Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 32
Binary Search Trees

FIT 1008
Introduction to Computer Science

Objectives

- To understand **Binary Search Trees**
- **Implement** Binary Search Trees:
  - ➡ search
  - ➡ insert
- **Advantages and disadvantages** of Binary Search Trees over sorted lists.

insert

```
>>> a = dict()
>>> a[123465] = "Julian"
>>> a[133123] = "Nicole"
>>> a[982211] = "David"
>>>
>>> a
{123465: 'Julian', 133123: 'Nicole', 982211: 'David'}
>>>
>>>
>>> a[133123]
'Nicole'
```

search

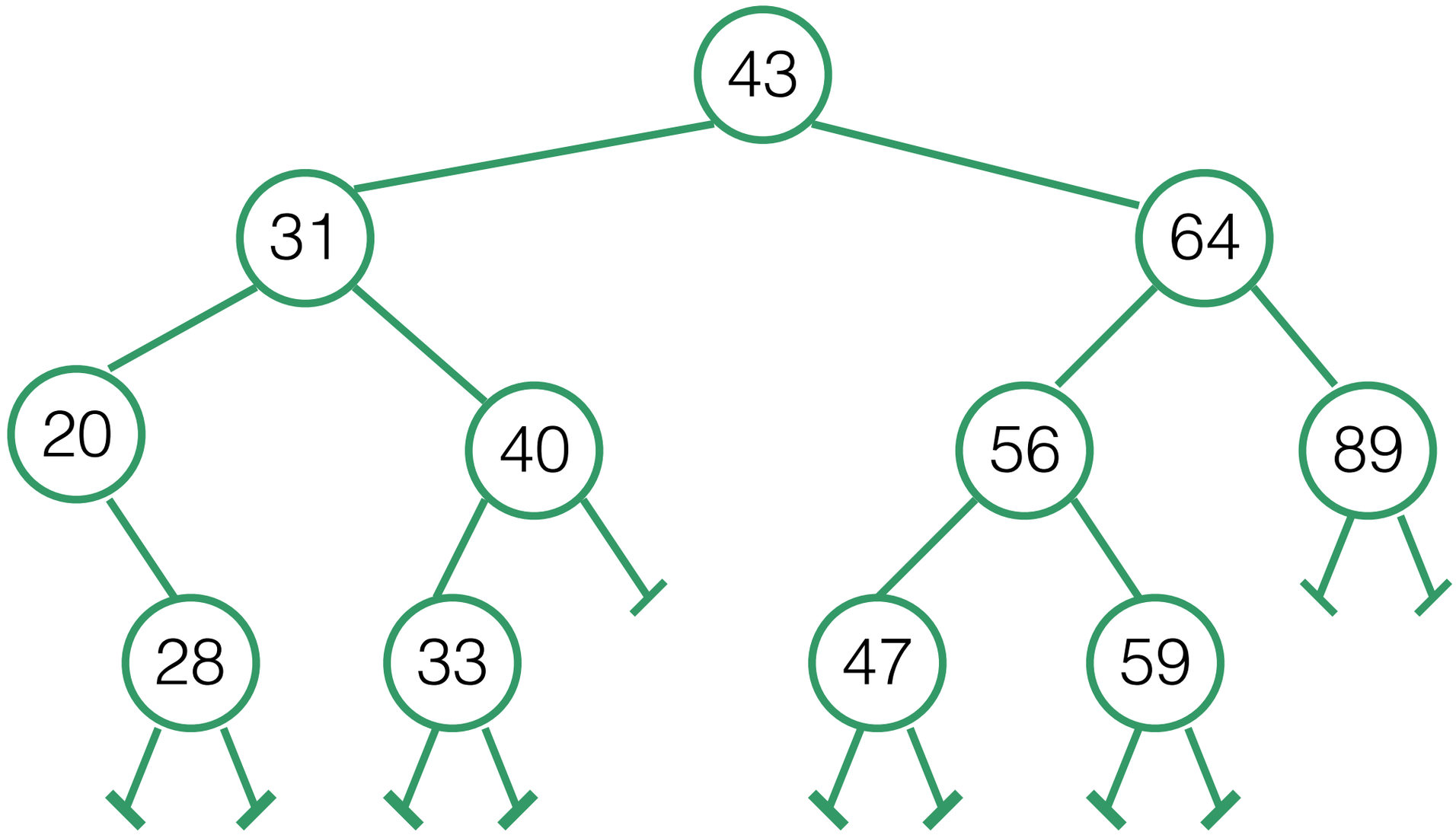
Python dictionaries are implemented using Hash Tables

You can also use a Binary Search Tree!

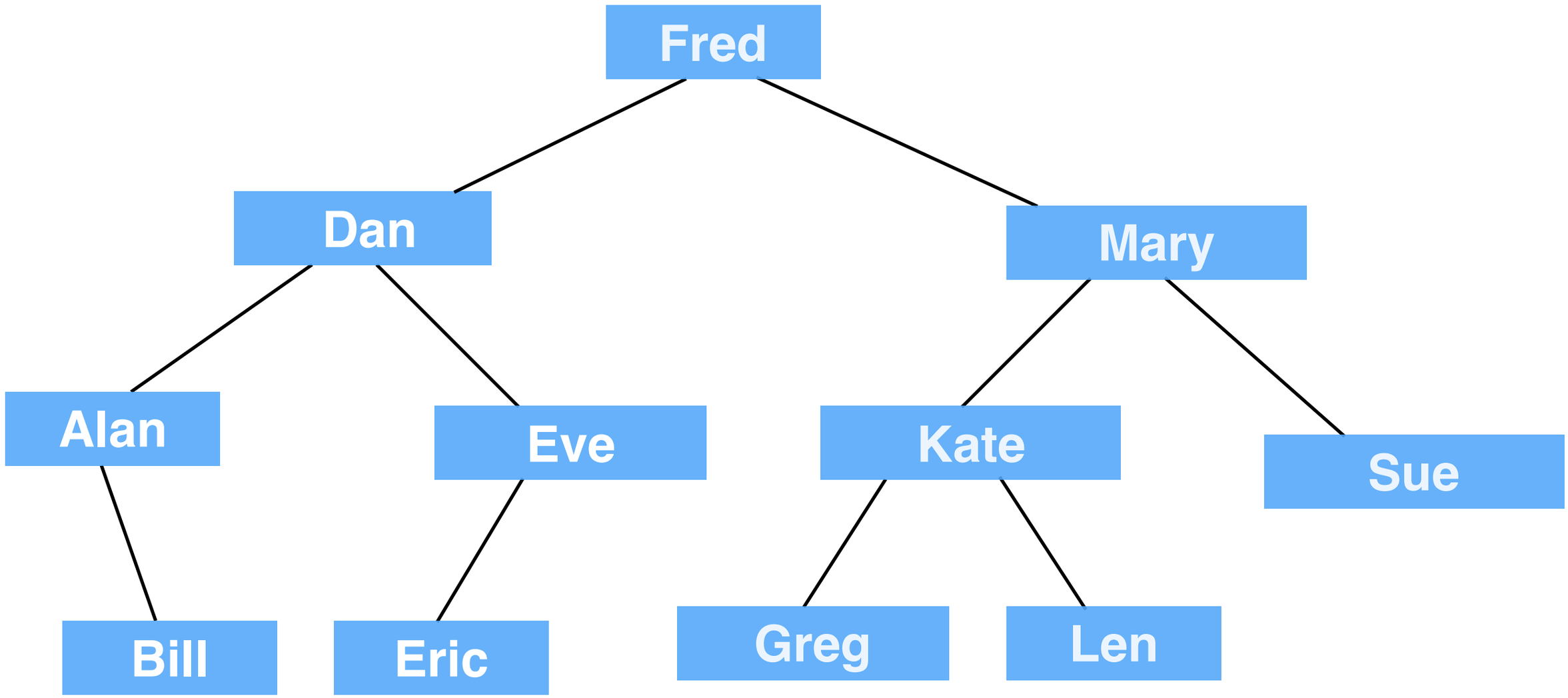
Binary Search Tree

A Binary Tree such that:

- Every node entry has a **key**
  - ➡ All **keys** in the **left subtree** of a node are **smaller than** the key of the node
  - ➡ All **keys** in the **right subtree** of a node are **greater than** the key of the node



key is an integer.



key is a string

(here not showing the associated items)

Why Binary Search Trees?

Sorted List Array-based:

- Good for search — O(log N) [binary search]
- Bad for inserting/deleting — O(N) [shuffling things around]

Linked Sorted List:

- Good for inserting/deleting – O(1) [modifying links]
- Bad for searching – O(N) [steps through the list]

Binary Search Trees:
good for searching **and** good for inserting/deleting

```python
class BinarySearchTreeNode:
    def __init__(self, key, item=None, left=None, right=None):
        self.key = key
        self.item = item
        self.left = left
        self.right = right
```

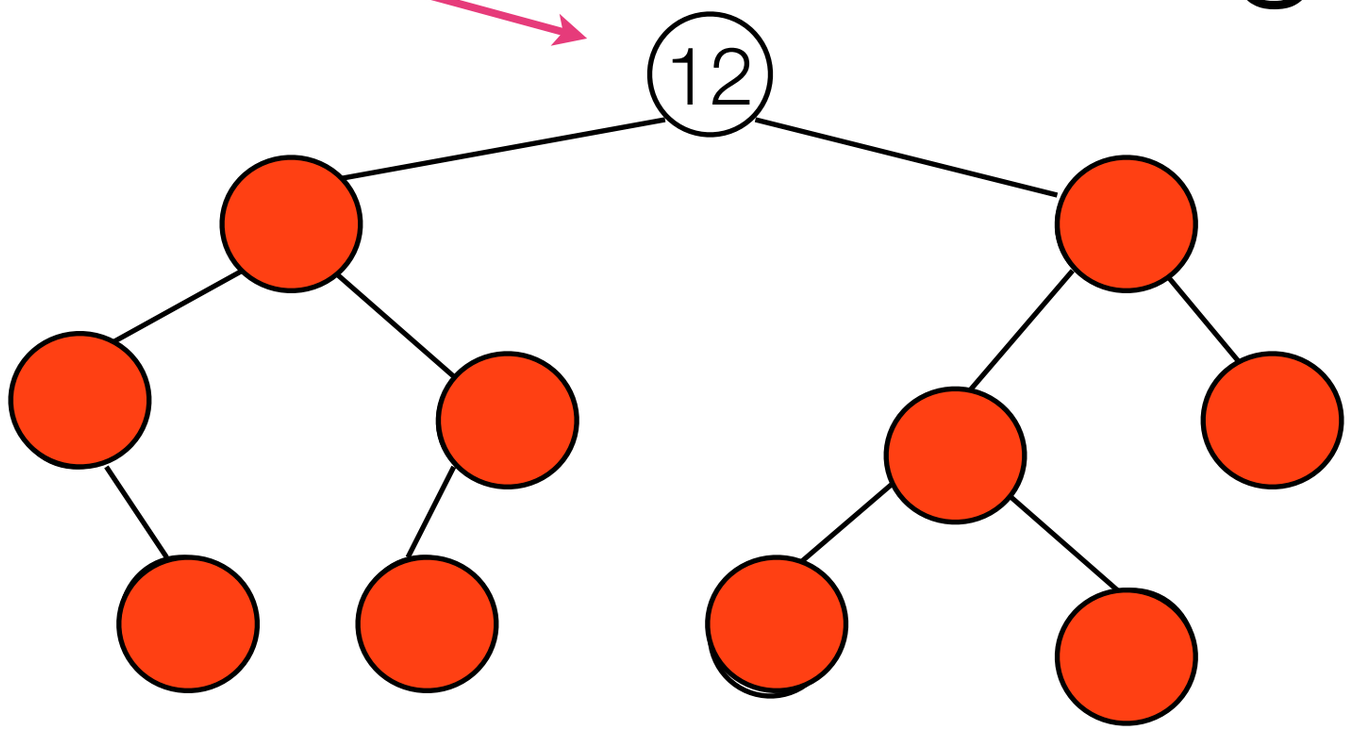
```python
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def is_empty(self):
        return self.root is None
```

Motivation: Search

Searching

root

target = 13

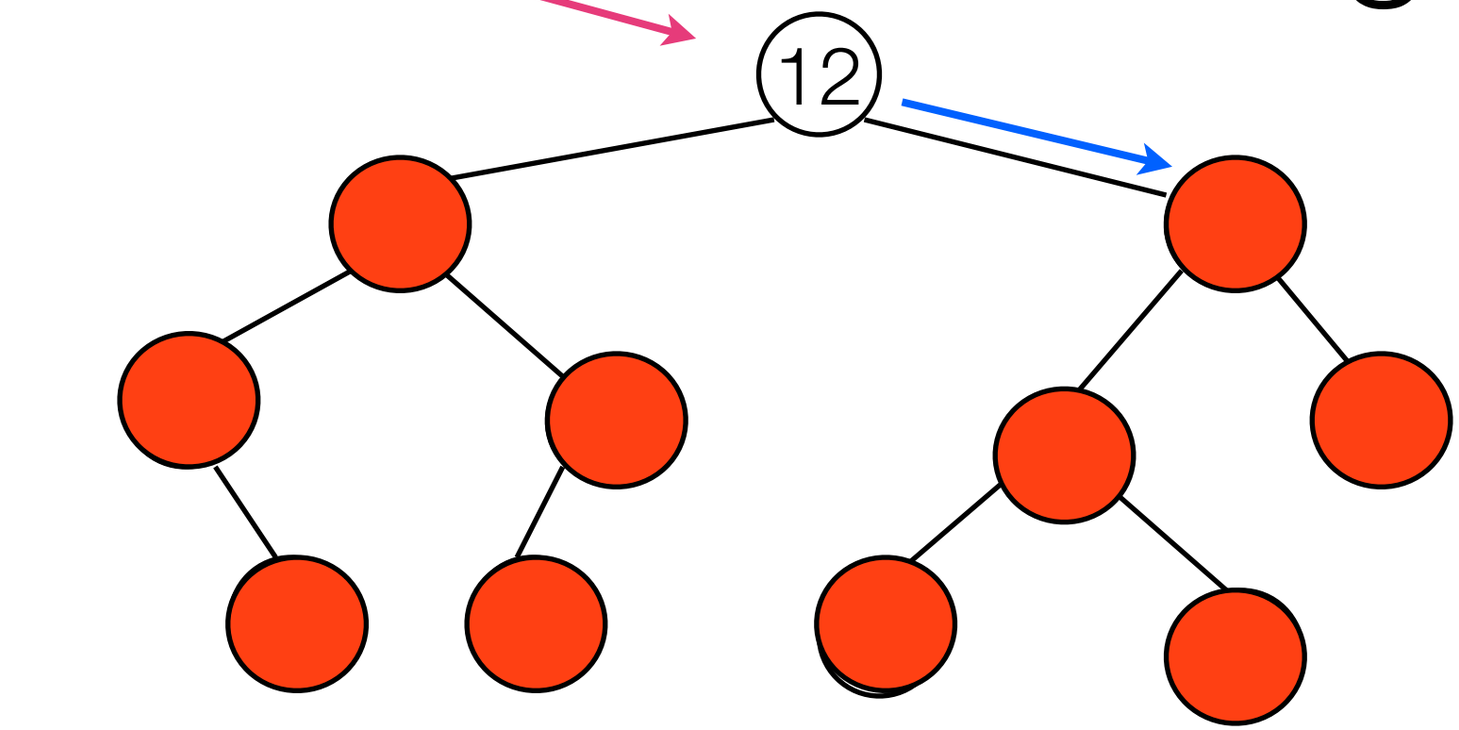


Only showing keys!

