The Art of Data Structures

Binary Search Trees

Alan Beadle
CSC 162: The Art of Data Structures
UNIVERSITY of ROCHESTER

Agenda

- Binary Search Trees
- Search Tree Operations
- Search Tree Implementation
- Search Tree Analysis

Binary Search Trees

Binary Search Tree

Maps

- So far, two different ways to get key-value pairs in a collection
- Recall that these collections implement the map abstract data type
- The two implementations of a map ADT we discussed were binary search on a list and hash tables

Binary Search Tree

Maps

- binary search trees as yet another way to map from a key to a value
- we are not interested in the exact placement of items in the tree, but we are interested in using the binary tree structure to provide for efficient searching

Binary Search Tree

Map *Specification*

- `Map()` creates a new, empty binary tree
- `put(key,val)` Add a new key-value pair to the tree
- `get(key)` Given a key, return the value stored in the tree or None otherwise
- `delete_key(key)` Delete the key-value pair from the tree

Binary Search Tree

Map *Specification (cont.)*

- `length()` Return the number of key-value pairs stored in the tree
- `has_key(key)` Return `True` if the given key is in the dictionary, `False` otherwise.
- `operators` We can use the above methods to overload the `[]` operators for both assignment and lookup; in addition, we can use `has_key` to override the `in` operator

Binary Search Tree
Definition

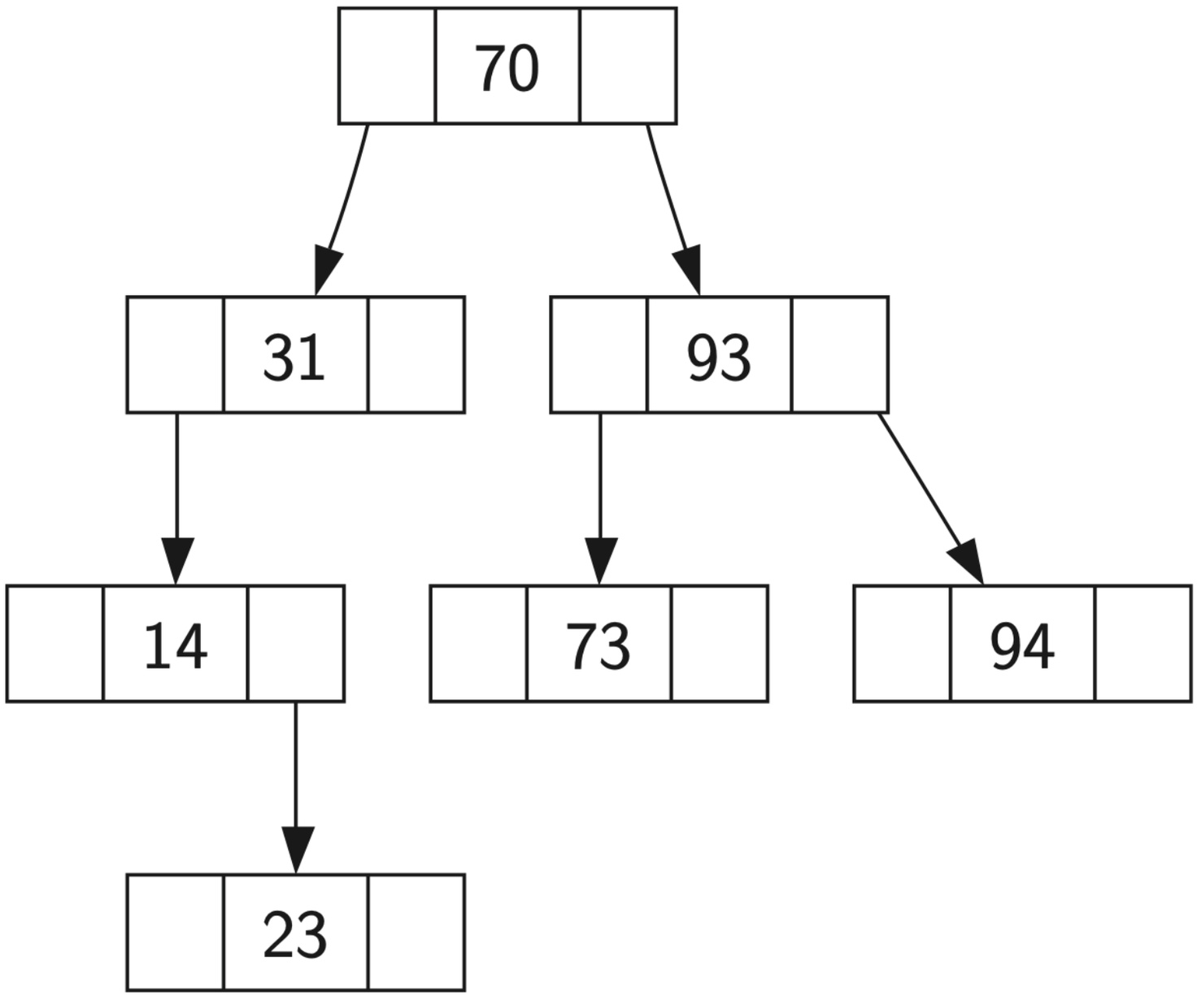
- A binary search tree relies on the property that:
  - keys that are less than the parent are found in the left subtree,
  - keys that are greater than the parent are found in the right subtree
- This the **bst property**

