

CSCI 232:

Data Structures and Algorithms

Binary Search Trees (BST) Part 2, Program 1

Reese Pearsall
Summer 2025

Announcements

Program 1 posted, due
Wednesday June 4th

Lab 3 due Sunday

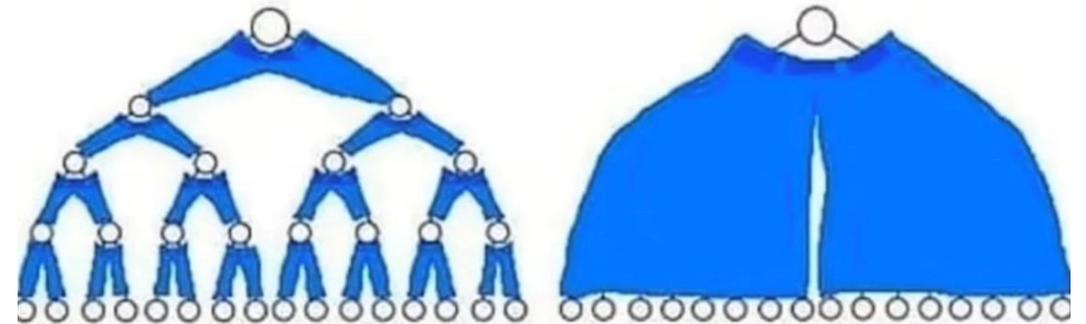
Please remove your **package**
statements before submitting

**If a binary tree wore pants
would he wear them...**

like this

or

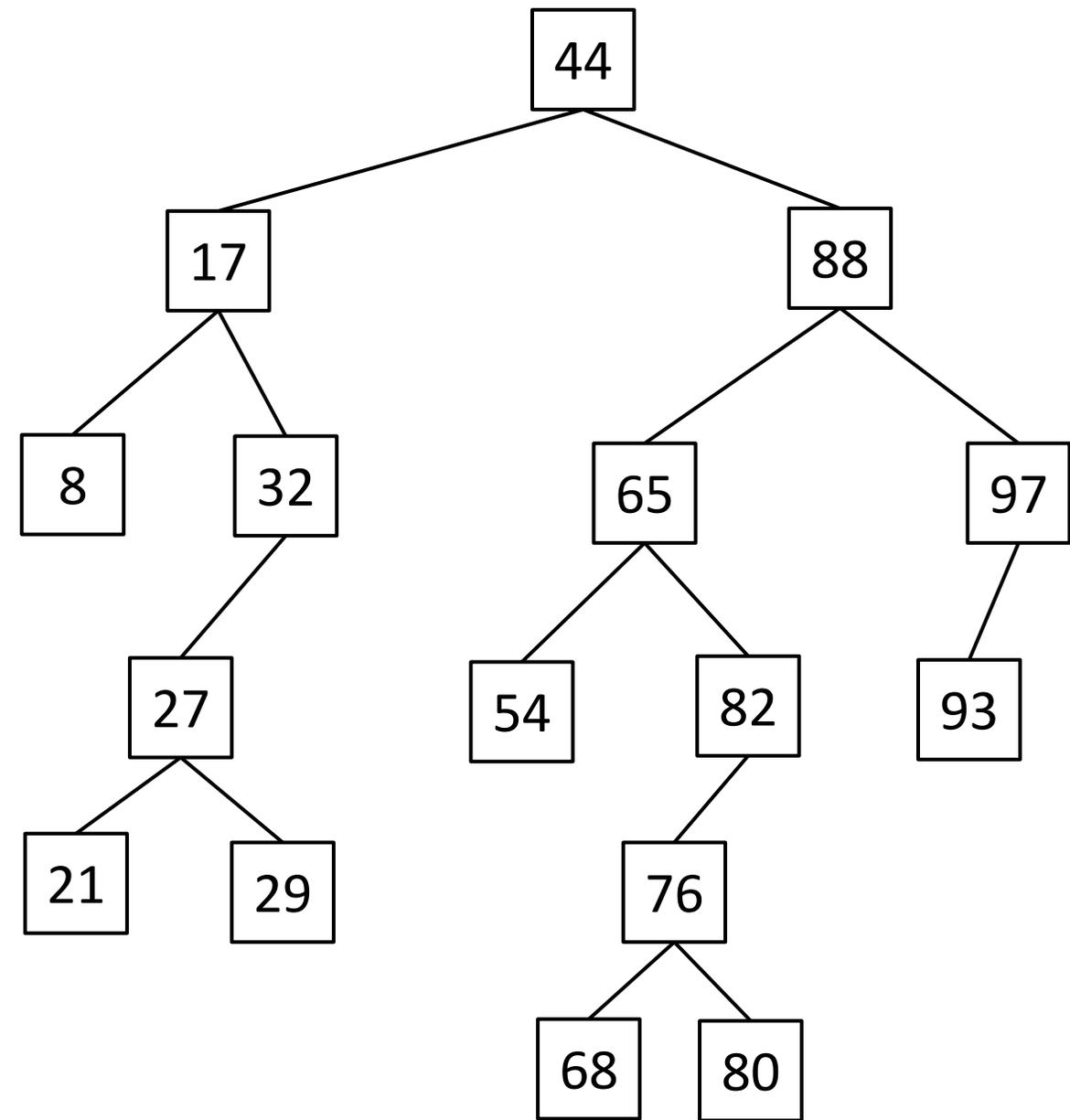
like this?



Binary Search Tree

Properties of a BST:

- Composed of **Comparable** data elements
- Each node as at most two children
- For a given node, all left-hand descendants have values that are less than the node
- For a given node, all right-hand descendants have values that are greater than the node
- No duplicate values



Binary Search Tree - Insertion

```
public void insert(int newValue) {
    if(root == null) {
        root = new Node(newValue);
    }
    else {
        Node currentNode = root;
        boolean placed = false;
        while(!placed) {
            if(currentNode.getValue() == newValue) {
                placed = true;
                System.out.println("No duplicate values allowed");
            }
            else if(newValue < currentNode.getValue()) {
                if(currentNode.getLeft() == null) {
                    currentNode.setLeft(new Node(newValue));
                    currentNode.getLeft().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getLeft();
                }
            }
            else {
                if(currentNode.getRight() == null) {
                    currentNode.setRight(new Node(newValue));
                    currentNode.getRight().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getRight();
                }
            }
        }
    }
}
```

Binary Search Tree - Insertion

```
public void insert(int newValue) {
    if(root == null) {
        root = new Node(newValue);
    }
    else {
        Node currentNode = root;
        boolean placed = false;
        while(!placed) {
            if(currentNode.getValue() == newValue) {
                placed = true;
                System.out.println("No duplicate values allowed");
            }
            else if(newValue < currentNode.getValue()) {
                if(currentNode.getLeft() == null) {
                    currentNode.setLeft(new Node(newValue));
                    currentNode.getLeft().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getLeft();
                }
            }
            else {
                if(currentNode.getRight() == null) {
                    currentNode.setRight(new Node(newValue));
                    currentNode.getRight().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getRight();
                }
            }
        }
    }
}
```

We repeatedly move left or right until we find the correct spot for our new node

Binary Search Tree - Insertion

```
public void insert(int newValue) {
    if(root == null) {
        root = new Node(newValue);
    }
    else {
        Node currentNode = root;
        boolean placed = false;
        while(!placed) {
            if(currentNode.getValue() == newValue) {
                placed = true;
                System.out.println("No duplicate values allowed");
            }
            else if(newValue < currentNode.getValue()) {
                if(currentNode.getLeft() == null) {
                    currentNode.setLeft(new Node(newValue));
                    currentNode.getLeft().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getLeft();
                }
            }
            else {
                if(currentNode.getRight() == null) {
                    currentNode.setRight(new Node(newValue));
                    currentNode.getRight().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getRight();
                }
            }
        }
    }
}
```

We repeatedly move left or right until we find the correct spot for our new node

Once we find the correct spot, we update some pointers

Binary Search Tree - Insertion

Running time?

```
public void insert(int newValue) {
    if(root == null) {
        root = new Node(newValue);
    }
    else {
        Node currentNode = root;
        boolean placed = false;
        while(!placed) {
            if(currentNode.getValue() == newValue) {
                placed = true;
                System.out.println("No duplicate values allowed");
            }
            else if(newValue < currentNode.getValue()) {
                if(currentNode.getLeft() == null) {
                    currentNode.setLeft(new Node(newValue));
                    currentNode.getLeft().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getLeft();
                }
            }
            else {
                if(currentNode.getRight() == null) {
                    currentNode.setRight(new Node(newValue));
                    currentNode.getRight().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getRight();
                }
            }
        }
    }
}
```

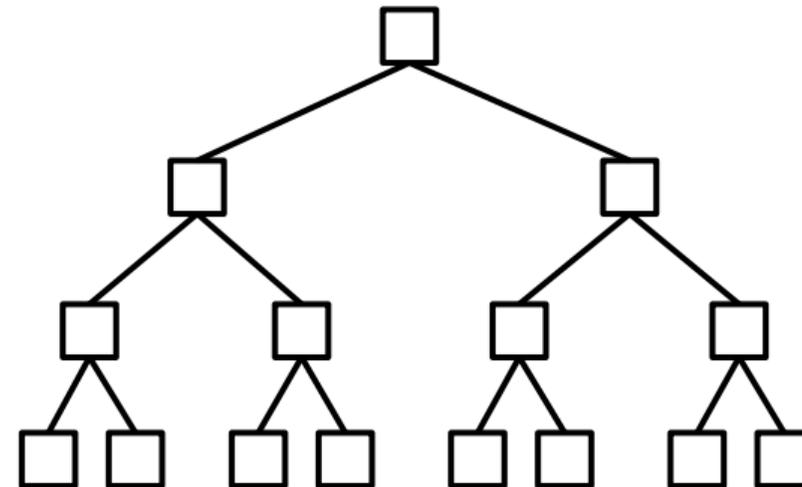
Binary Search Tree - Insertion

```
public void insert(int newValue) {
    if(root == null) {
        root = new Node(newValue);
    }
    else {
        Node currentNode = root;
        boolean placed = false;
        while(!placed) {
            if(currentNode.getValue() == newValue) {
                placed = true;
                System.out.println("No duplicate values allowed");
            }
            else if(newValue < currentNode.getValue()) {
                if(currentNode.getLeft() == null) {
                    currentNode.setLeft(new Node(newValue));
                    currentNode.getLeft().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getLeft();
                }
            }
            else {
                if(currentNode.getRight() == null) {
                    currentNode.setRight(new Node(newValue));
                    currentNode.getRight().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getRight();
                }
            }
        }
    }
}
```

Running time?

We will always be inserting a leaf node, so worst cast scenario we will need to travel the **height** of the tree

If we have a “balanced tree” the height of the tree, is $\log(n)$ $n = \#$ of nodes



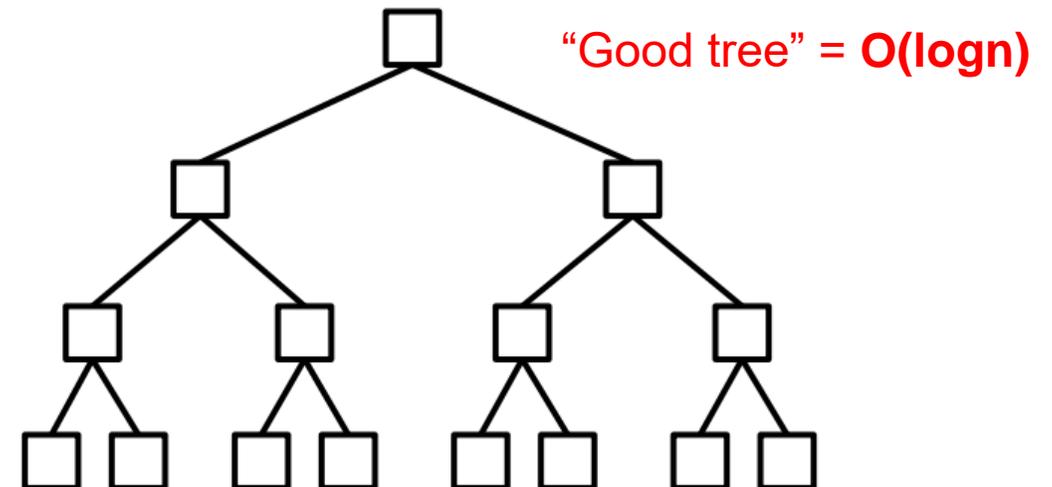
Binary Search Tree - Insertion

```
public void insert(int newValue) {
    if(root == null) {
        root = new Node(newValue);
    }
    else {
        Node currentNode = root;
        boolean placed = false;
        while(!placed) {
            if(currentNode.getValue() == newValue) {
                placed = true;
                System.out.println("No duplicate values allowed");
            }
            else if(newValue < currentNode.getValue()) {
                if(currentNode.getLeft() == null) {
                    currentNode.setLeft(new Node(newValue));
                    currentNode.getLeft().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getLeft();
                }
            }
            else {
                if(currentNode.getRight() == null) {
                    currentNode.setRight(new Node(newValue));
                    currentNode.getRight().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getRight();
                }
            }
        }
    }
}
```

Running time?

We will always be inserting a leaf node, so worst cast scenario we will need to travel the **height** of the tree

If we have a “balanced tree” the height of the tree, is $\log(n)$ $n = \#$ of nodes



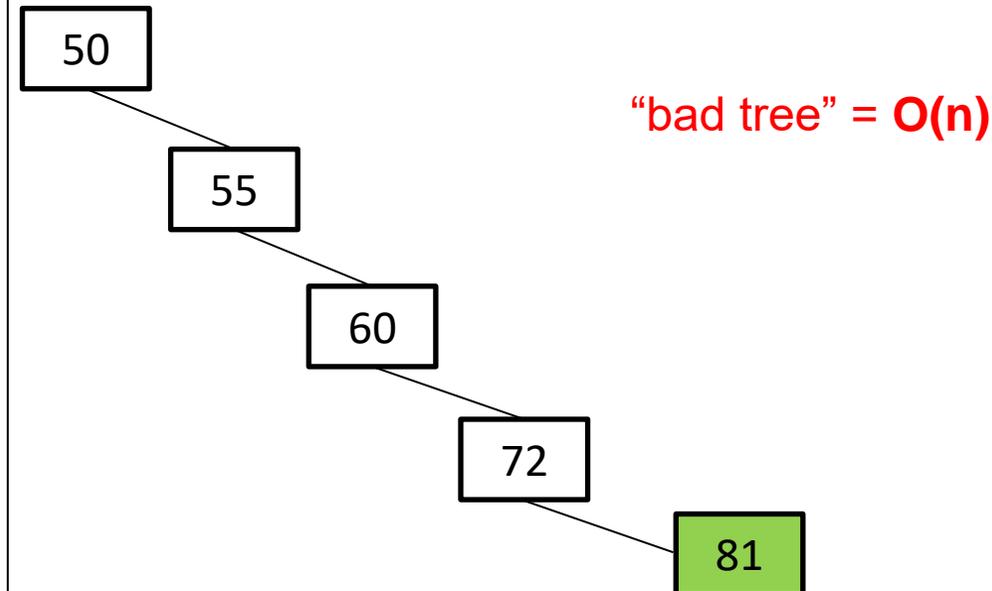
Binary Search Tree - Insertion

```
public void insert(int newValue) {
    if(root == null) {
        root = new Node(newValue);
    }
    else {
        Node currentNode = root;
        boolean placed = false;
        while(!placed) {
            if(currentNode.getValue() == newValue) {
                placed = true;
                System.out.println("No duplicate values allowed");
            }
            else if(newValue < currentNode.getValue()) {
                if(currentNode.getLeft() == null) {
                    currentNode.setLeft(new Node(newValue));
                    currentNode.getLeft().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getLeft();
                }
            }
            else {
                if(currentNode.getRight() == null) {
                    currentNode.setRight(new Node(newValue));
                    currentNode.getRight().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getRight();
                }
            }
        }
    }
}
```

Running time?

We will always be inserting a leaf node, so worst cast scenario we will need to travel the **height** of the tree

If we have a “bad tree” the height of the tree, is $O(n-1)$ $n = \#$ of nodes



Binary Search Tree - Insertion

```
public void insert(int newValue) {
    if(root == null) {
        root = new Node(newValue);
    }
    else {
        Node currentNode = root;
        boolean placed = false;
        while(!placed) {
            if(currentNode.getValue() == newValue) {
                placed = true;
                System.out.println("No duplicate values allowed");
            }
            else if(newValue < currentNode.getValue()) {
                if(currentNode.getLeft() == null) {
                    currentNode.setLeft(new Node(newValue));
                    currentNode.getLeft().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getLeft();
                }
            }
            else {
                if(currentNode.getRight() == null) {
                    currentNode.setRight(new Node(newValue));
                    currentNode.getRight().setParent(currentNode);
                    placed = true;
                }
                else {
                    currentNode = currentNode.getRight();
                }
            }
        }
    }
}
```

Running time?

We will always be inserting a leaf node, so worst cast scenario we will need to travel the **height** of the tree

“Bad” tree $\rightarrow O(n)$
“Good” tree $\rightarrow O(\log n)$

$O(h) \rightarrow h = \text{height of tree}$

Running time for adding to an array?

$O(n)$



Binary Search Tree - Insertion

Running time?

We will always be inserting a leaf node, so worst cast scenario we will need to travel the **height** of the tree

```
public void insert(int newValue) {  
    if(ro  
        r  
    }  
    else  
        Node  
        boole  
        while  
    }  
    e  
}
```

If we can find a way to keep a tree “balanced”, we can achieve **$O(\log n)$** insertion time, and **$O(\log n)$** searching time

“Bad” tree $\rightarrow O(n)$
“Good” tree $\rightarrow O(\log n)$

$O(h) \rightarrow h = \text{height of tree}$

Running time for adding to an array?

$O(n)$



Binary Search Tree - Insertion

Running time?

We will always be inserting a leaf node, so worst cast scenario we will need to travel the **height** of the tree

```
public void insert(int newValue) {  
    if(ro  
        r  
    }  
    else  
        Node  
        boole  
        while  
    }  
    e  
}
```

If we can find a way to keep a tree “balanced”, we can achieve **$O(\log n)$** insertion time, and **$O(\log n)$** searching time

There is a way! Coming soon

“Bad” tree $\rightarrow O(n)$
“Good” tree $\rightarrow O(\log n)$

$O(h) \rightarrow h = \text{height of tree}$

Running time for adding to an array?

