
- BSTs allow for fast searching of nodes

LOOKING FOR **h**

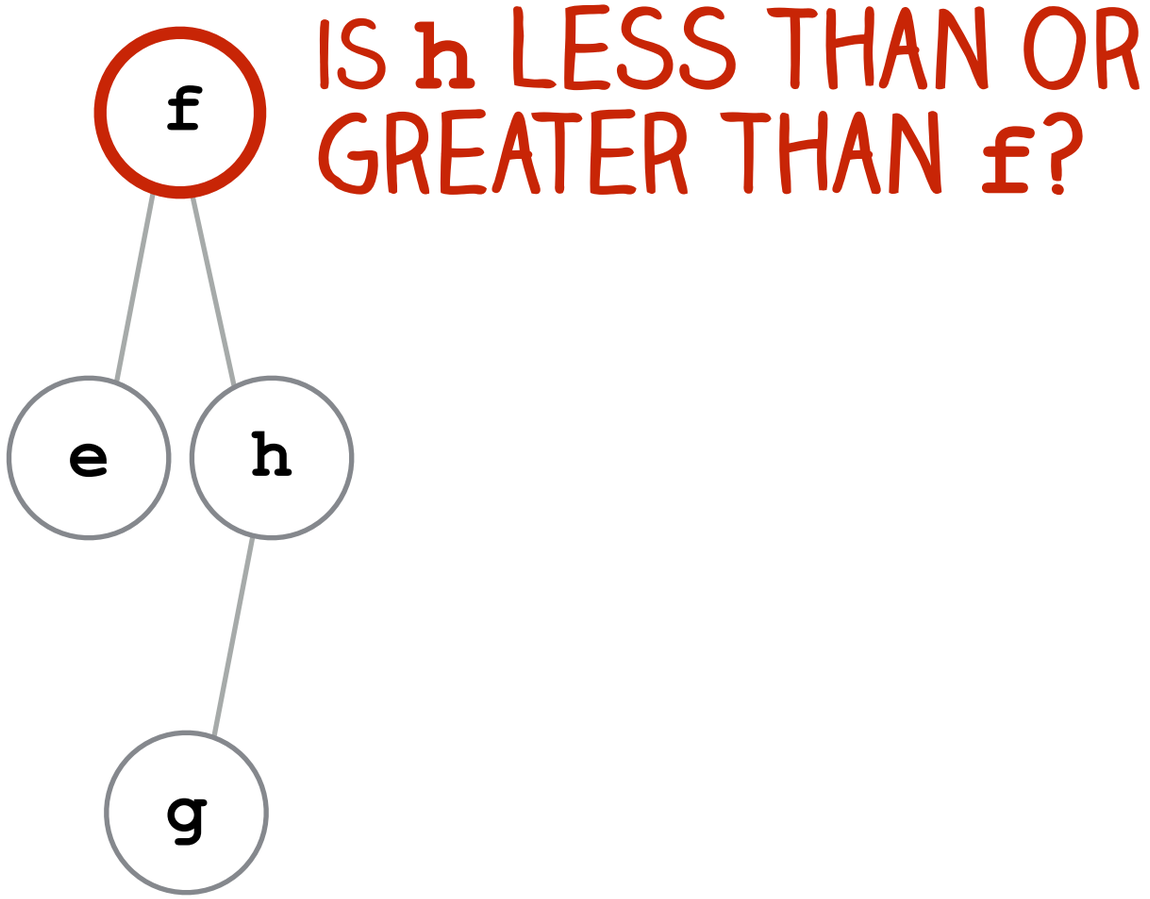
IS **h** LESS THAN OR GREATER THAN **i**?



LOOKING FOR **h**



LOOKING FOR **h**



LOOKING FOR **h**

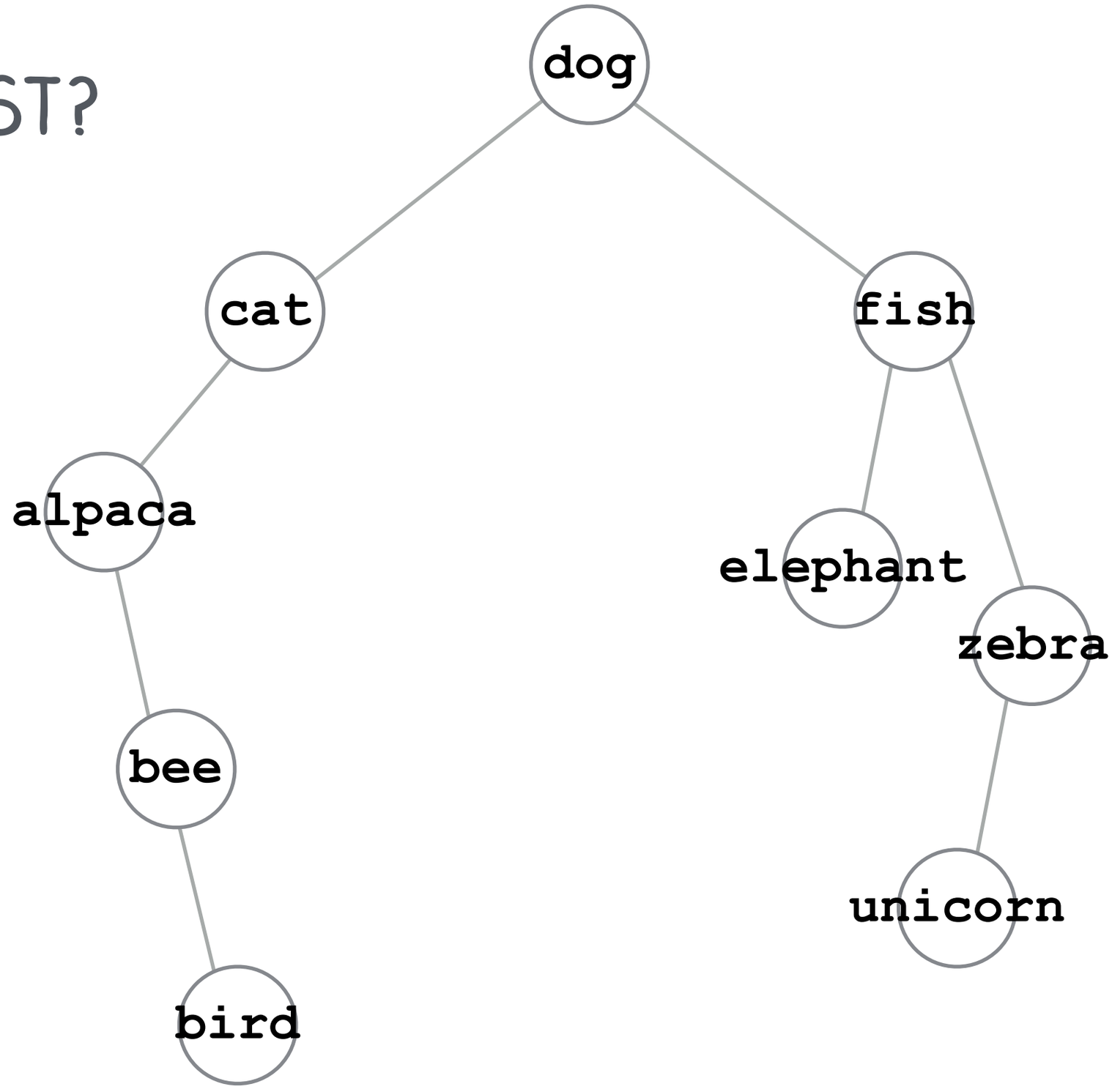
-NOTE: at each step we eliminate approximately half of the tree

  -what does this imply about the search complexity?

-what if the item is not found in the tree? how do we know when to stop and return `false`?

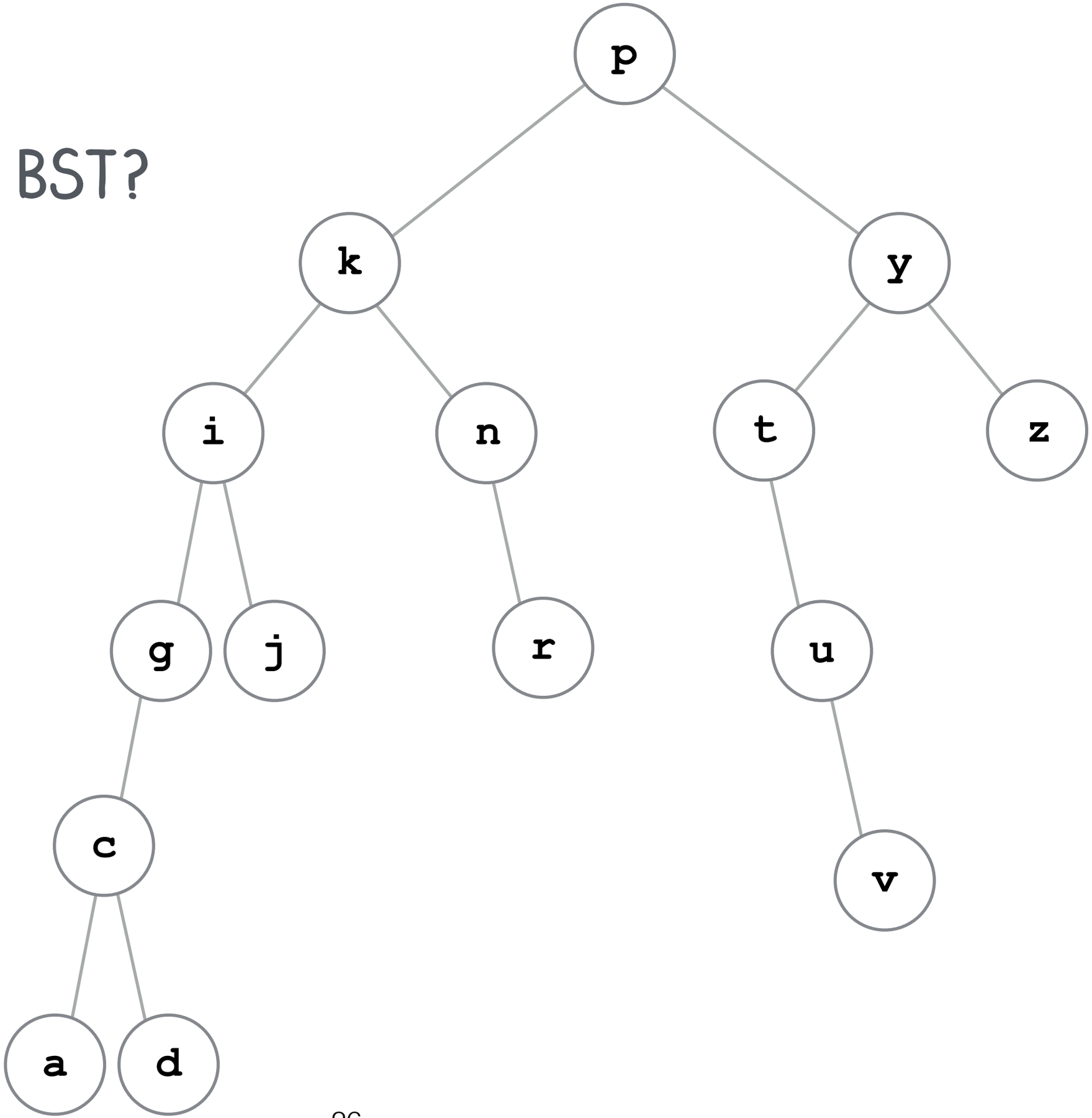
IS THIS A BST?

