COSC 2306
Data Programming

Trees

Binary Search Trees

- Data in each node
  - Larger than the data in its left child
  - Smaller than the data in its right child

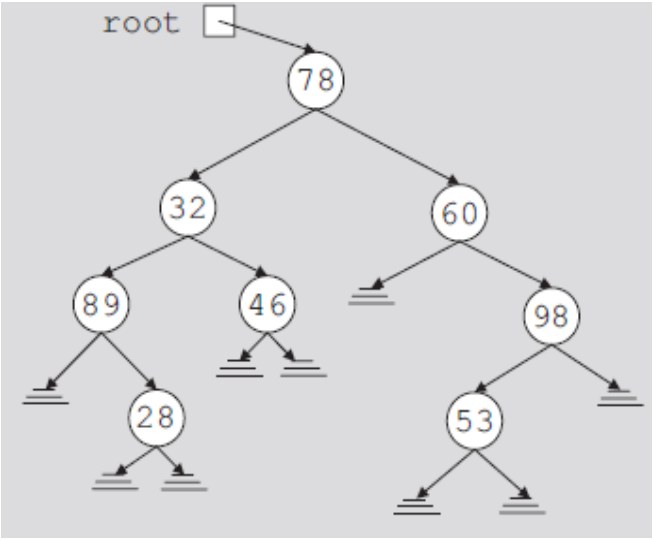

FIGURE 11-6 Arbitrary binary tree

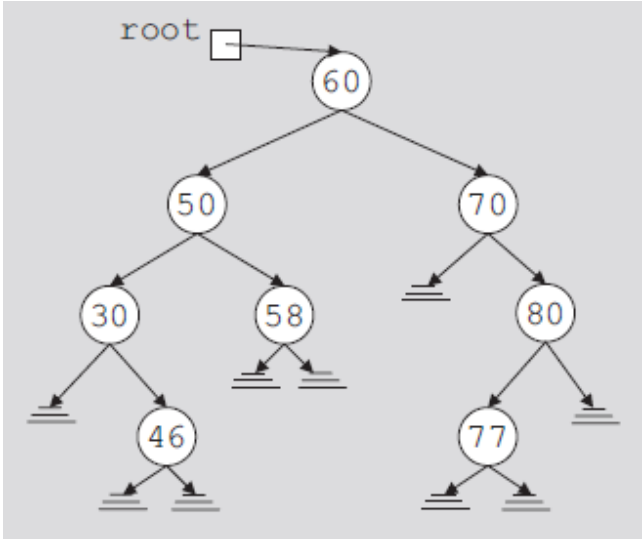

FIGURE 11-7 Binary search tree

Binary Search Trees

- A binary search tree, T, is either empty or the following is true:
  - T has a special node called the root node
  - T has two sets of nodes, L_T and R_T , called the left subtree and right subtree of T, respectively
  - L_T and R_T are binary search trees
  - The key in the root node is larger than every key in the left subtree and smaller than every key in the right subtree

Binary Search Trees - Put

- Starting from the root, search the binary tree comparing the new key to the key in the current node
  - If new key < the current node, search the left subtree
  - If new key > the current node, search the right subtree
- When there is no left (or right) child to search, we have found the position where the new node should be installed