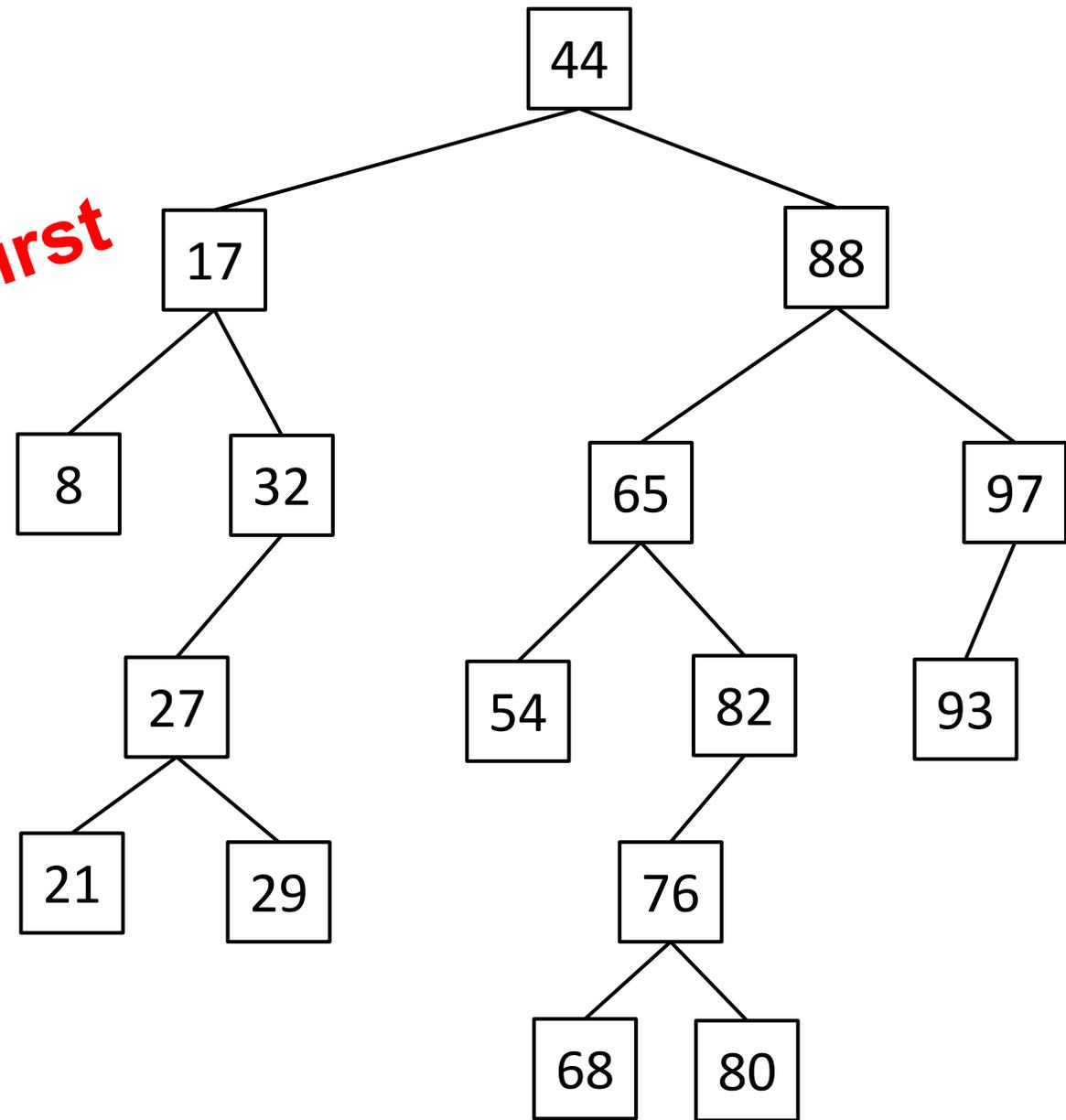
$O(n)$



Binary Search Tree- Traversal

```
public void depthFirst(Node n) {  
    if(n != null) {  
        System.out.println(n.getValue());  
        depthFirst(n.getLeft());  
        depthFirst(n.getRight());  
    }  
}
```

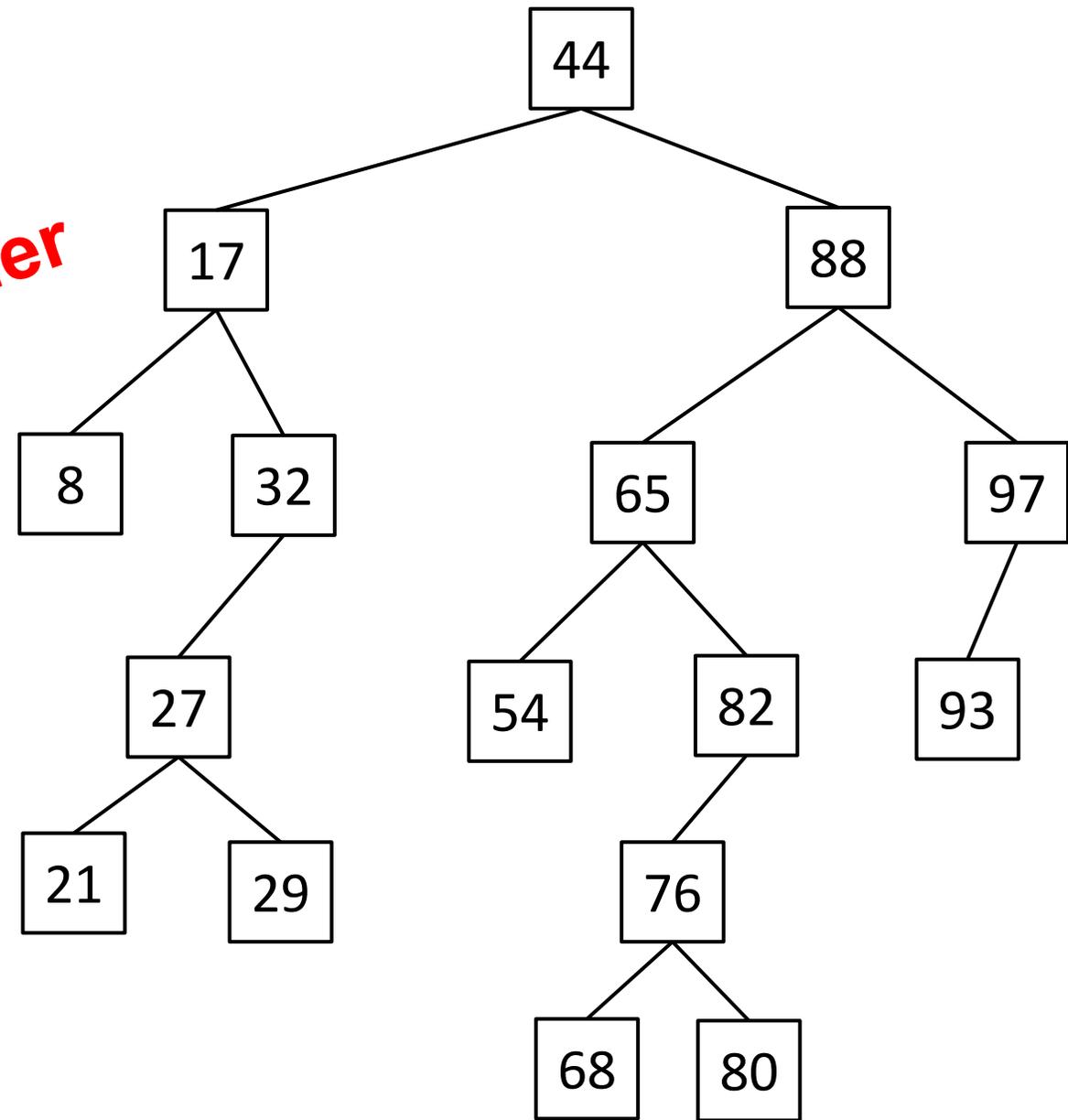
Depth First



Binary Search Tree- Traversal

```
public void preorder(Node n) {  
    if(n != null) {  
        System.out.println(n.getValue());  
        preorder(n.getLeft());  
        preorder(n.getRight());  
    }  
}
```

Preorder

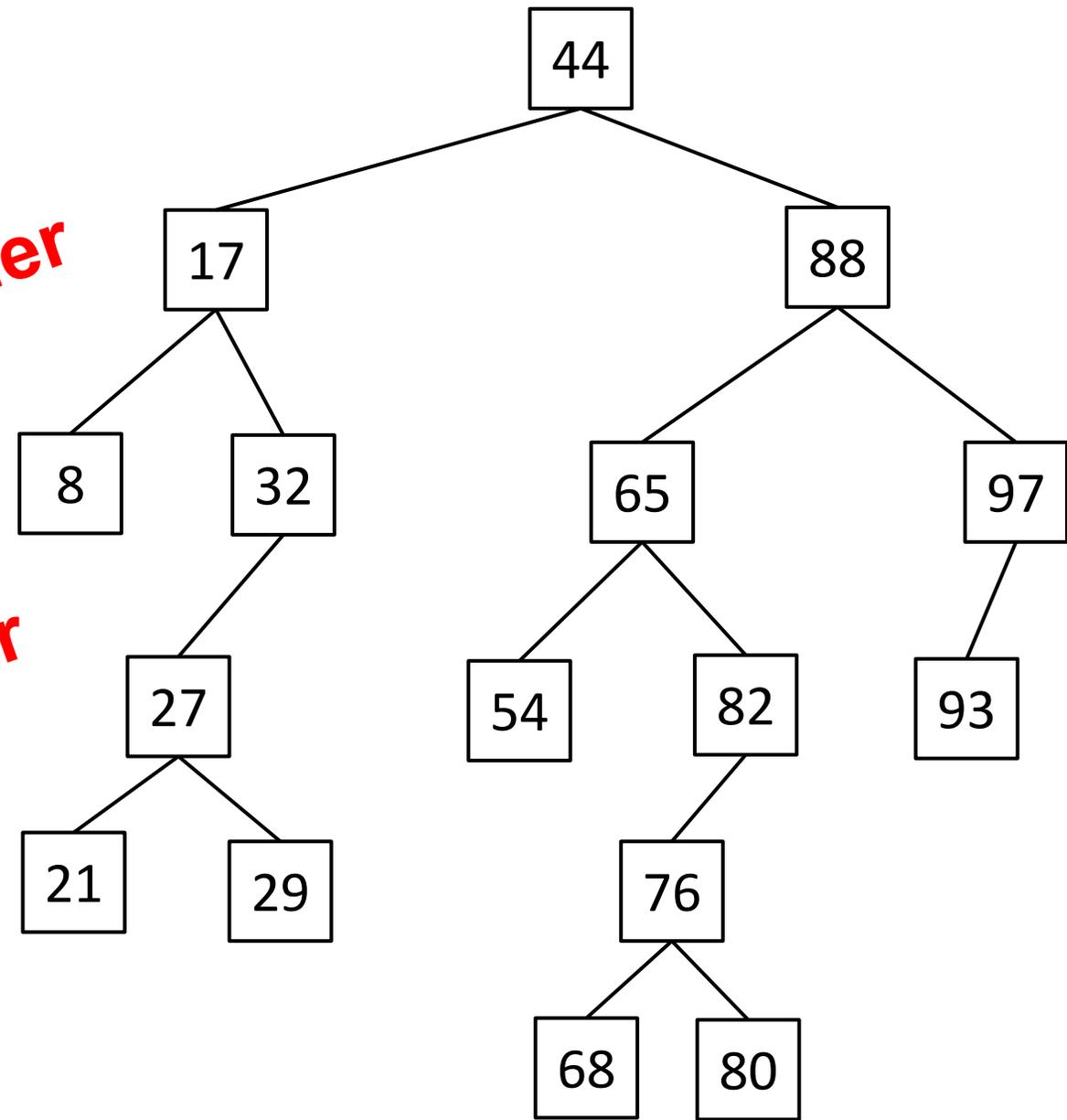


Binary Search Tree- Traversal

```
public void preorder(Node n) {  
    if(n != null) {  
        System.out.println(n.getValue());  
        preorder(n.getLeft());  
        preorder(n.getRight());  
    }  
}  
  
public void inorder(Node n) {  
    if(n != null) {  
        inorder(n.getLeft());  
        System.out.println(n.getValue());  
        inorder(n.getRight());  
    }  
}
```

Preorder

Inorder



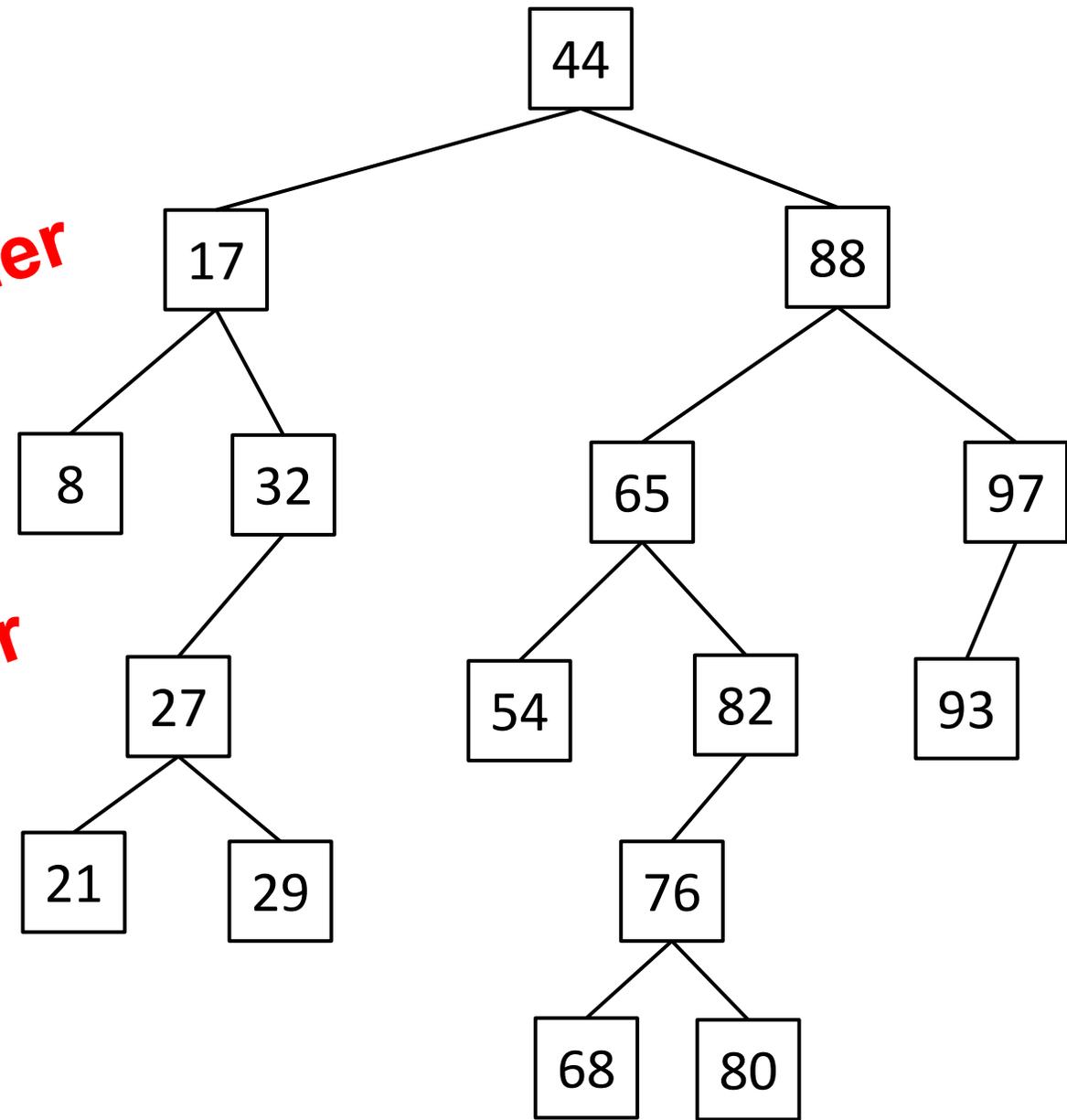
Binary Search Tree- Traversal

```
public void preorder(Node n) {  
    if(n != null) {  
        System.out.println(n.getValue());  
        preorder(n.getLeft());  
        preorder(n.getRight());  
    }  
}  
  
public void inorder(Node n) {  
    if(n != null) {  
        inorder(n.getLeft());  
        System.out.println(n.getValue());  
        inorder(n.getRight());  
    }  
}  
  
public void postorder(Node n) {  
    if(n != null) {  
        postorder(n.getLeft());  
        postorder(n.getRight());  
        System.out.println(n.getValue());  
    }  
}
```

Preorder

Inorder

Postorder



Binary Search Tree- Traversal

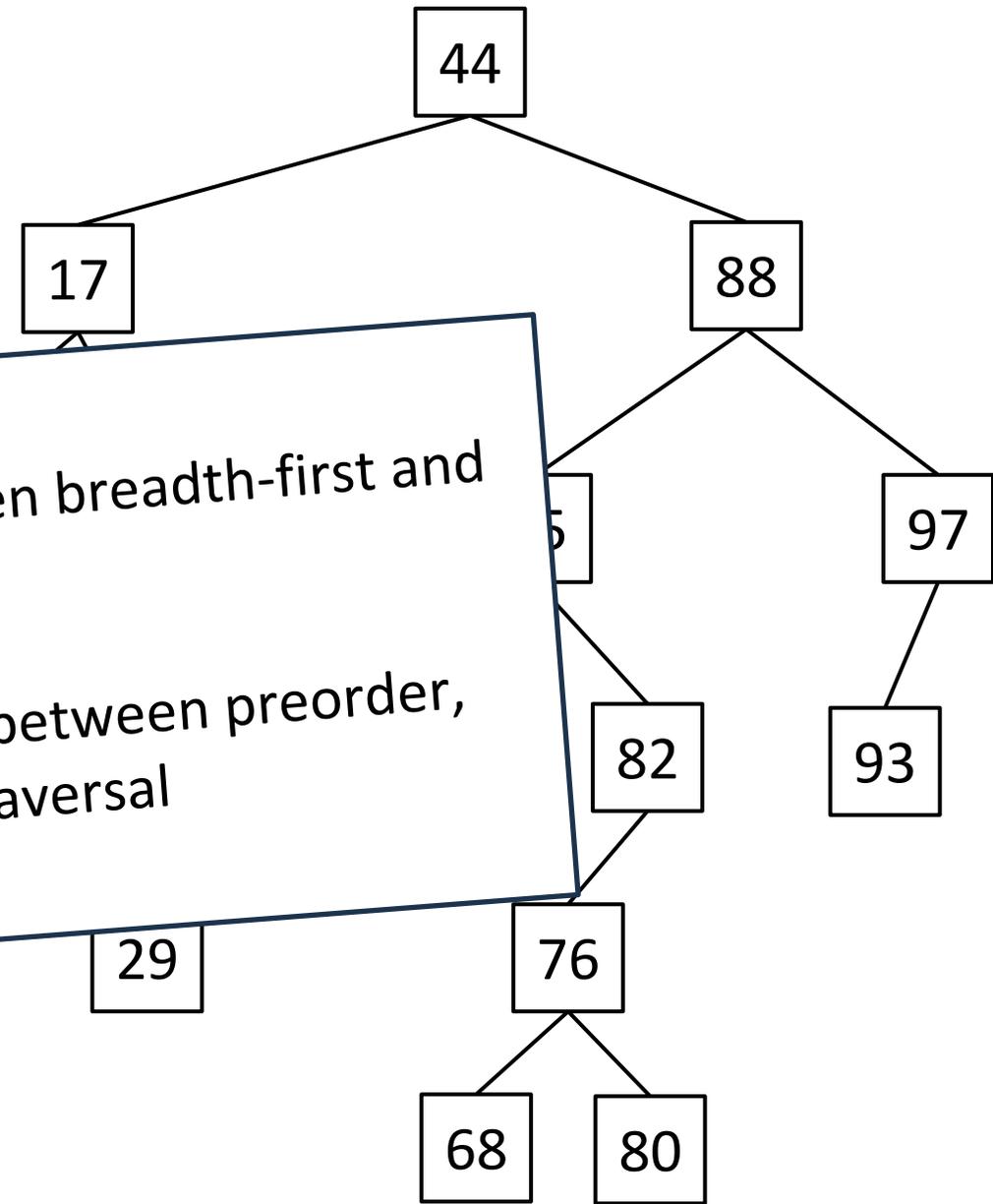
```
public void preorder(Node n) {  
    if(n != null) {  
        System.out.println(n.getValue());  
        preorder(n.getLeft());  
        preorder(n.getRight());  
    }  
}  
  
public void inorder(Node n) {  
    if(n != null) {  
        inorder(n.getLeft());  
        System.out.println(n.getValue());  
        inorder(n.getRight());  
    }  
}  
  
public void postorder(Node n) {  
    if(n != null) {  
        postorder(n.getLeft());  
        postorder(n.getRight());  
        System.out.println(n.getValue());  
    }  
}
```