Binary Search Tree
Example

- The previous binary search tree represents the nodes that exist after we have inserted the following keys:
  - 70, 31, 93, 94, 14, 23, 73

Binary Search Tree

A Simple Binary Tree



Binary Search Tree
Required Classes

- We'll need to work with an empty binary search tree, so two classes are needed:
  - `BinarySearchTree`
  - `TreeNode`

Binary Search Tree

`BinarySearchTree` *Implementation*

```
class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()
```

Binary Search Tree

`TreeNode` *Implementation*

```
class TreeNode:
    def __init__(self, key, val, left=None, right=None, parent=None):
        self.key = key
        self.payload = val
        self.left_child = left
        self.right_child = right
        self.parent = parent

    def has_left_child(self):
        return self.left_child

    def has_right_child(self):
        return self.right_child

    def is_left_child(self):
        return self.parent and self.parent.left_child == self
```

Binary Search Tree

`TreeNode` *Implementation (cont.)*

```
    def is_right_child(self):
        return self.parent and self.parent.right_child == self

    def is_root(self):
        return not self.parent
```

```
    def is_leaf(self):
        return not (self.right_child or self.left_child)

    def has_any_children(self):
        return self.right_child or self.left_child

    def has_both_children(self):
        return self.right_child and self.left_child
```

Binary Search Tree

`TreeNode` *Implementation (cont.)*

```
def replace_node_data(self, key, value, lc, rc):
    self.key = key
    self.payload = value
    self.left_child = lc
    self.right_child = rc
    if self.has_left_child():
        self.left_child.parent = self
    if self.has_right_child():
        self.right_child.parent = self
```

Binary Search Tree

`TreeNode` Properties

- Every `TreeNode` instance keeps track of its parent node
- We leverage keyword arguments to provide optional, customizing parameters

Binary Search Tree

BinarySearchTree.put

```
def put(self, key, val):
    if self.root:
        self._put(key, val, self.root)
    else:
        self.root = TreeNode(key, val)
    self.size = self.size + 1
```

```
def _put(self, key, val, current_node):
    if key < current_node.key:
        if current_node.has_left_child():
            self._put(key, val, current_node.left_child)
        else:
            current_node.left_child = TreeNode(key, val, parent=current_node)
    else:
        if current_node.has_right_child():
            self._put(key, val, current_node.right_child)
        else:
            current_node.right_child = TreeNode(key, val, parent=current_node)
```