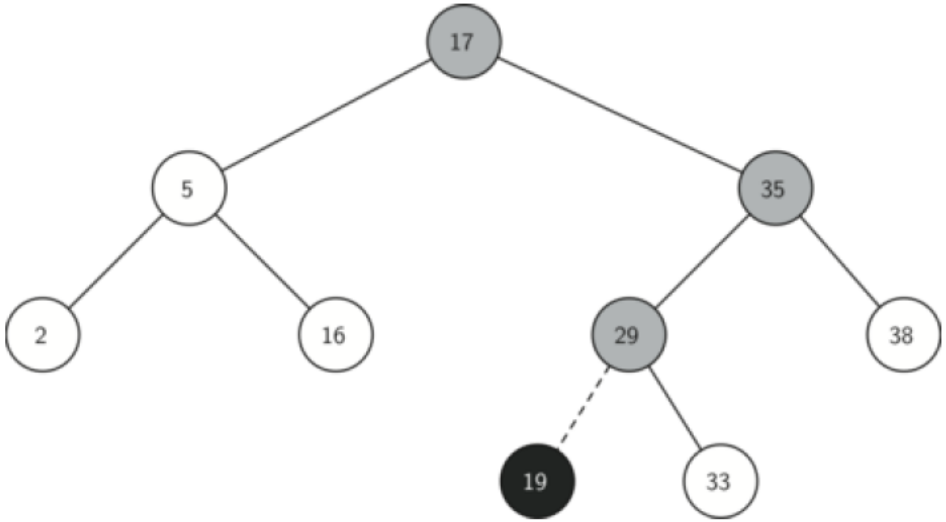

Figure 6.19: Inserting a Node with Key = 19

Binary Search Trees - Delete

We have to address 3 different scenarios

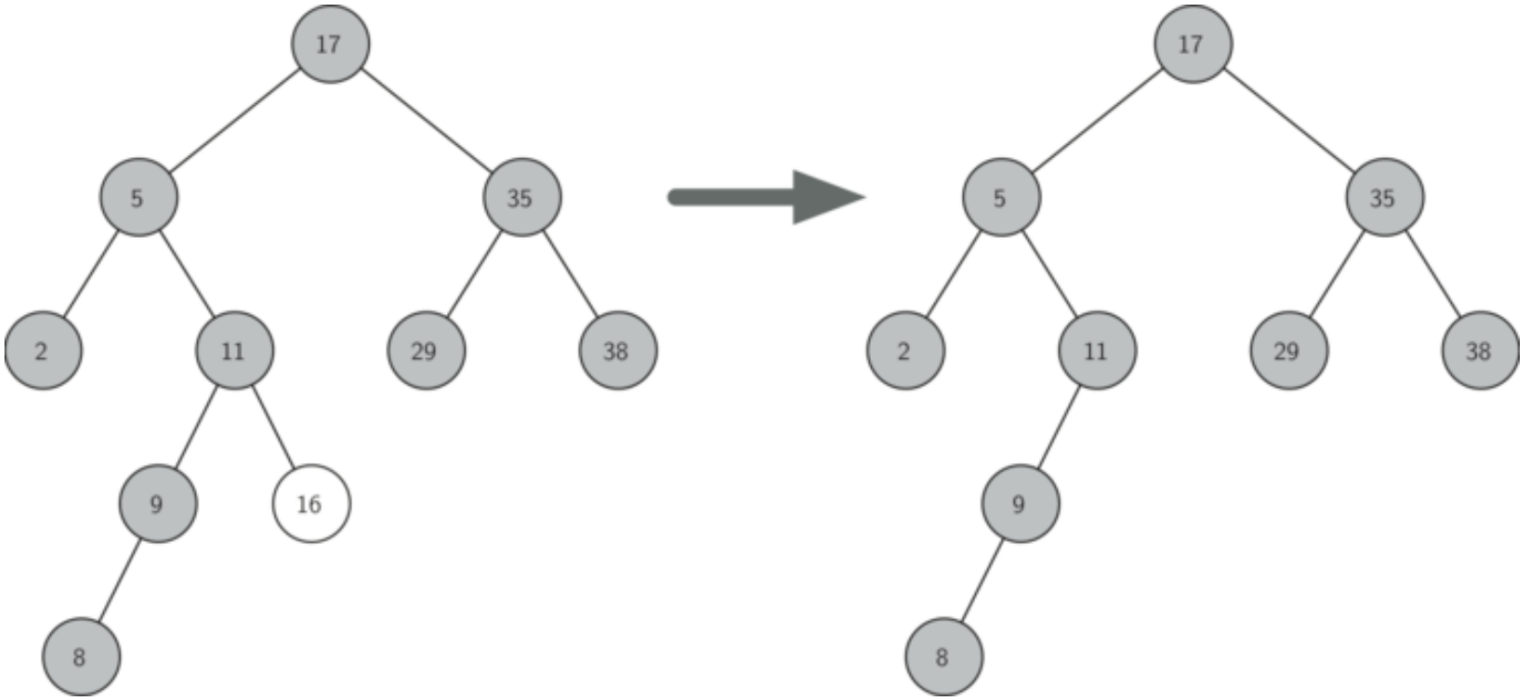

Figure 3: Deleting Node 16, a Node without Children

Assign node to null

Binary Search Trees - Delete

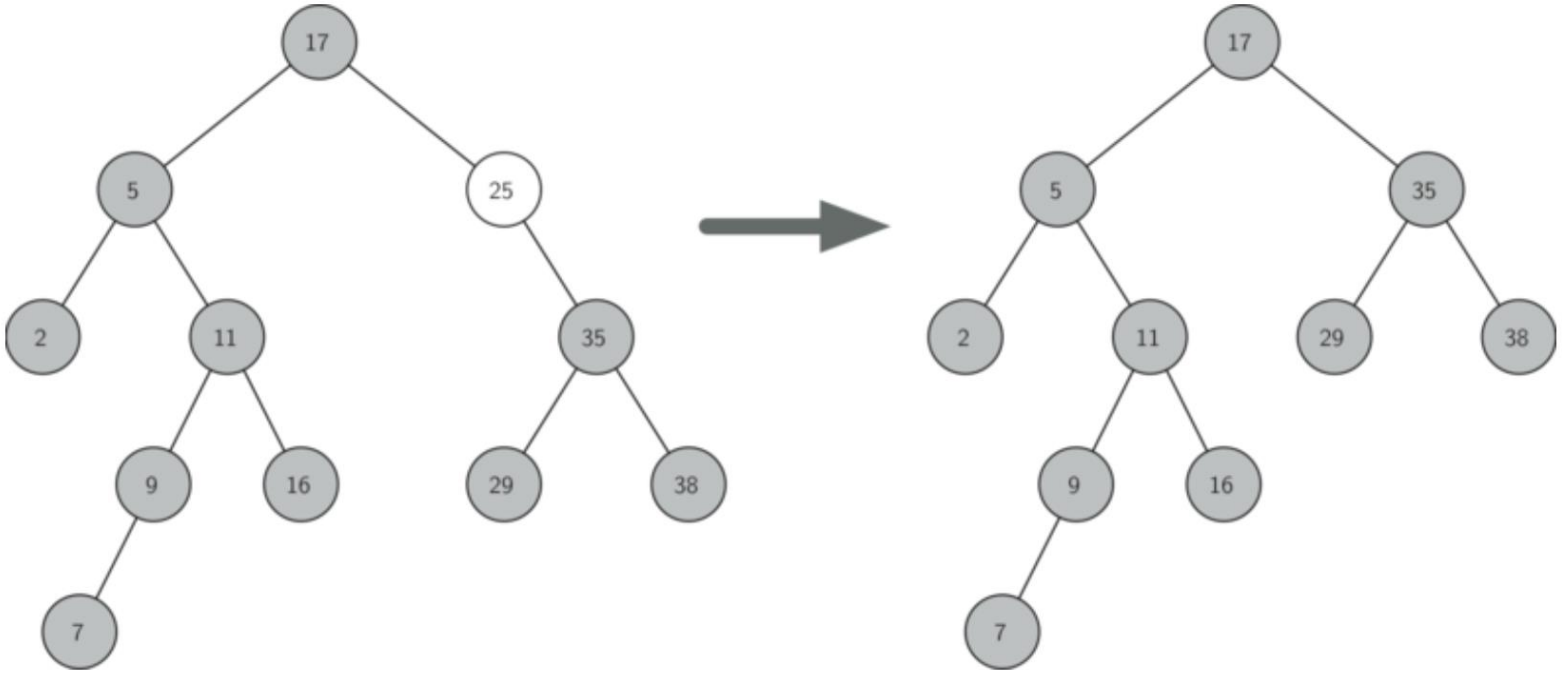

Figure 4: Deleting Node 25, a Node That Has a Single Child

Copy the child to the node and delete the node