Preorder

You should know the difference between breadth-first and depth-first traversal

You should also know the difference between preorder, inorder, and postorder traversal

Postorder



Binary Search Tree- Traversal

```
public void preorder(Node n) {  
    if(n != null) {  
        System.out.println(n.getValue());  
        preorder(n.getLeft());  
        preorder(n.getRight());  
    }  
}  
  
public void inorder(Node n) {  
    if(n != null) {  
        inorder(n.getLeft());  
        System.out.println(n.getValue());  
        inorder(n.getRight());  
    }  
}  
  
public void postorder(Node n) {  
    if(n != null) {  
        postorder(n.getLeft());  
        postorder(n.getRight());  
        System.out.println(n.getValue());  
    }  
}
```

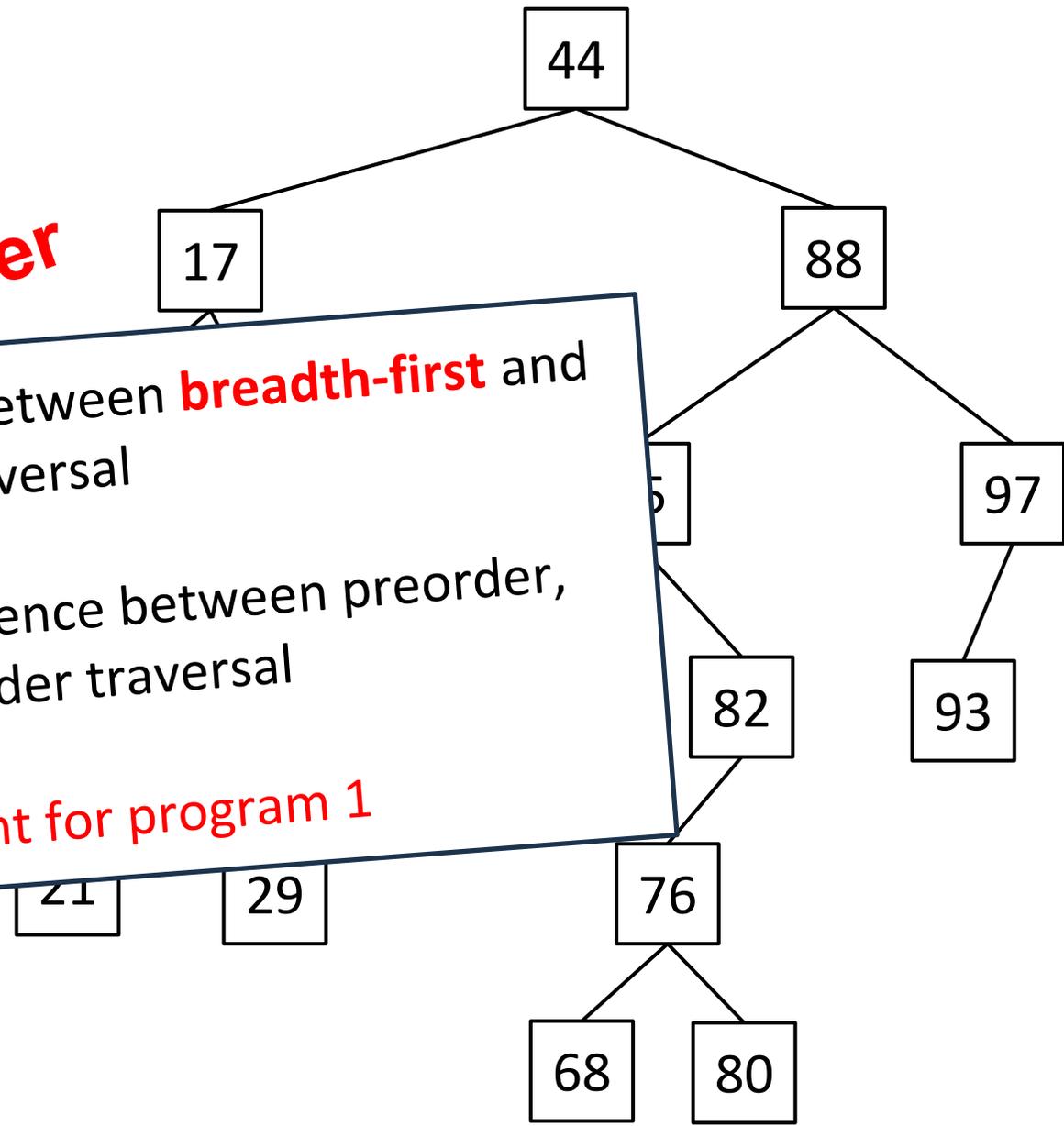
Preorder

You should know the difference between **breadth-first** and depth-first traversal

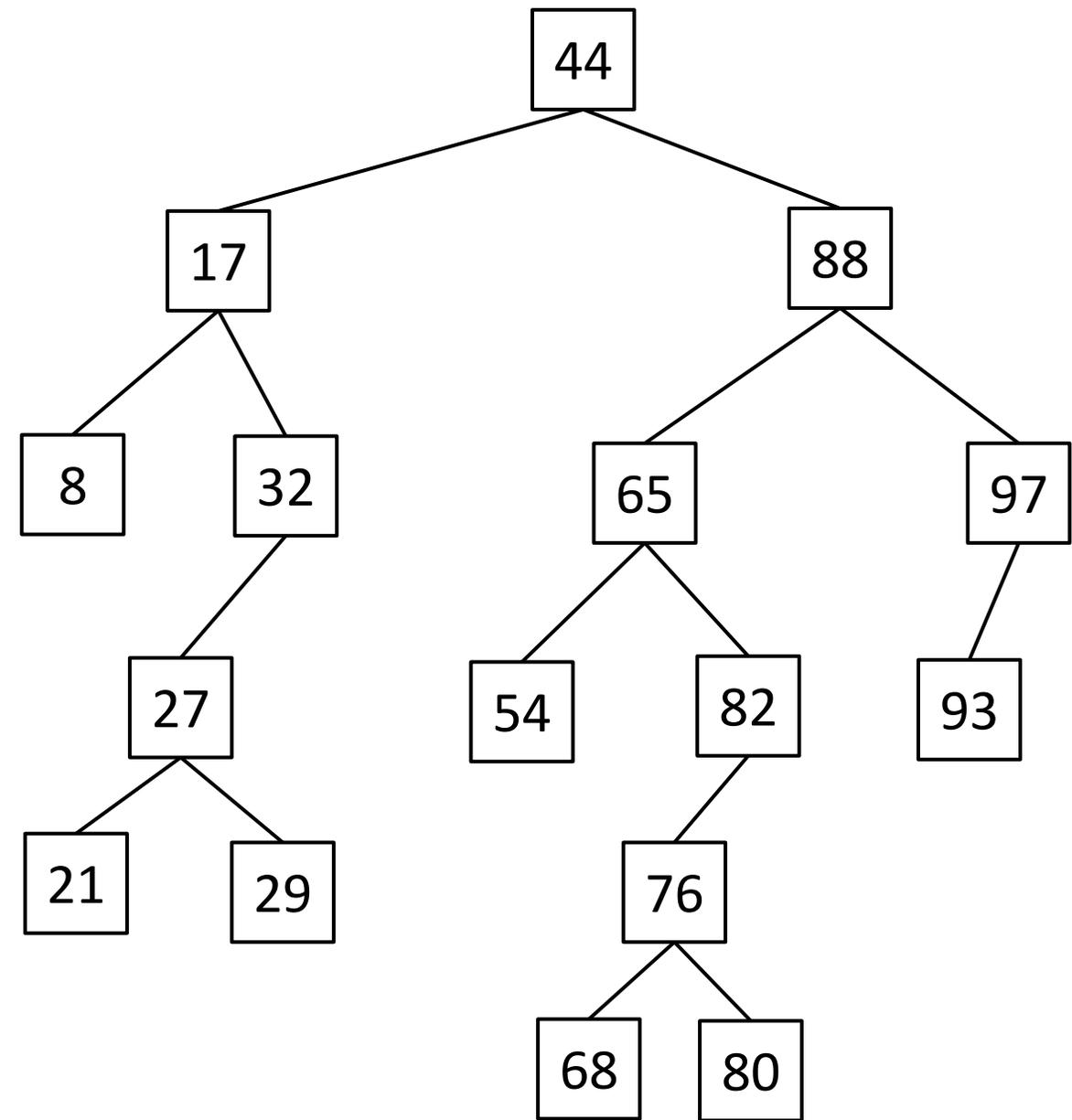
You should also know the difference between preorder, **inorder**, and postorder traversal