A) **yes**
B) **no**



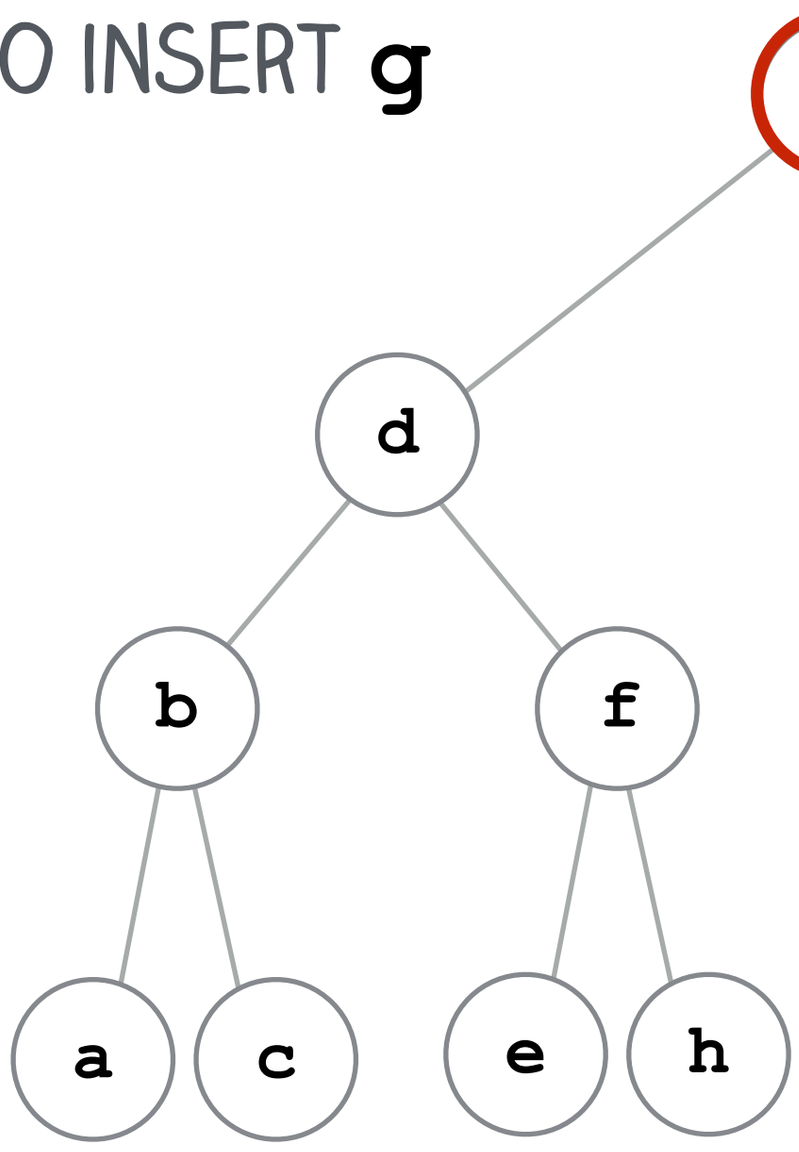
IS THIS A BST?

A) **yes**
B) **no**



insertion

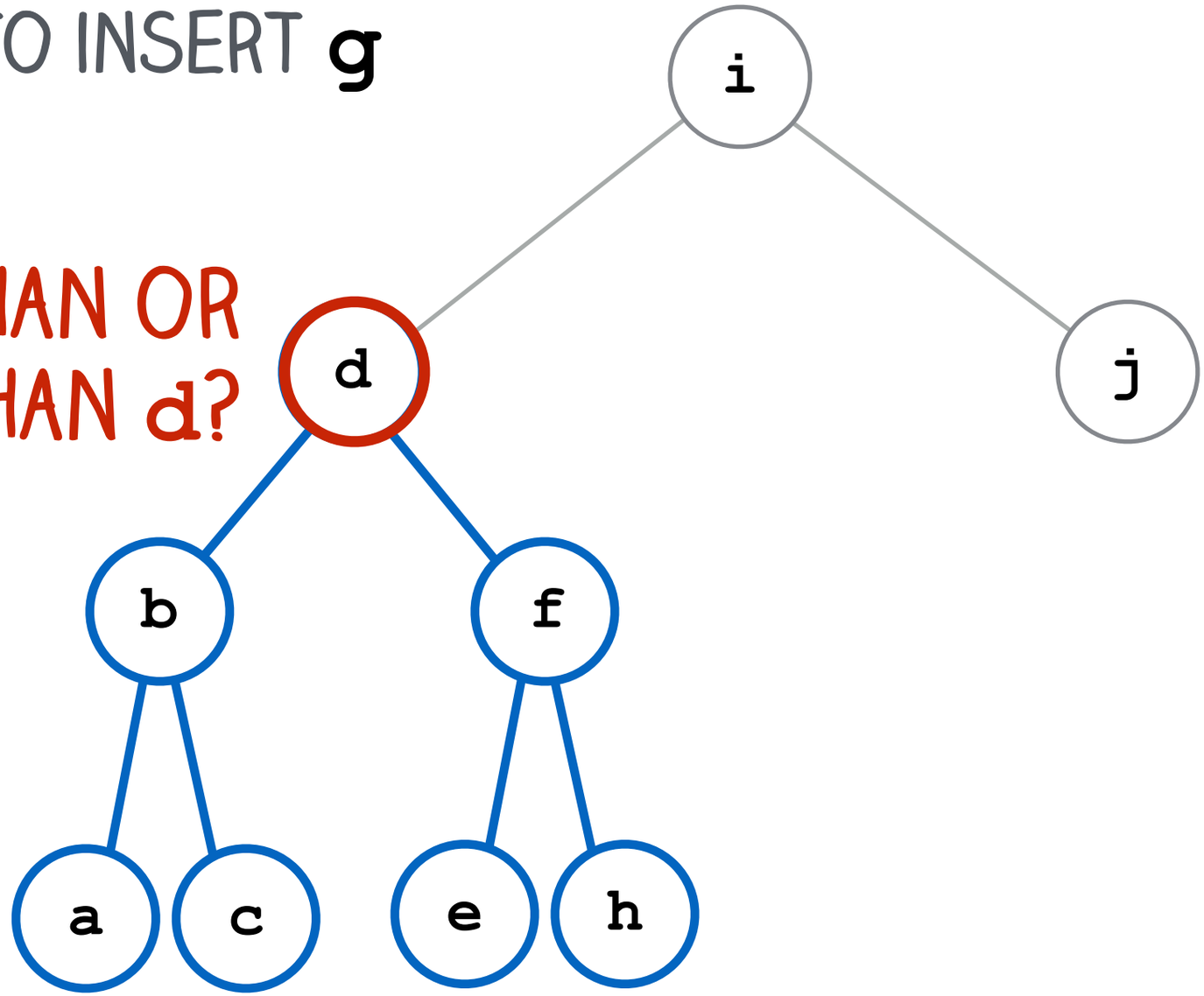
WE WANT TO INSERT **g**



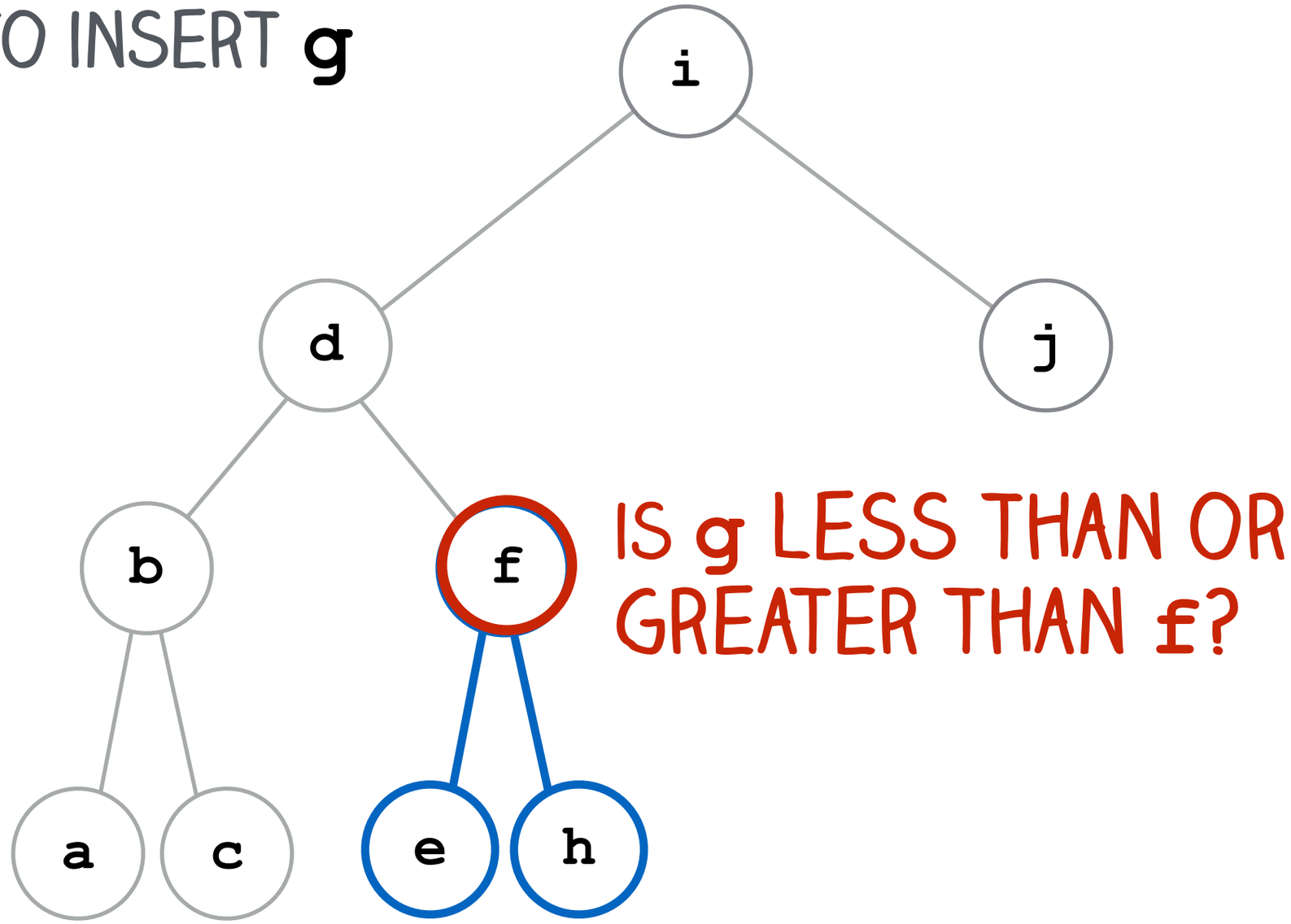
IS **g** LESS THAN OR GREATER THAN **i**?

WE WANT TO INSERT **g**

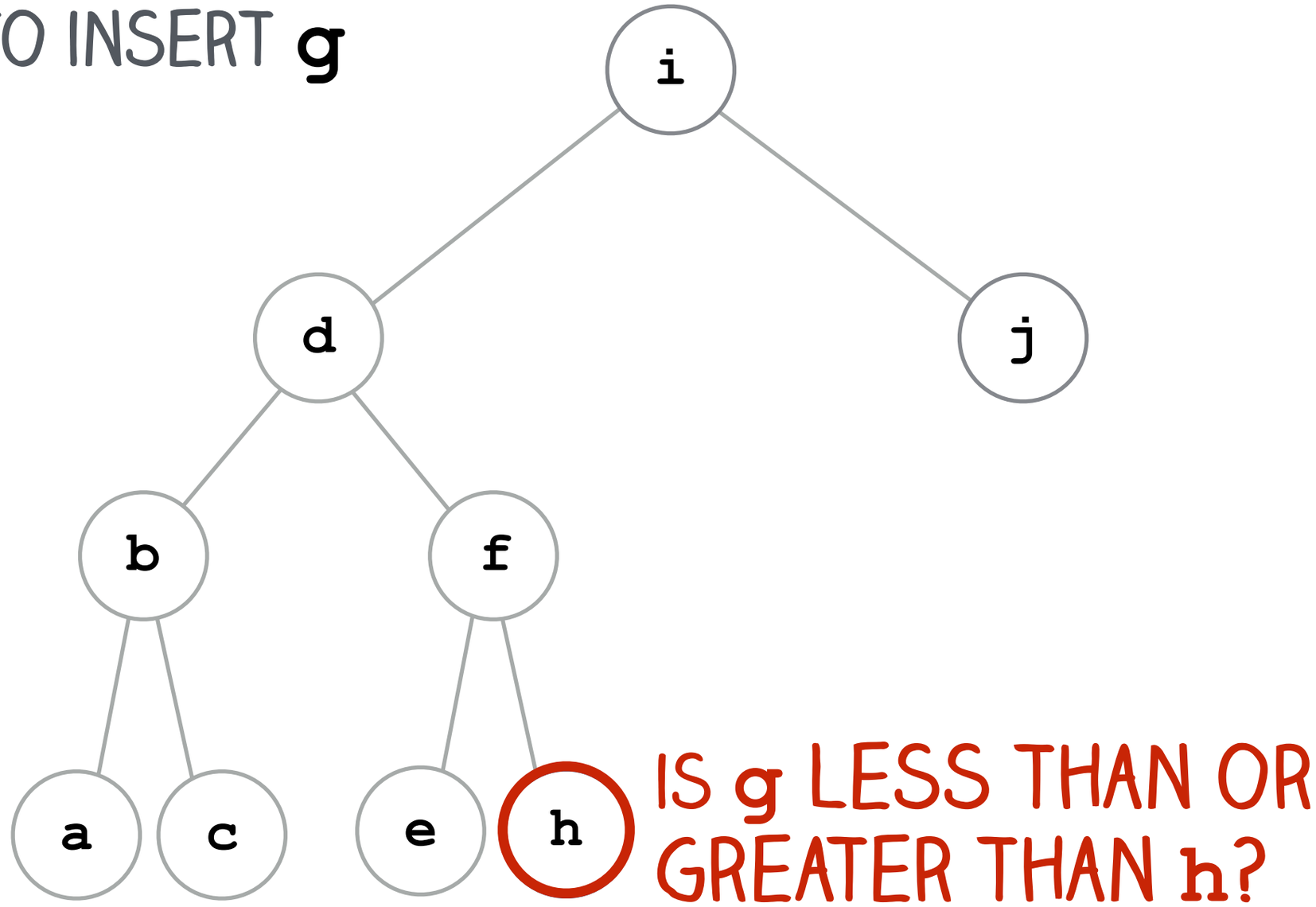
IS **g** LESS THAN OR GREATER THAN **d**?