Binary Search Trees - Delete

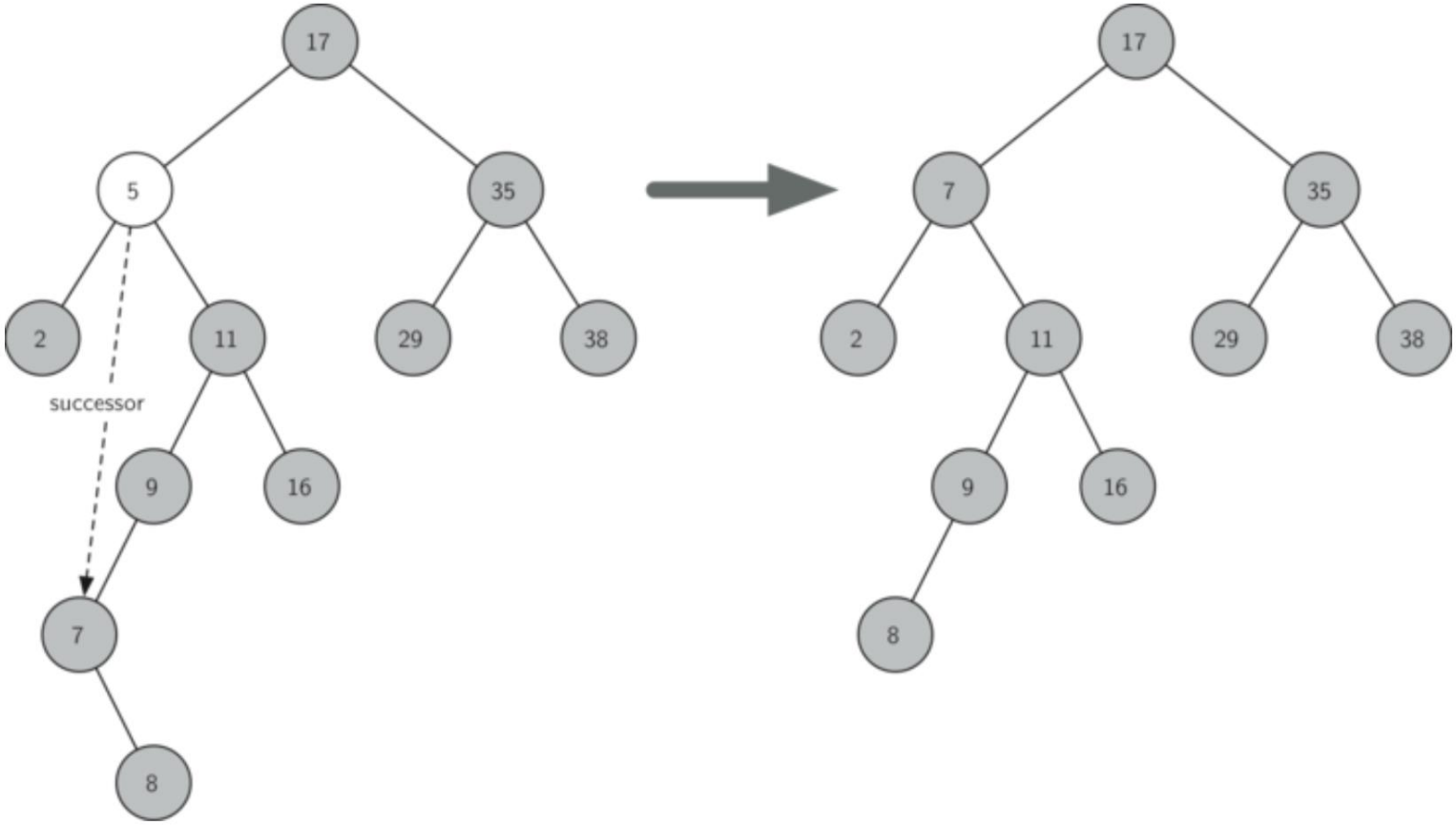

Figure 5: Deleting Node 5, a Node with Two Children

The successor is the smallest key in the right subtree

Binary Search Trees - Class

```
# Basic BinarySearchTree class - incomplete
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 Trees - Class

- Define helper functions to
  - Classify a node according to its own position as a child (left or right)
  - The kind of children the node has
  - Explicitly keep track of the parent as an attribute of each node
- Node
  - A node's name, called "key"
  - A node can also have additional information, called "payload"
- Underscore
  - Single underscore _ before an object name: object is private, and shouldn't be directly accessed from outside the class
  - Import does not include this class

Binary Search Trees - Class