These will be important for program 1

Postorder



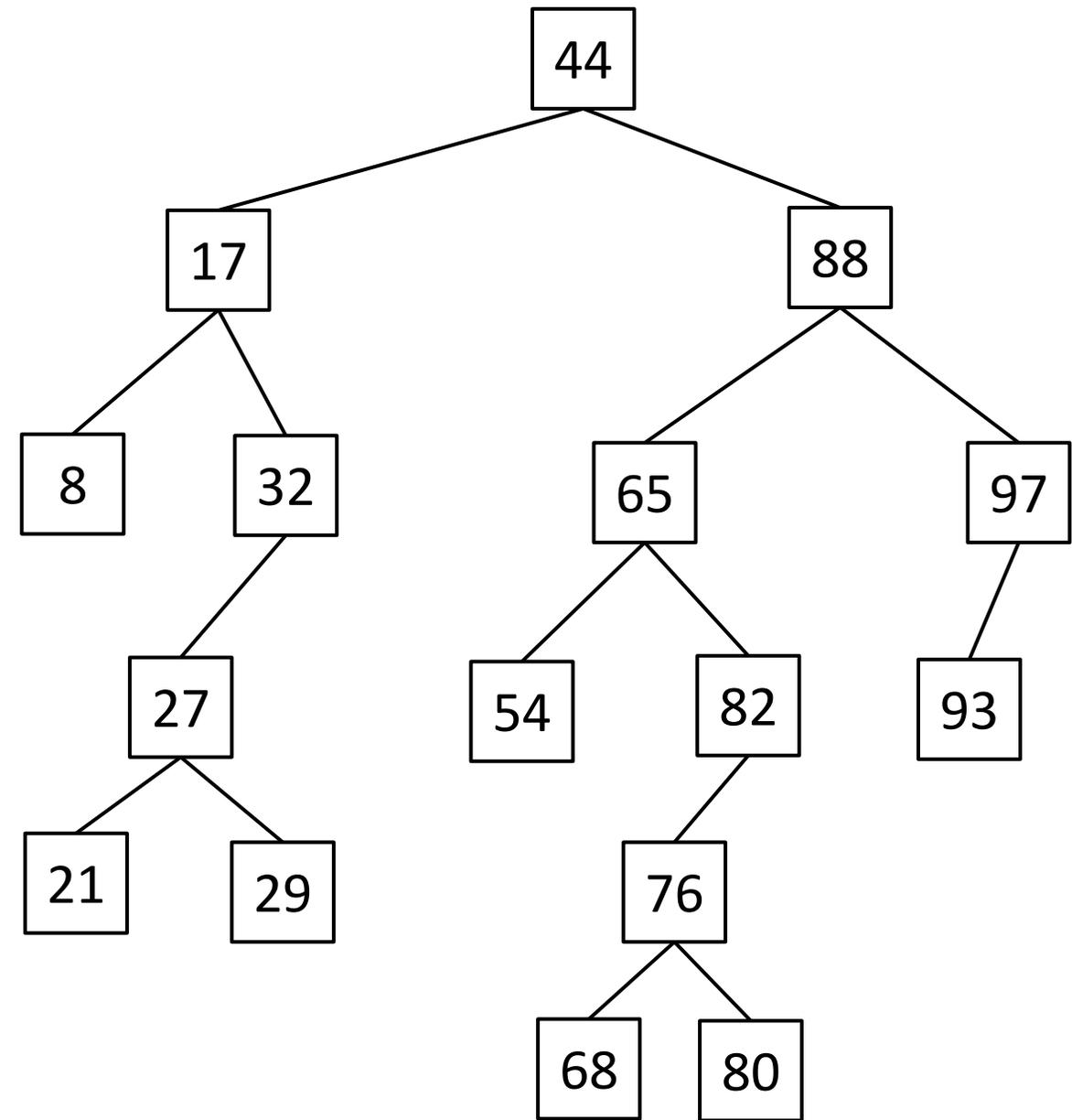
Binary Search Tree- Removal



Binary Search Tree- Removal

`remove(68);`

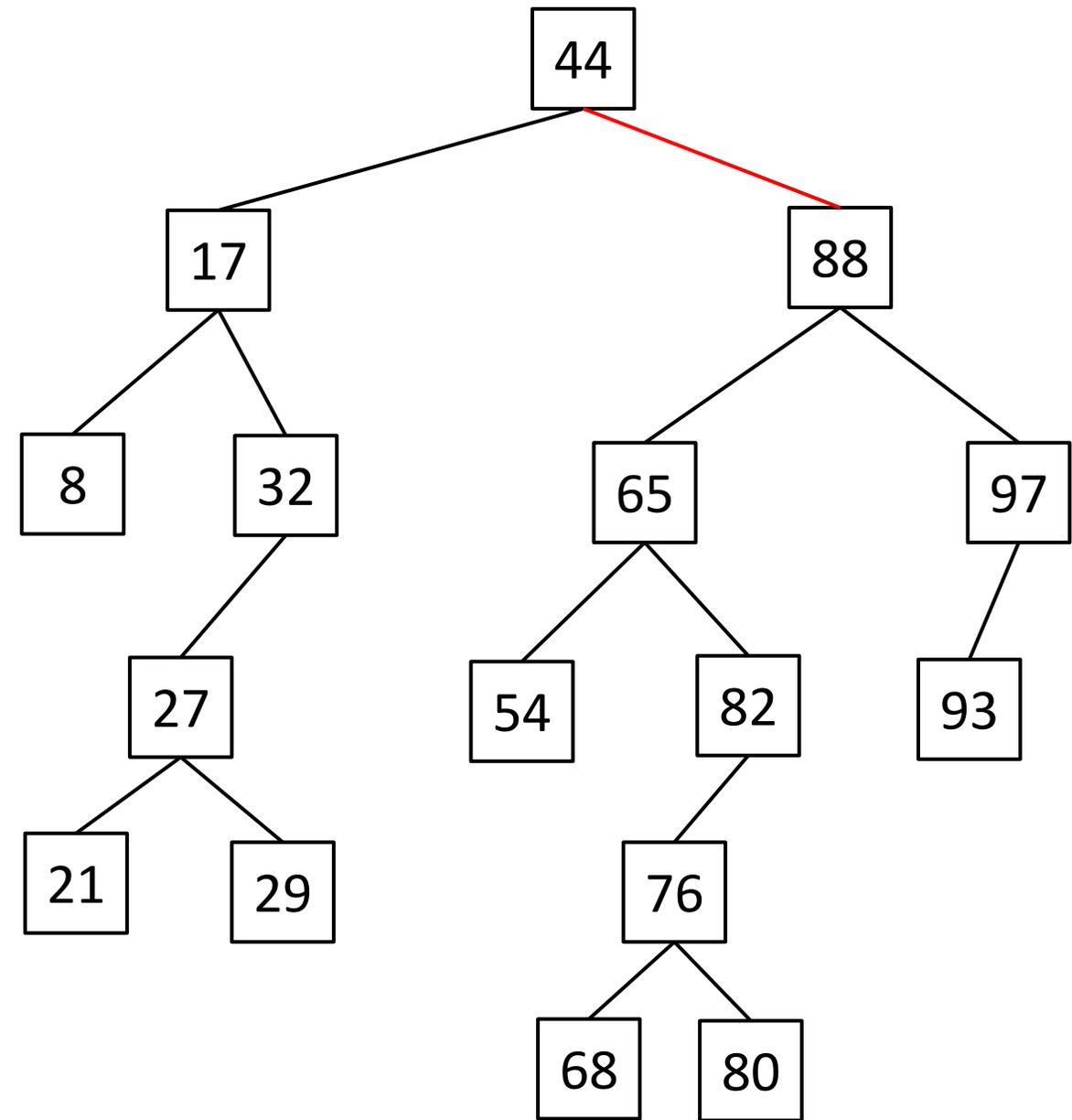
Step 1: Find the node in the tree



Binary Search Tree- Removal

`remove(68);`

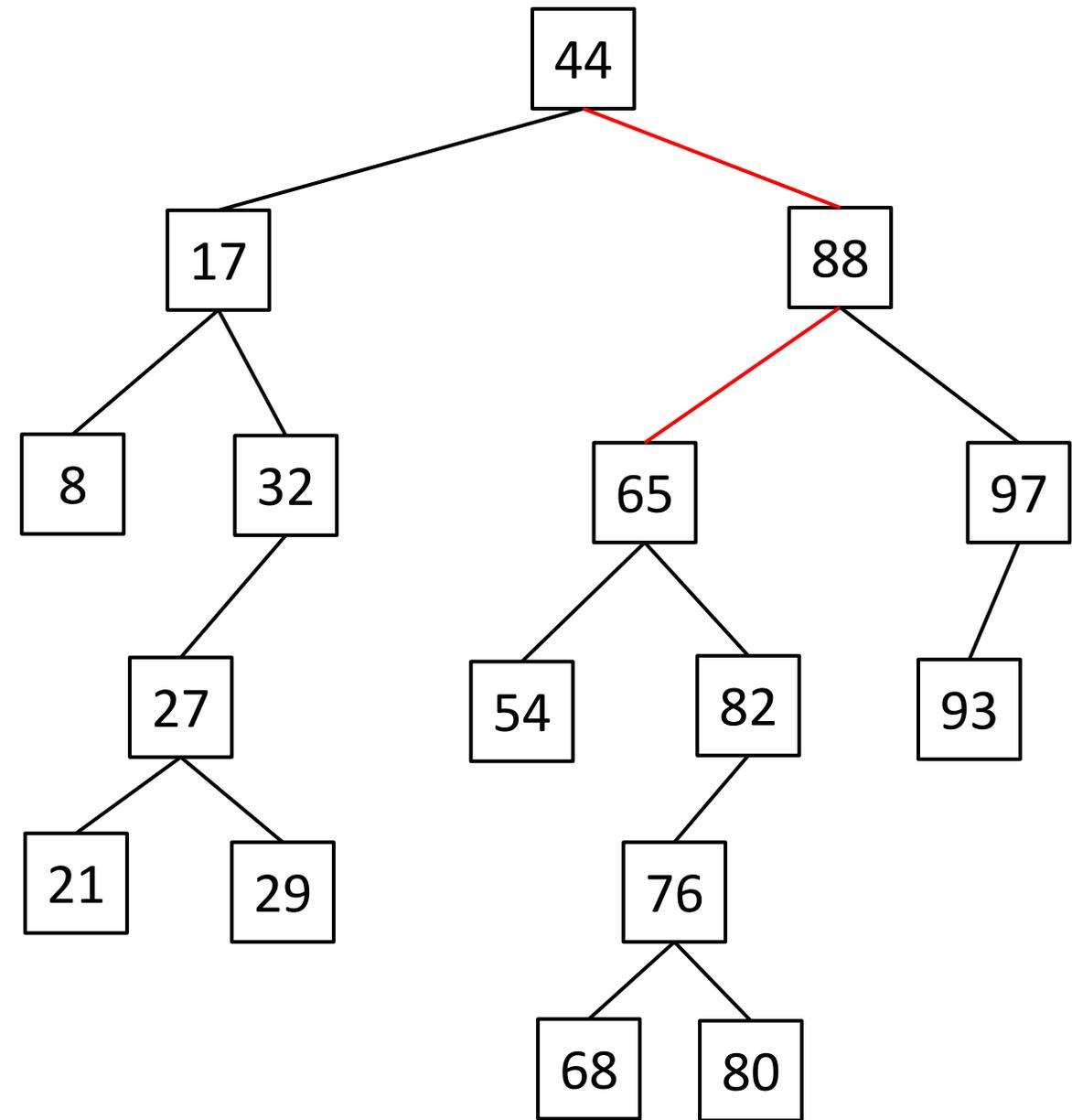
Step 1: Find the node in the tree



Binary Search Tree- Removal

`remove(68);`

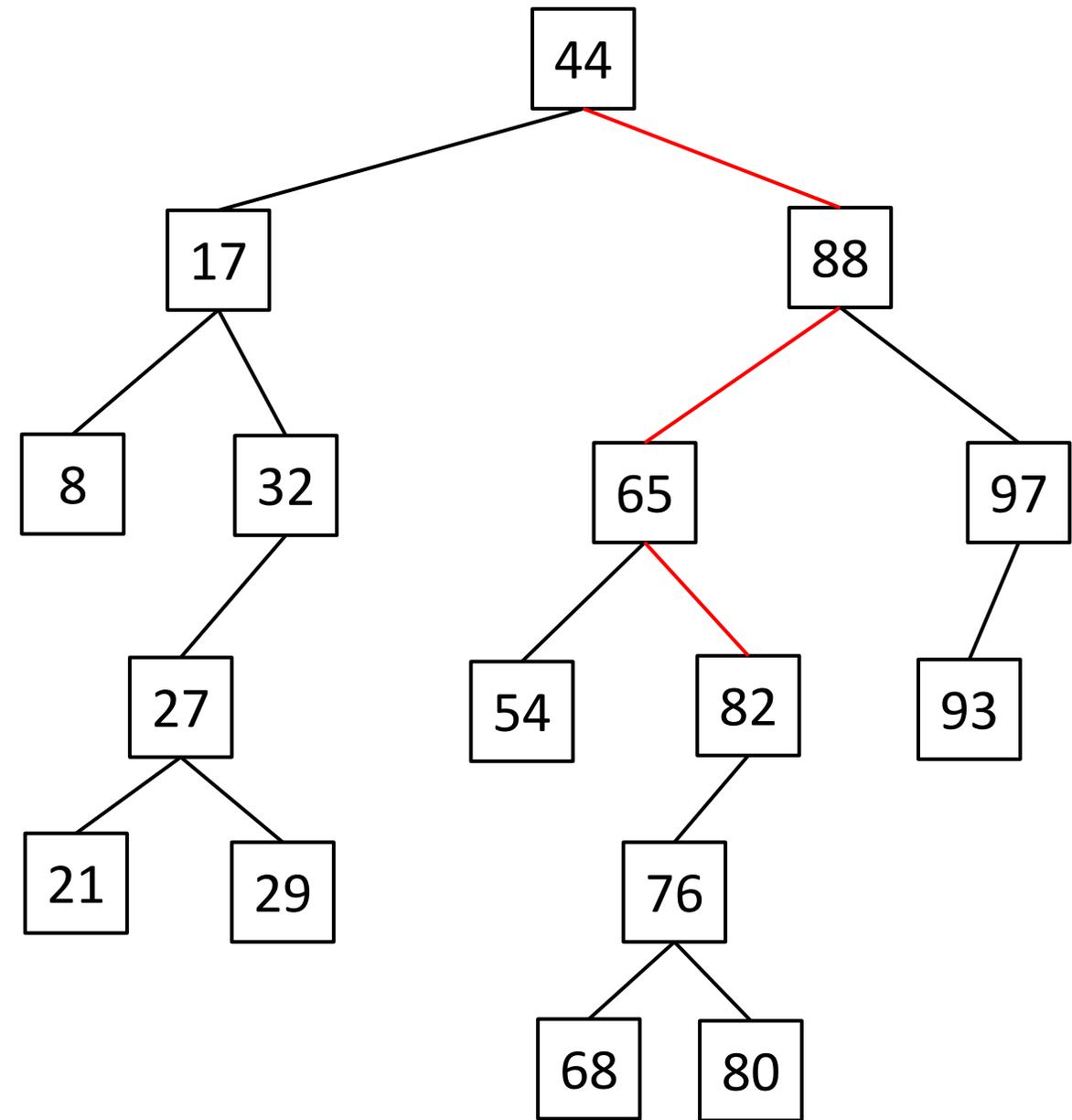
Step 1: Find the node in the tree



Binary Search Tree- Removal

`remove(68);`

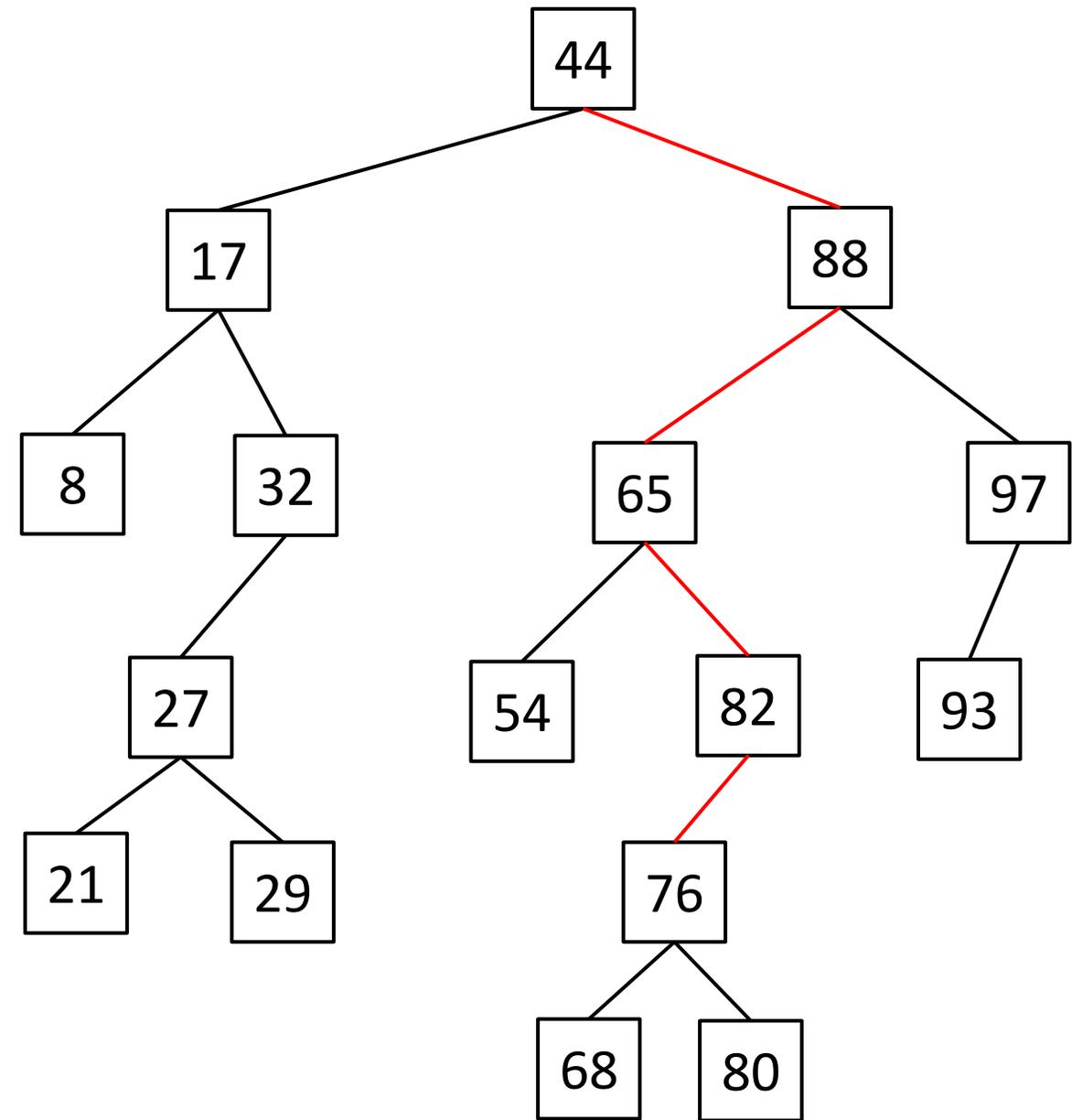
Step 1: Find the node in the tree



Binary Search Tree- Removal

`remove(68);`

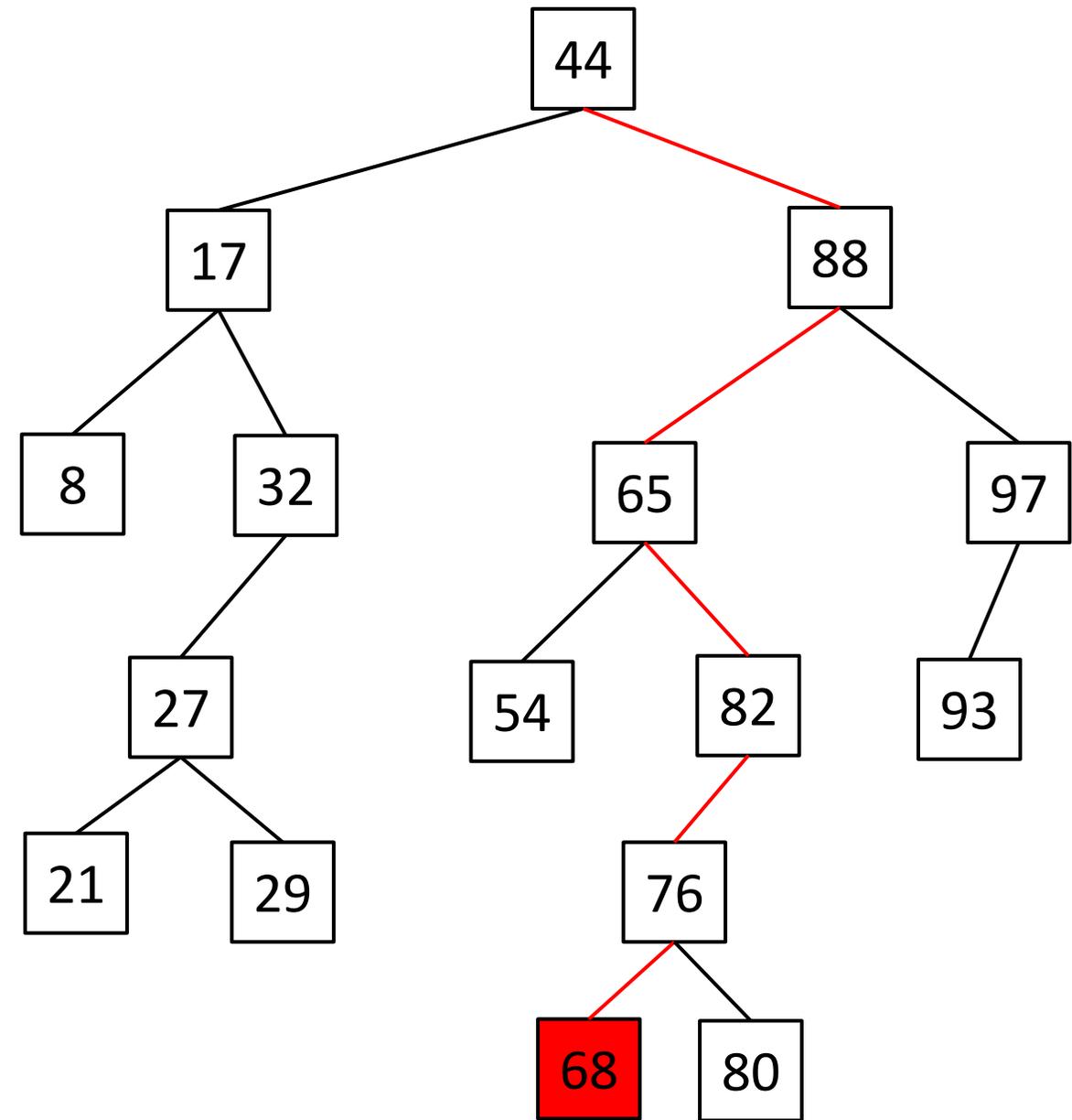
Step 1: Find the node in the tree



Binary Search Tree- Removal

`remove(68);`

Step 1: Find the node in the tree

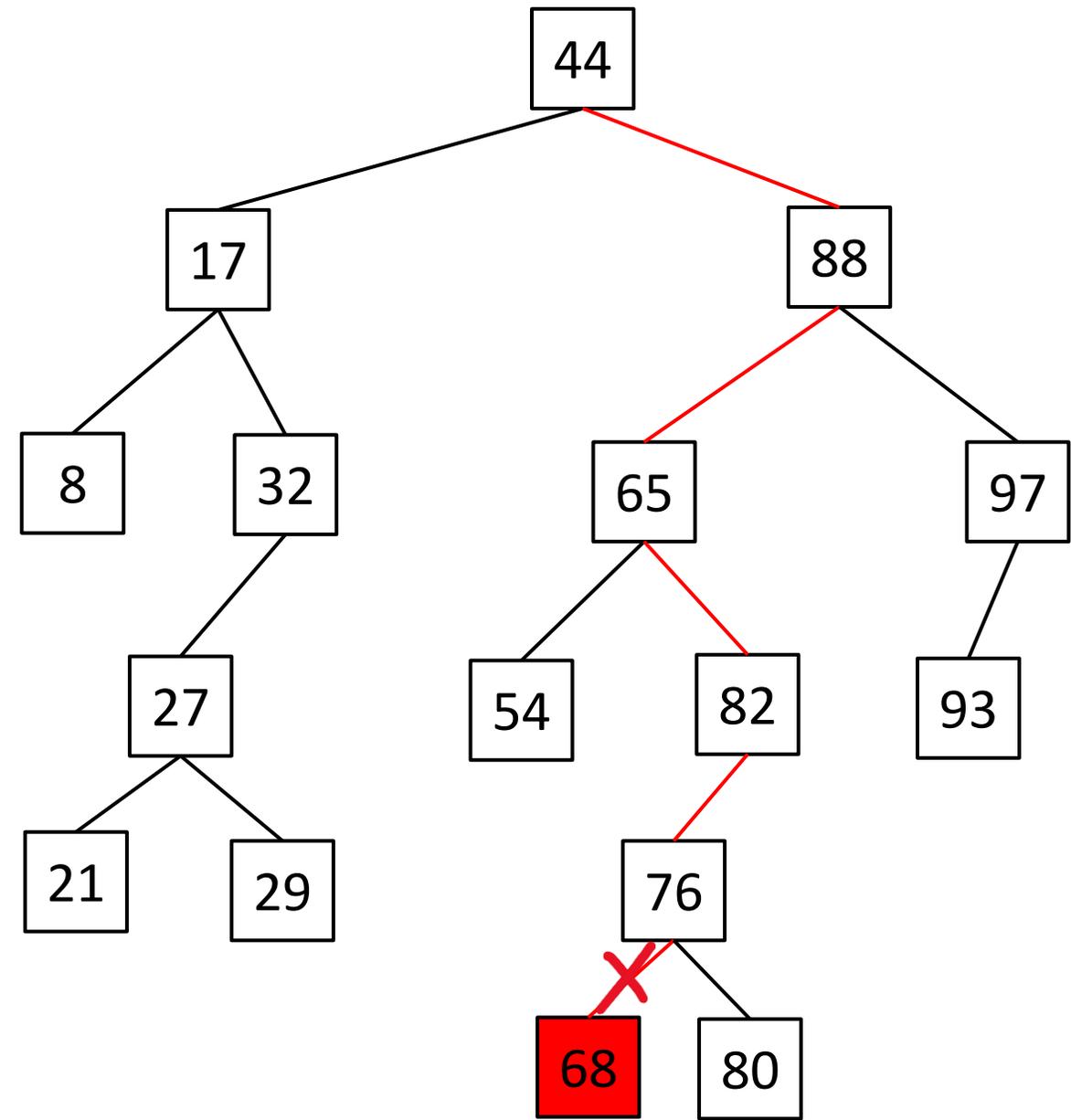


Binary Search Tree- Removal

`remove(68);`

Step 1: Find the node in the tree

Step 2: Change parent to point to null



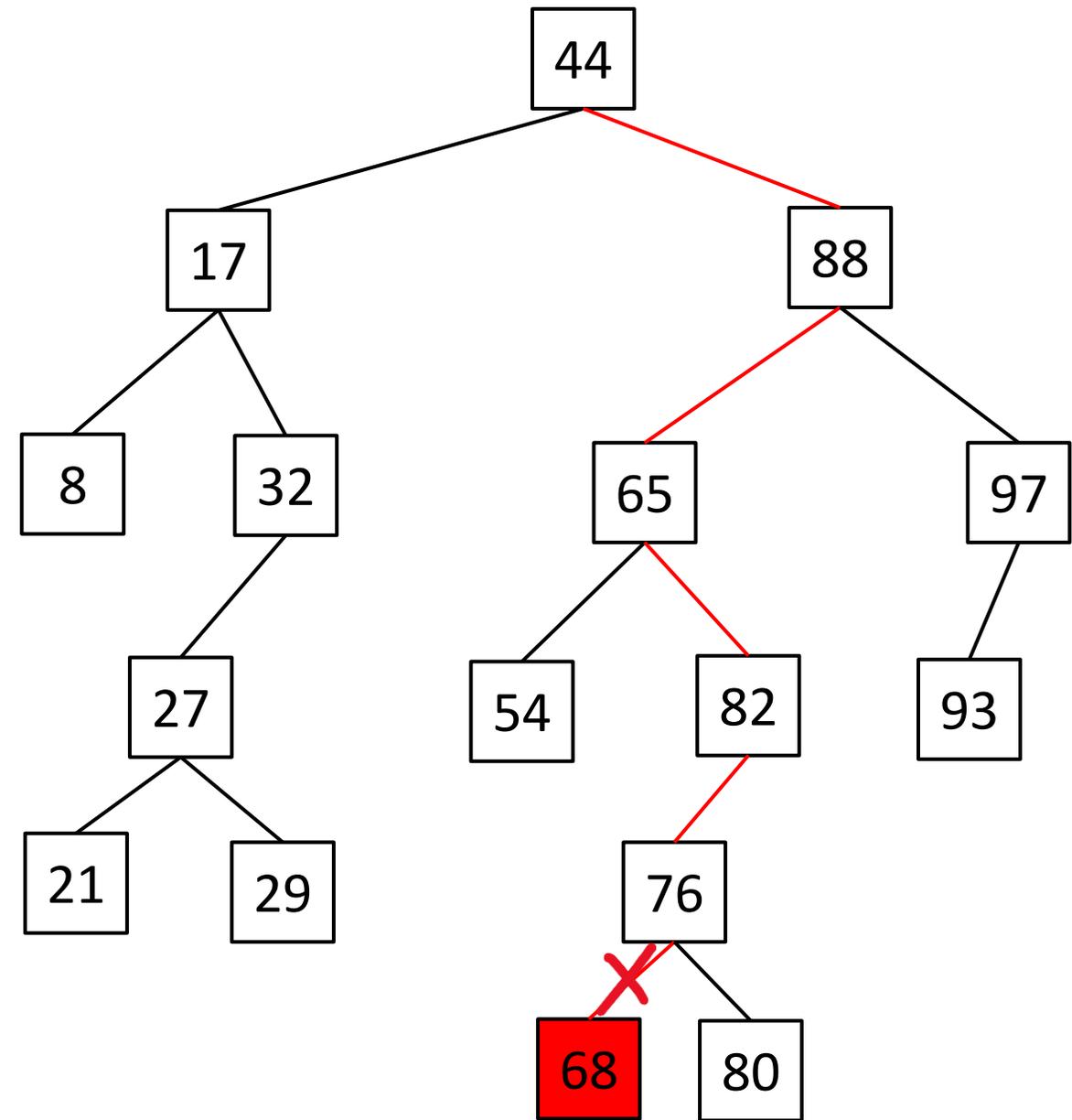
Binary Search Tree- Removal

`remove(68);`

Step 1: Find the node in the tree

Step 2: Change parent to point to null

Does this always work?