WE WANT TO INSERT **g**

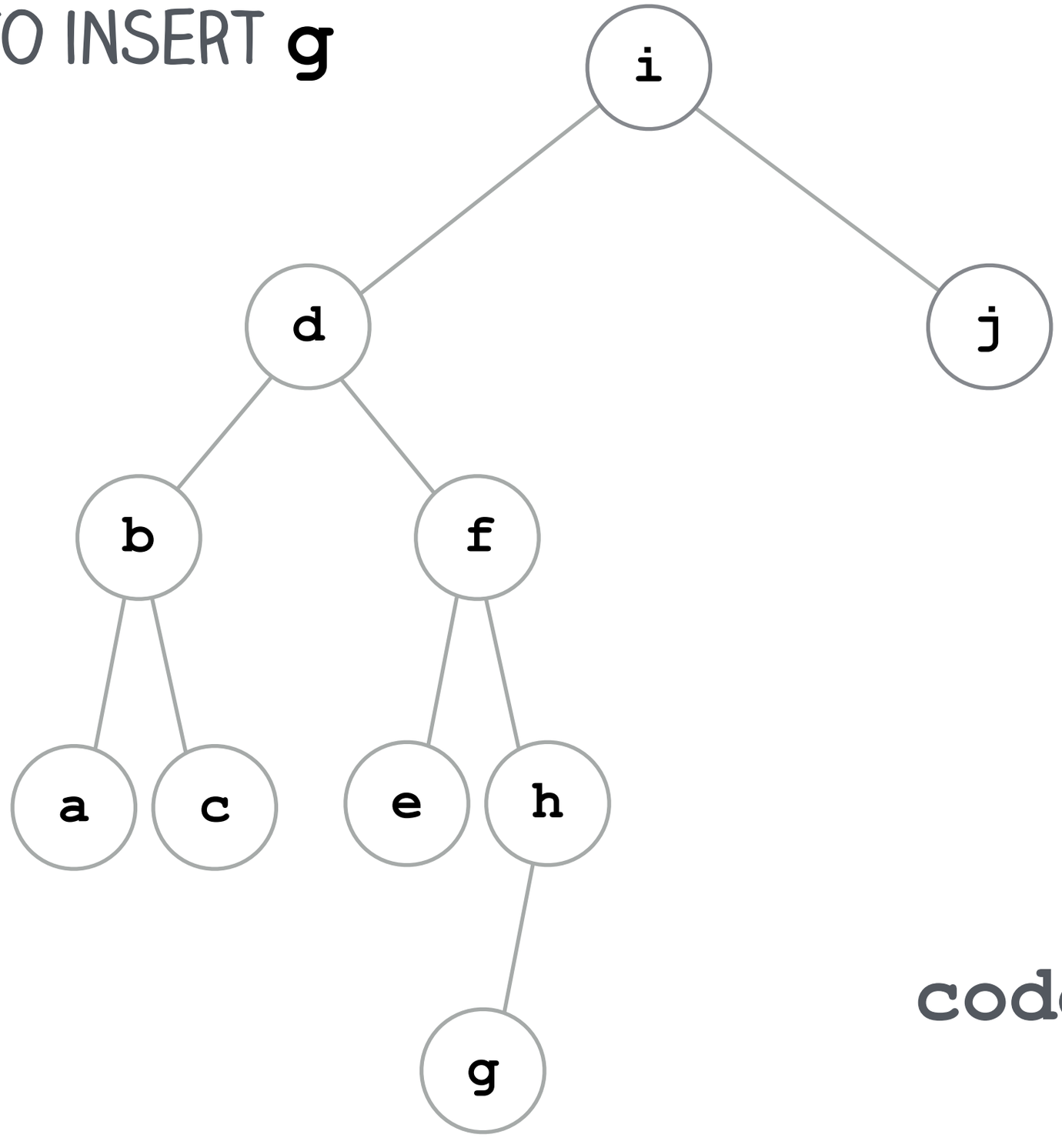


WE WANT TO INSERT **g**



IS **g** LESS THAN OR GREATER THAN **h**?

WE WANT TO INSERT **g**



code demo...

performance

-big-O complexity of BST searching

  -we disregard one subtree at each step

    -roughly half of the remaining nodes!

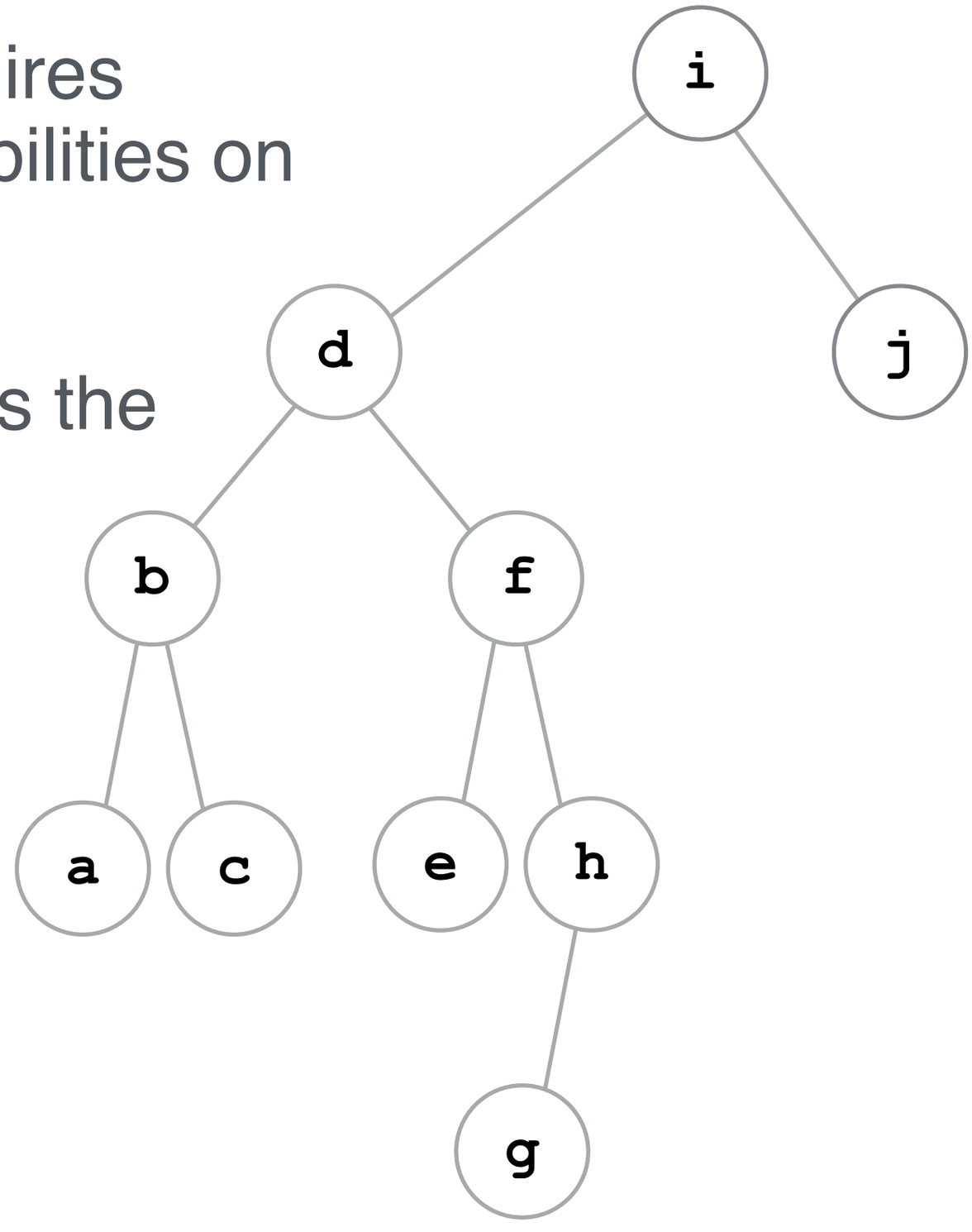
  -O(log N)

-what is the big-O complexity of BST insertion?

-searching and insertion are both **O(log N)**

- **O(log N)** performance requires eliminating half of the possibilities on each step…

- are all left and right subtrees the same size?

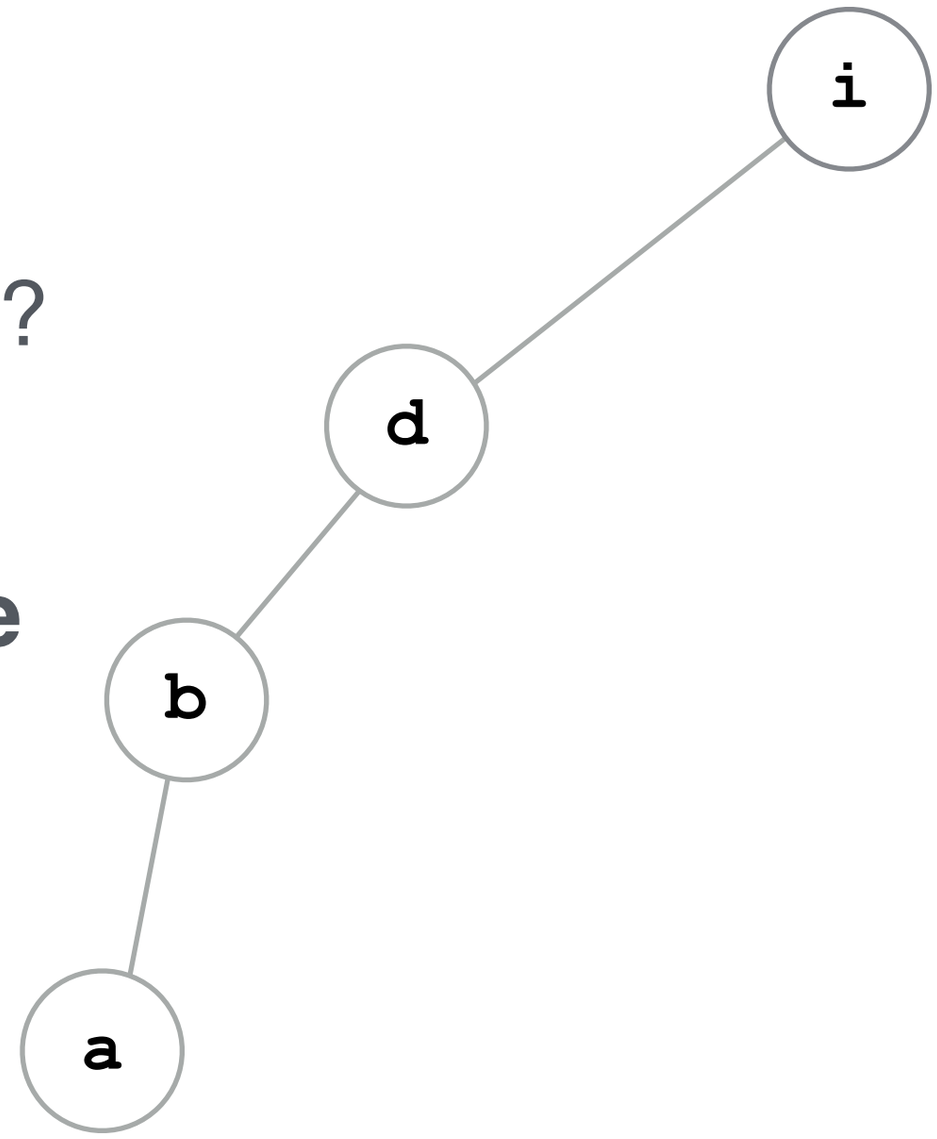


-is this a valid BST?