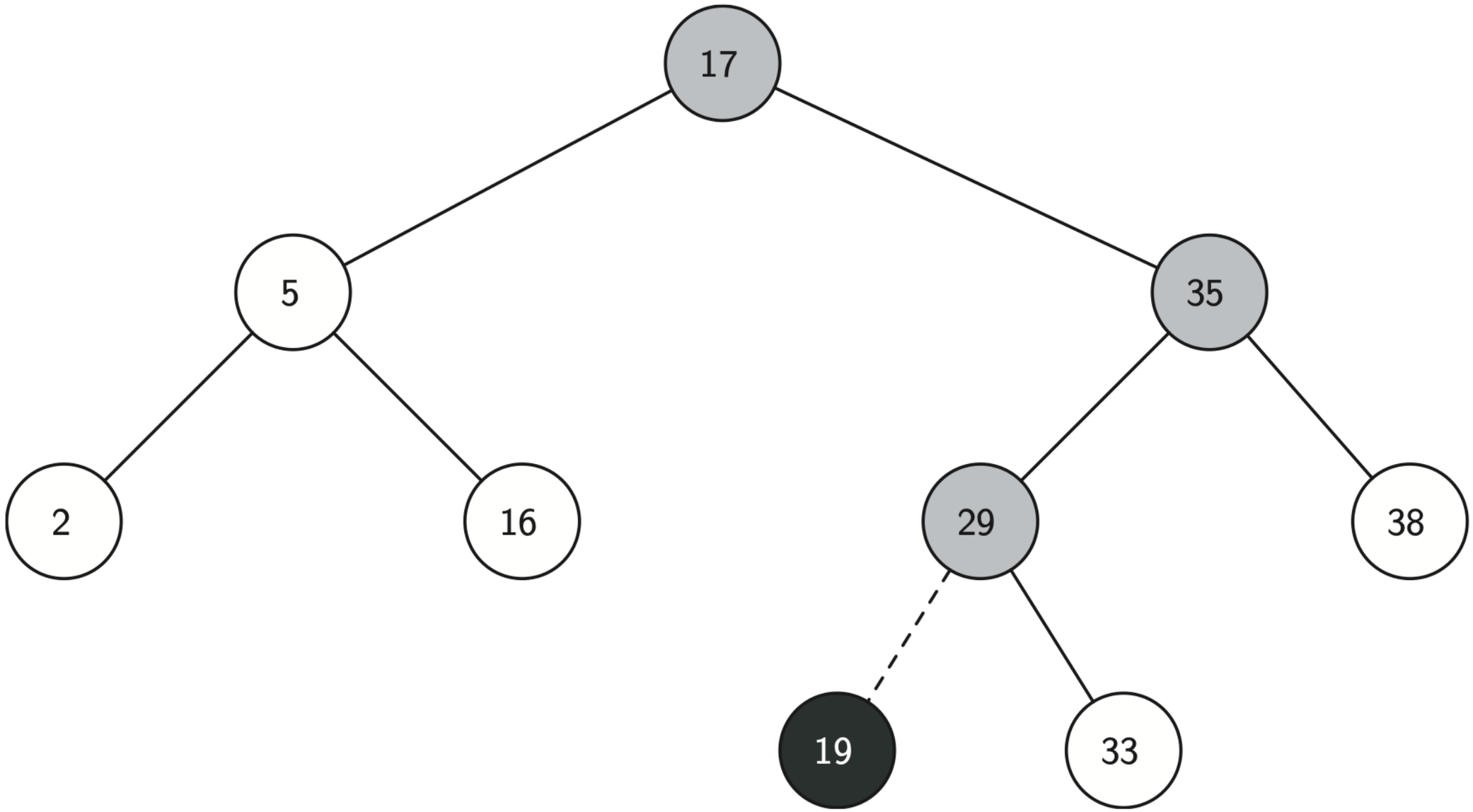
Binary Search Tree

BinarySearchTree.__setitem__

```
def __setitem__(self, k, v):
    self.put(k, v)
```

Binary Search Tree

Inserting a Node with Key = 19



Binary Search Tree

BinarySearchTree.get

```
def get(self, key):
    if self.root:
        res = self._get(key, self.root)
        if res:
            return res.payload
        else:
            return None
    else:
        return None

def _get(self, key, current_node):
    if not current_node:
        return None
    elif current_node.key == key:
        return current_node
    elif key < current_node.key:
        return self._get(key, current_node.left_child)
    else:
        return self._get(key, current_node.right_child)
```