Searching

root

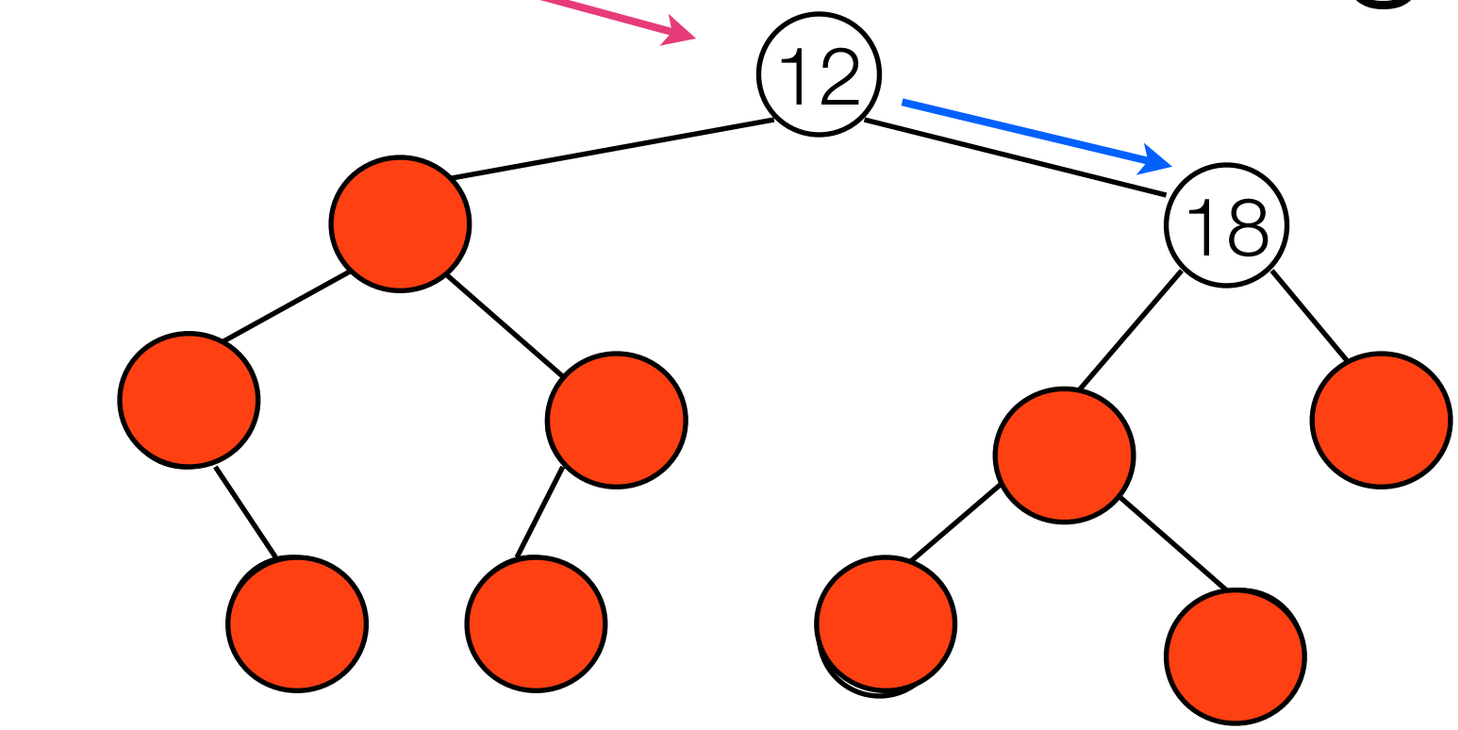
target = 13



Searching

root

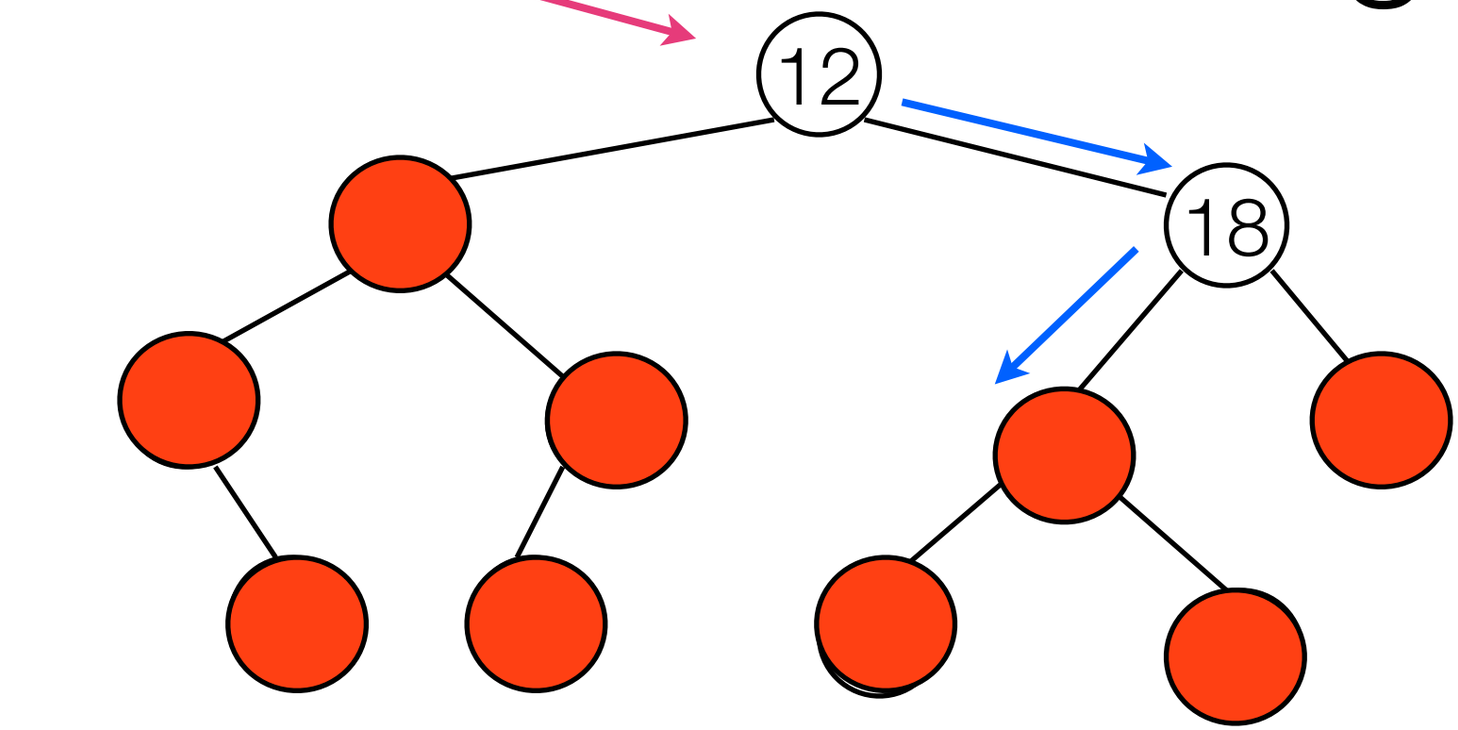
target = 13



Searching

root

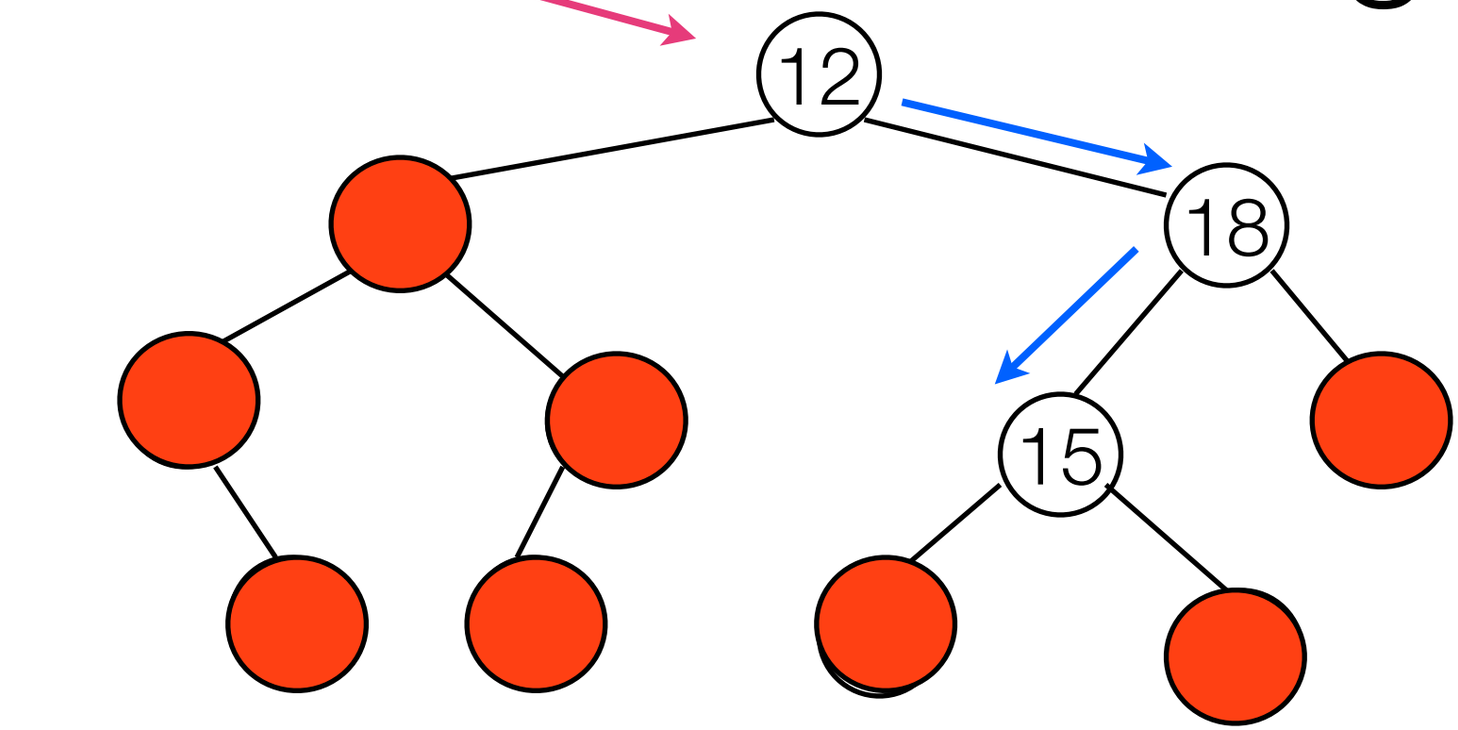
target = 13



Searching

root

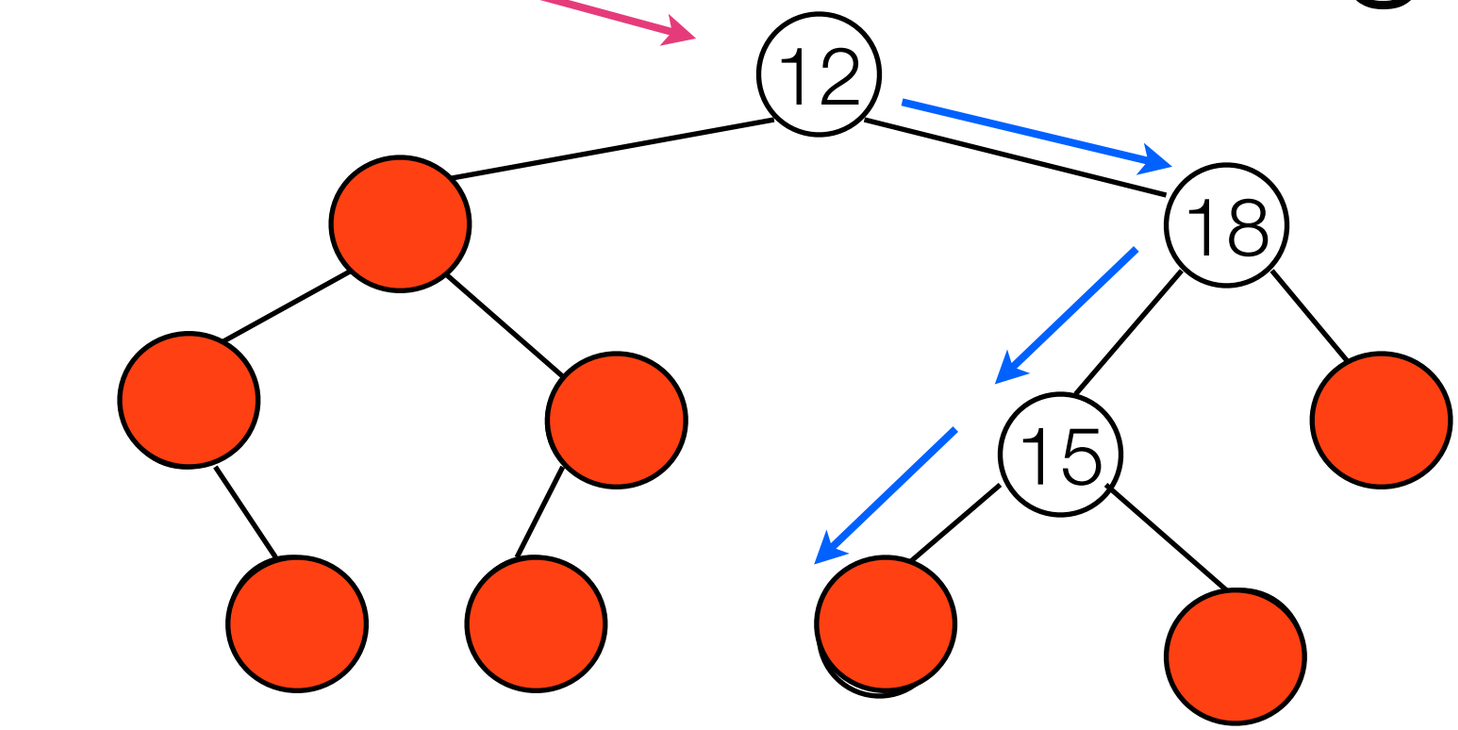
target = 13



Searching

root

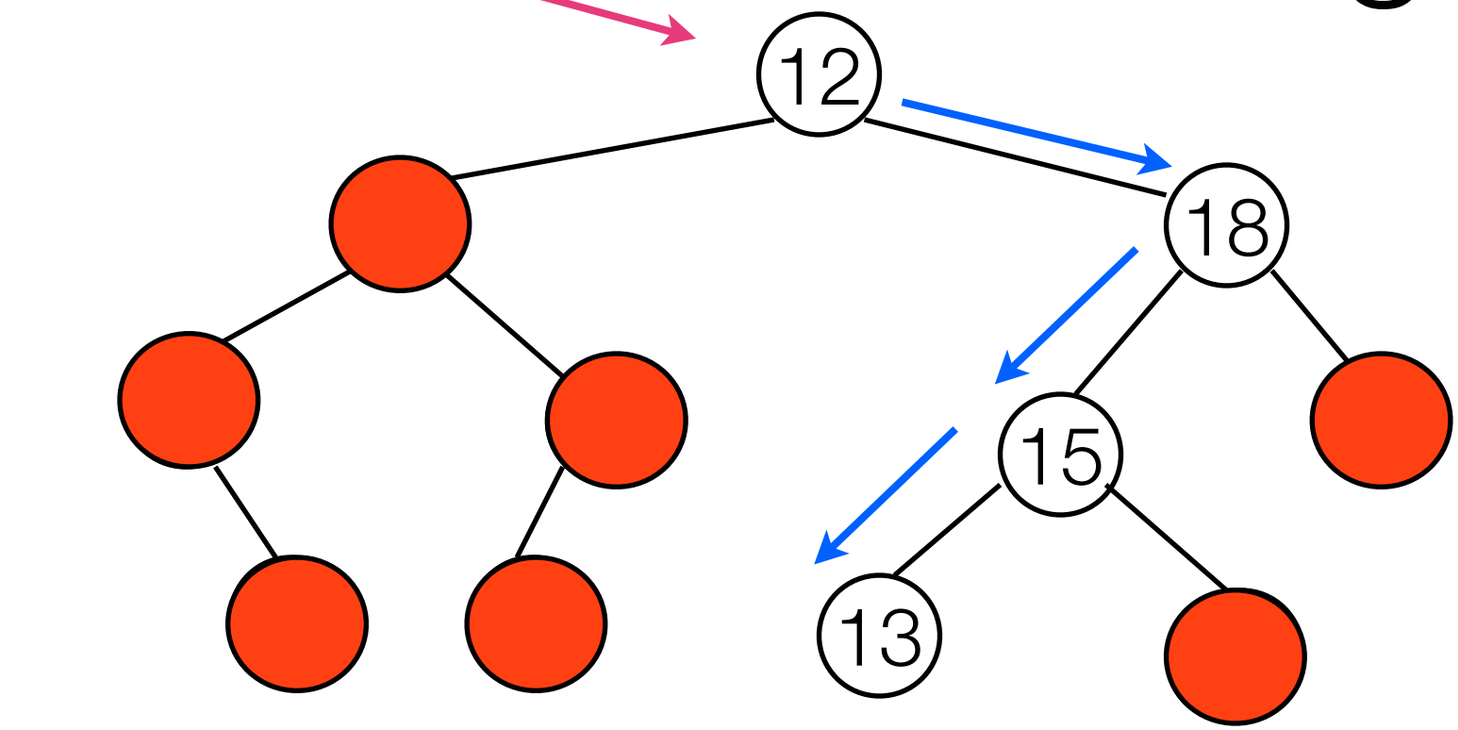
target = 13



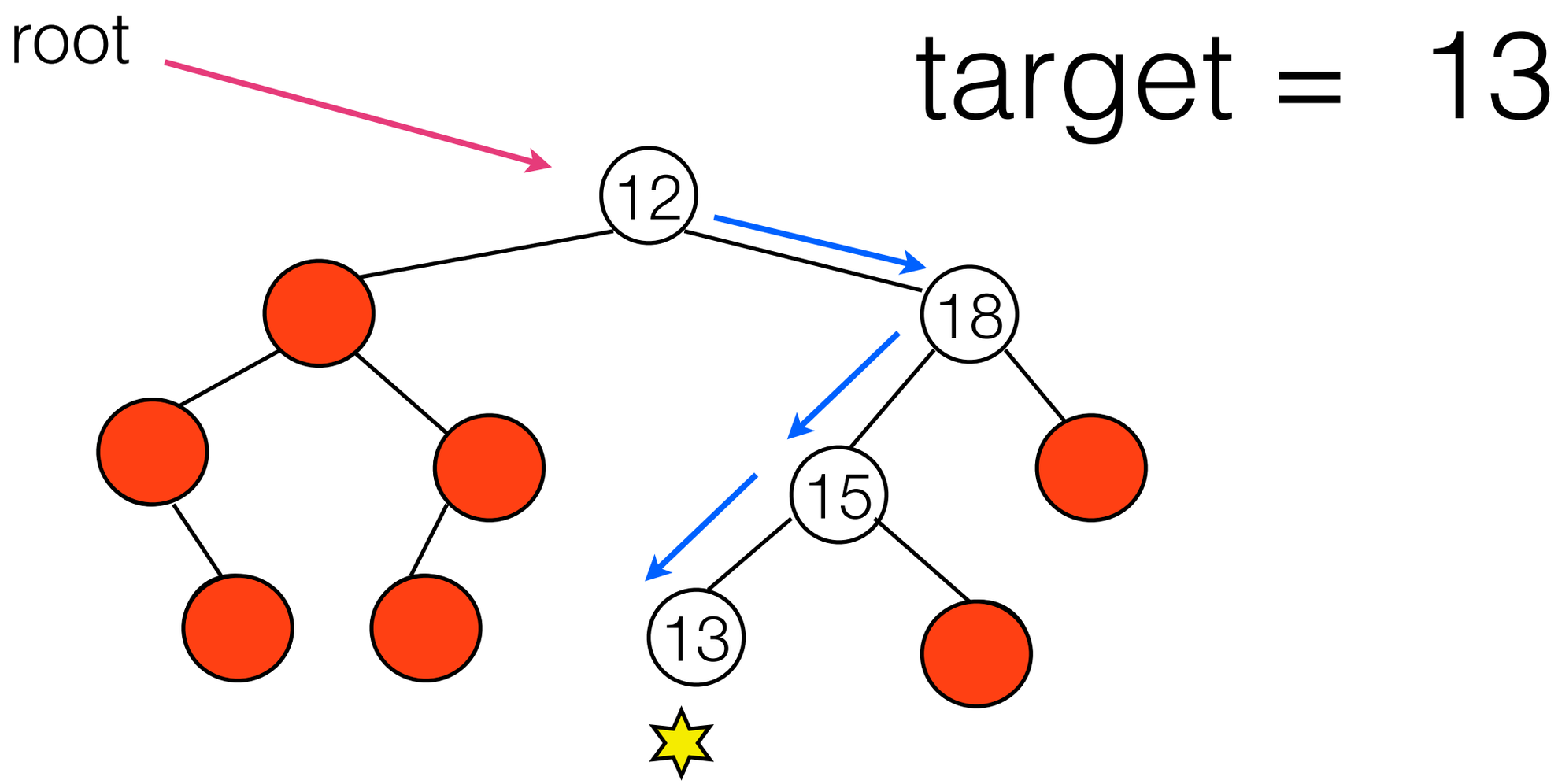
Searching

root

target = 13



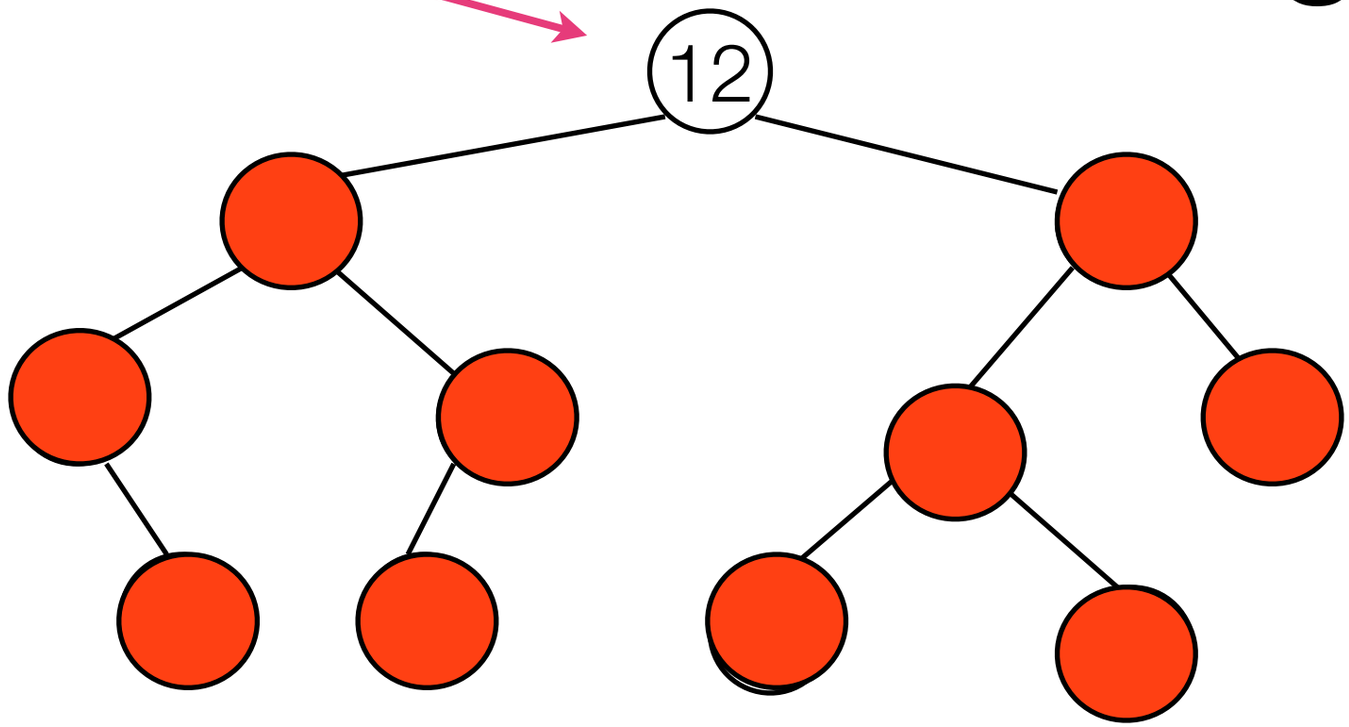
Searching



Searching

root

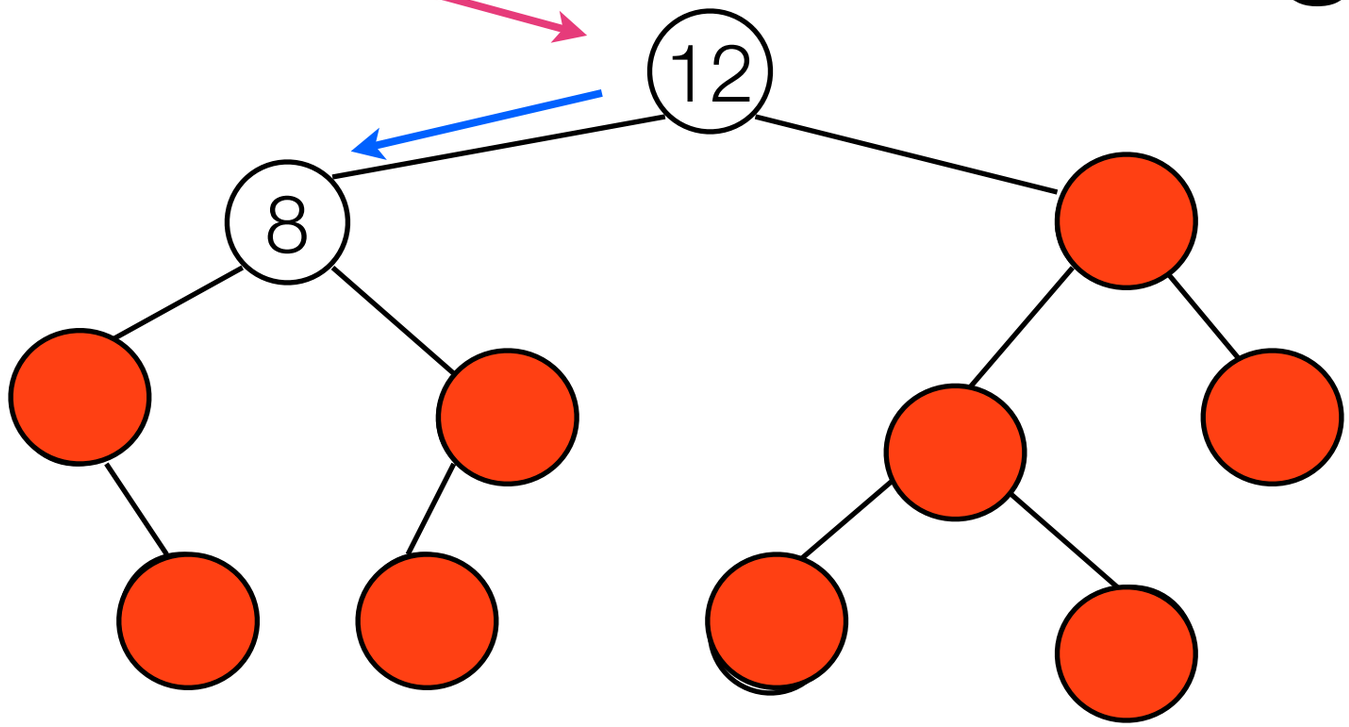
target = 2



Searching

root

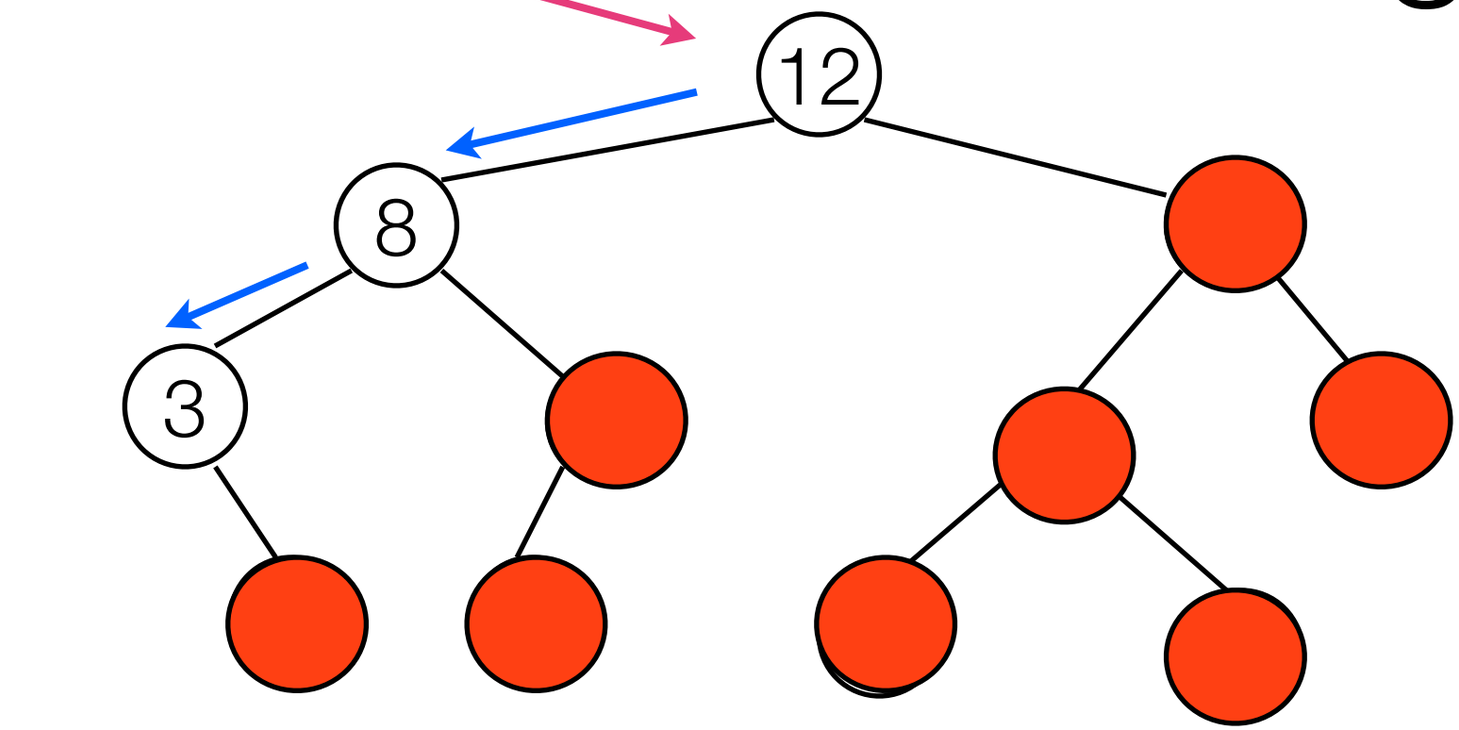
target = 2