-what is the big-O complexity of searching and insertion for this tree?

-the order in which nodes are inserted determines the structure of the tree

-these nodes were inserted in descending order



code demo...

insertion & searching

-average case: O(log N)

-inserted in random order

-worst case:O(N)

-inserted in ascending or descending order

-best case: O(log N)

-how does this compare to a sorted array?

BST vs **array**

-arrays can have **O(log N)** searching performance as well

-how?

-what is the cost of inserting into an array?

-we can put it at the end and resort: **O(log N)**

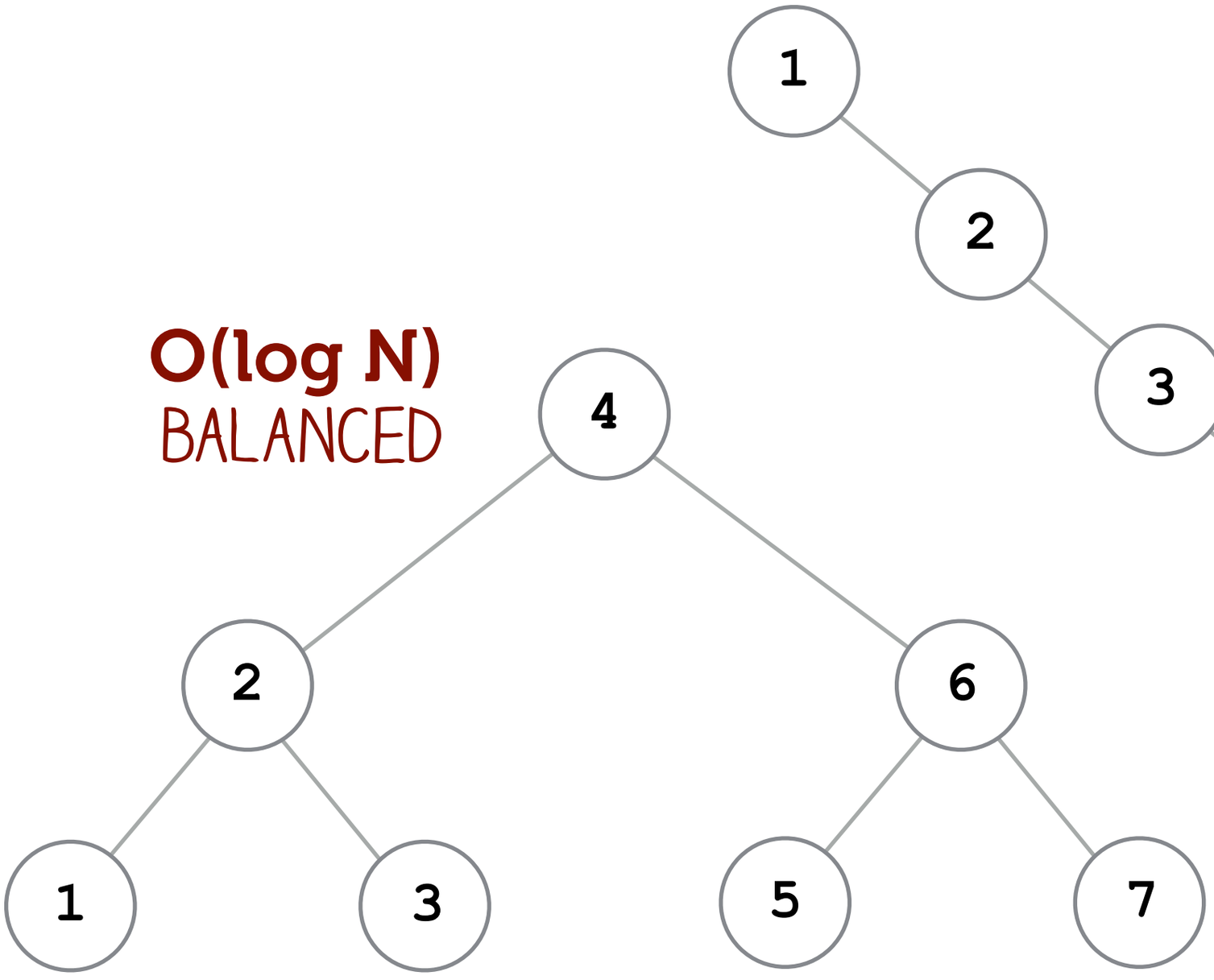
-or insert it in the right spot: **O(N)**

BST WINS FOR INSERTION! YAY!

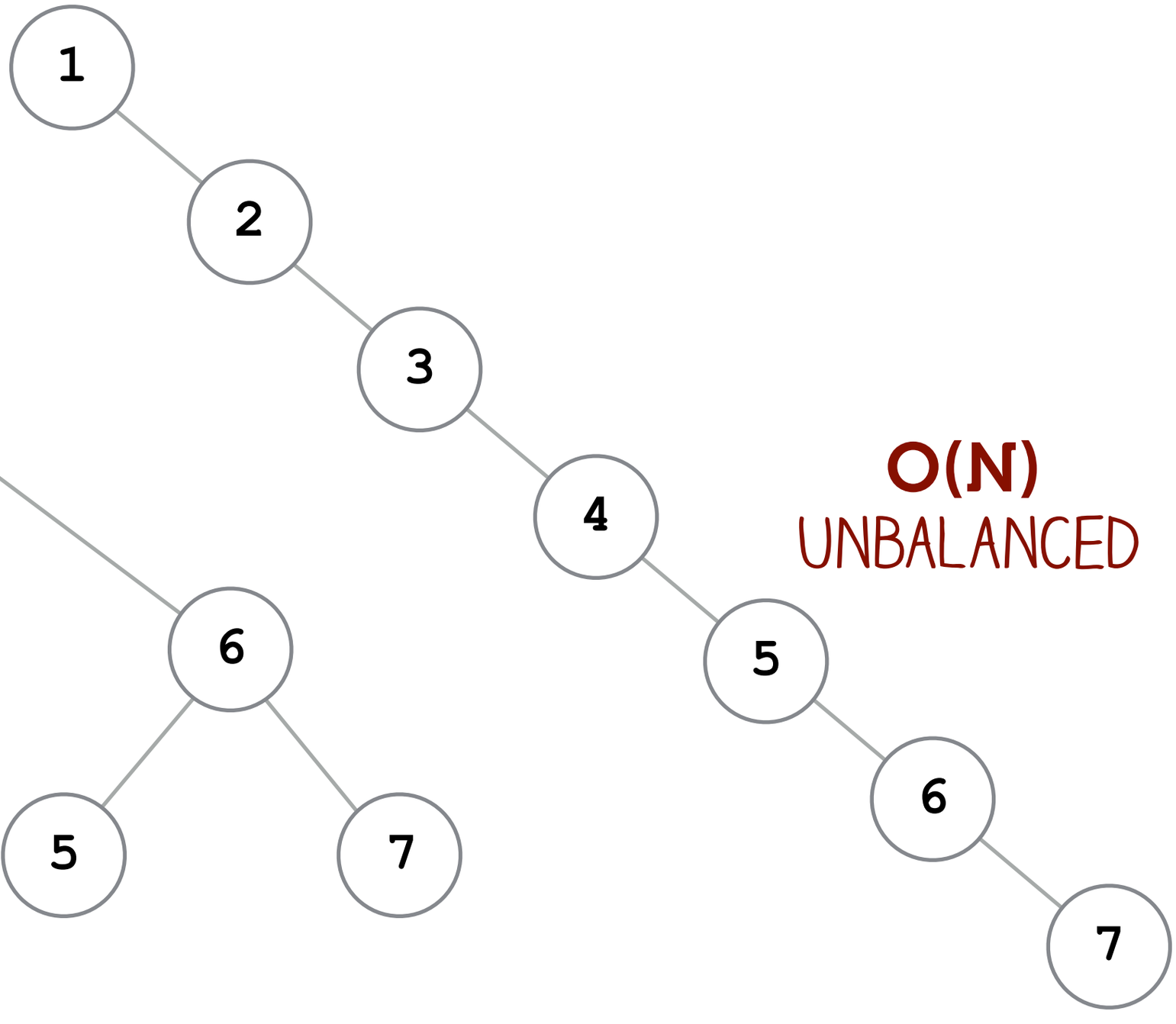
balanced VS unbalanced

ALL OPERATIONS DEPEND ON HOW WELL-BALANCED THE TREE IS

O(log N)
BALANCED



O(N)
UNBALANCED



deletion

-since we must maintain the properties of a tree structure, deletion is more complicated than with an array or linked-list

-there are three different cases:

1. deleting a leaf node
2. deleting a node with one child subtree
3. deleting a node with two children subtrees

-first step of deletion is to find the node to delete

-just a regular BST search

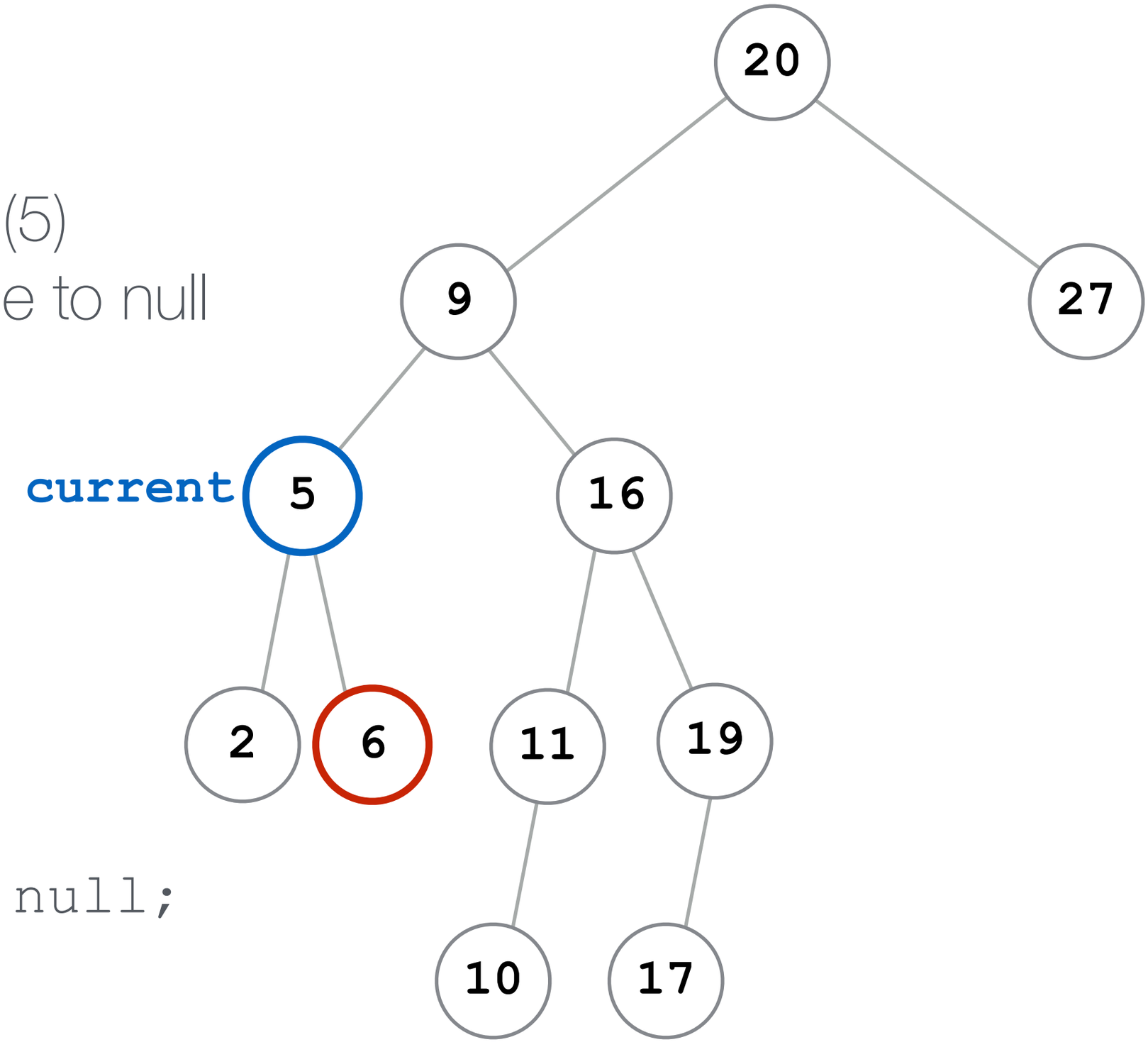
-BUT, stop at the *parent* of the node to be deleted