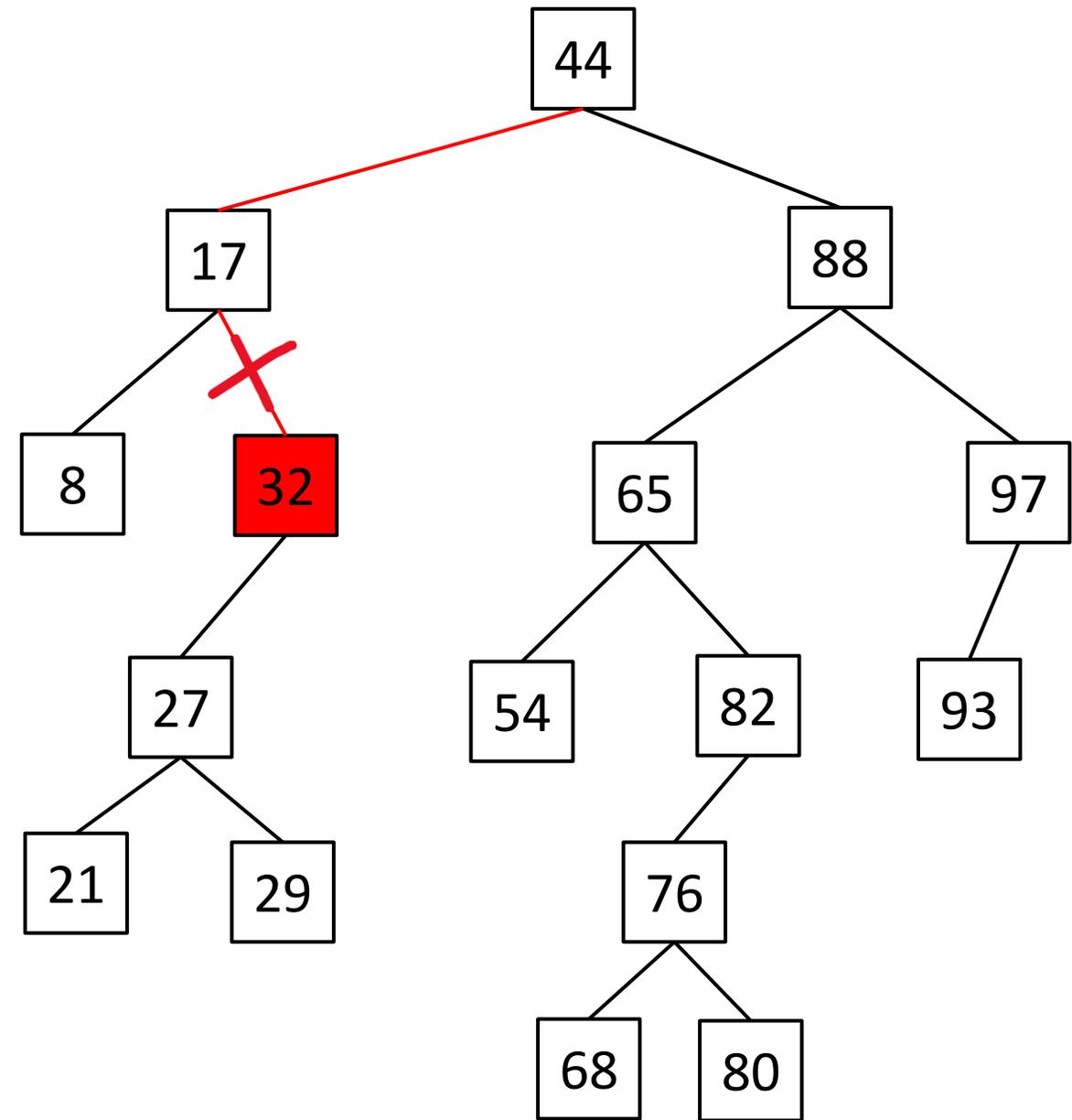


Binary Search Tree- Removal

`remove(32);`

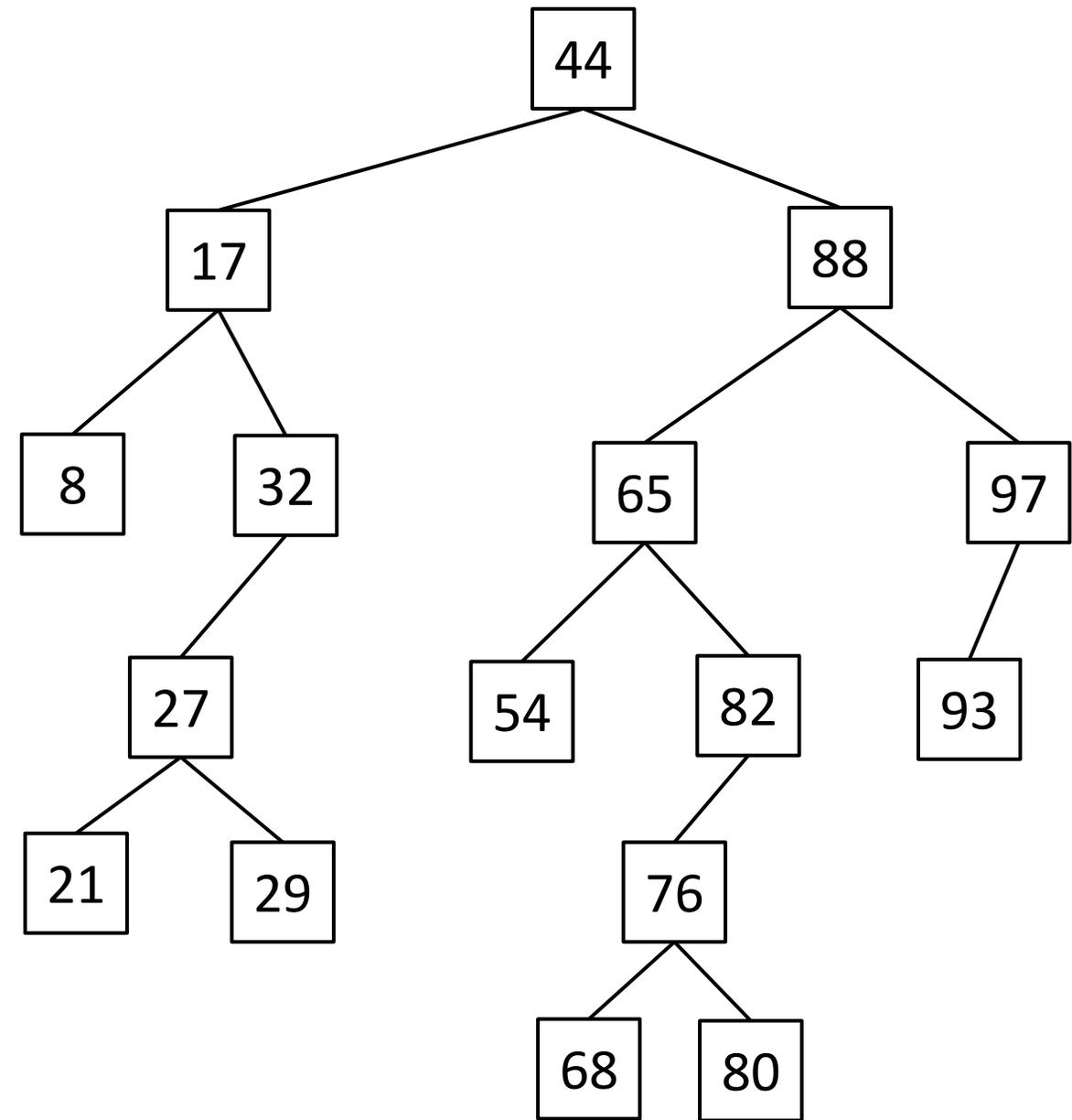
- Step 1: Find the node in the tree
- Step 2: Change parent to point to null

This does not always work



Binary Search Tree- Removal

- Case 1: Node has no children
- Case 2: Node has one child
- Case 3: Node has two children



Binary Search Tree- Removal

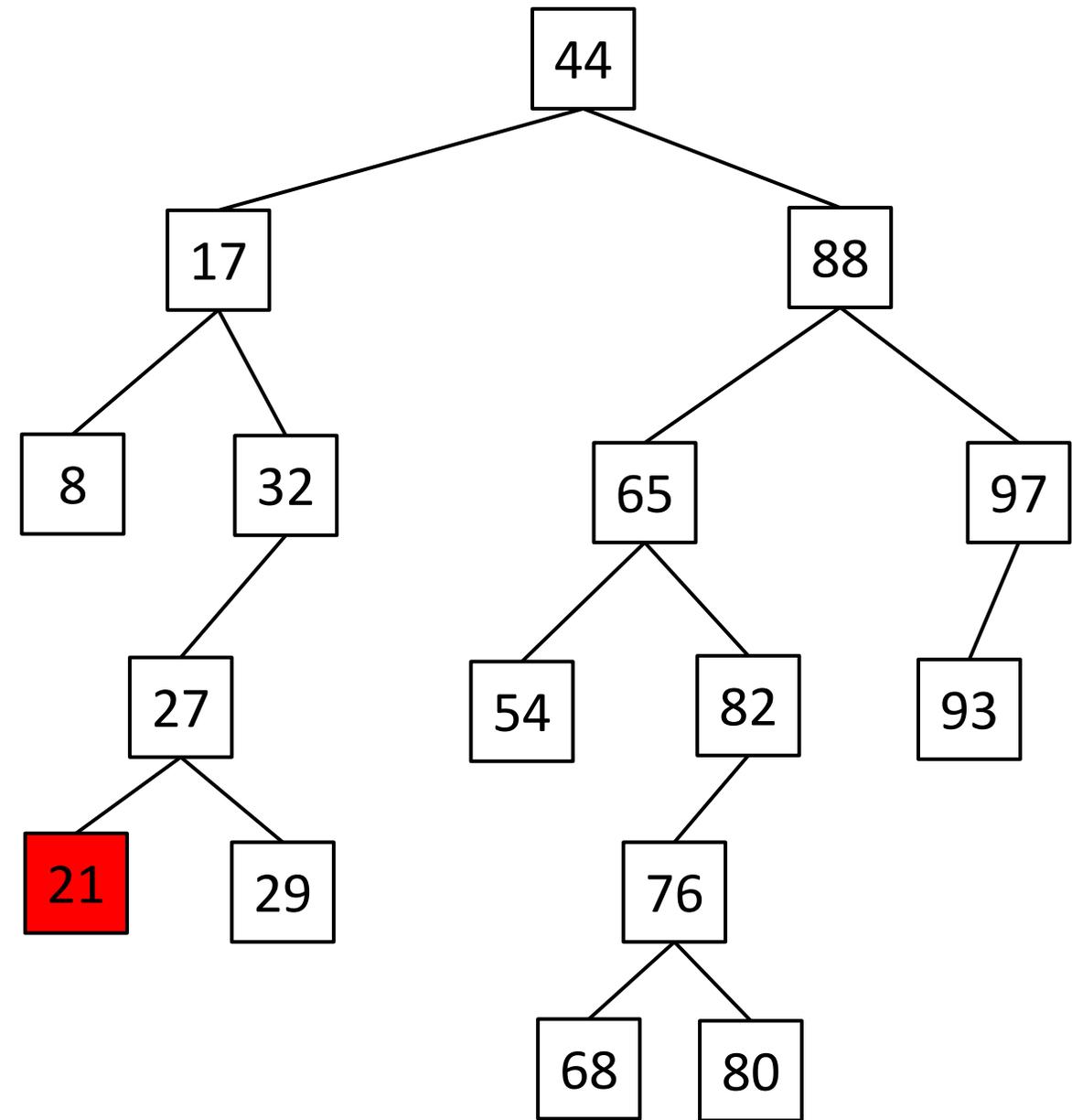
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(21);`

How do we know it has no children?



Binary Search Tree- Removal

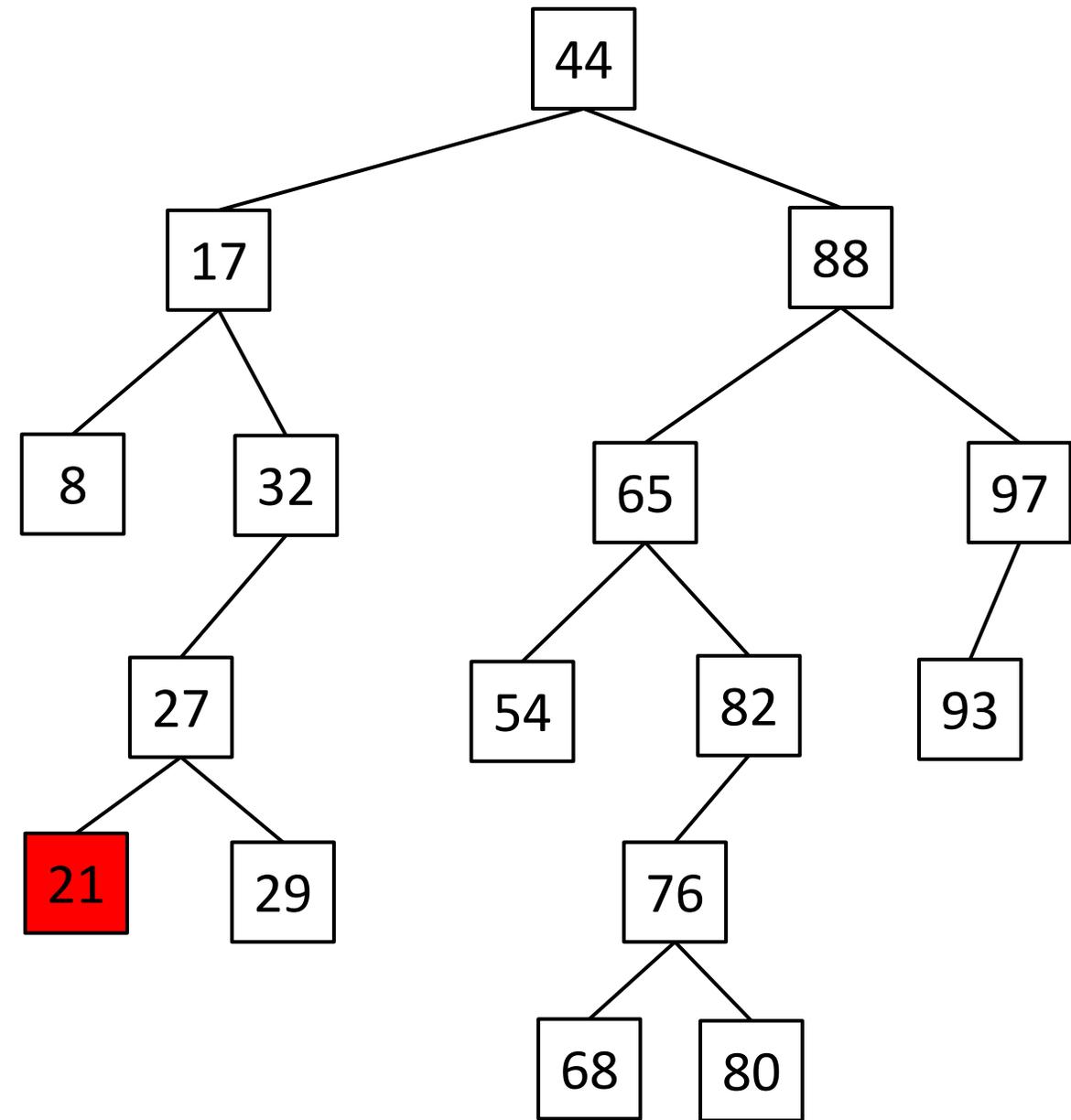
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(21);`