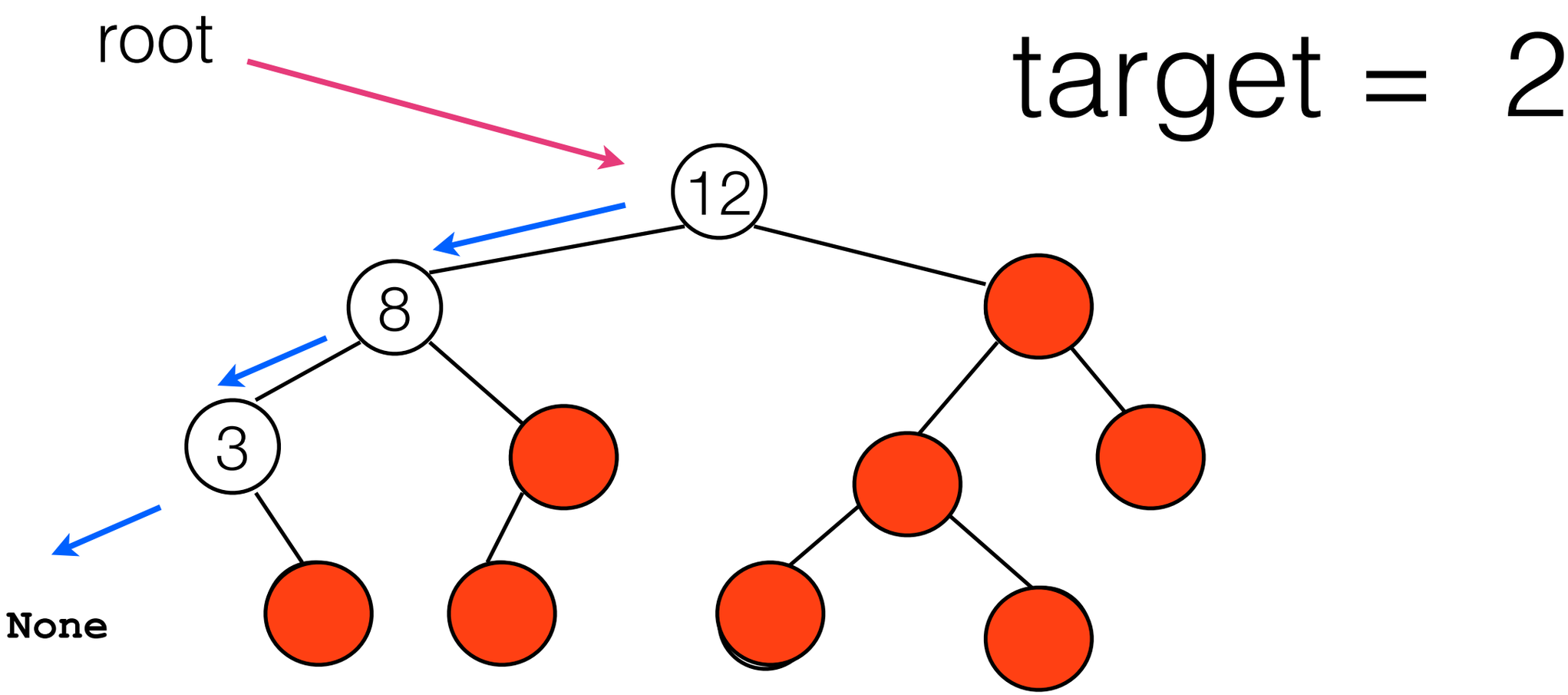
Searching

root

target = 2

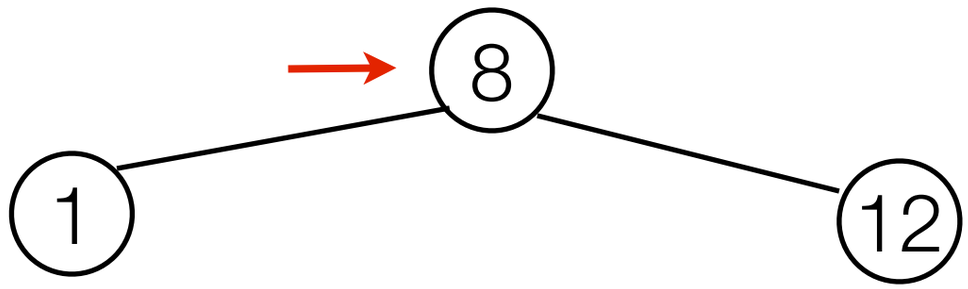


Searching

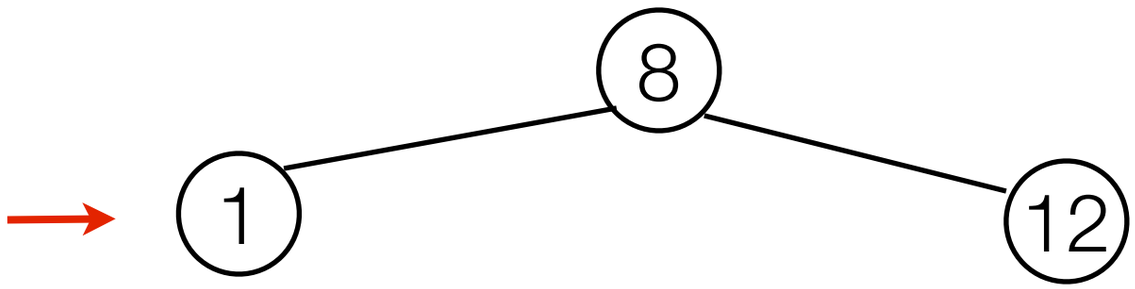


Insert

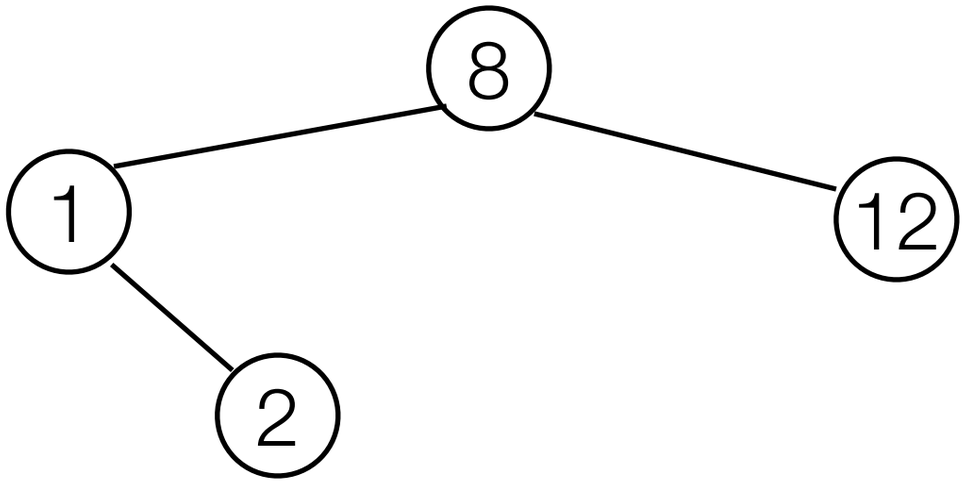
Insert 2



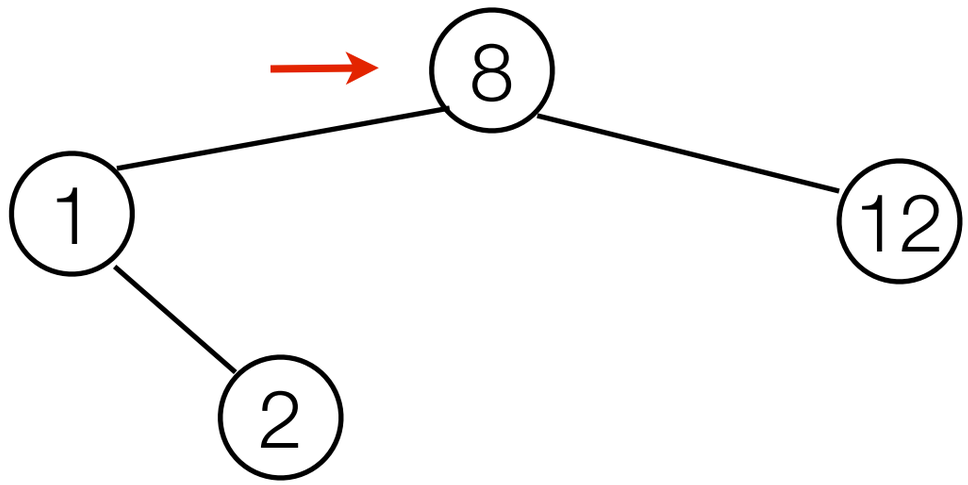
Insert 2



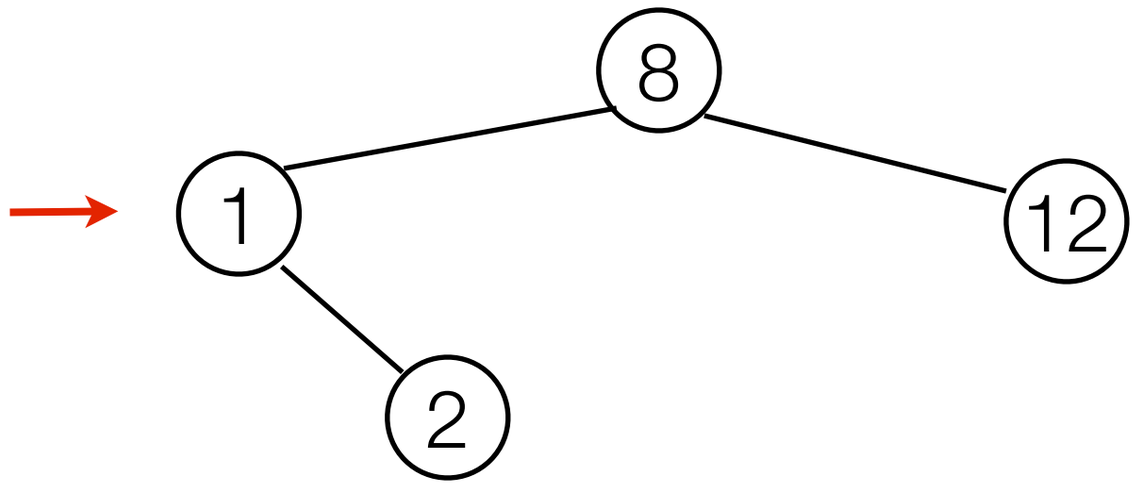
Insert 2



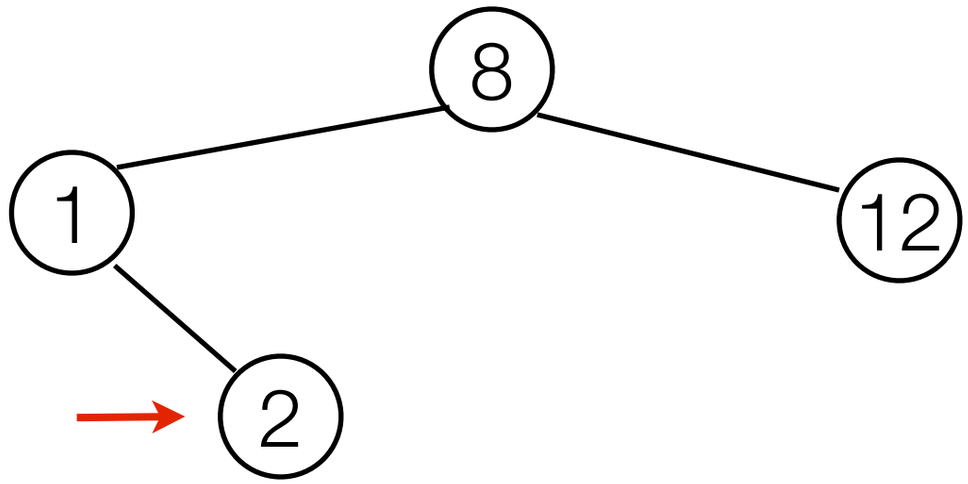
Insert 7



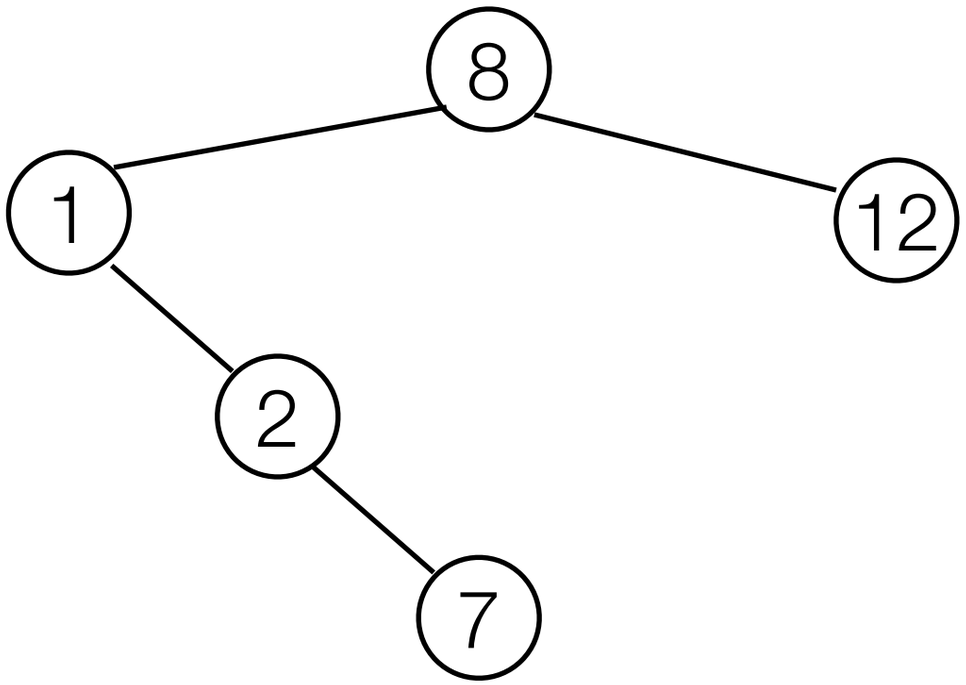
Insert 7



Insert 7



Insert 7



Our BST does not allow for duplicates, so we need to do something if we find the key in the tree…

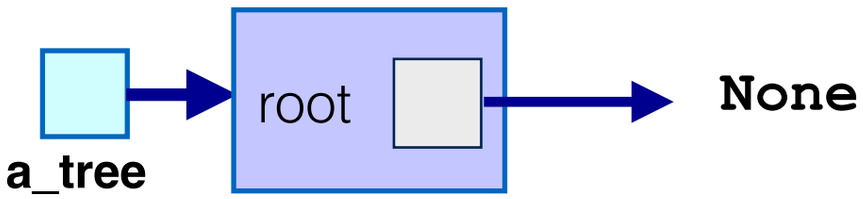
Insert algorithm

Input:
key and associated value to insert.