CASE 1: **deleting a leaf node**

DELETE NODE 6

-stop at parent node (5)

-set parent's reference to null



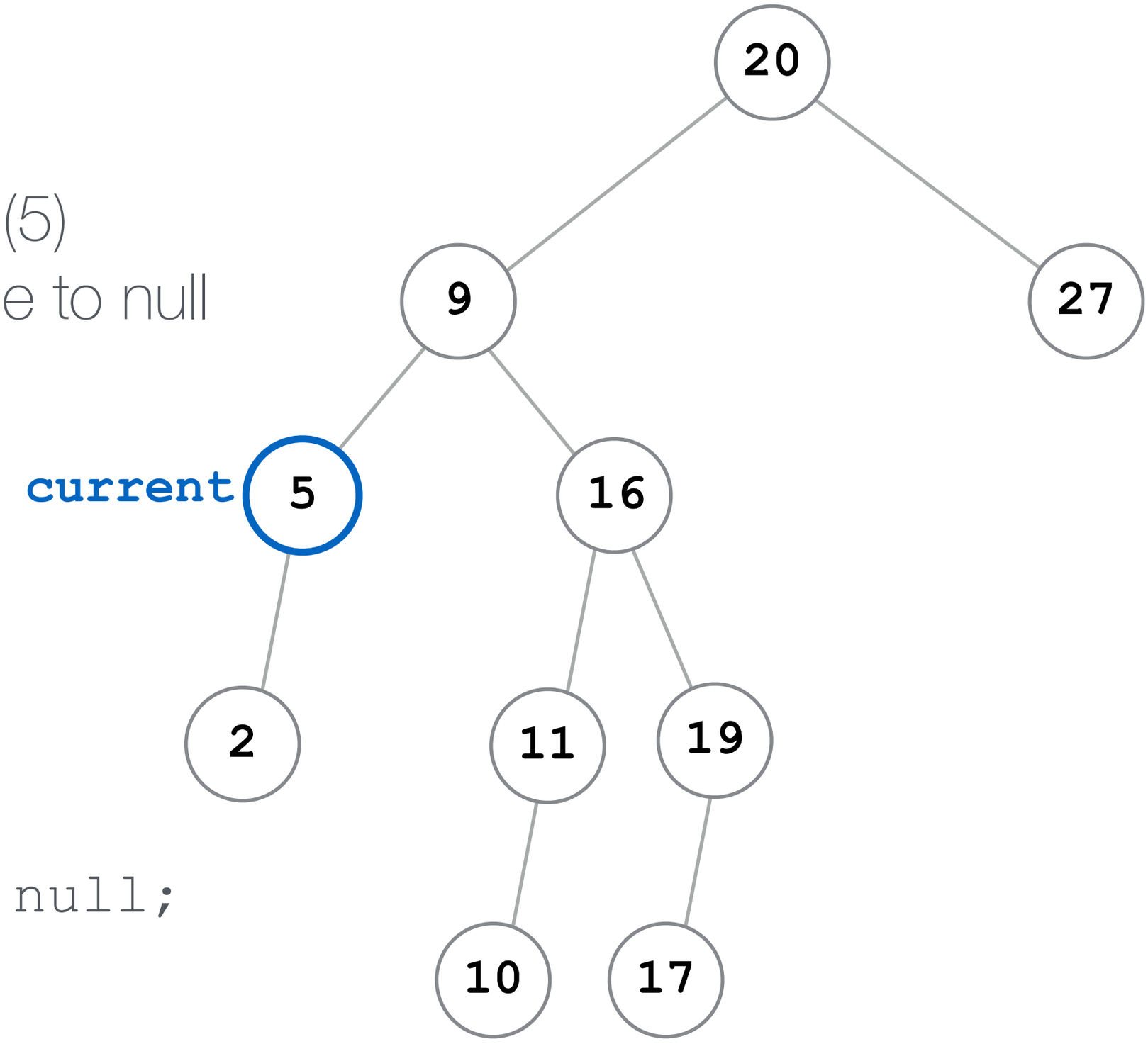

```
current.right = null;
```

CASE 1: **deleting a leaf node**

DELETE NODE 6

-stop at parent node (5)

-set parent's reference to null

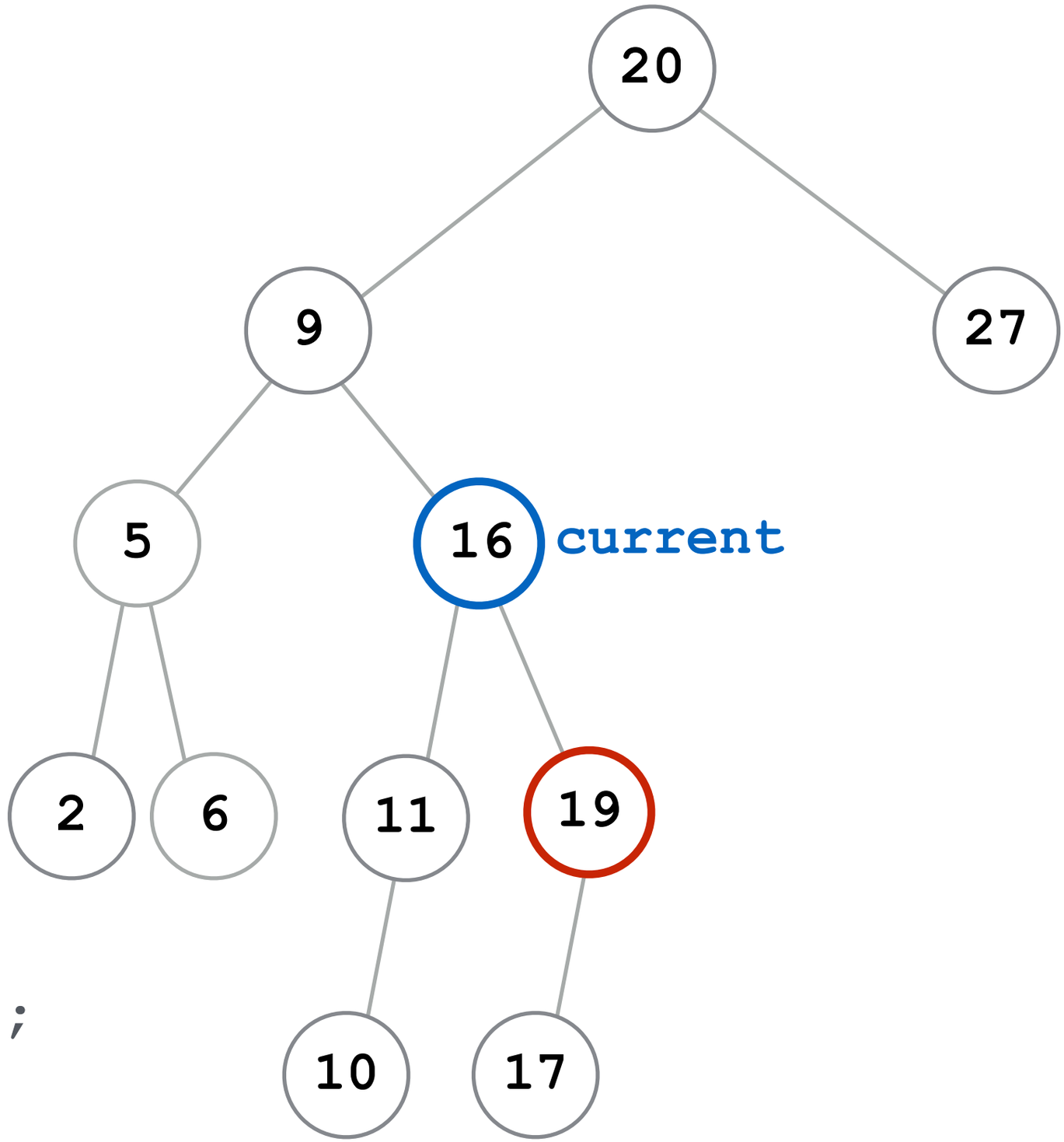



```
current.right = null;
```

CASE 2: **delete node with 1 child**

DELETE NODE 19

-stop at parent node (16)
-set parent's reference to node's child
-multiple cases depending on which side the child and grandchild are on!

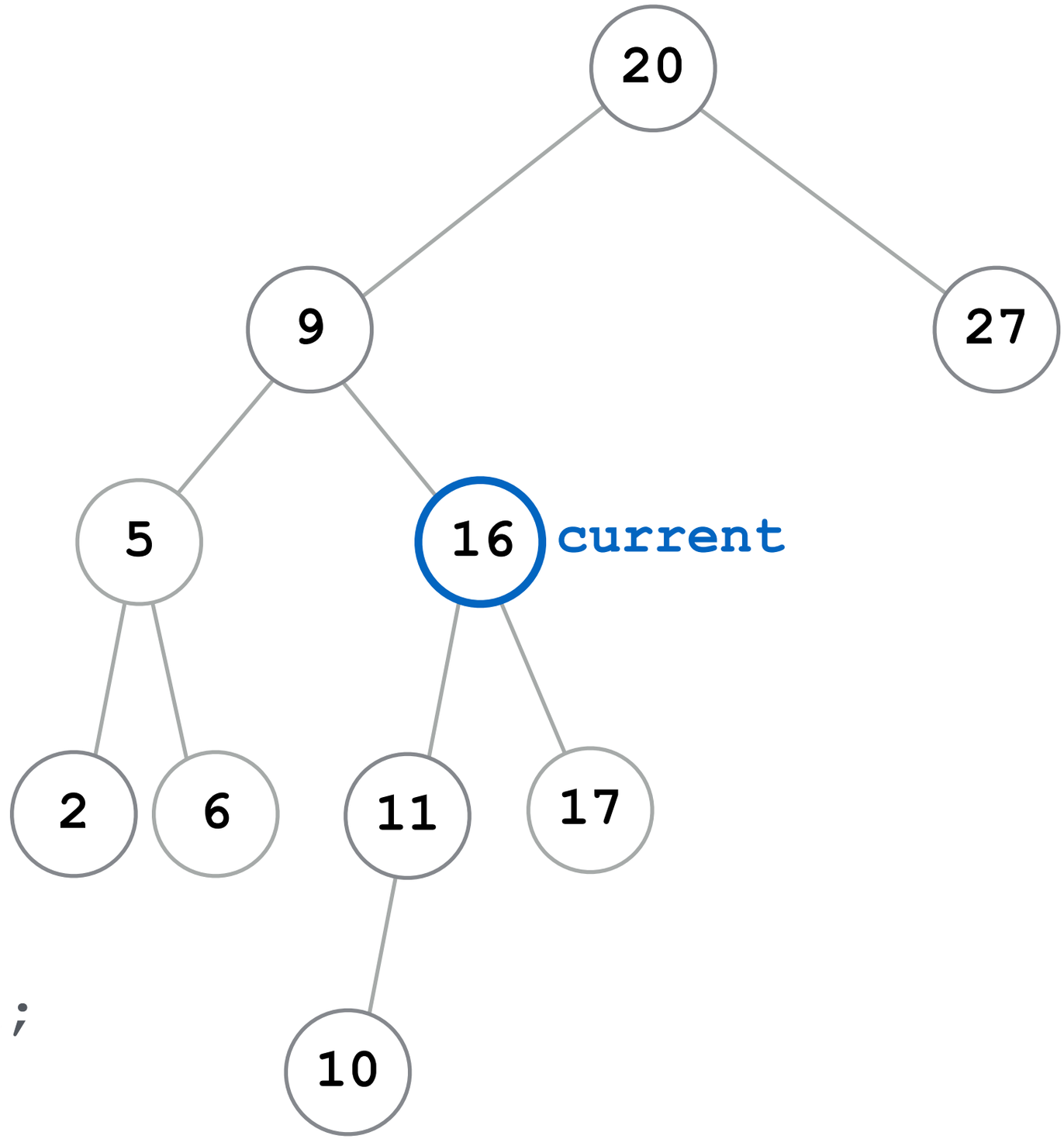



```
current.right =
  current.right.left;
```

CASE 2: **delete node with 1 child**

DELETE NODE 19

-stop at parent node (16)
-set parent's reference to node's child
-multiple cases depending on which side the child and grandchild are on!