How do we know it has no children?
If its left and right child are both null



Binary Search Tree- Removal

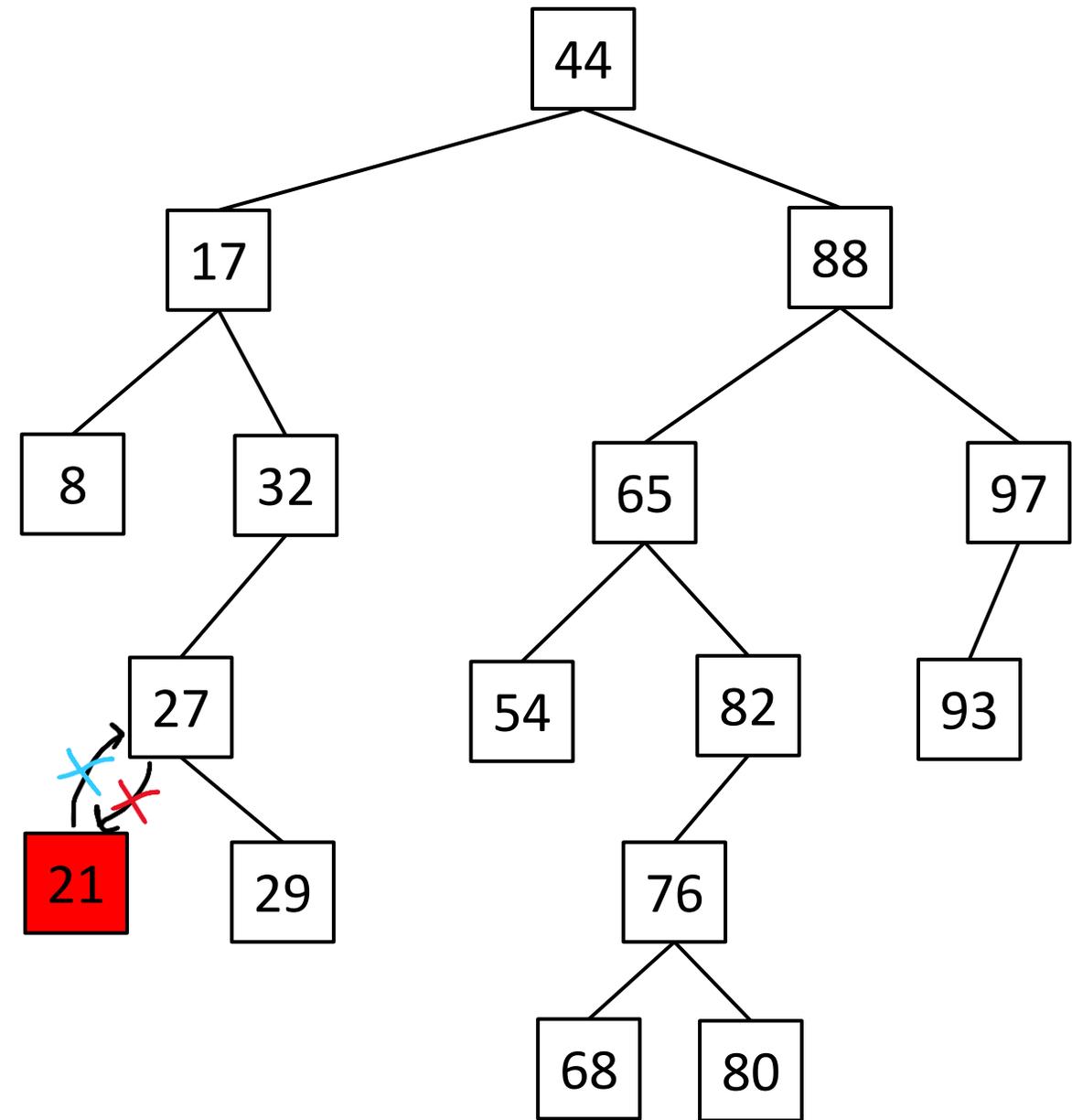
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(21);`

1. Update parent's **child** to point to **null**
2. Update Node's **parent** to point to **null**



Binary Search Tree- Removal

Case 1: Node has no children

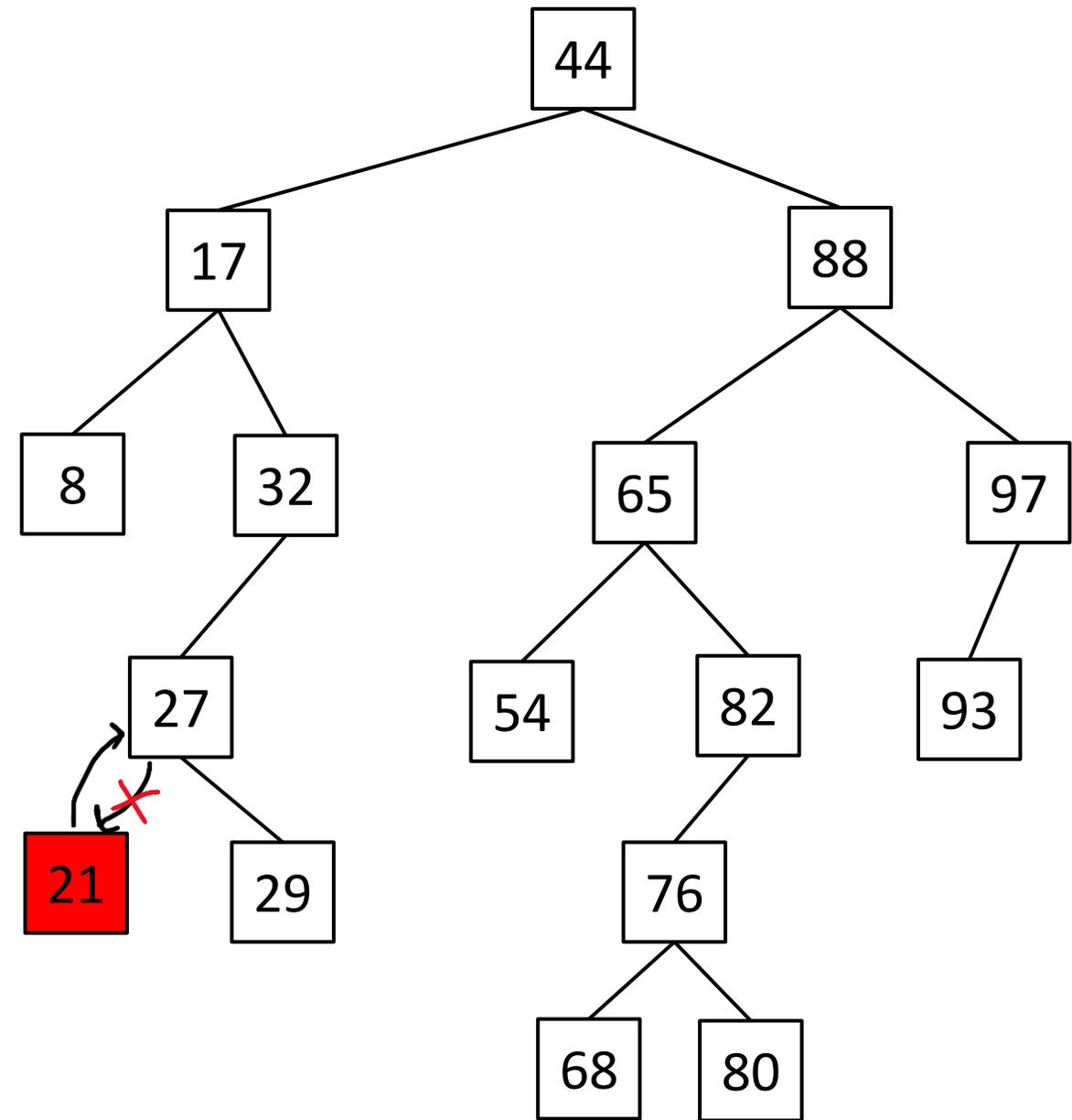
Case 2: Node has one child

Case 3: Node has two children

`remove(21);`

1. Update parent's **child** to point to **null**

2. Update Node's ~~parent~~ to point to **null**



Binary Search Tree- Removal

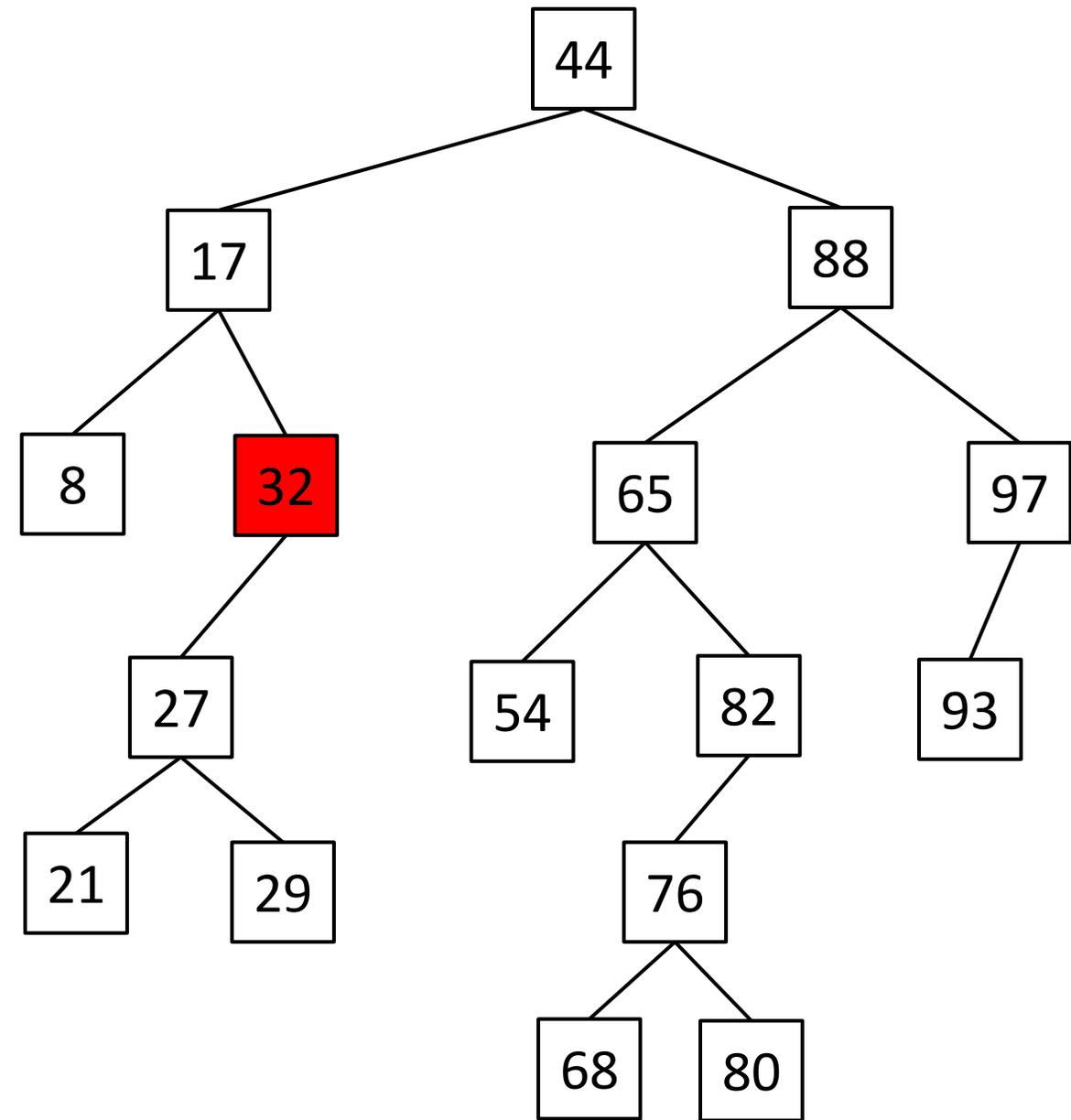
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(32);`

???



Binary Search Tree- Removal

Case 1: Node has no children

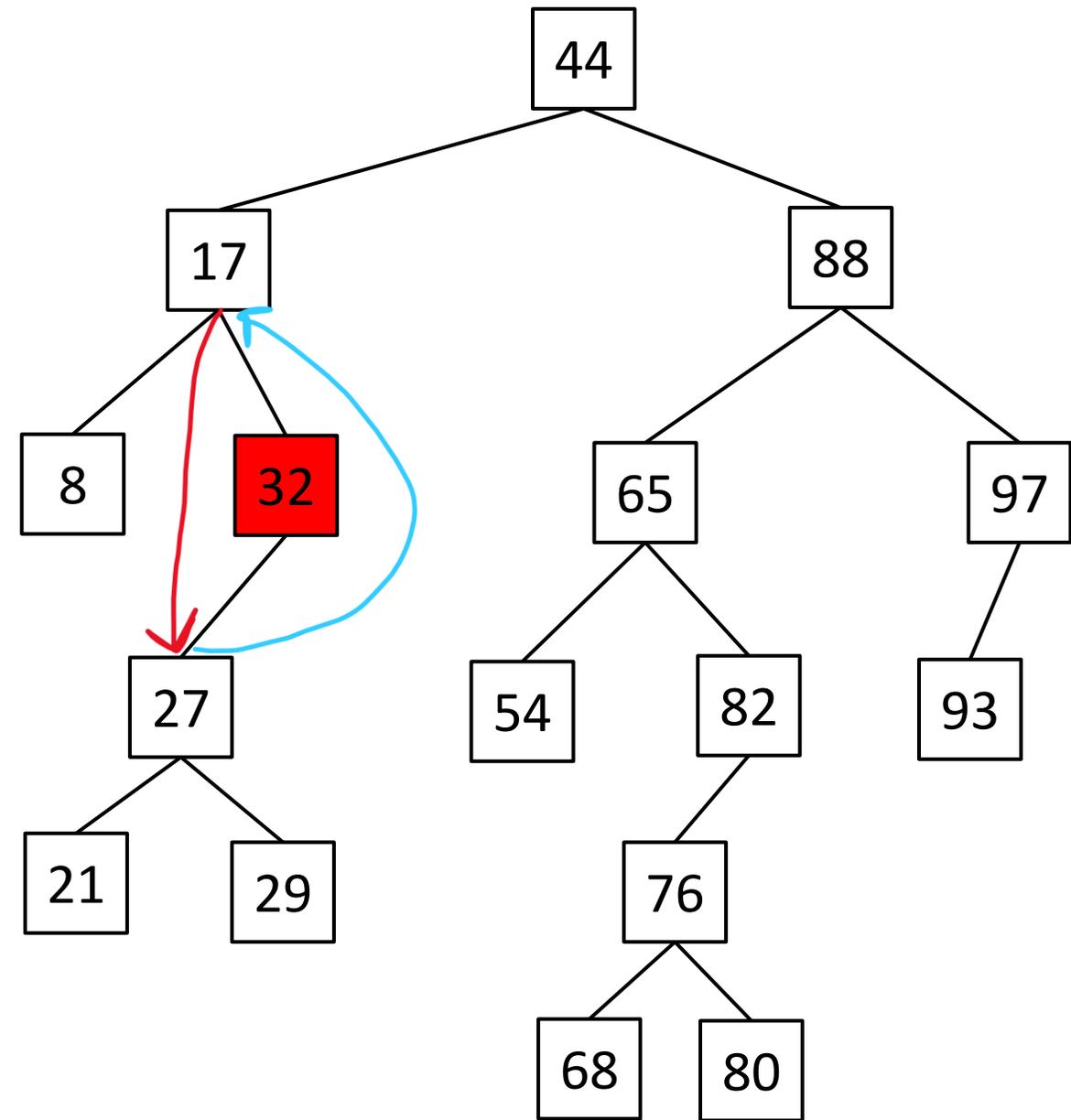
Case 2: Node has one child

Case 3: Node has two children

`remove(32);`

Change the Node's parent to point to the only child

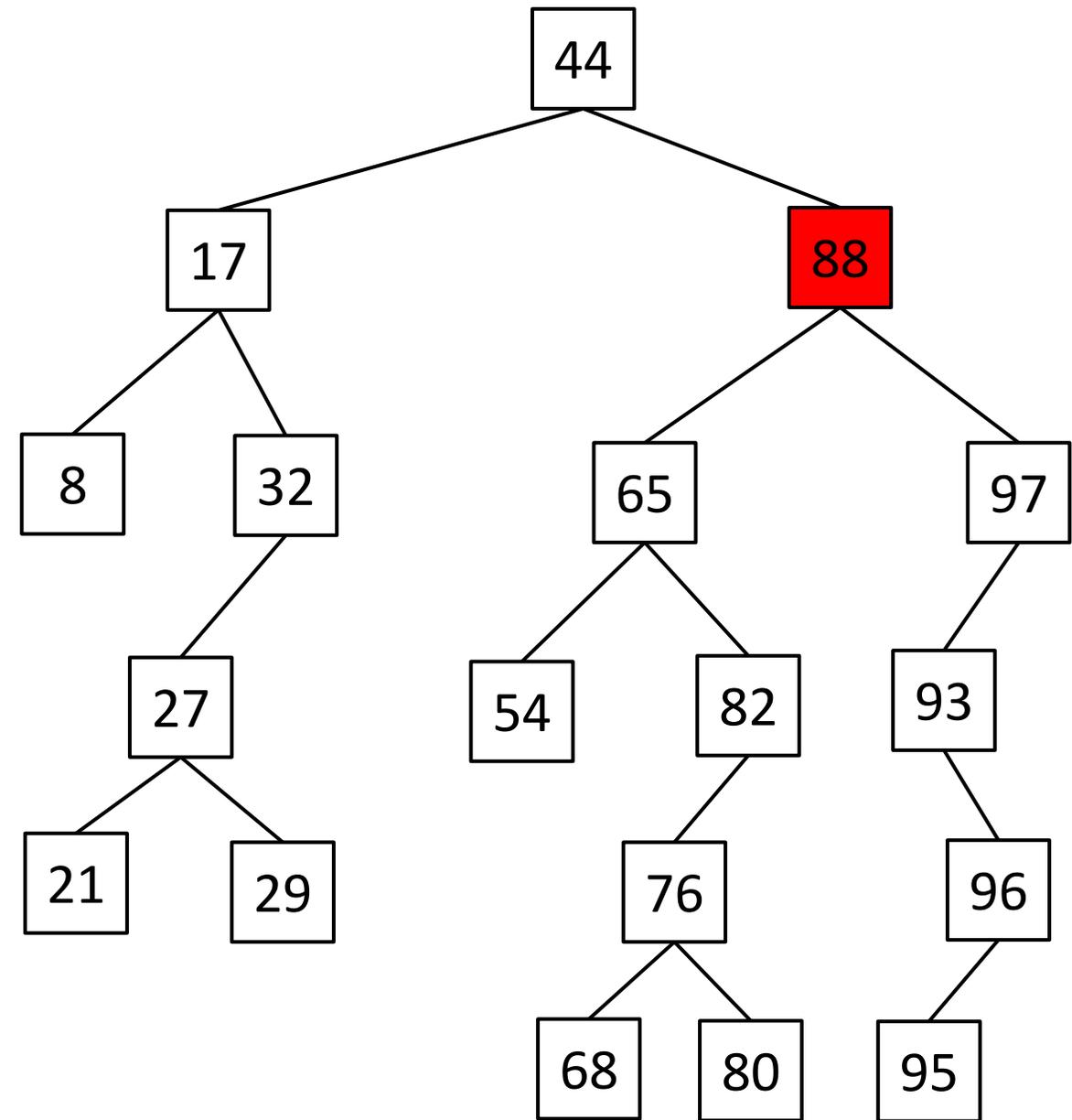
Update the Node's only child parent to point to the Node's Parent



Binary Search Tree- Removal

- Case 1: Node has no children
- Case 2: Node has one child
- Case 3: Node has two children