Idea:
Find the right spot (search) then create new node.

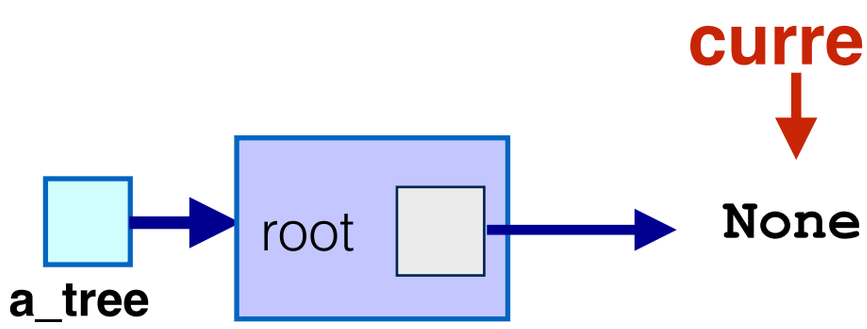
- Try to find the key…
  - **Found**? Raise an exception, keys must be unique….or replace value.
  - **Not found**? parent of **`None`** should be the parent of new node, which needs to be created.

```python
def insert(self, key, item):
    self.root = self._insert_aux(self.root, key, item)


def _insert_aux(self, current, key, item):
     if current is None: # base case: at the leaf
         current = BinarySearchTreeNode(key,item)
     elif key < current.key:
         self._insert_aux(current.left,key,item)
     elif key > current.key:
         self._insert_aux(current.right,key,item)
     else: # key == current.key
         raise ValueError("Duplicate Item")
```



a_tree.insert(57, “Coco”)

```
def insert(self, key, item):
    self.root = self._insert_aux(self.root, key, item)


def _insert_aux(self, current, key, item):
     if current is None: # base case: at the leaf
         current = BinarySearchTreeNode(key,item)
     elif key < current.key:
         self._insert_aux(current.left,key,item)
     elif key > current.key:
         self._insert_aux(current.right,key,item)
     else: # key == current.key
         raise ValueError("Duplicate Item")
```

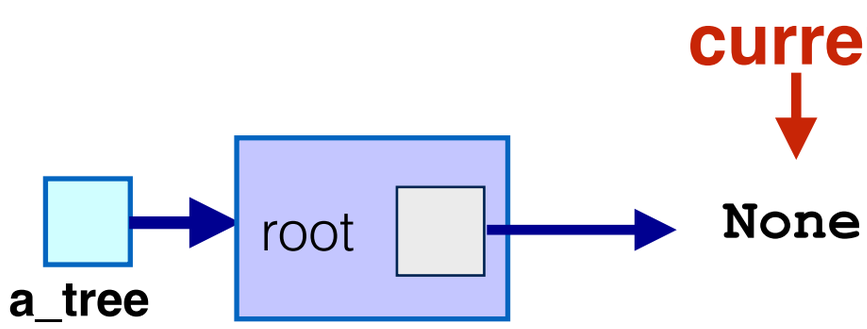


key → 57

item → “Coco”

```
a_tree.insert(57, “Coco”)
```

```python
def insert(self, key, item):
    self.root = self._insert_aux(self.root, key, item)


def _insert_aux(self, current, key, item):
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key,item)
    elif key < current.key:
        self._insert_aux(current.left,key,item)
    elif key > current.key:
        self._insert_aux(current.right,key,item)
    else: # key == current.key
        raise ValueError("Duplicate Item")
```

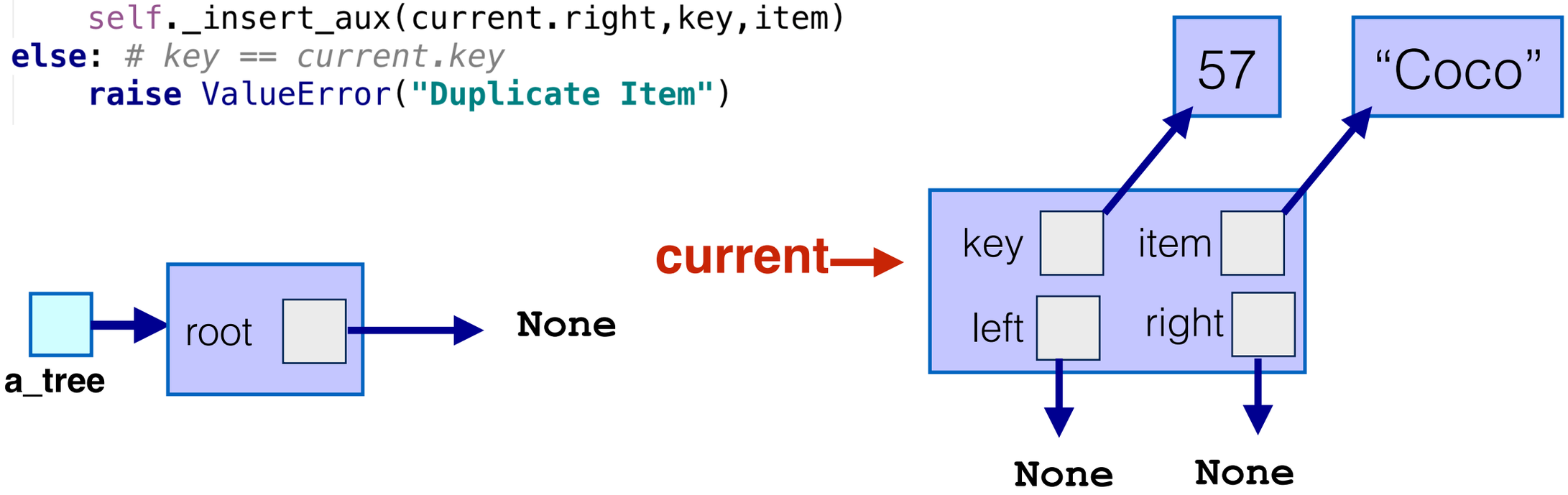


key → 57

item → “Coco”

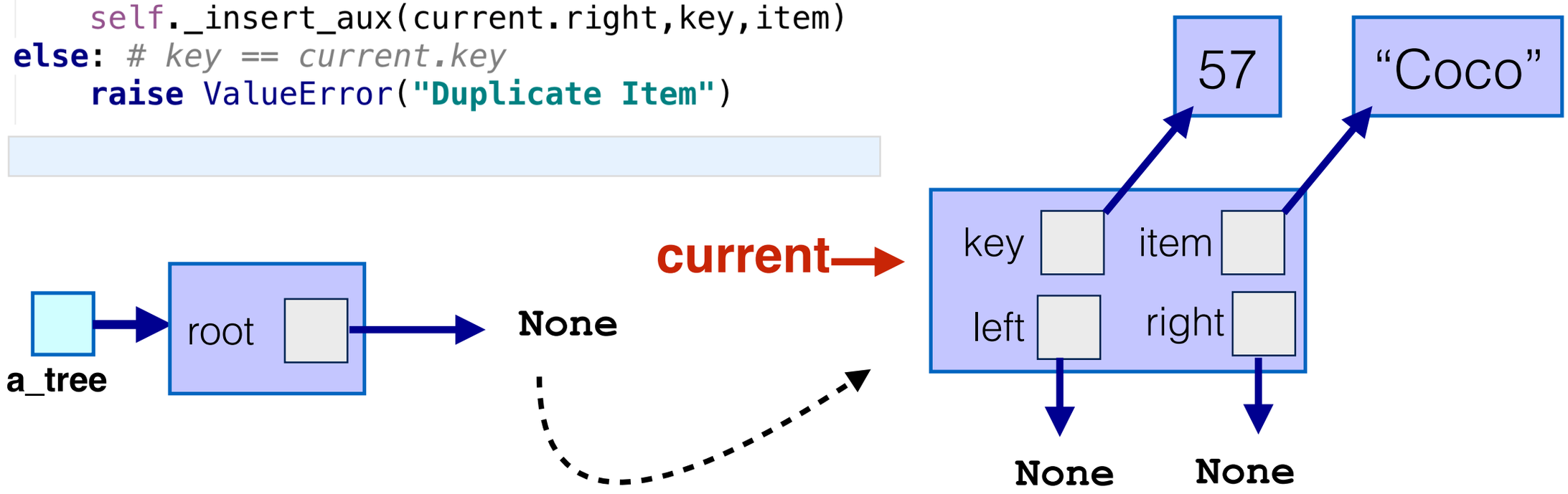
```
a_tree.insert(57, “Coco”)
```

```
def insert(self, key, item):
    self.root = self._insert_aux(self.root, key, item)


def _insert_aux(self, current, key, item):
     if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key,item)
     elif key < current.key:
        self._insert_aux(current.left,key,item)
     elif key > current.key:
        self._insert_aux(current.right,key,item)
     else: # key == current.key
        raise ValueError("Duplicate Item")
```




```
a_tree.insert(57, “Coco”)
```

```python
def insert(self, key, item):
    self.root = self._insert_aux(self.root, key, item)


def _insert_aux(self, current, key, item):
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key,item)
    elif key < current.key:
        self._insert_aux(current.left,key,item)
    elif key > current.key:
        self._insert_aux(current.right,key,item)
    else: # key == current.key
        raise ValueError("Duplicate Item")
```



missing link!

current needs to be returned!

```
def insert(self, key, item):
    self.root = self._insert_aux(self.root, key, item)


def _insert_aux(self, current, key, item):
     if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key,item)
    elif key < current.key:
        current.left = self._insert_aux(current.left,key,item)
    elif key > current.key:
        current.right = self._insert_aux(current.right,key,item)
    else: # key == current.key
        raise ValueError("Duplicate Item")
    return current
```

```
def insert(self, key, item):
    self.root = self._insert_aux(self.root, key, item)


def _insert_aux(self, current, key, item):
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key,item)
    elif key < current.key:
        current.left = self._insert_aux(current.left,key,item)
    elif key > current.key:
        current.right = self._insert_aux(current.right,key,item)
    else: # key == current.key
        raise ValueError("Duplicate Item")
    return current
```

__setitem__

```
def __setitem__(self, key, item):
    self.root = self._setitem_aux_(self.root, key, item)


def _setitem_aux_(self, current, key, item):
    if current is None: # base case: at the leaf
        current = BinarySearchTreeNode(key,item)
    elif key < current.key:
        current.left = self._setitem_aux_(current.left,key,item)
    elif key > current.key:
        current.right = self._setitem_aux_(current.right,key,item)
    else: # key == current.key
        current.item = item
    return current
```

Search algorithm

- If we reach an empty node, item is not there… return **`False`**.
- Else, if target key is equal to the current node's key, return **`True`**
- Else, if target key is **less** than current node's key, **search the left sub-tree**
- Else, if target key is **greater** than current node's key, **search the right sub-tree**

search can be implemented by

`__contains__`

and

`__getitem__`

```
def __contains__(self, key):
    return self._contains_aux(key, self.root)

def _contains_aux(self, key, current_node):
    if current_node is None:  # base case
        return False
    elif key == current_node.key:
            return True
    elif key < current_node.key:
        return self._contains_aux(key, current_node.left)
    elif key > current_node.key:
        return self._contains_aux(key, current_node.right)
```

Keys implement “rich comparison”

object.**__lt__**(*self*, *other*)
object.**__le__**(*self*, *other*)
object.**__eq__**(*self*, *other*)
object.**__ne__**(*self*, *other*)
object.**__gt__**(*self*, *other*)
object.**__ge__**(*self*, *other*)

These are the so-called “rich comparison” methods. The correspondence between operator symbols and method names is as follows: `x<y` calls `x.__lt__(y)`, `x<=y` calls `x.__le__(y)`, `x==y` calls `x.__eq__(y)`, `x!=y` calls `x.__ne__(y)`, `x>y` calls `x.__gt__(y)`, and `x>=y` calls `x.__ge__(y)`.

https://docs.python.org/3/reference/datamodel.html

we want to get the item associated to a key…

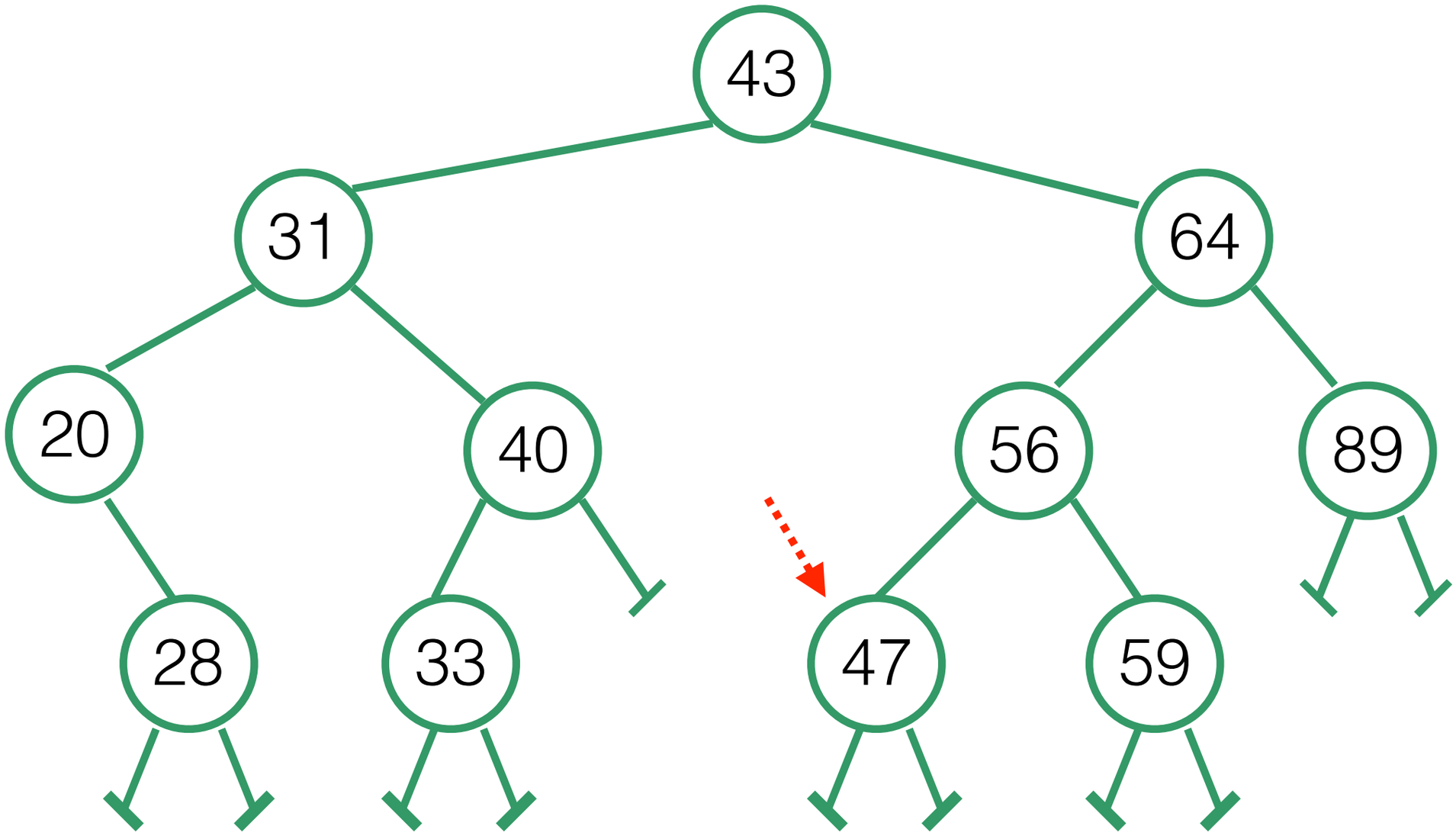
`__getitem__`

__getitem__

```
def __getitem__(self, key):
    return self._getitem_aux(self.root, key)

def _getitem_aux(self, current, key):
if current is None:  # base case: empty
    raise KeyError("Key not found")
elif key == current.key: # base case: found
    return current.item
elif key < current.key:
    return self.getitem_aux(current.left, key)
else:  # key > current.key
 return self.getitem_aux(current.right, key)
```