Binary Search Tree

BinarySearchTree.__getitem__

```
def __getitem__(self, key):
    return self.get(key)
```

Binary Search Tree

BinarySearchTree.__contains__

```
def __contains__(self, key):
    if self._get(key, self.root):
        return True
    else:
        return False
```

Binary Search Tree

BinarySearchTree.delete

```
def delete(self, key):
  if self.size > 1:
      node_to_remove = self._get(key, self.root)
      if node_to_remove:
          self.remove(node_to_remove)
          self.size = self.size-1
      else:
          raise KeyError('Error, key not in tree')
elif self.size == 1 and self.root.key == key:
    self.root = None
    self.size = self.size - 1
else:
    raise KeyError('Error, key not in tree')
```

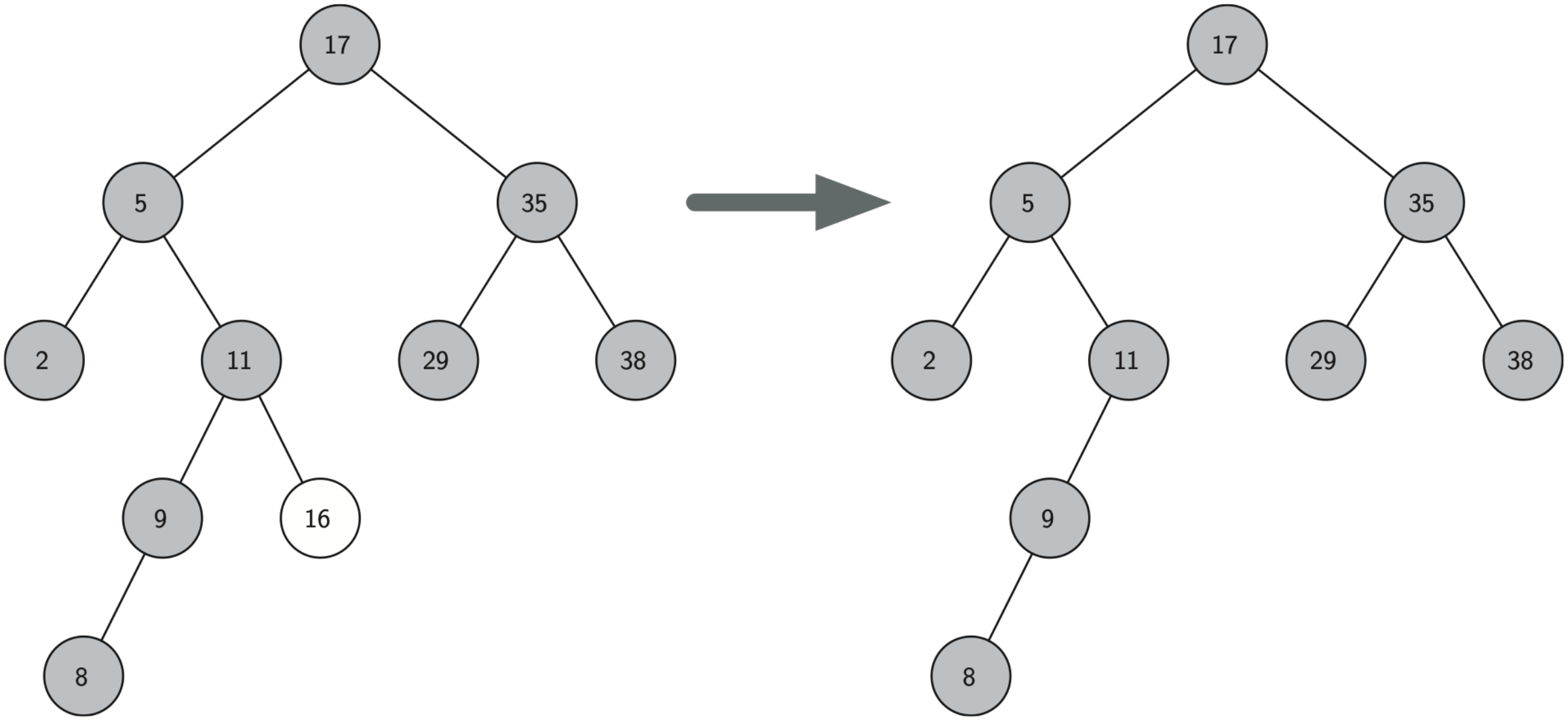
Binary Search Tree

BinarySearchTree.__delitem__

```
def __delitem__(self, key):
    self.delete(key)
```