`remove(88);`



Binary Search Tree- Removal

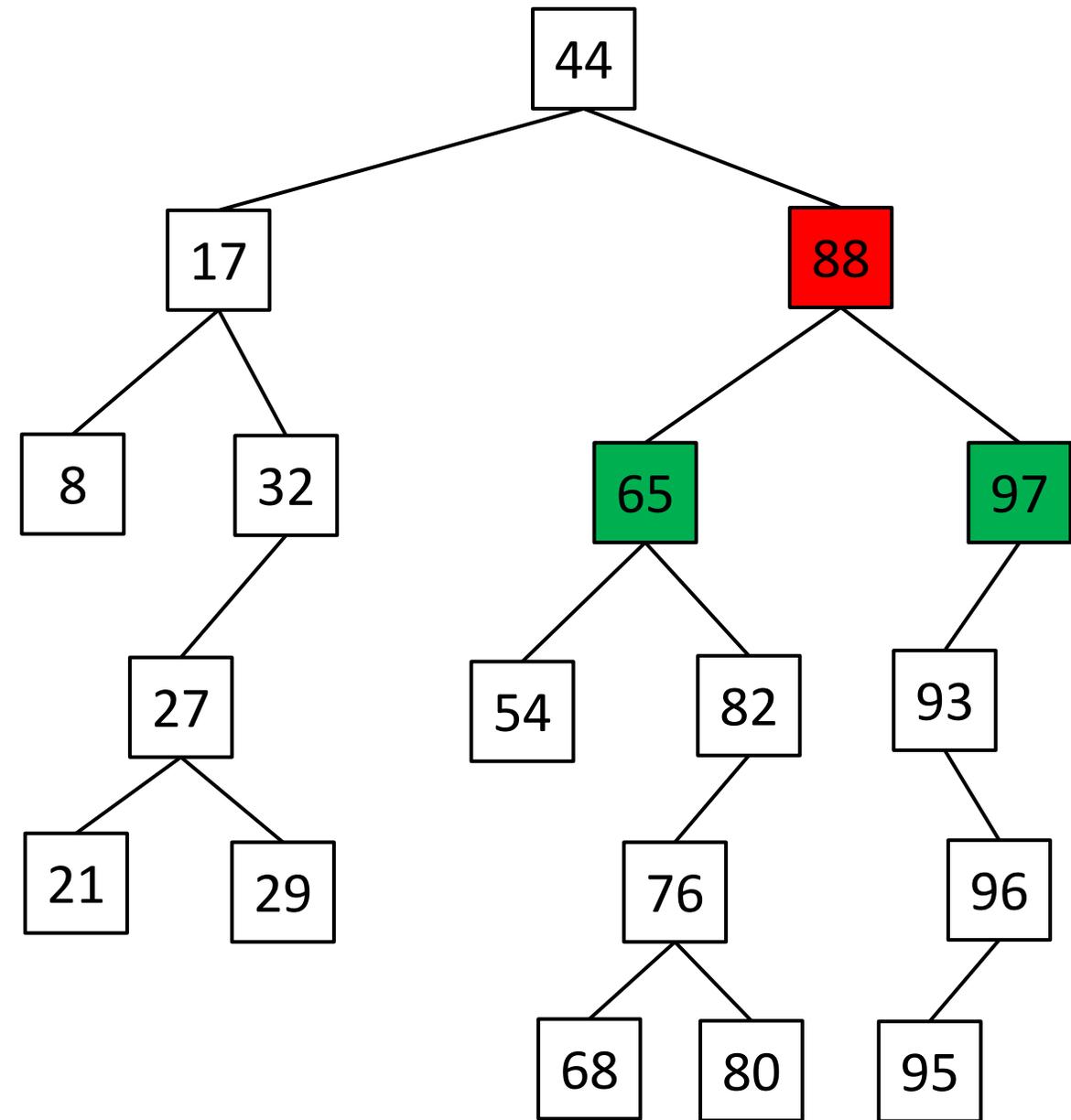
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(88);`

Which child to use?

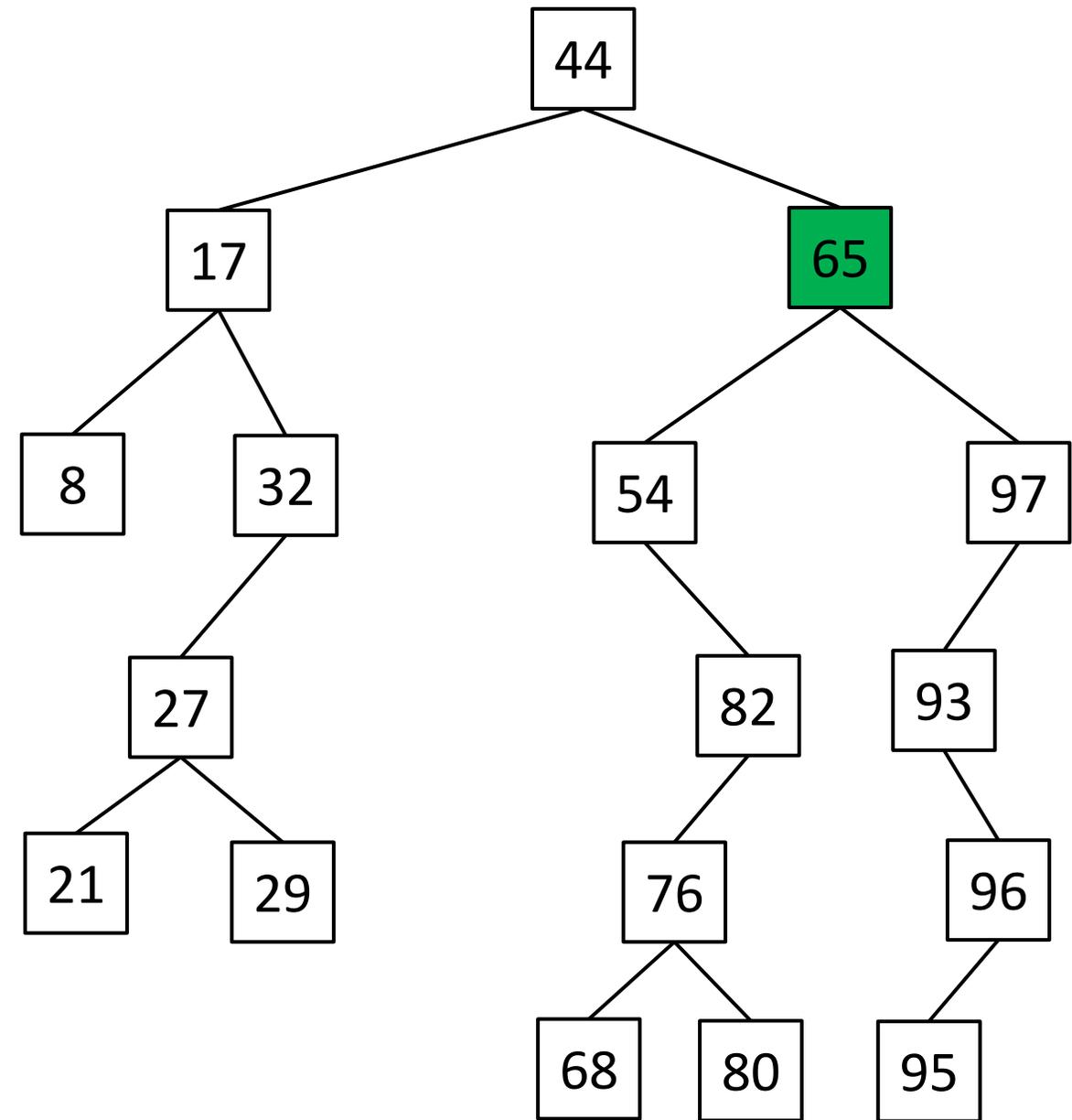


Binary Search Tree- Removal

- Case 1: Node has no children
- Case 2: Node has one child
- Case 3: Node has two children

`remove(88);`

Which child to use?



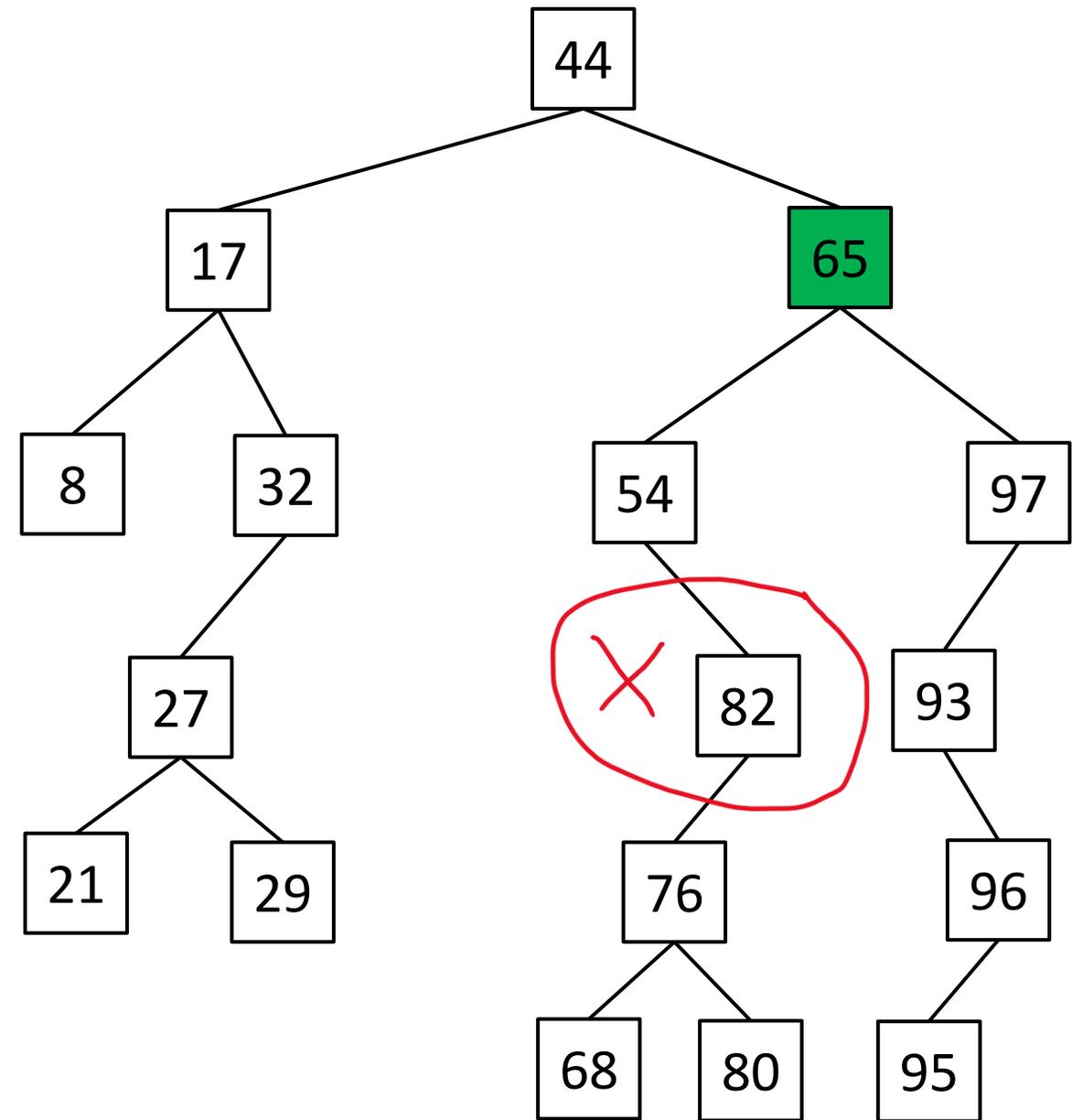
Binary Search Tree- Removal

- Case 1: Node has no children
- Case 2: Node has one child
- Case 3: Node has two children

`remove(88);`

Which child to use?

Left child
doesn't work!