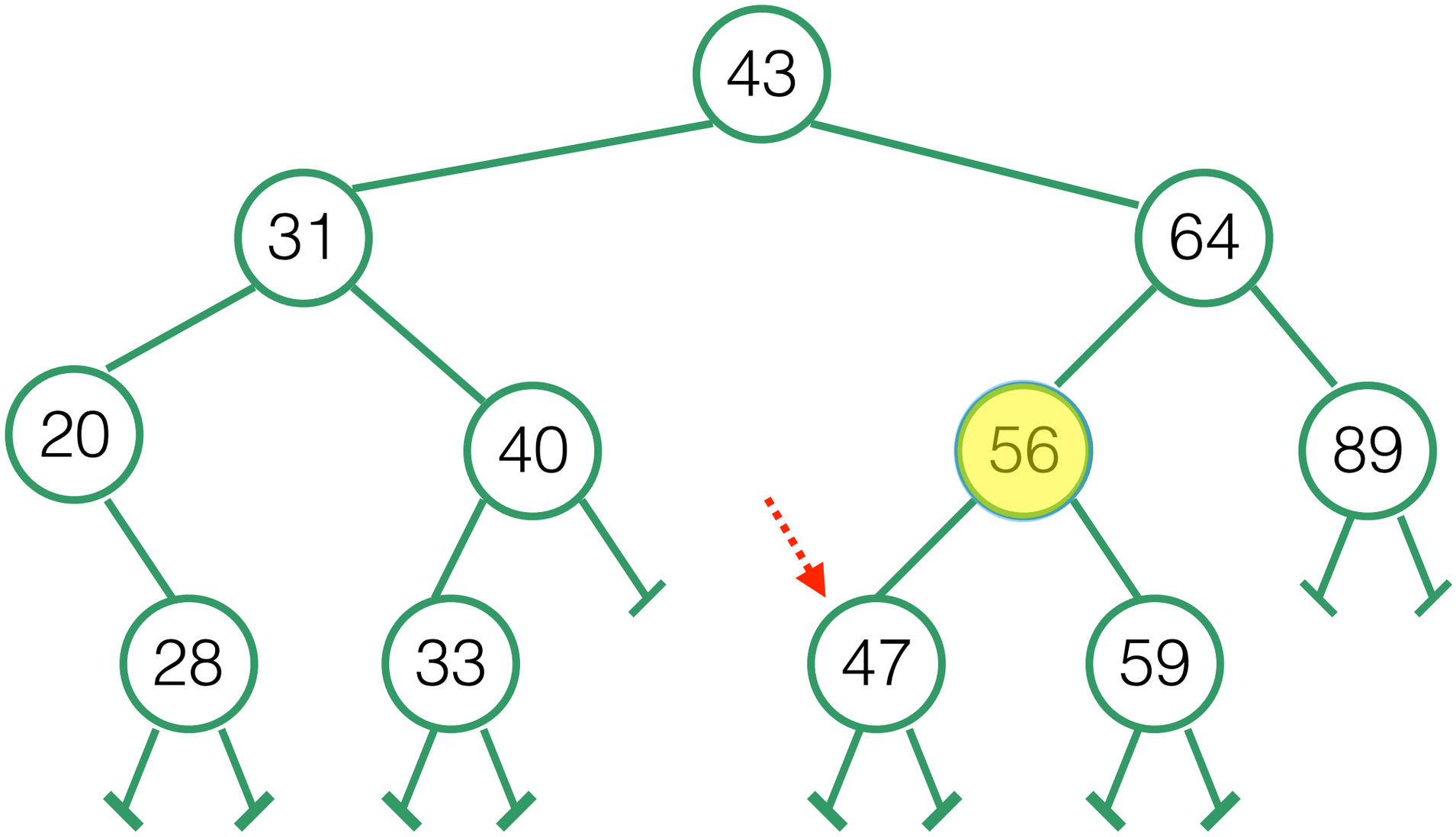
Delete

Delete 47



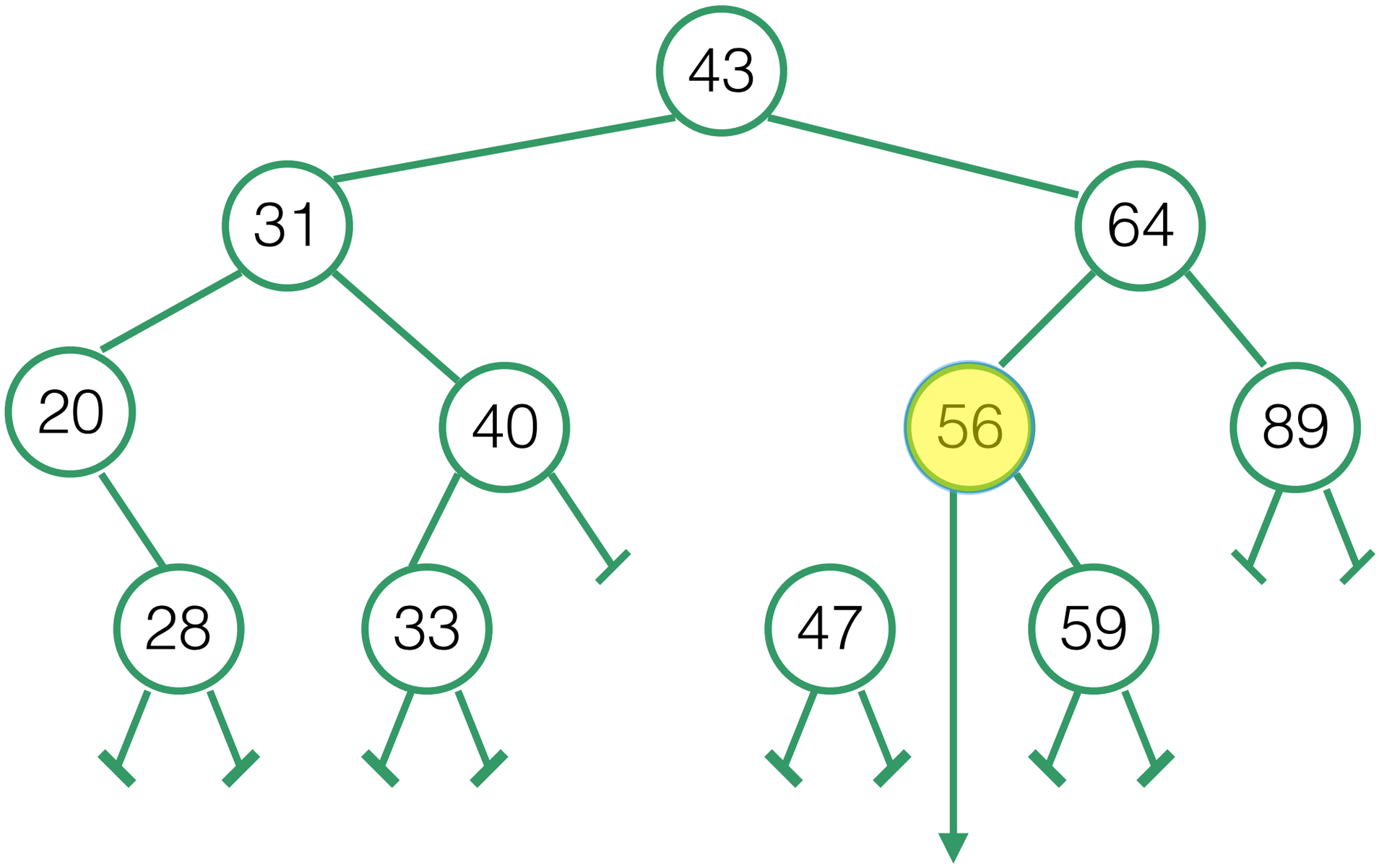
only showing **key**

Delete 47



only showing **key**

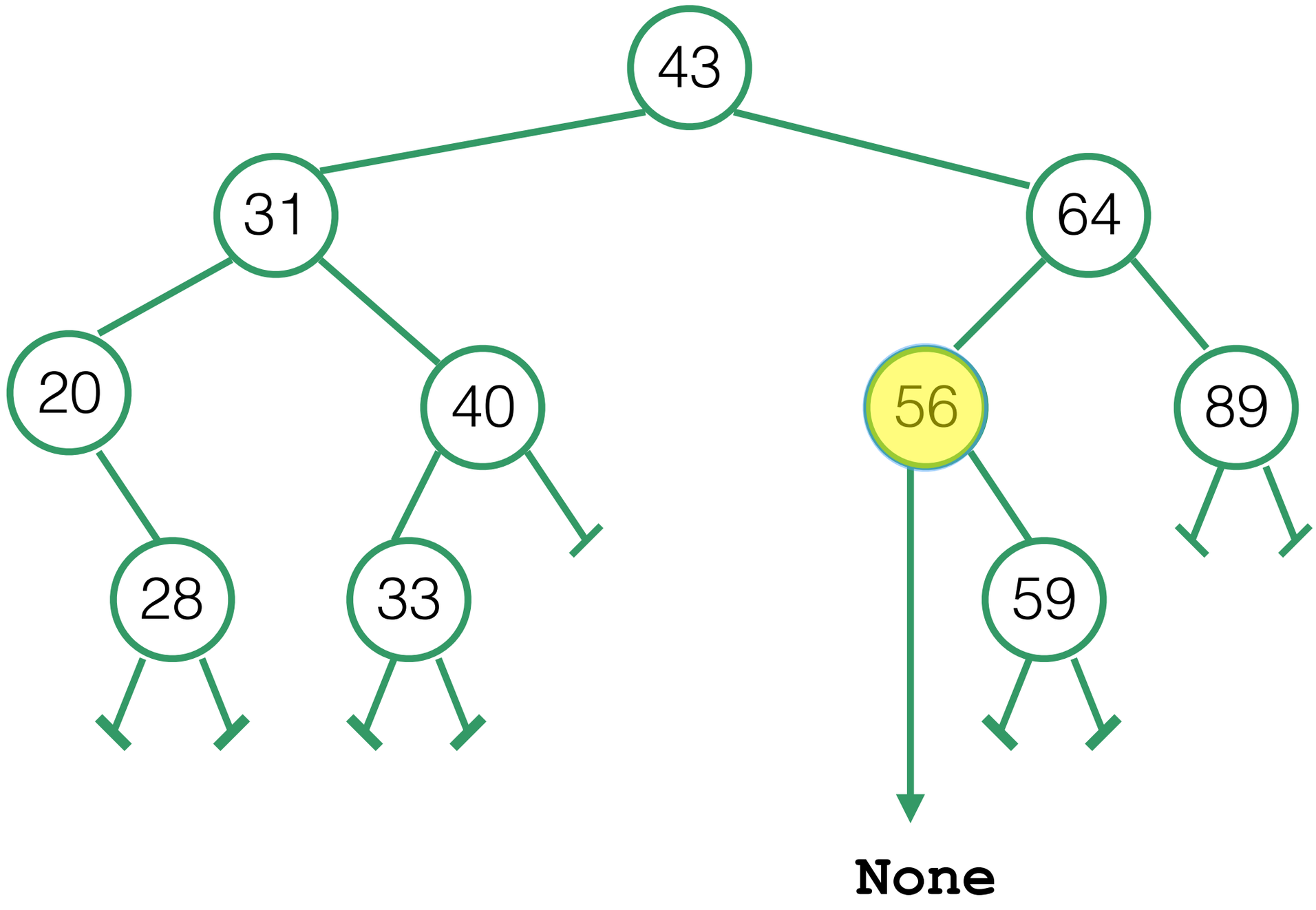
Delete 47



None

only showing **key**

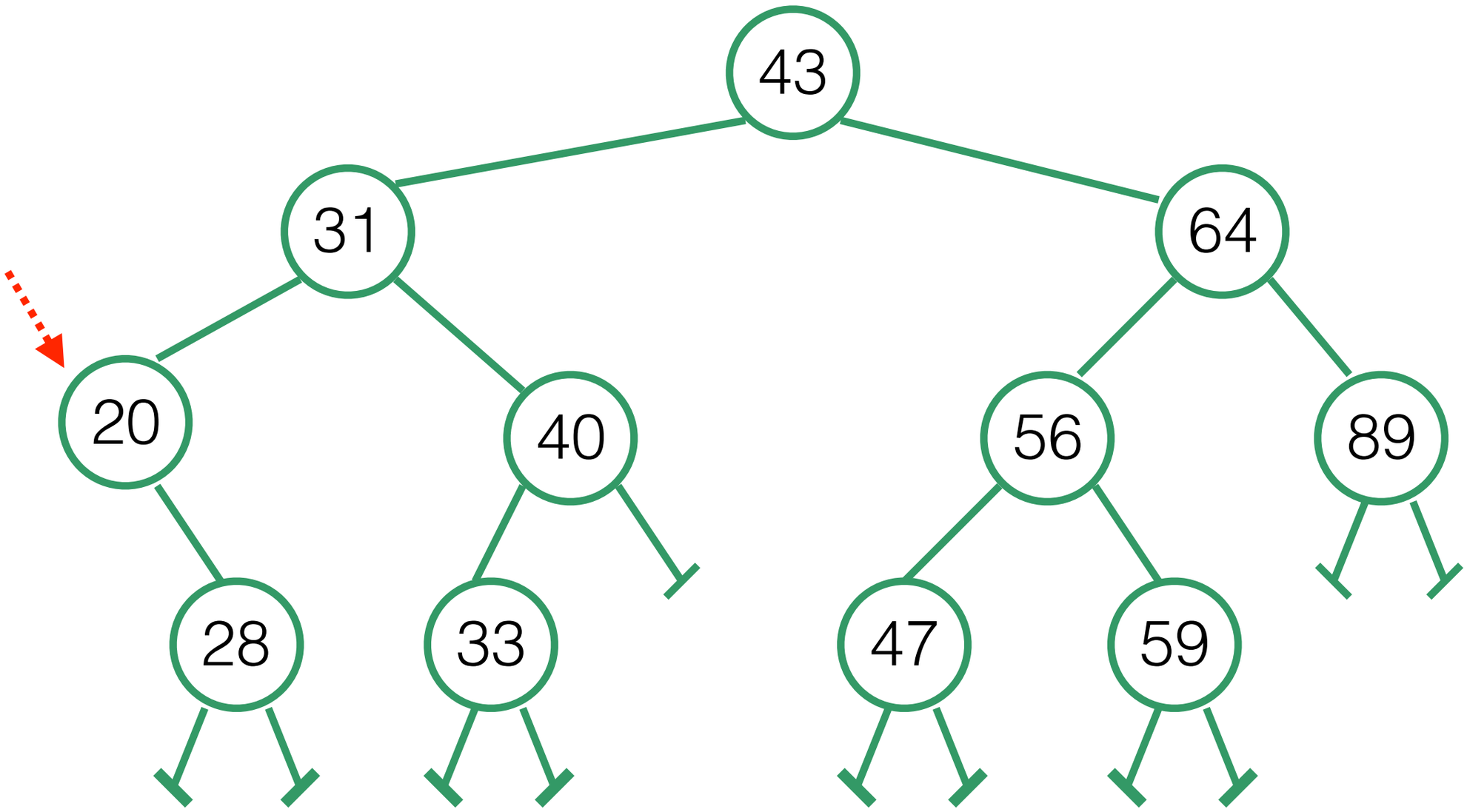
Delete 47



Node with no children:

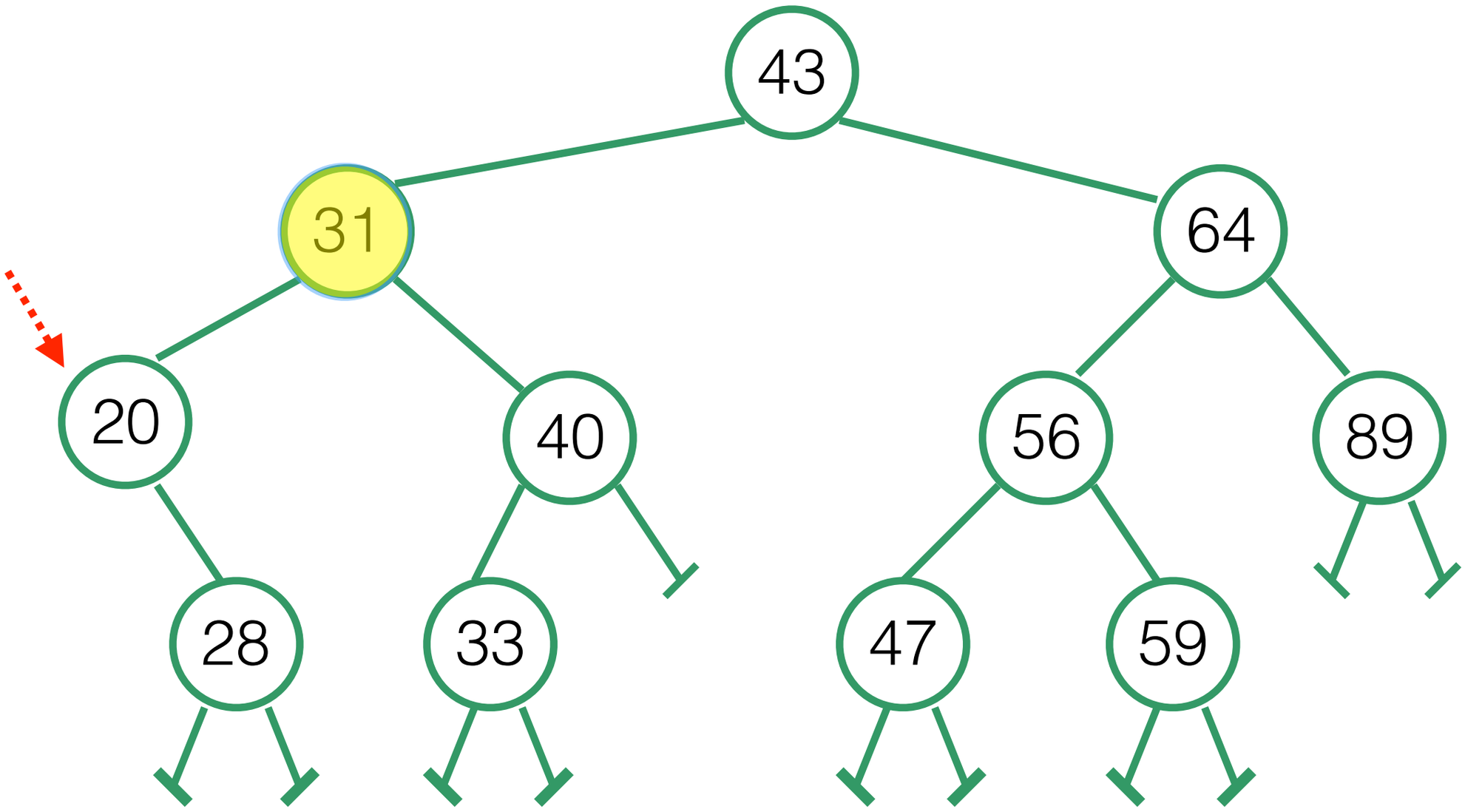
`Find parent - point to None`

Delete 20



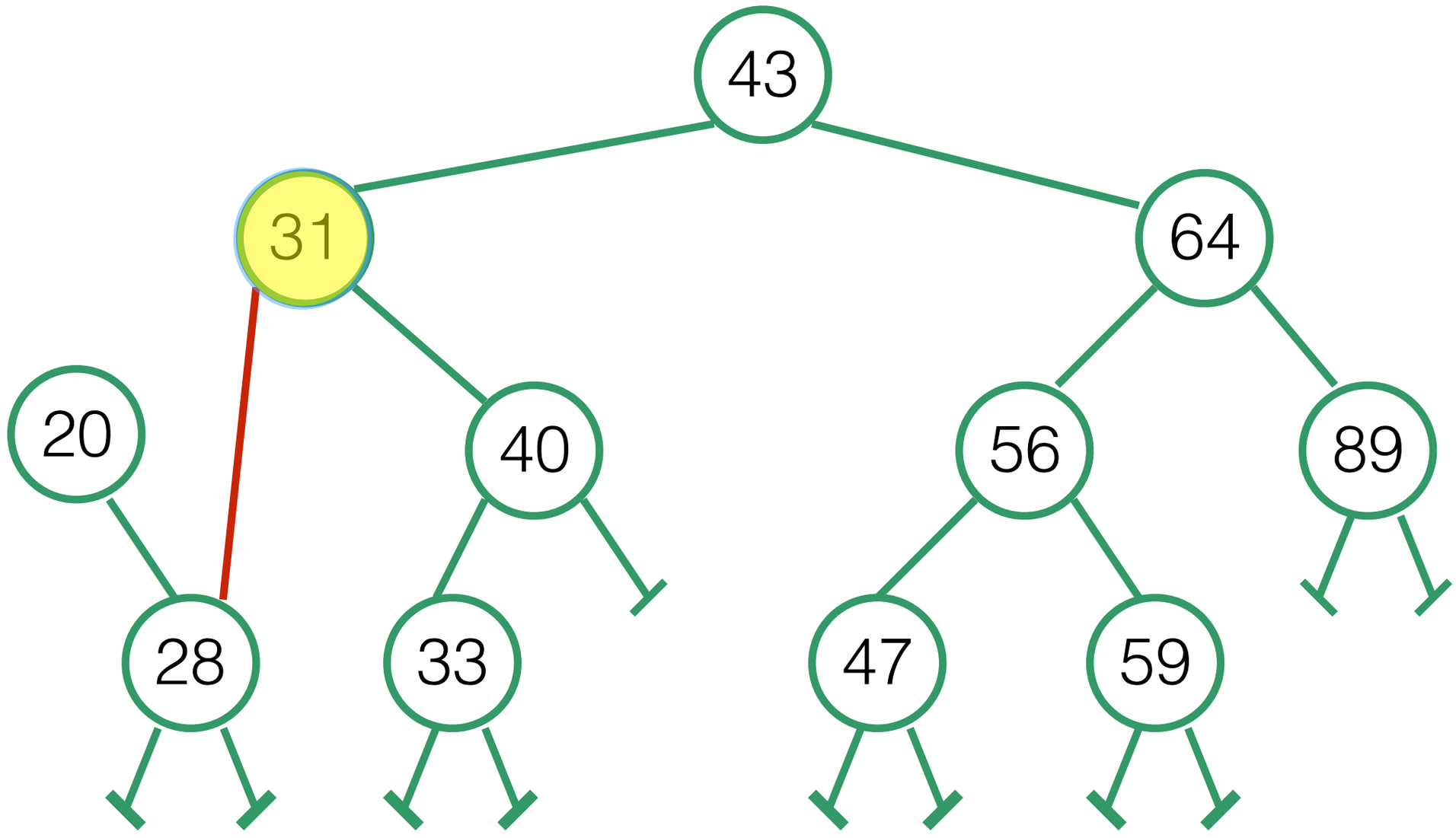
only showing **key**

Delete 20



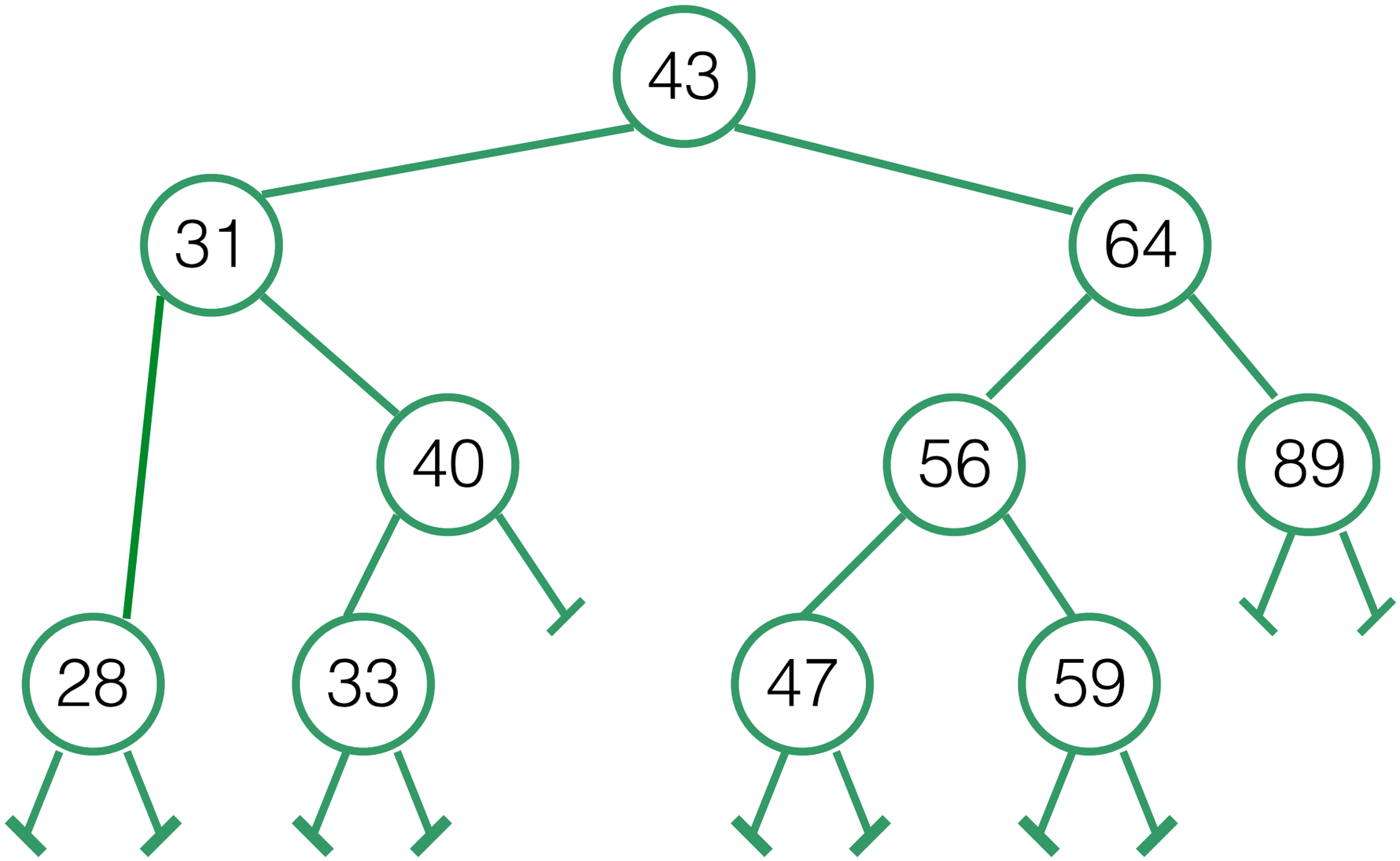
only showing **key**

Delete 20



only showing **key**

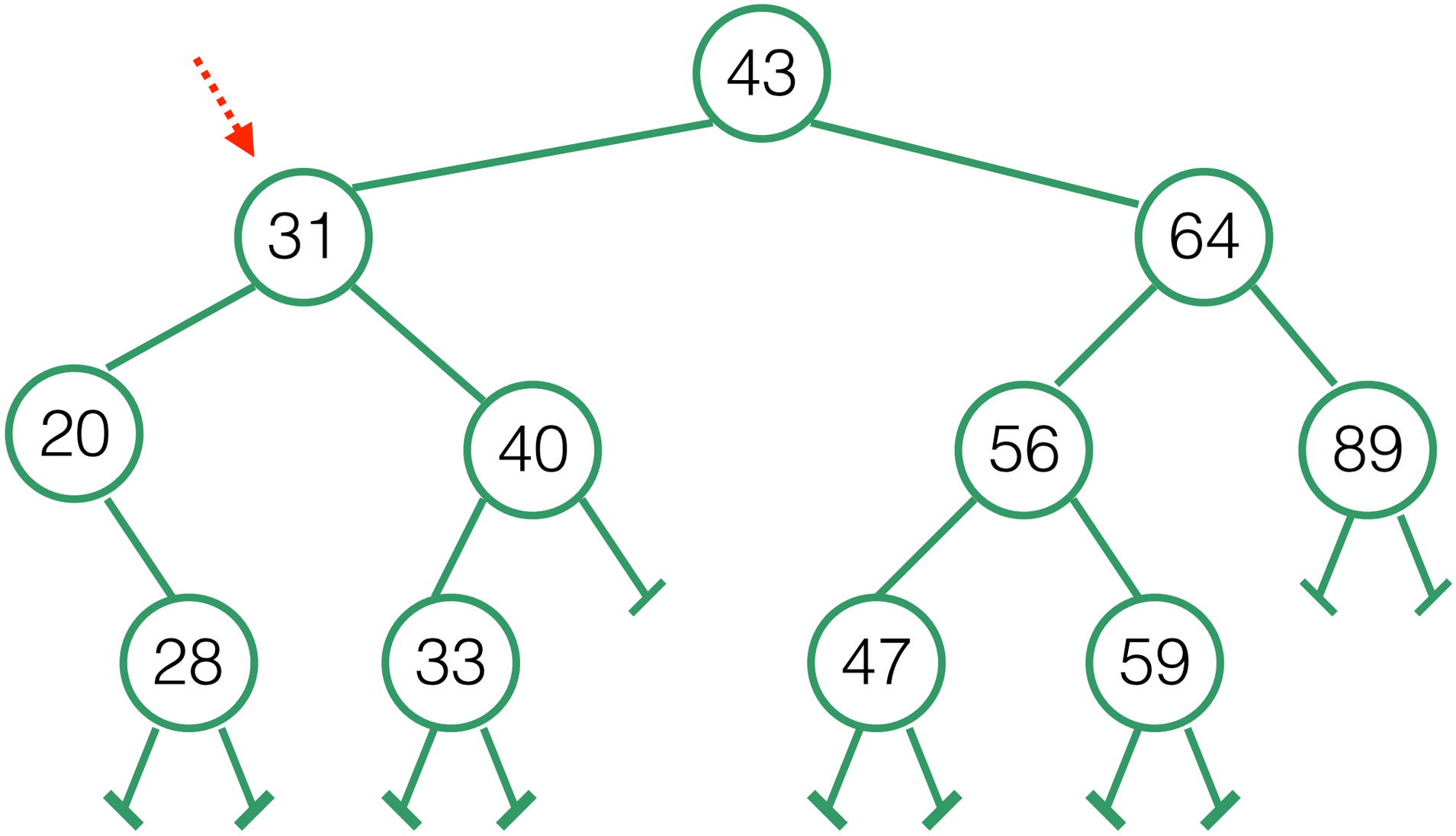
Delete 20



Node with one child:

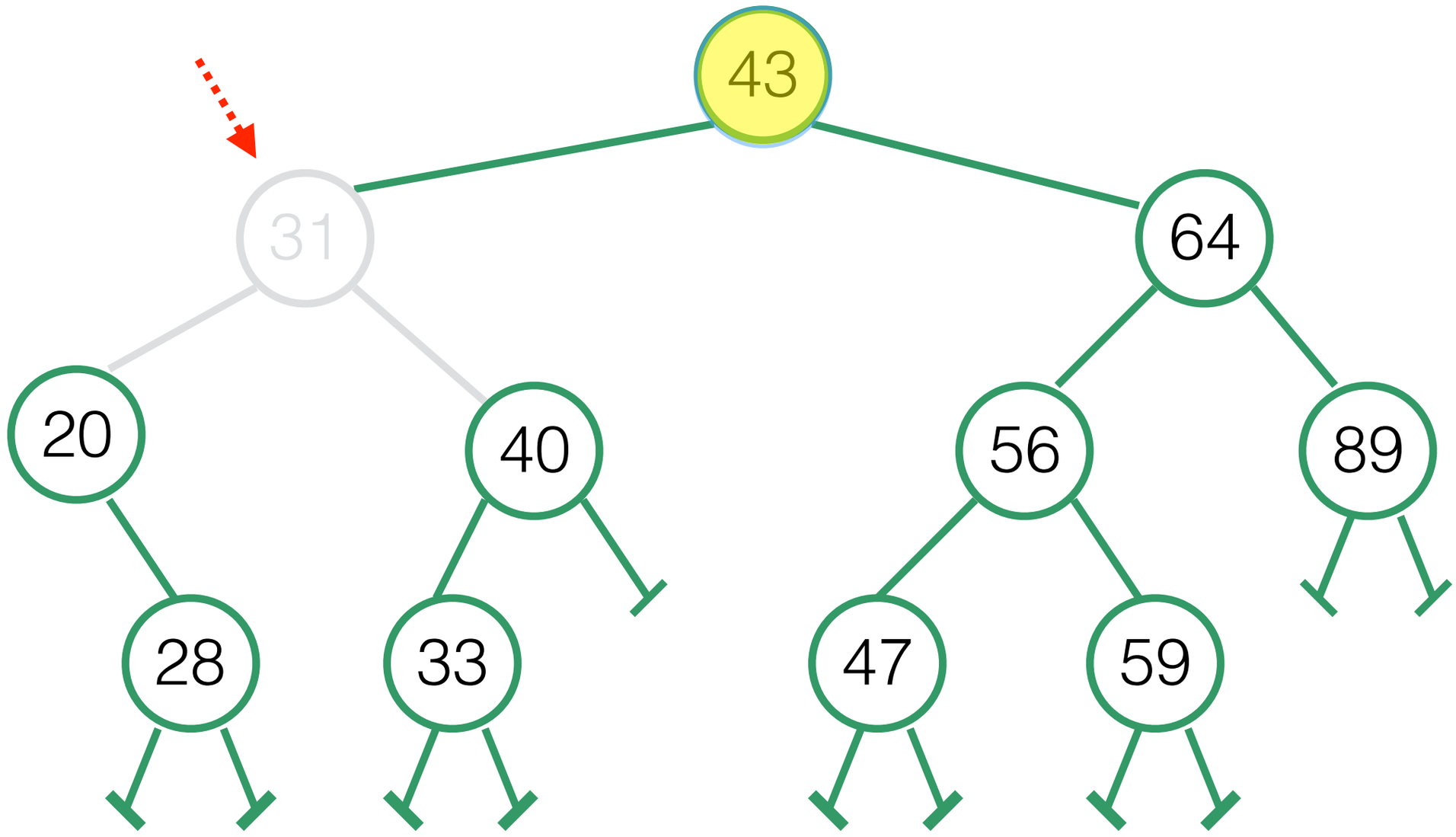
Find parent - point to child of deleted node