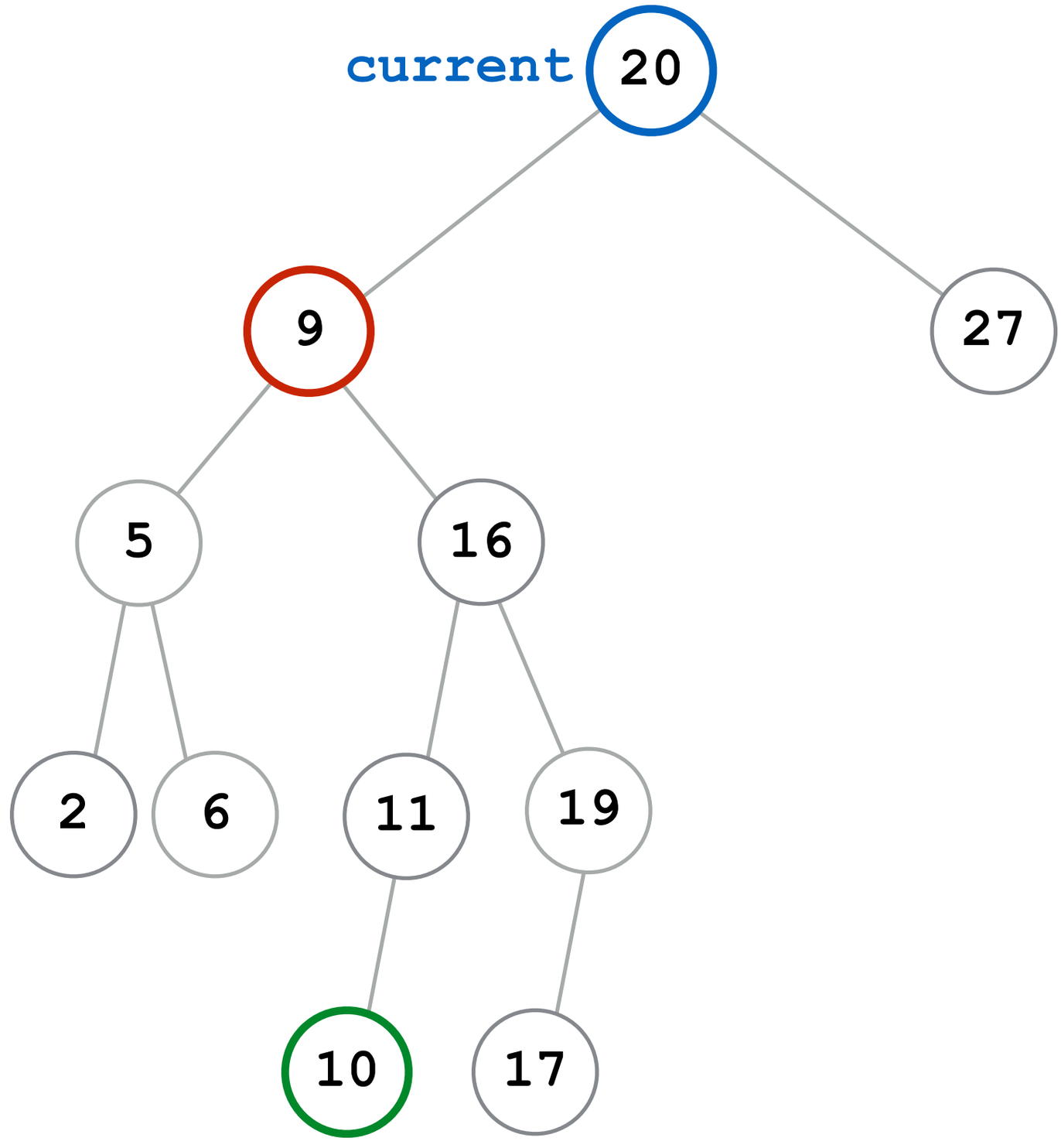



```
current.right =
  current.right.left;
```

CASE 3: **delete node with 2 children**

DELETE NODE 9

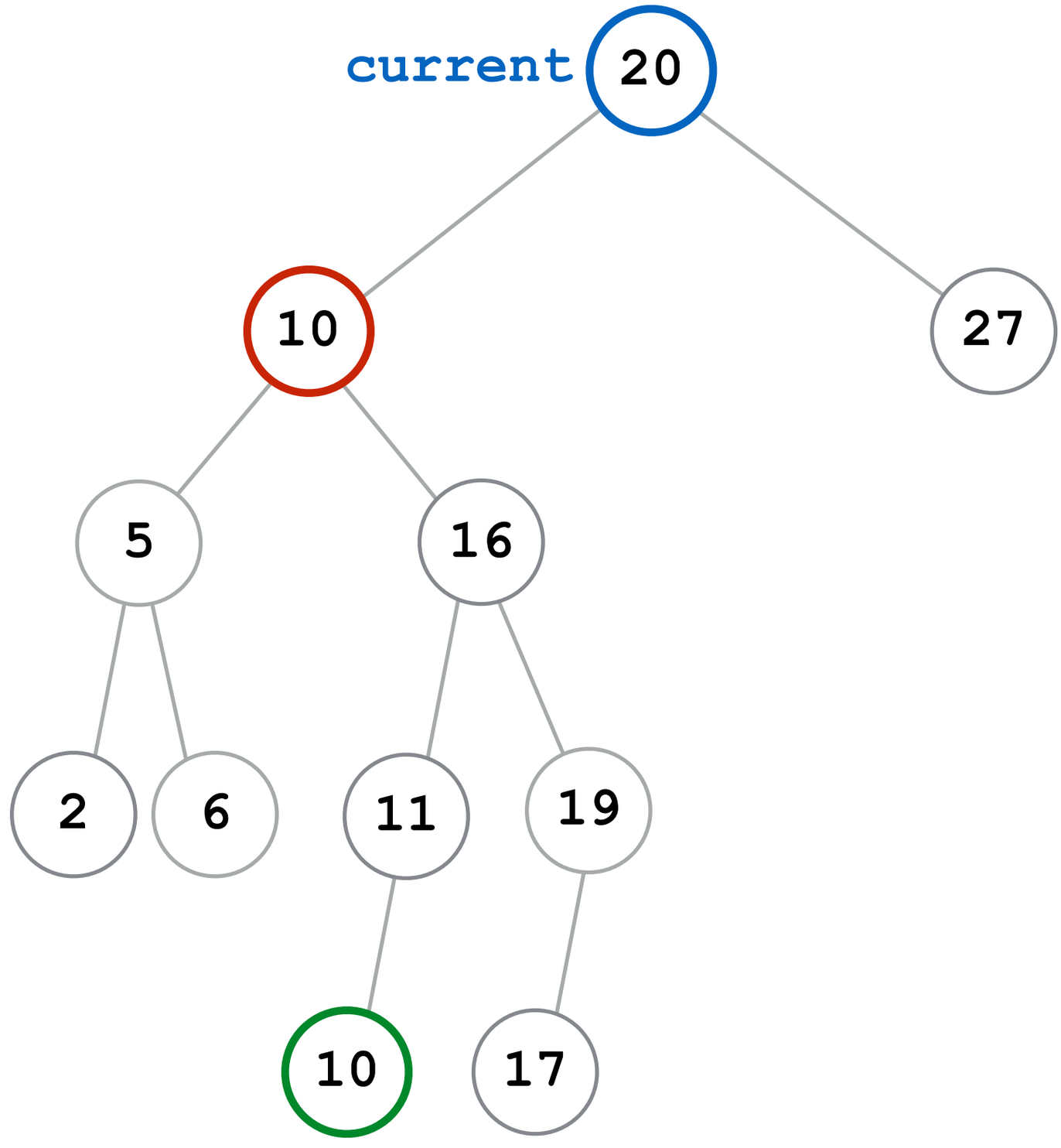
-stop at parent node (20)
-replace the node with smallest item in its right subtree(10)



CASE 3: **delete node with 2 children**

DELETE NODE 9

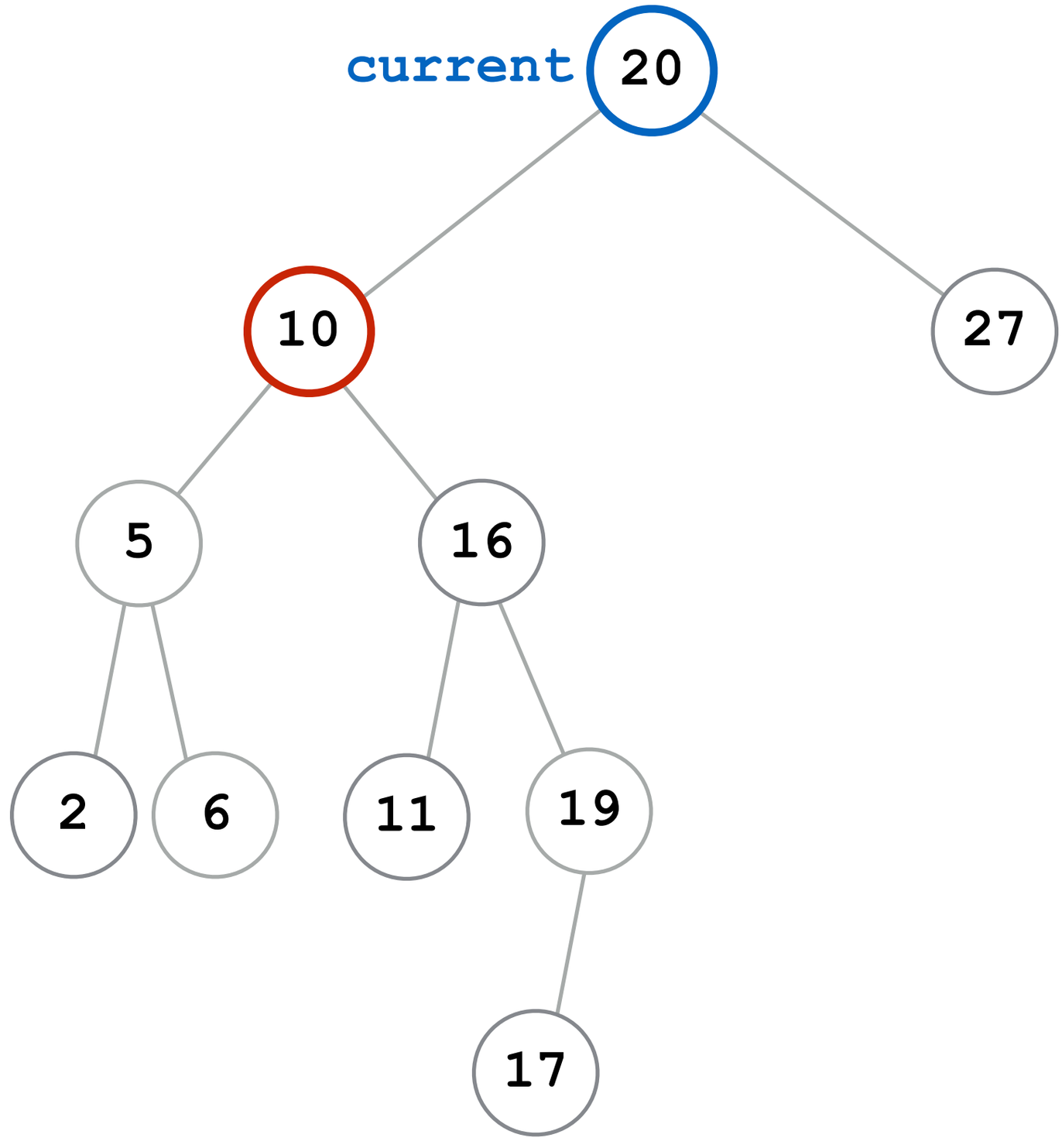
-stop at parent node (20)
-replace the node with smallest item in its right subtree(10)
-perform a delete on on ***successor*** (10)