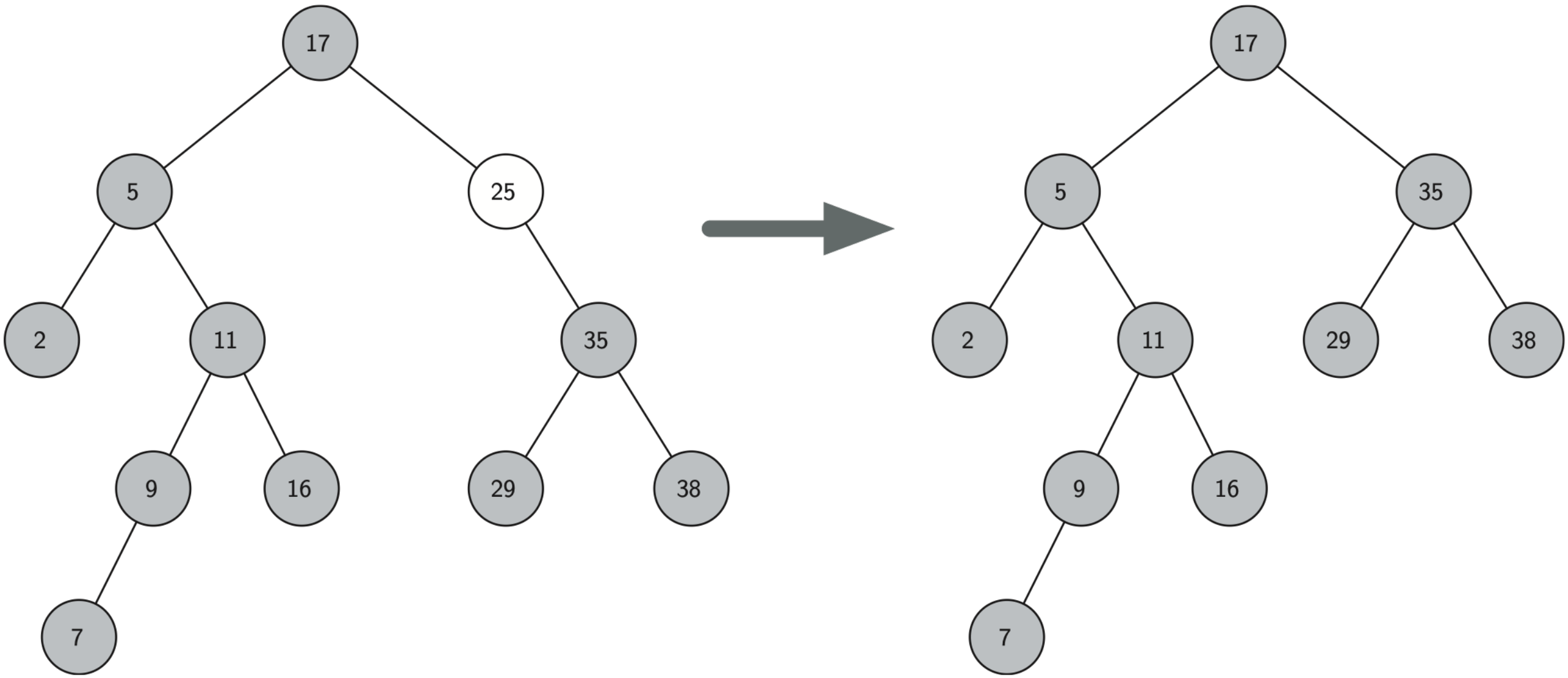
Binary Search Tree

Deleting Node 16, Childless



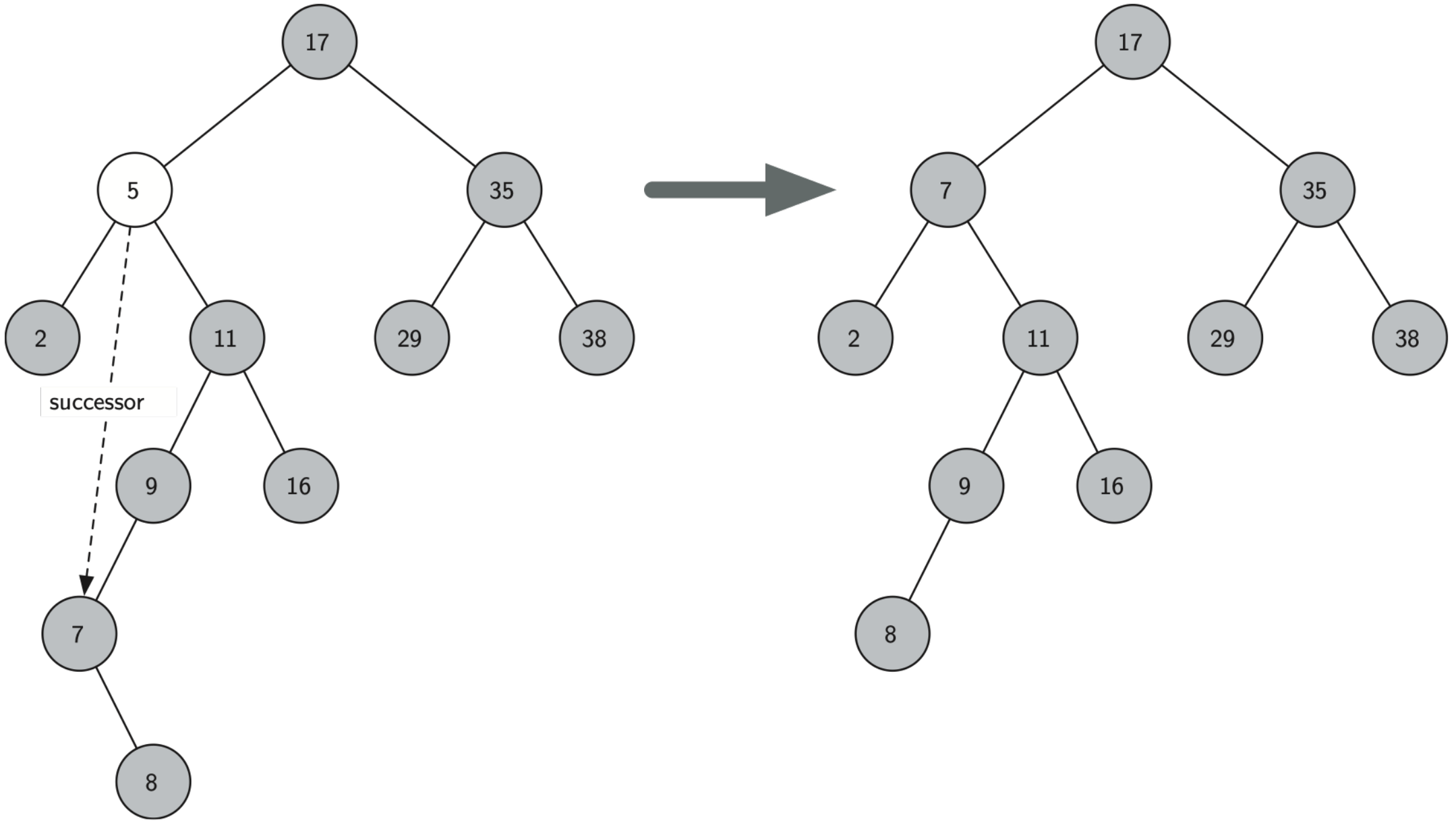
Binary Search Tree

Deleting Node 25, Single Child



Binary Search Tree

Deleting Node 5, Two Children



Binary Search Tree

TreeNode.splice_out

```
def splice_out(self):
    if self.is_leaf():
        if self.is_left_child():
                self.parent.left_child = None
        else:
                self.parent.right_child = None
    elif self.has_any_children():
        if self.has_left_child():
            if self.is_left_child():
                self.parent.left_child = self.left_child
            else:
                self.parent.right_child = self.left_child
            self.left_child.parent = self.parent
        else:
            if self.is_left_child():
                self.parent.left_child = self.right_child
            else:
                self.parent.right_child = self.right_child
            self.right_child.parent = self.parent
```