Delete 31



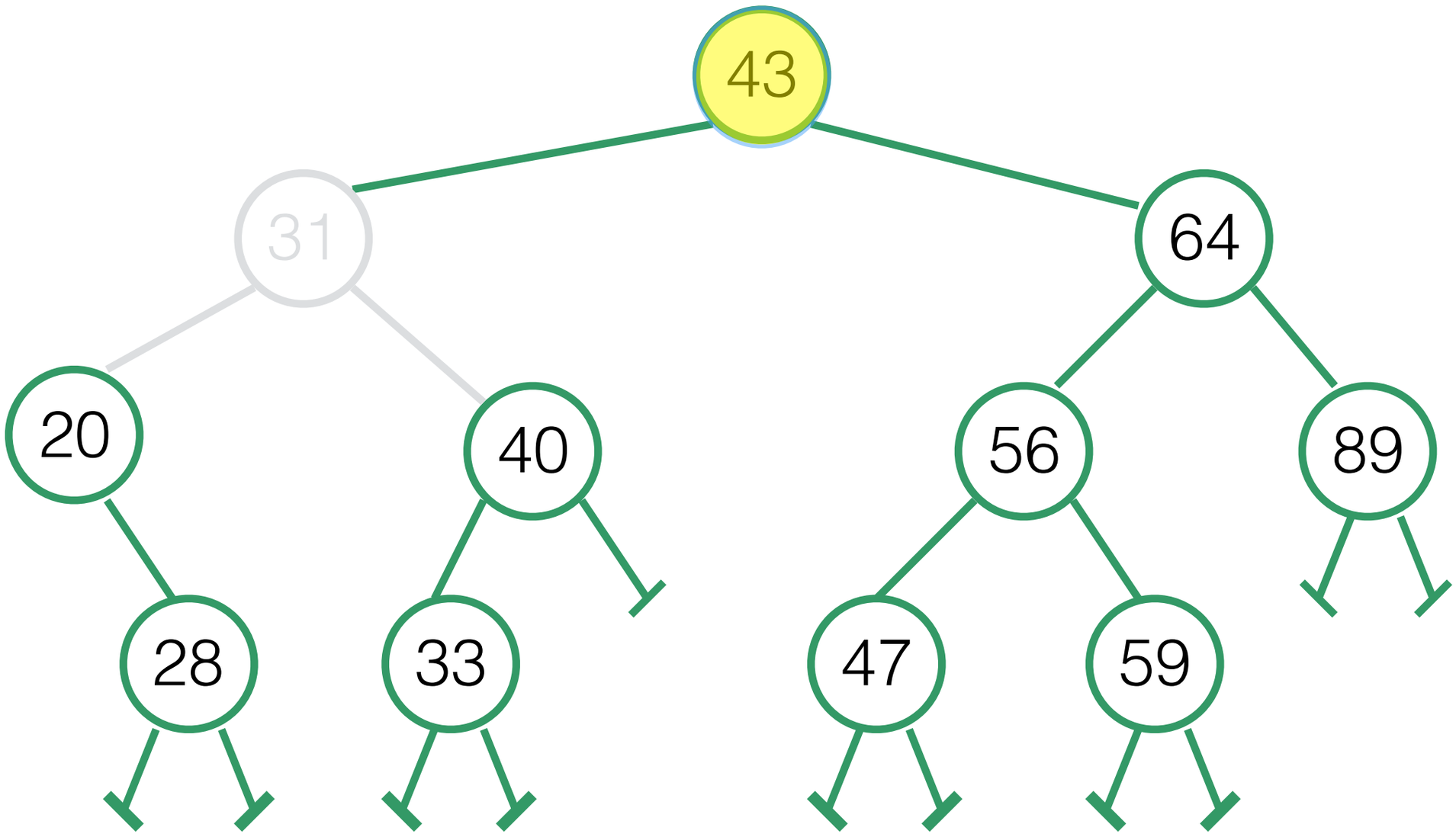
only showing **key**

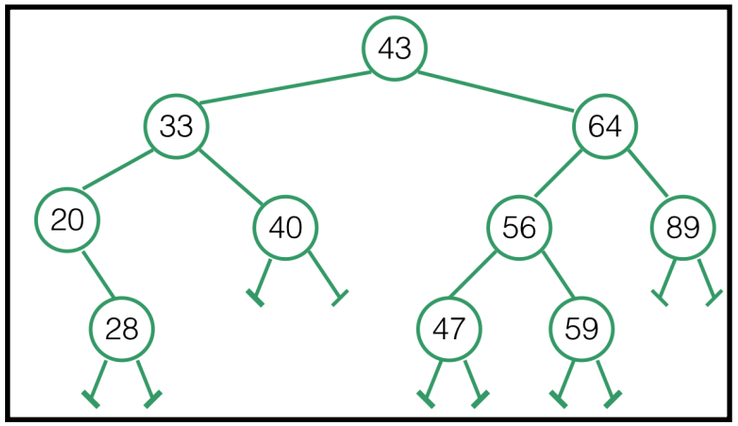
Delete 31



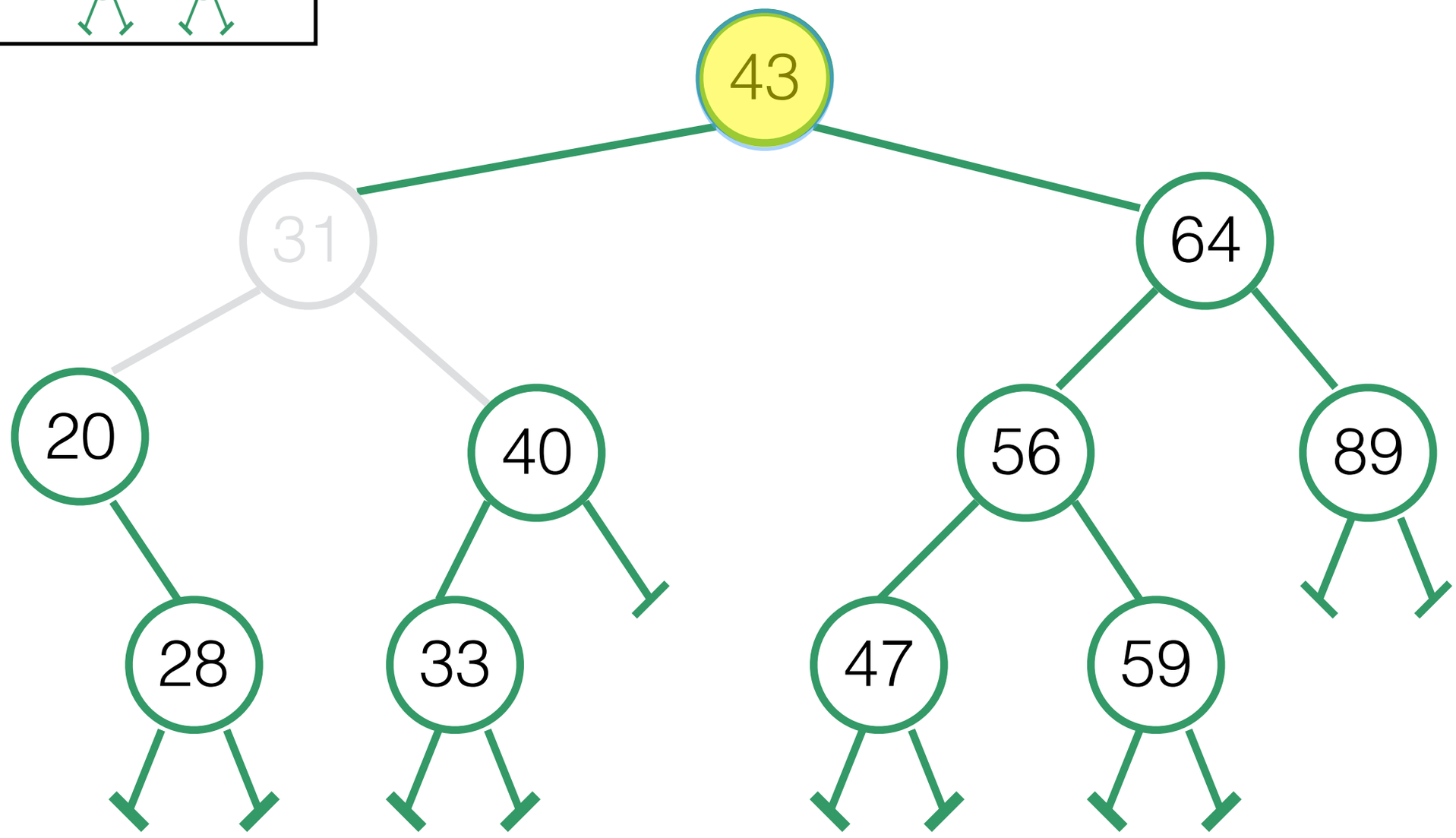
only showing **key**

Delete 31

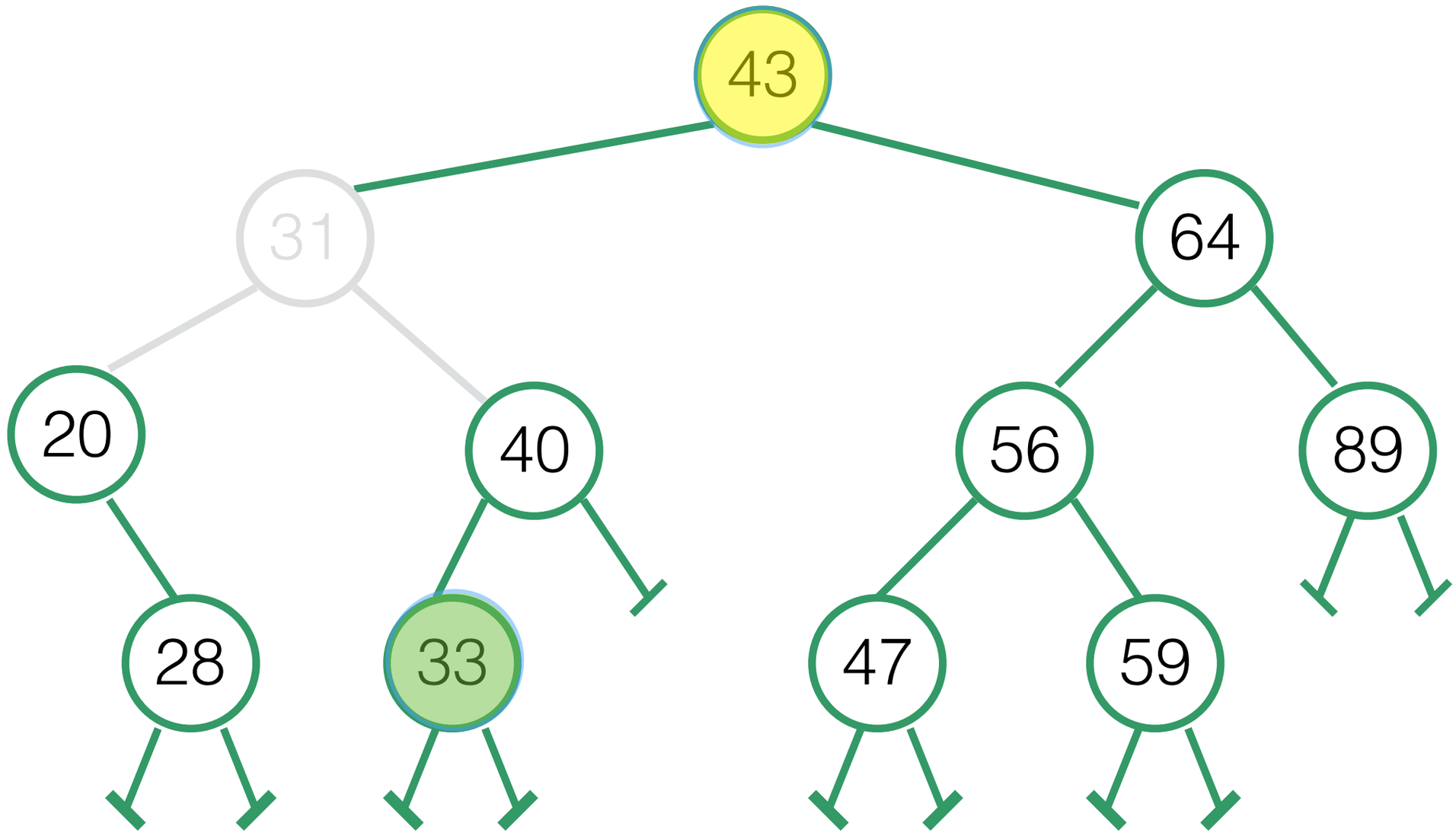




Delete 31

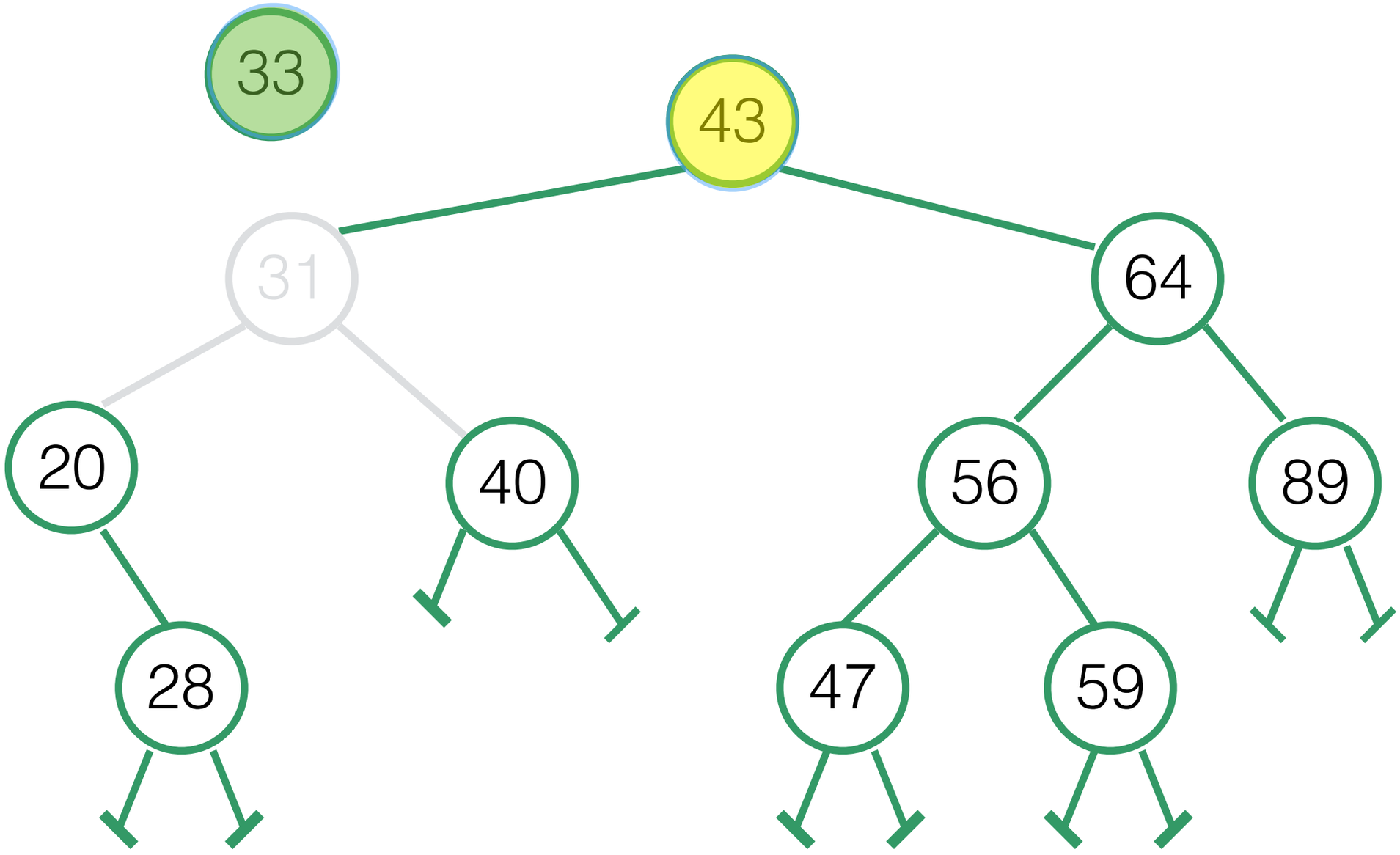


Delete 31

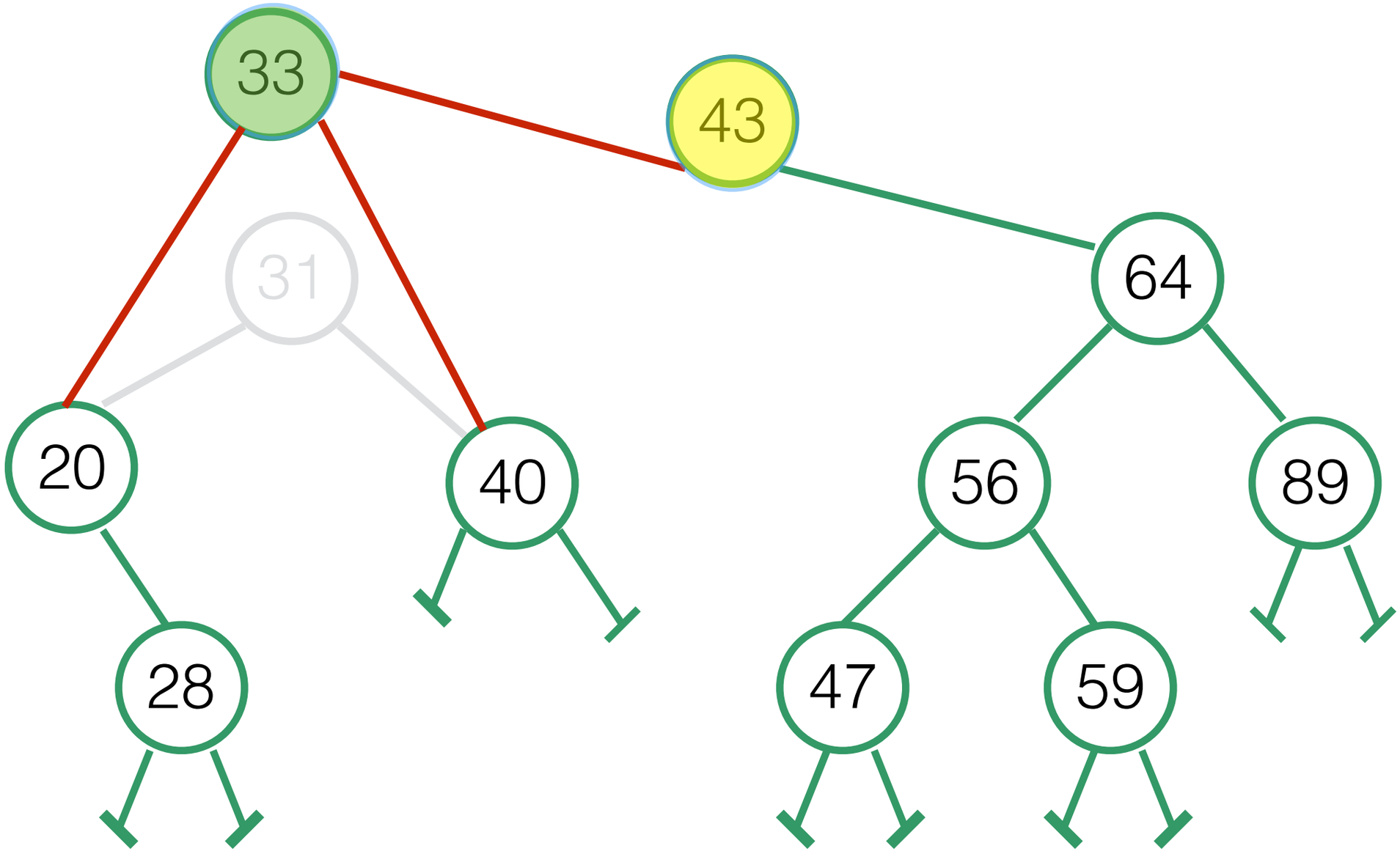


Successor of a node: node with **next** larger key.