```
# Completed TreeNode class
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
    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)
```

Binary Search Trees - Class

```
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
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 Trees - 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:  #search the left subtree
        if current_node.has_left_child():  #current node has subtree
            self._put(key, val, current_node.left_child)  #recursive search
        else:  #found the spot to insert
            current_node.left_child = TreeNode(key, val, parent=current_node)
    else:  #search the right subtree
        if current_node.has_right_child(): #current node has subtree
            self._put(key, val, current_node.right_child) #recursive search
        else:  #found the spot to insert
            current_node.right_child = TreeNode(key, val, parent=current_node)
```

Binary Search Trees - 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: #did not find the node
        return None
    elif current_node.key == key: #found the node
        return current_node
    elif key < current_node.key: #search left subtree
        return self._get(key, current_node.left_child) #recursive search
    else: #search right subtree
        return self._get(key, current_node.right_child) #recursive search
def __getitem__(self, key):
    return self.get(key)
```

Binary Search Trees - Delete

```
def delete(self, key):
    if self.size > 1:  #if the node is NOT root
        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:  #if the node is root
        self.root = None
        self.size = self.size - 1
    else:  #if the node is not present
        raise KeyError('Error, key not in tree')

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

Binary Search Trees - Delete

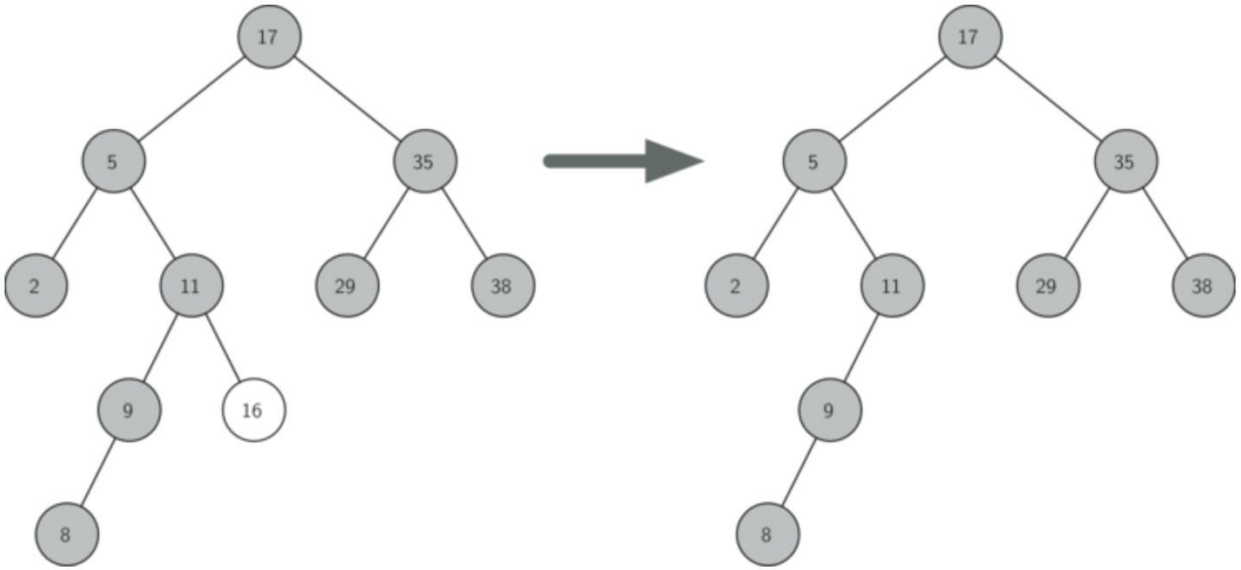

Figure 3: Deleting Node 16, a Node without Children