CASE 3: **delete node with 2 children**

DELETE NODE 9

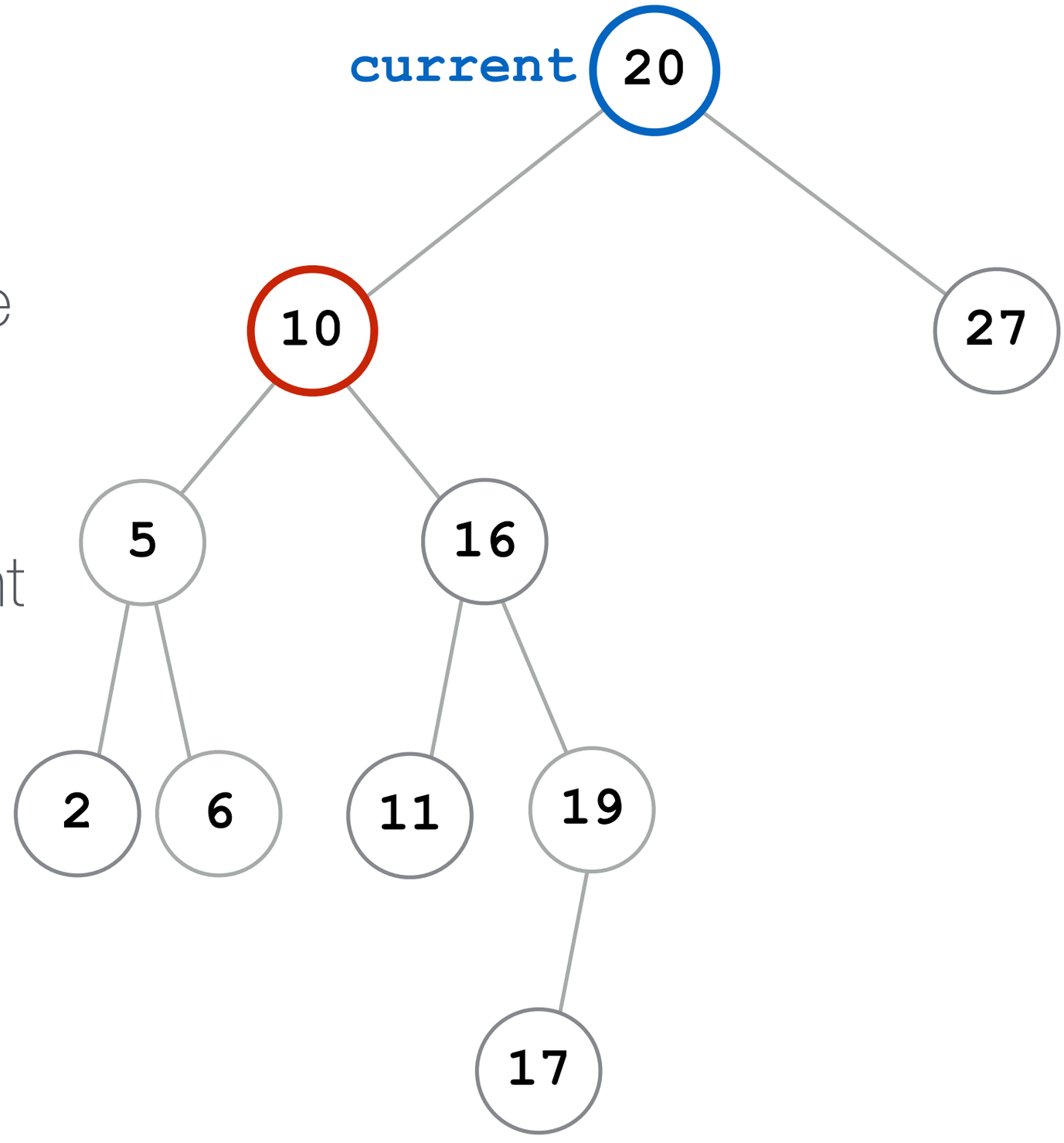
-stop at parent node (20)
-replace the node with smallest item in its right subtree(10)
-perform a delete on ***successor*** (10)
(guaranteed not to have a left child! case 1 or 2…)



CASE 3: **delete node with 2 children**

Replacing the deleted node with the smallest child in right subtree guarantees the BST structure… why?

The smallest item in the right subtree is greater than any item in the left subtree, ***and*** smaller than any item in the new right subtree.



deletion performance

WHAT IS THE COST OF DELETING A NODE FROM A BST?

-first, find the node we want to delete: **O(log N)**

-cost of:

-case 1 (delete leaf):

SET A SINGLE REFERENCE TO NULL: **O(1)**

-case 2 (delete node with 1 child):

BYPASS A REFERENCE: **O(1)**

-case 3 (delete node with 2 children):

FIND THE SUCCESSOR: **O(log N)**

DELETE THE DUPLICATE SUCCESSOR: **O(1)**

some other useful properties

-how can we find the smallest node?

-start at the root, go as far left as possible

-**O(log N)** on a balanced tree

-how can we find the largest node?

-start at the root, go as far right as possible

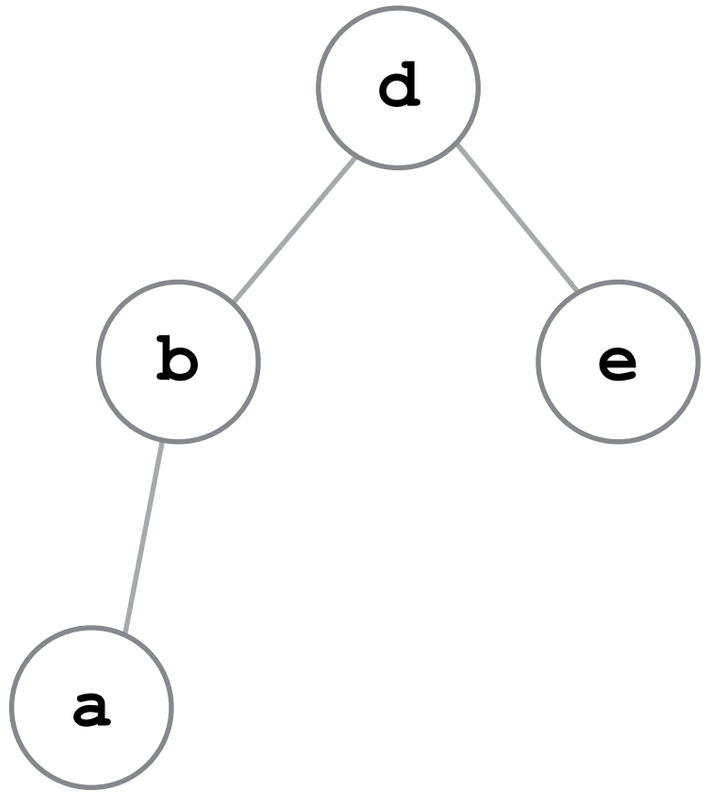
-**O(log N)** on a balanced tree

-how can we print nodes in ascending order?

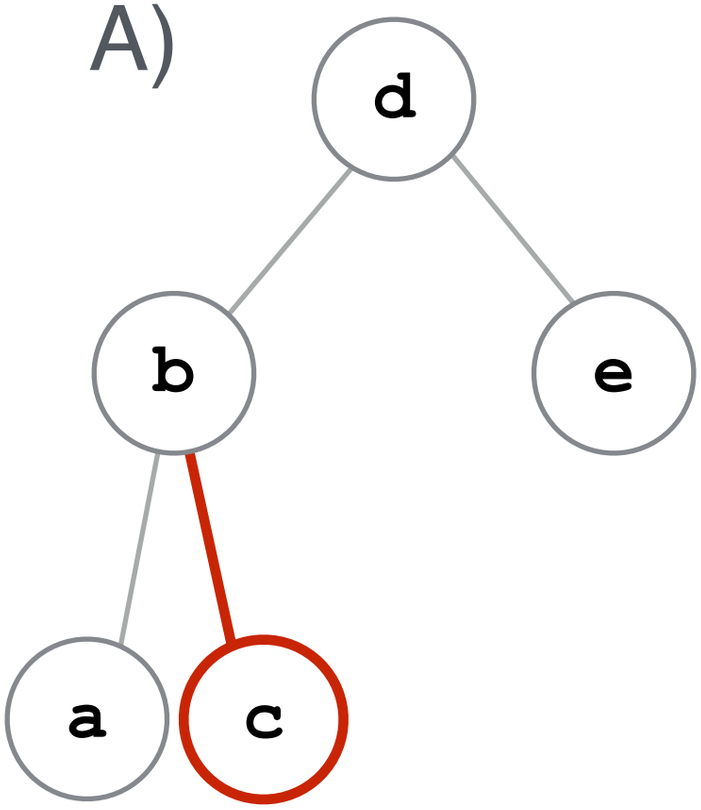
-in-order traversal

quick review…

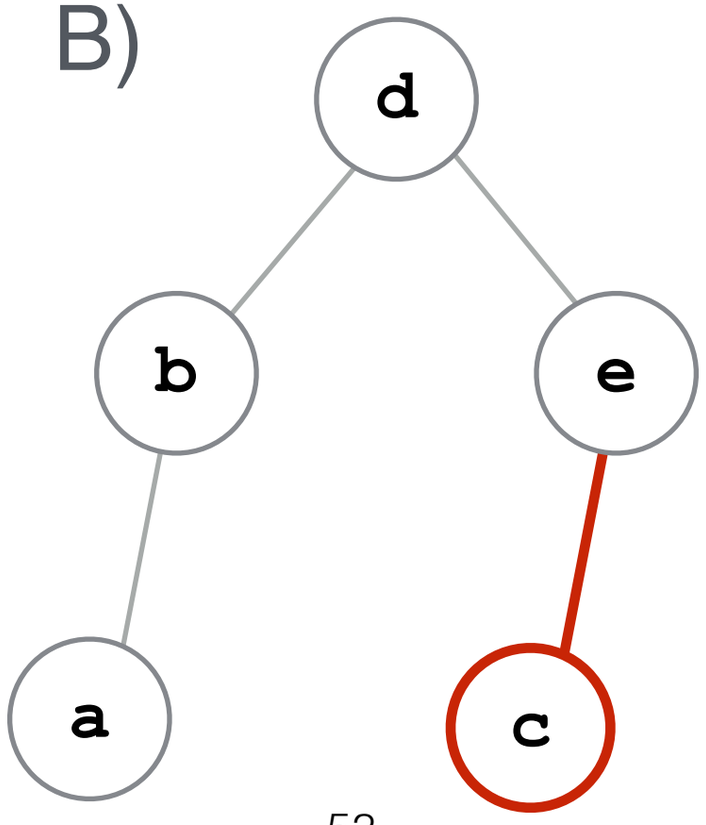
WHICH OF THE FOLLOWING TREES IS THE RESULT OF ADDING `c` TO THIS BST?