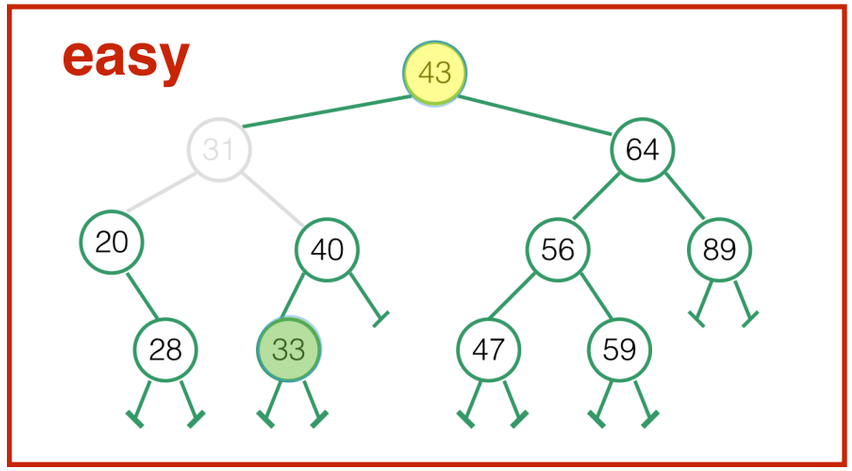
Delete 31

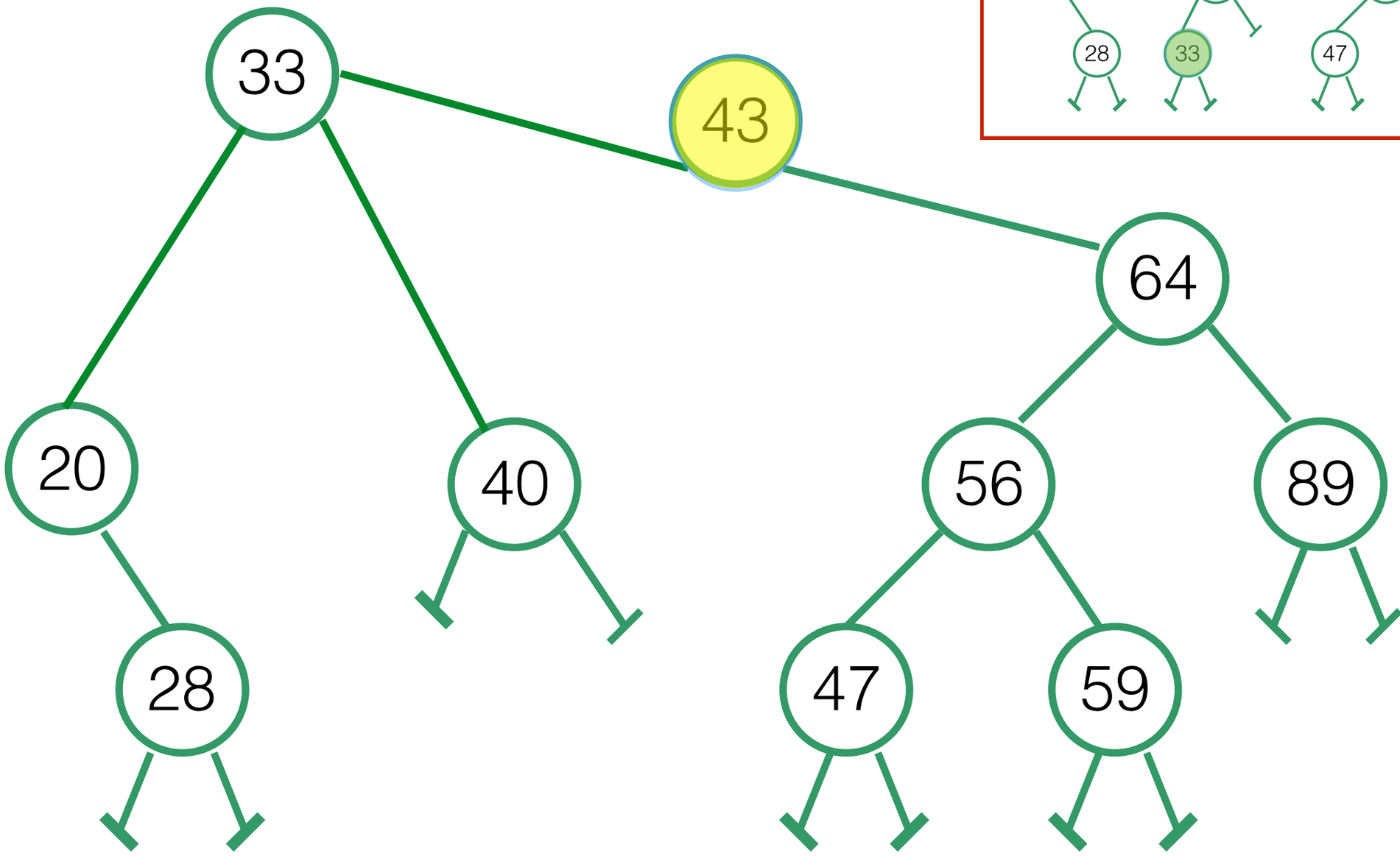


Delete 31



Delete 31

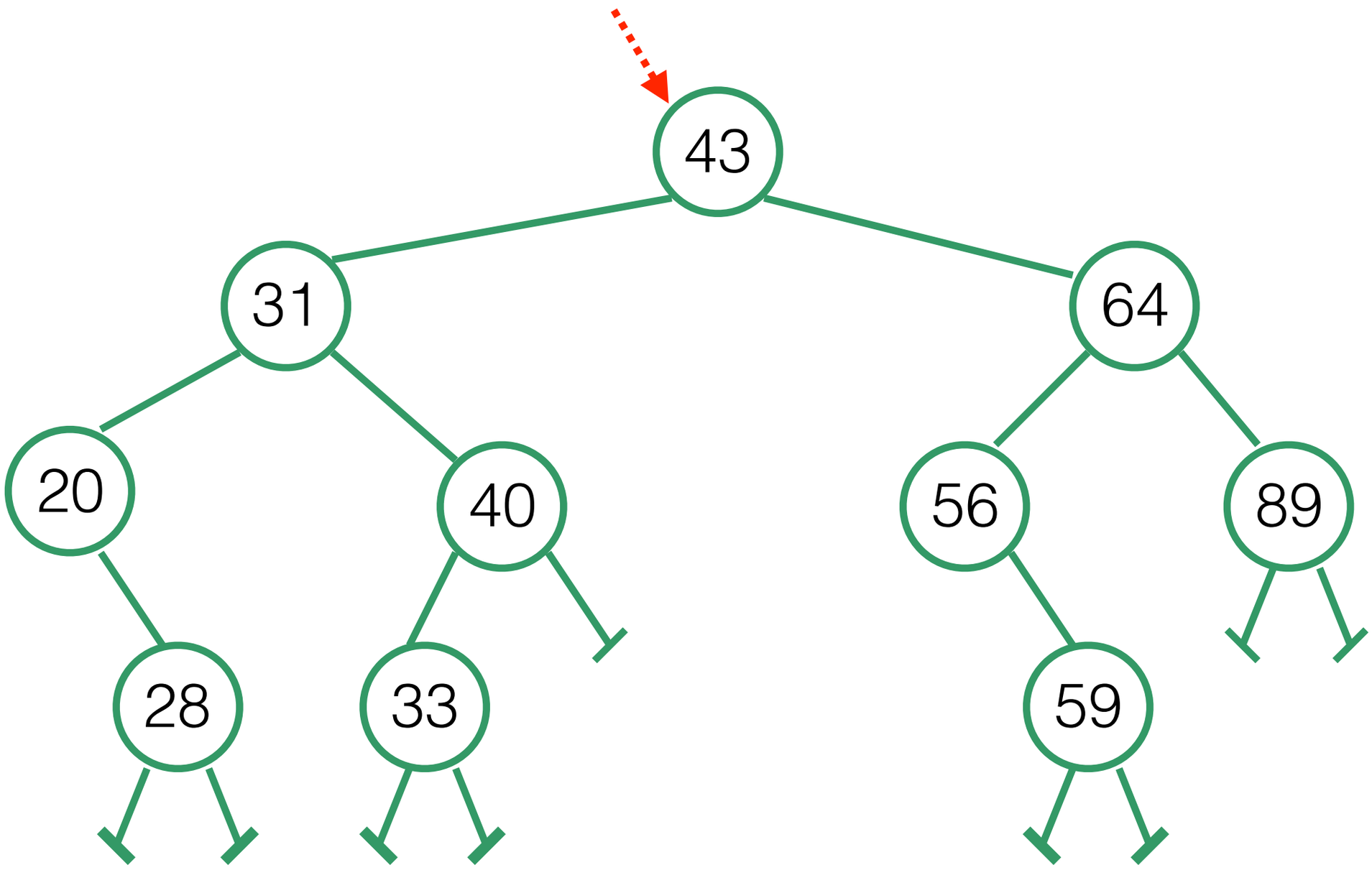




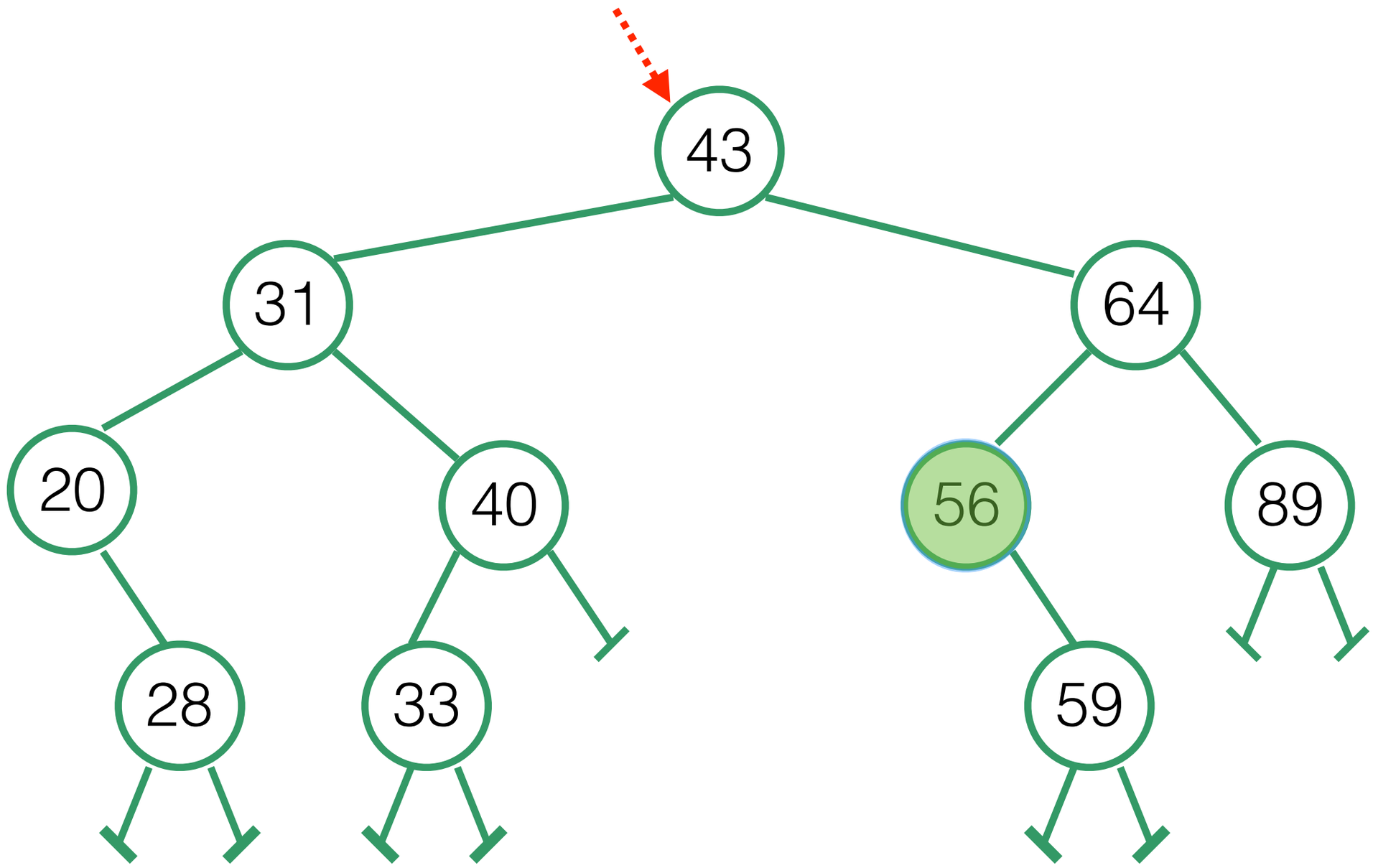
Node with two children:

Find parent and successor - successor is the new parent of the (orphan) children

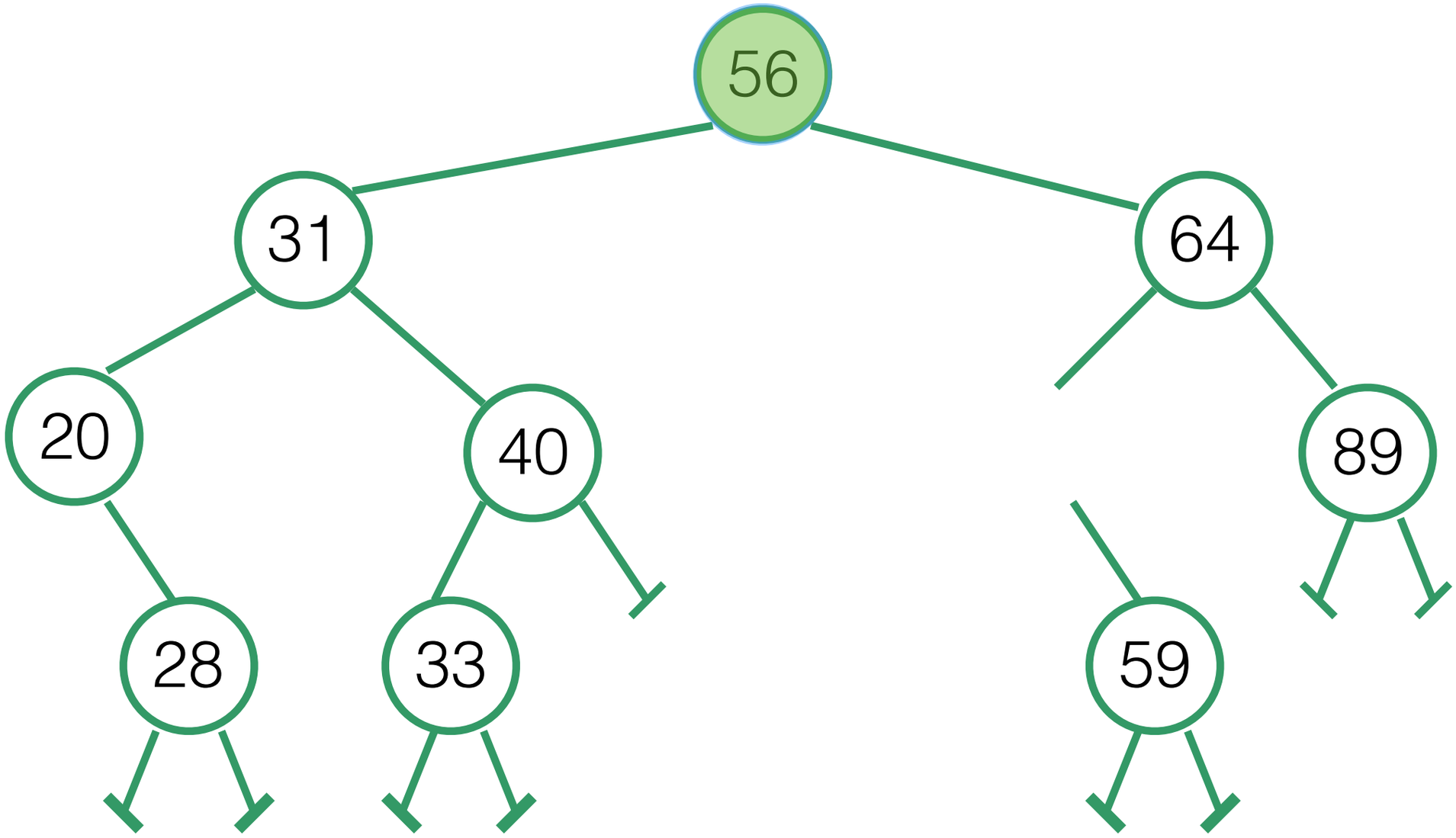
Delete 43



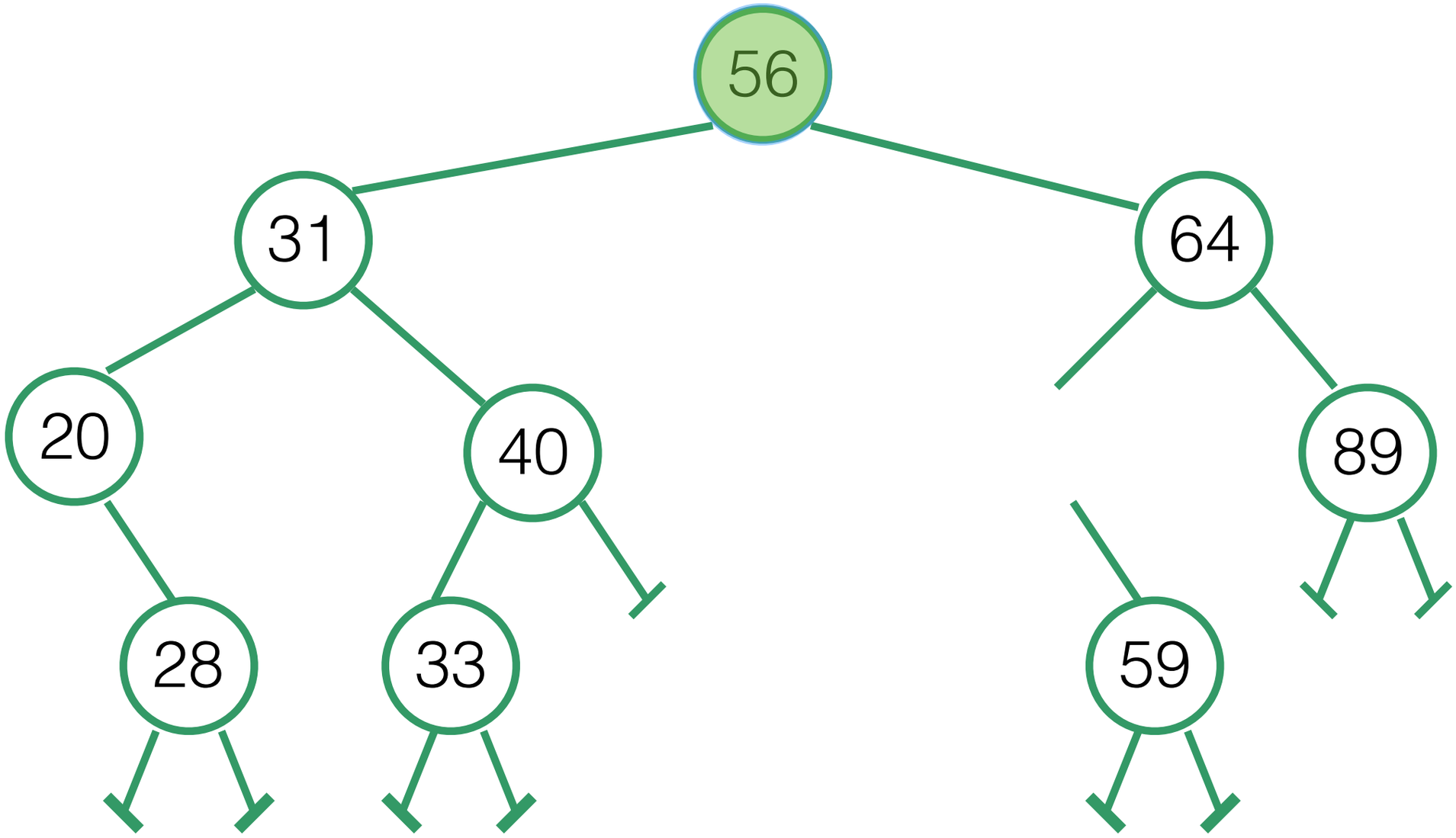
Delete 43



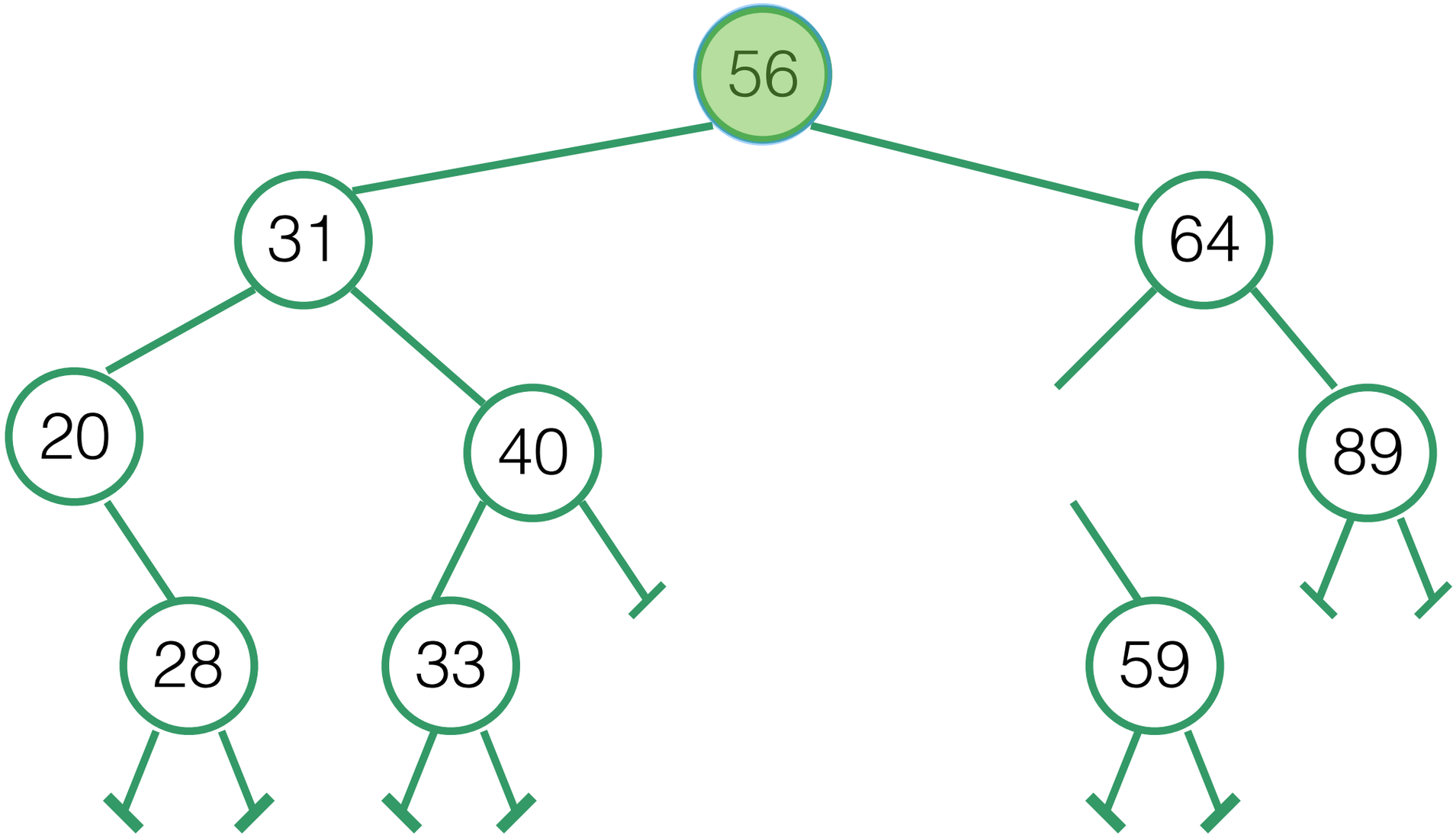
Delete 43



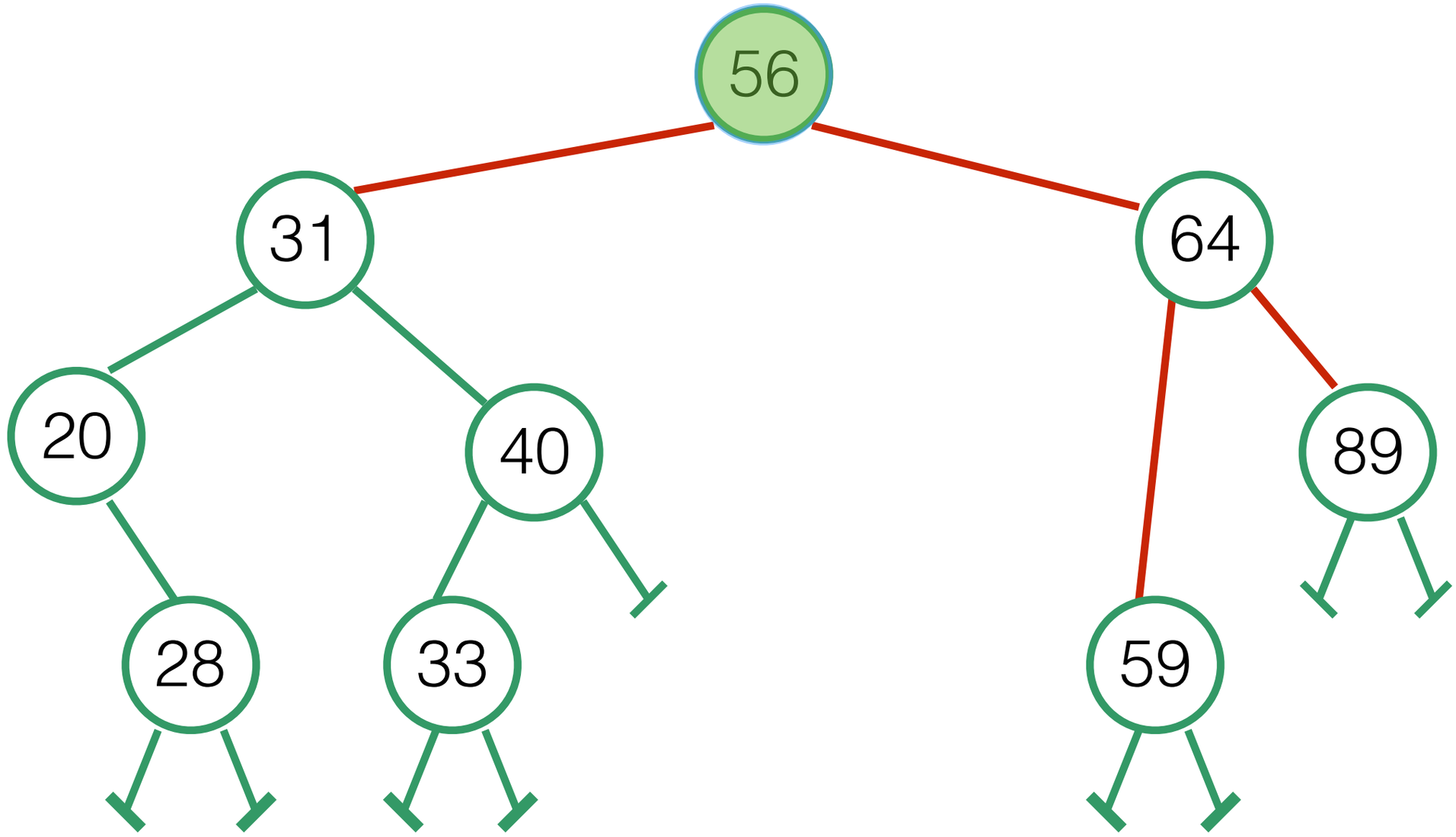
Delete 43



Delete 43



Delete 43



Delete

Input: key of element to delete.

Idea: Find key and successor…

- Try to find the key…
  - ➡ **If it is a leaf?** Set parent's reference to None
  - ➡ **It has one child?** Parent's reference set to child ("bypass").
  - ➡ **It has two children?** Find successor. Successor takes position of deleted node. If successor leaves an orphan child, it should be linked to the successor's parent.

Summary

- Binary search trees: search, insertion and deletion