Binary Search Tree

TreeNode.find_successor

```
def find_successor(self):
    succ = None
    if self.has_right_child():
        succ = self.right_child.find_min()
    else:
        if self.parent:
            if self.is_left_child():
                succ = self.parent
            else:
                self.parent.right_child = None
                succ = self.parent.find_successor()
                self.parent.right_child = self
    return succ
```

Binary Search Tree

TreeNode.find_min

```
def find_min(self):
    current = self
    while current.has_left_child():
        current = current.left_child
    return current
```

Binary Search Tree

BinarySearchTree.remove

```
def remove(self, current_node):
   if current_node.is_leaf(): #leaf
     if current_node == current_node.parent.left_child:
         current_node.parent.left_child = None
     else:
         current_node.parent.right_child = None
  elif current_node.has_both_children(): #interior
    succ = current_node.find_successor()
    succ.splice_out()
    current_node.key = succ.key
    current_node.payload = succ.payload

  else: # this node has one child
    if current_node.has_left_child():
      if current_node.is_left_child():
          current_node.left_child.parent = current_node.parent
          current_node.parent.left_child = current_node.left_child
```

Binary Search Tree

`BinarySearchTree.remove` *(cont.)*

```
elif current_node.is_right_child():
    current_node.left_child.parent = current_node.parent
    current_node.parent.right_child = current_node.left_child
 else:
    current_node.replace_node_data(current_node.left_child.key,
                                   current_node.left_child.payload,
                                   current_node.left_child.left_child,
                                   current_node.left_child.right_child)
else:
  if current_node.is_left_child():
      current_node.right_child.parent = current_node.parent
      current_node.parent.left_child = current_node.right_child
  elif current_node.is_right_child():
      current_node.right_child.parent = current_node.parent
      current_node.parent.right_child = current_node.right_child
  else:
      current_node.replace_node_data(current_node.right_child.key,
                                     current_node.right_child.payload,
                                     current_node.right_child.left_child,
                                     current_node.right_child.right_child)
```

Binary Search Tree

`BinarySearchTree.__iter__`

- Using Python's iterator pattern: generator functions
- `yield` keyword

Binary Search Tree

`BinarySearchTree.__iter__`

```
def __iter__(self):
    if self:
        if self.has_left_child():
            for elem in self.left_child:
                yield elem
        yield self.key
        if self.has_right_child():
            for elem in self.right_child:
                yield elem
```

Binary Search Trees

Analysis

Binary Search Trees

Analysis

- Look at `put()`
  - Tree height limits performance
  - Added in random order, tree height is $log_2 n$
  - The number of nodes at any particular level is 2^d where d is the depth of the level
  - Total nodes in a perfectly balanced binary tree is $2^{h+1} - 1$, where h represents the height of the tree

Binary Search Trees

Analysis

- Look at `put()`
  - Tree height limits performance
  - Added in random order, tree height is $log_2 n$
  - The number of nodes at any particular level is 2^d where d is the depth of the level
  - Total nodes in a perfectly balanced binary tree is $2^{h+1} - 1$, where h represents the height of the tree

Binary Search Trees

Analysis

- A perfectly balanced tree has the same number of nodes in the left sub- tree as the right subtree
- In a balanced binary tree, the worst-case per- formance of `put` is $O(log_2\ n)$
- This gives us the height of the tree, and represents the maximum number of comparisons that put will need to do as it searches for the proper place to insert a new node

Binary Search Trees

Analysis

- It is possible to construct a search tree that has height *n* simply by inserting the keys in sorted order
- In this case the performance of the `put` method is *O(n)*