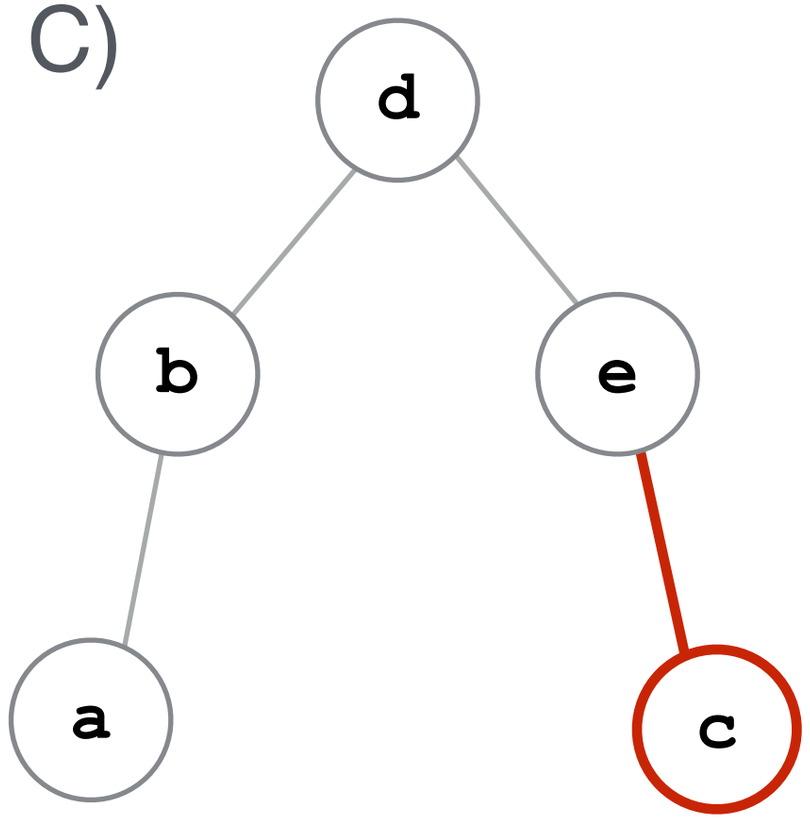
A)



B)

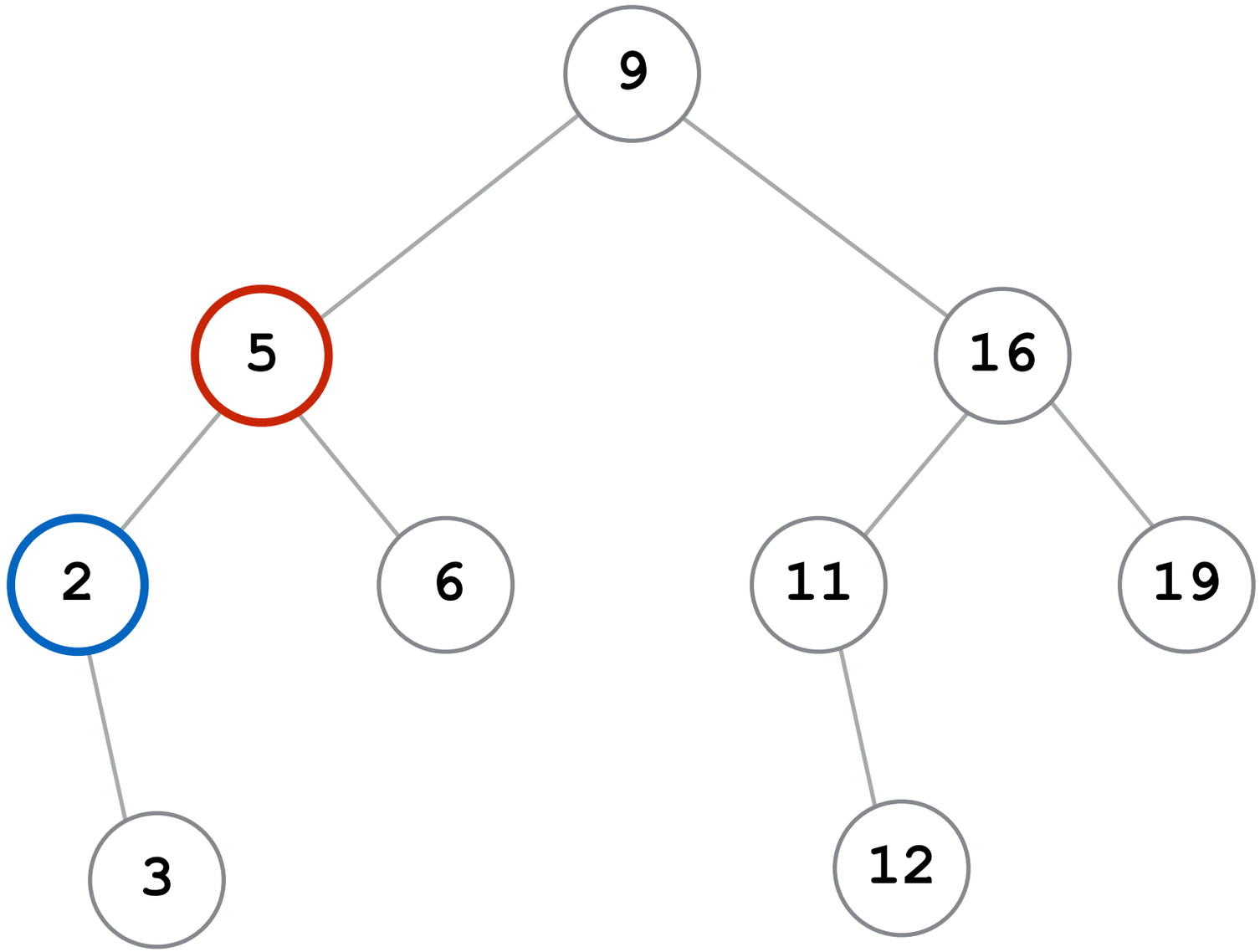


C)



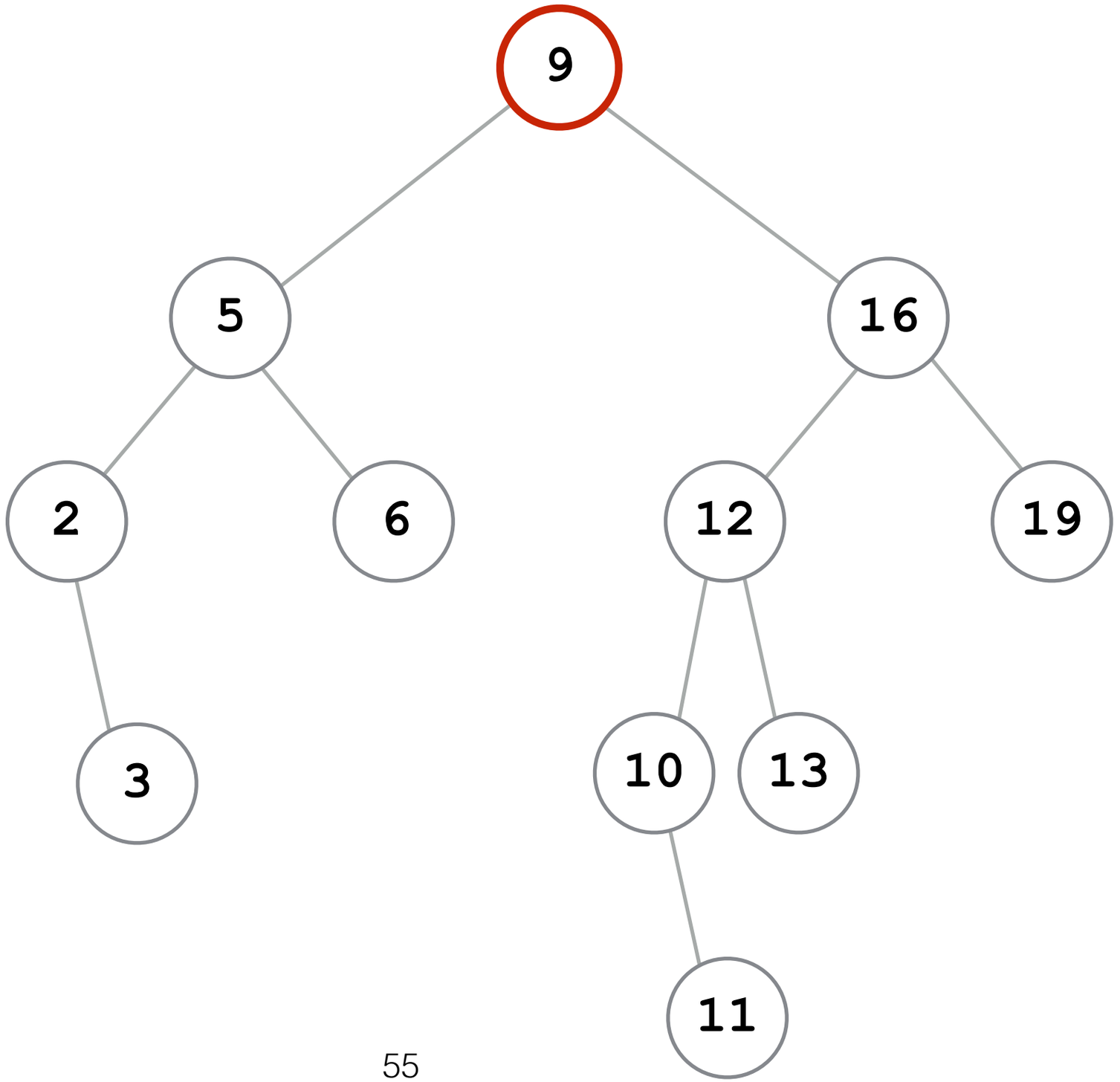
WHAT WILL 5'S LEFT CHILD BE AFTER DELETING 2?

A) **3**
B) **6**
C) **null**



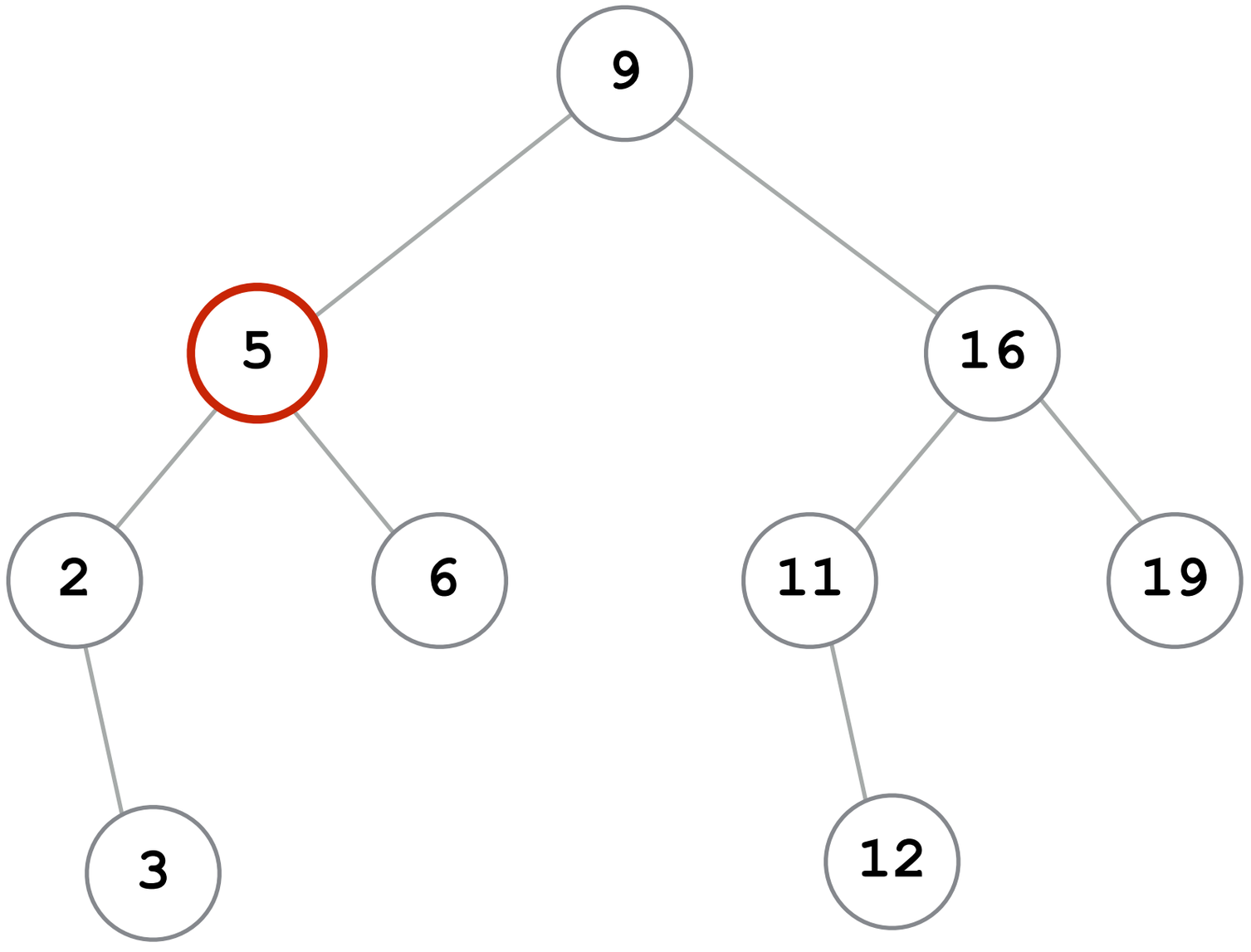
WHAT NODE WILL REPLACE 9 AFTER DELETING 9?

A) **6**
B) **10**
C) **13**
D) **19**



WHAT NODE WILL REPLACE 5 AFTER DELETING 5?

A) **2**
B) **3**
C) **6**
D) **12**



next time…

-reading

-chapter 14 in book

-homework

-assignment 7 due tonight

-assignment 8 out later today