```
if current_node.is_leaf():
    if current_node == current_node.parent.left_child:
        current_node.parent.left_child = None
    else:
        current_node.parent.right_child = None
```

Binary Search Trees - Delete

else: # this node has one child

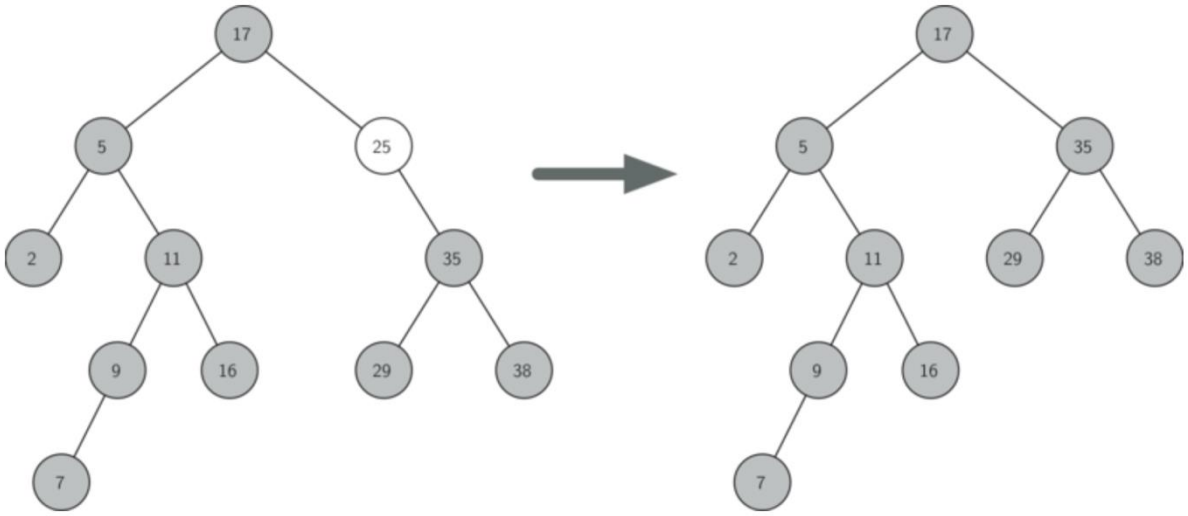

Figure 4: Deleting Node 25, a Node That Has a Single Child

1. If the current node is a left child:
   - update the parent reference of the left child -> the parent of the current node
   - update the left child reference of the parent -> the current node's left child
2. If the current node is a right child:
   - update the parent reference of the right child -> the parent of the current node
   - update the right child reference of the parent -> the current node's right child
3. If the current node has no parent:
   - it must be the root
   - replace the root node with the new node

Binary Search Trees - Delete