Binary Search Trees

Analysis

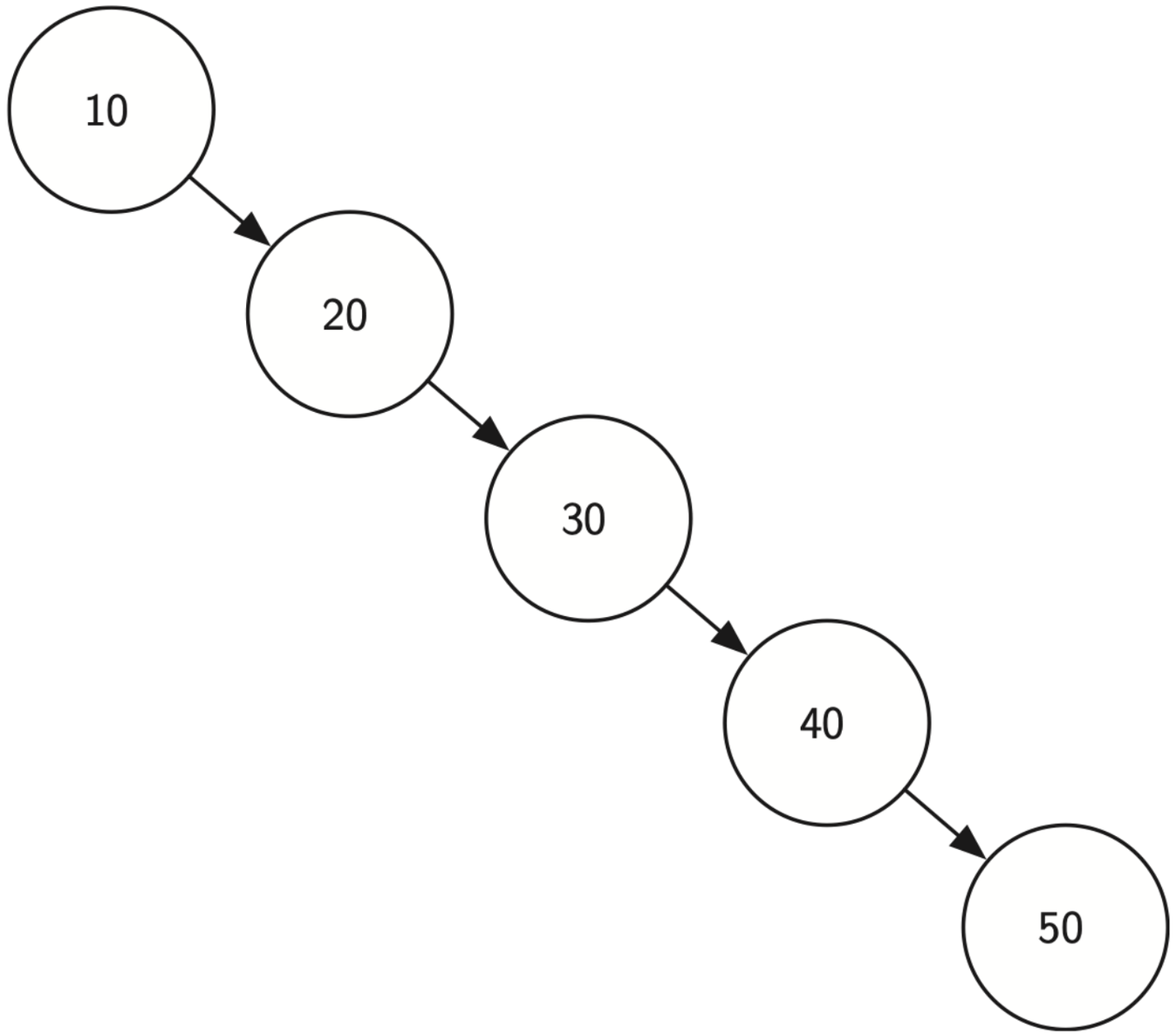
- Since `get` searches the tree to find the key, in the worst case the tree is searched all the way to the bottom and no key is found

- For `del`, the worst-case scenario to find the successor is also just the height of the tree which means that you would simply double the work

  - Since doubling is a constant factor it does not change worst case analysis of *O(n)* for an unbalanced tree

Binary Search Trees

A Skewed Binary Search Tree



Questions?