```
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
        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)
```

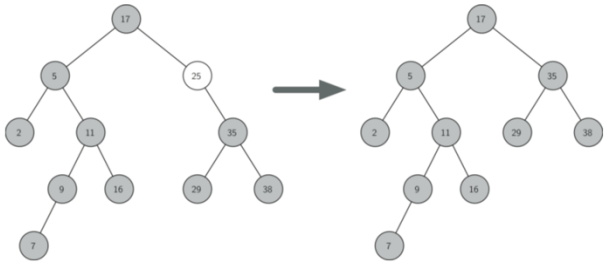

Figure 4: Deleting Node 25, a Node That Has a Single Child

Binary Search Trees - Delete

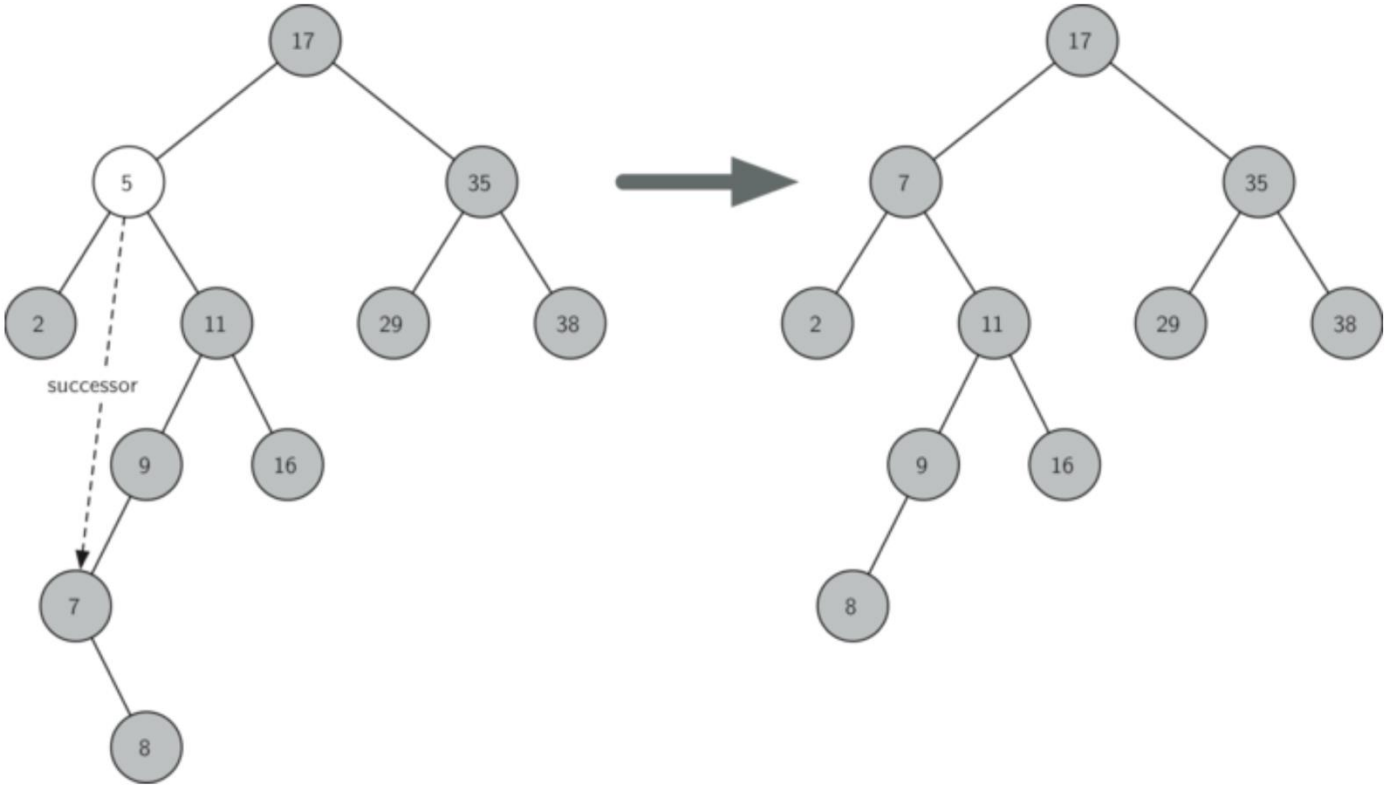

Figure 5: Deleting Node 5, a Node with Two Children

The successor is the smallest key in the right subtree

Binary Search Trees - Delete

```
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
```

```
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
```

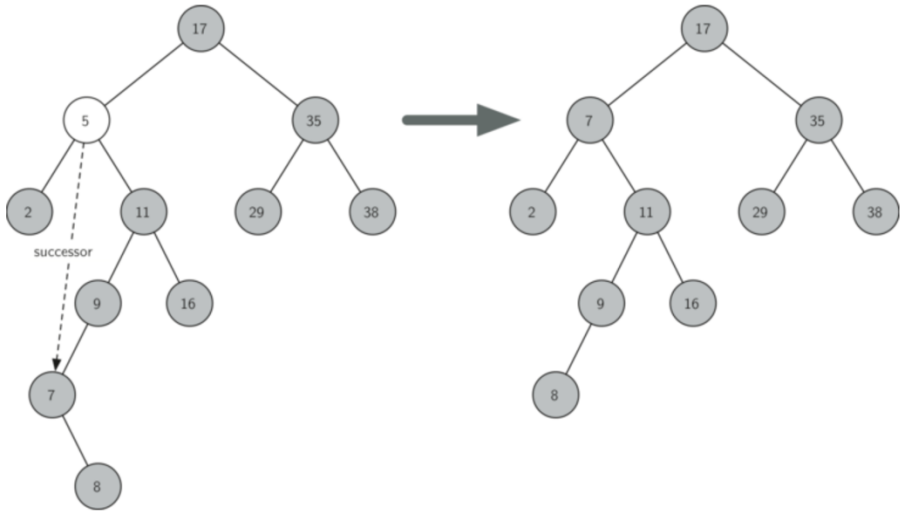

Figure 5: Deleting Node 5, a Node with Two Children

Binary Search Trees - Delete

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