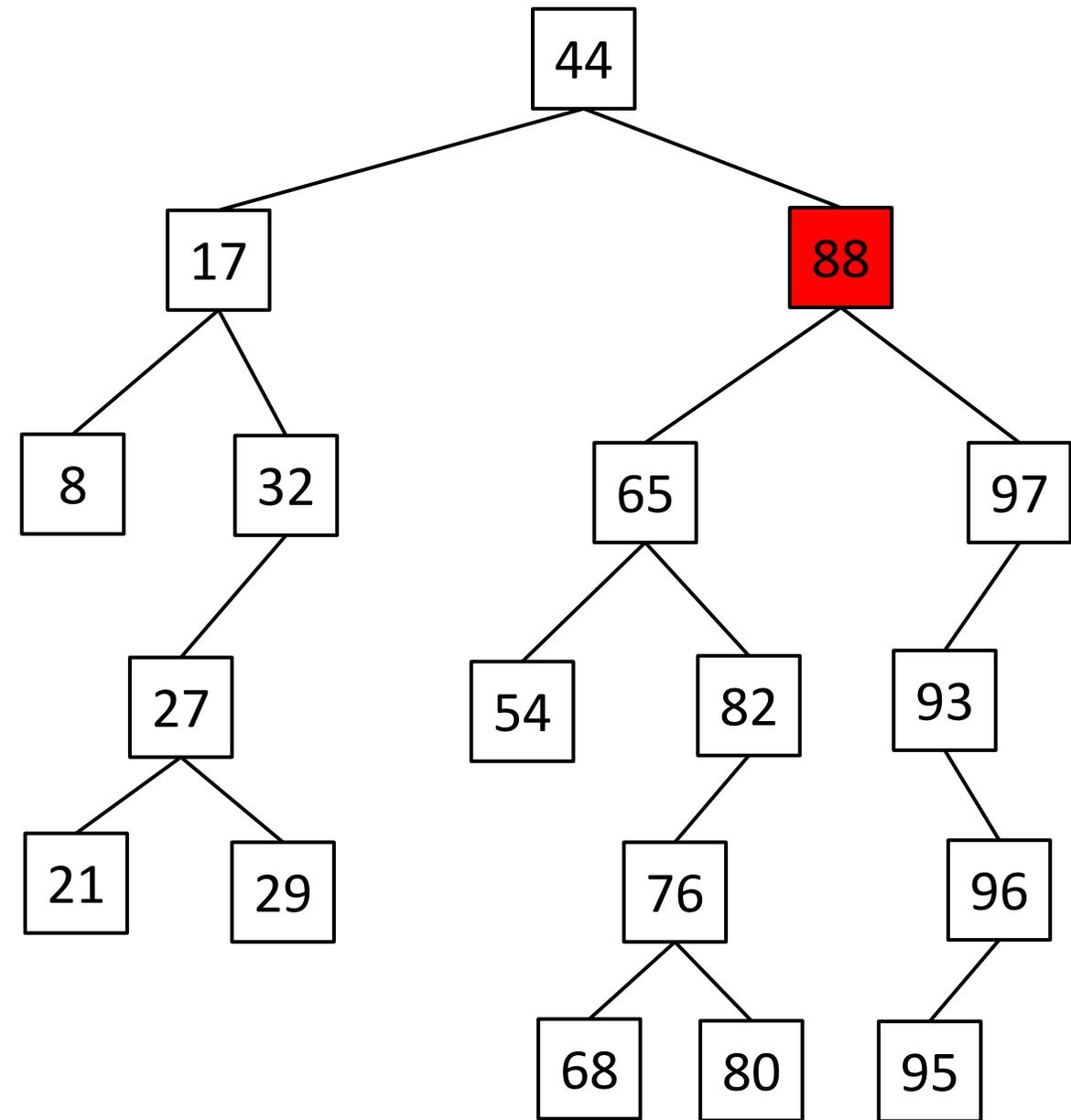
Binary Search Tree- Removal

Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(88);`



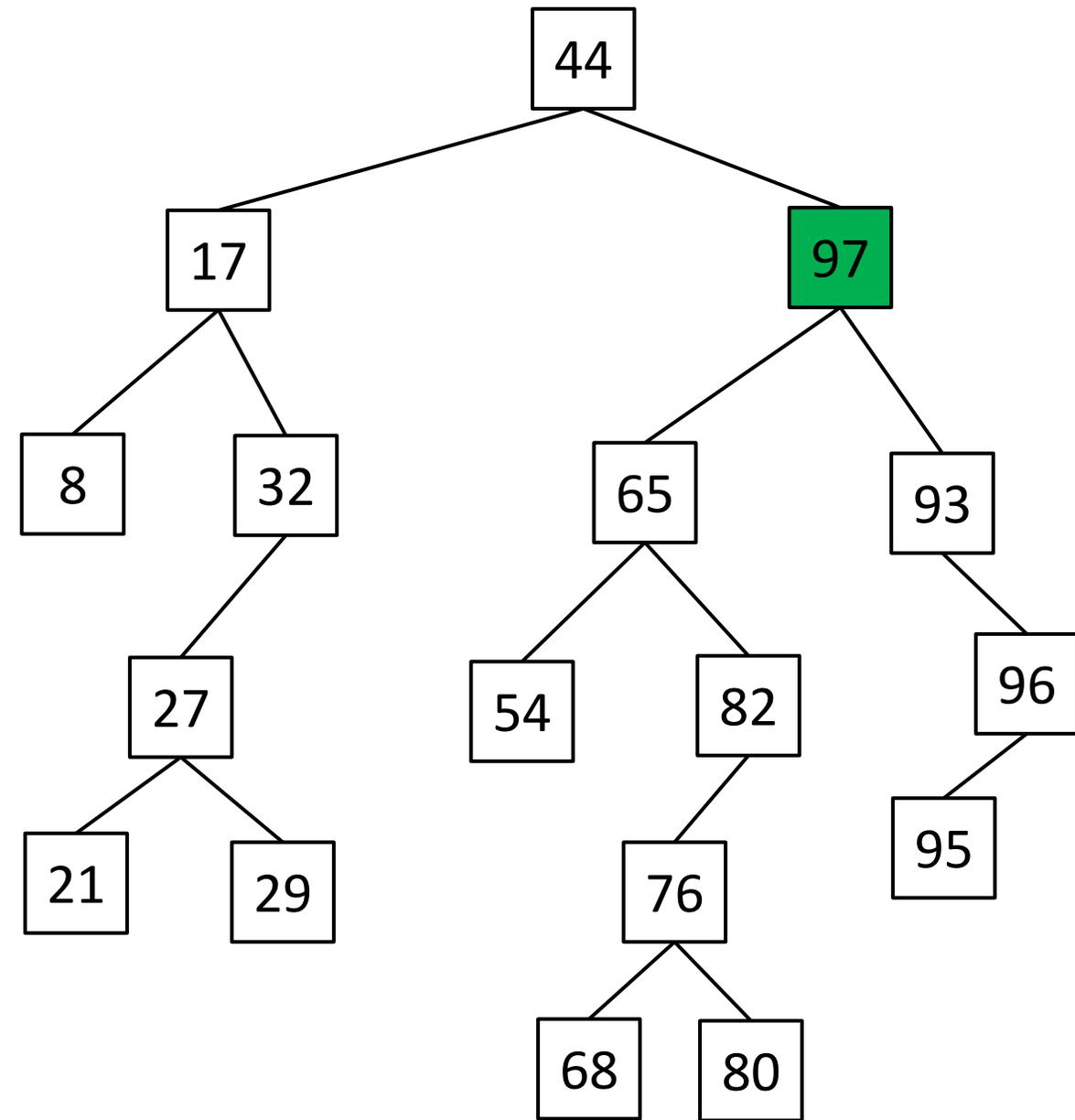
Binary Search Tree- Removal

Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(88);`

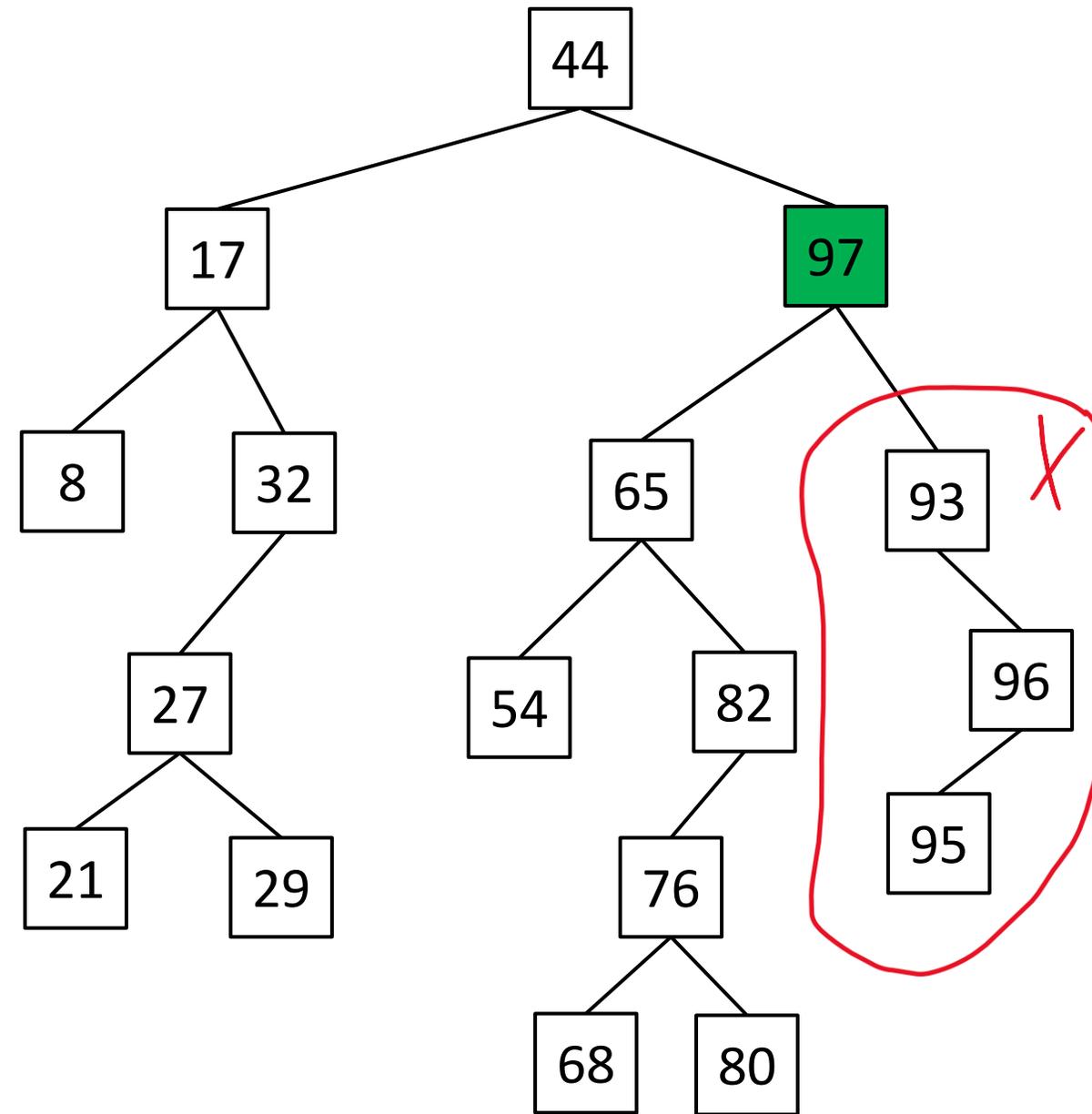


Binary Search Tree- Removal

- Case 1: Node has no children
- Case 2: Node has one child
- Case 3: Node has two children

`remove(88);`

Right child
doesn't work!



Binary Search Tree- Removal

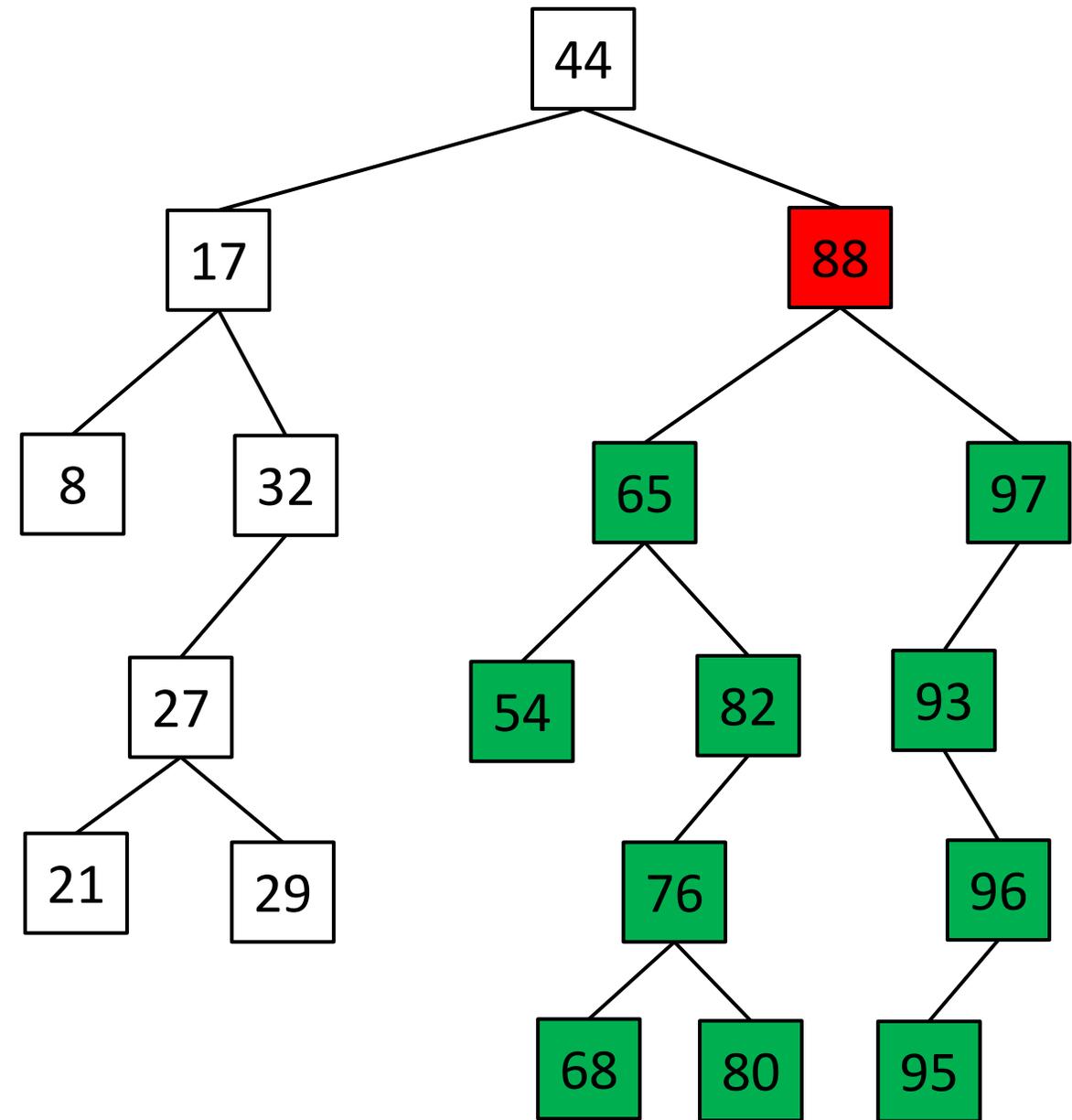
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(88);`

Which child descendant to use?



Binary Search Tree- Removal

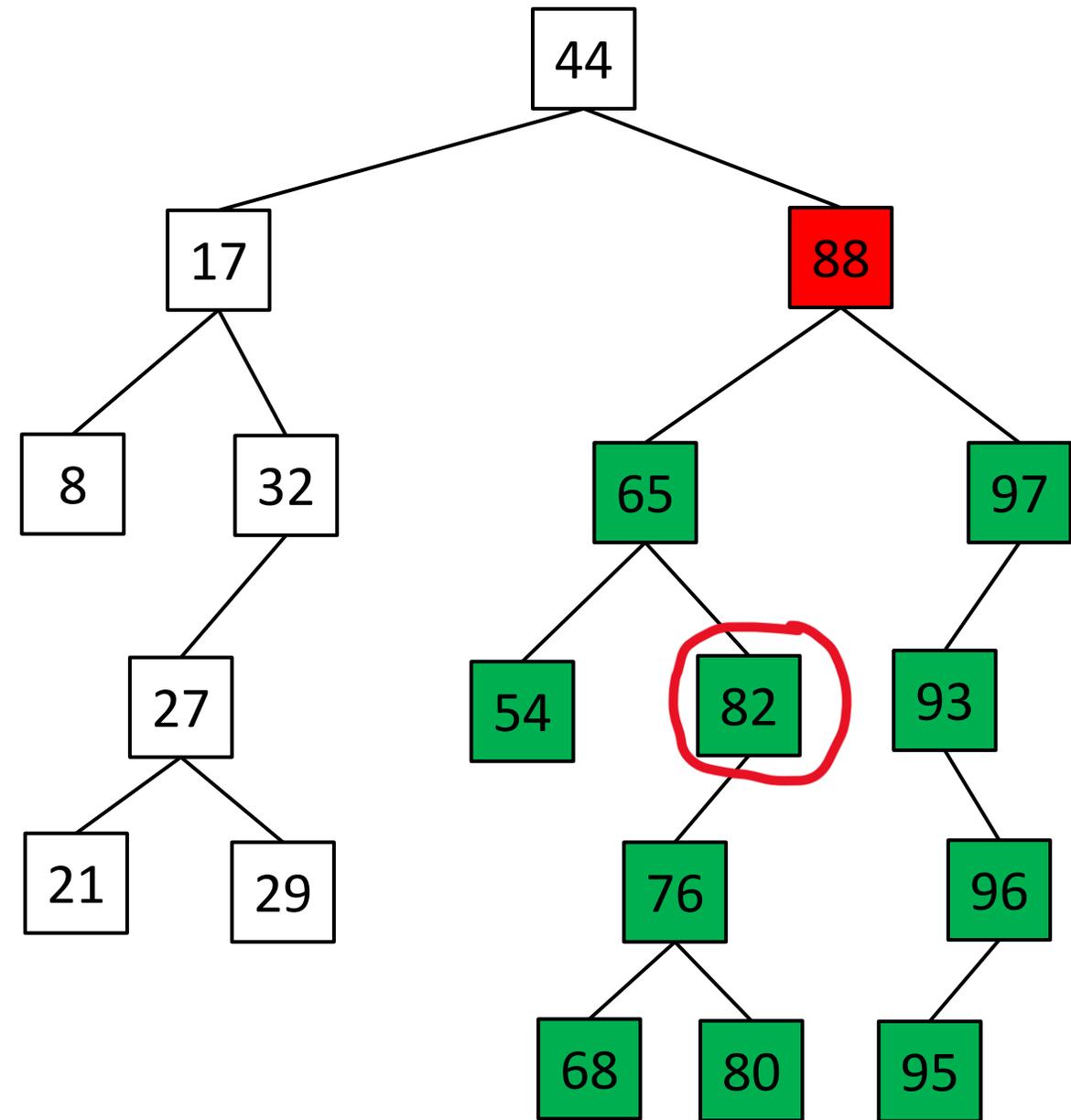
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(88);`

Which child descendant to use?



Binary Search Tree- Removal

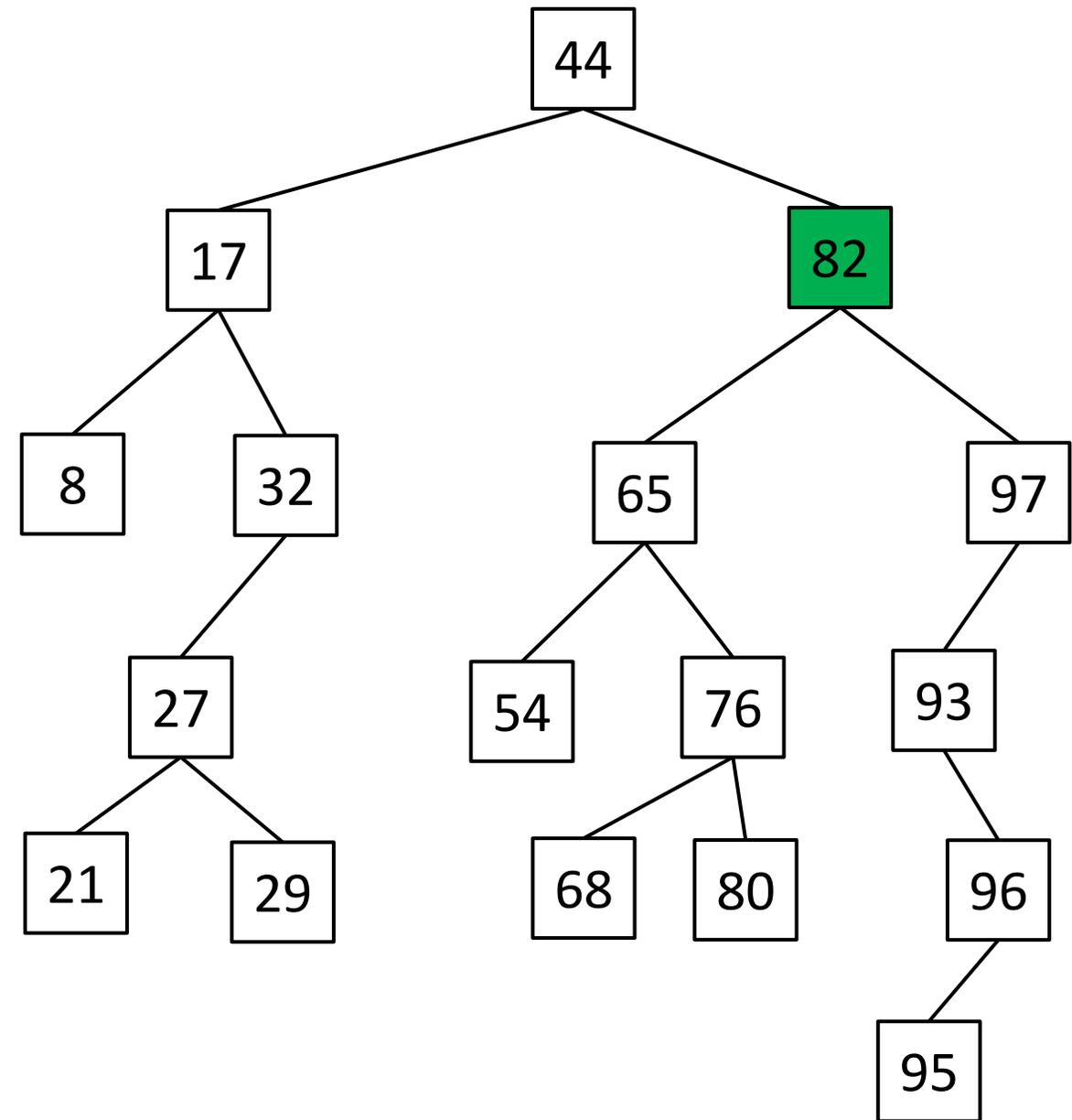
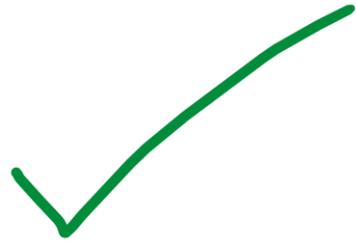
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(88);`

Which child descendant to use?



Binary Search Tree- Removal

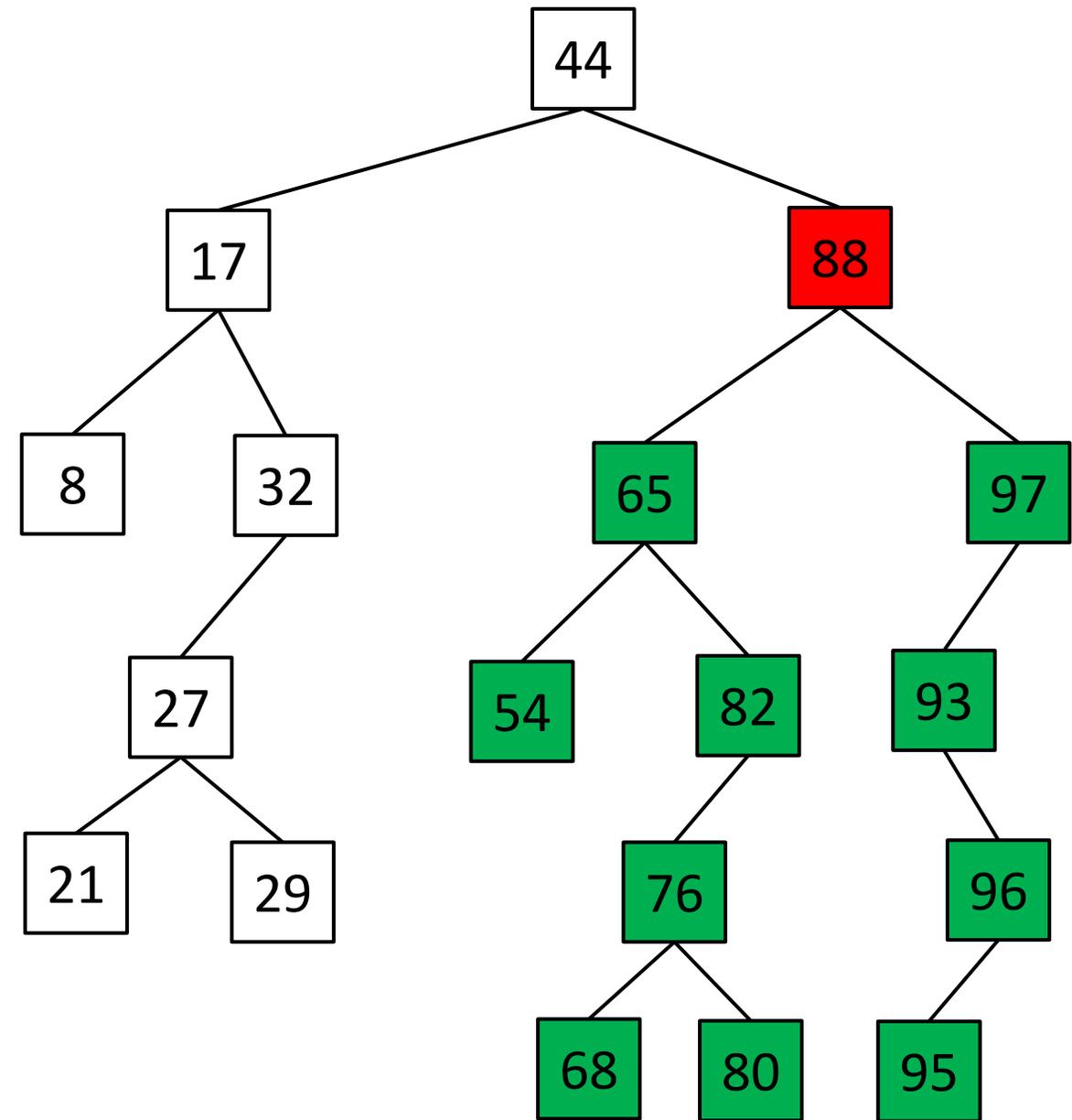
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(88);`

Which child descendant to use?



Binary Search Tree- Removal

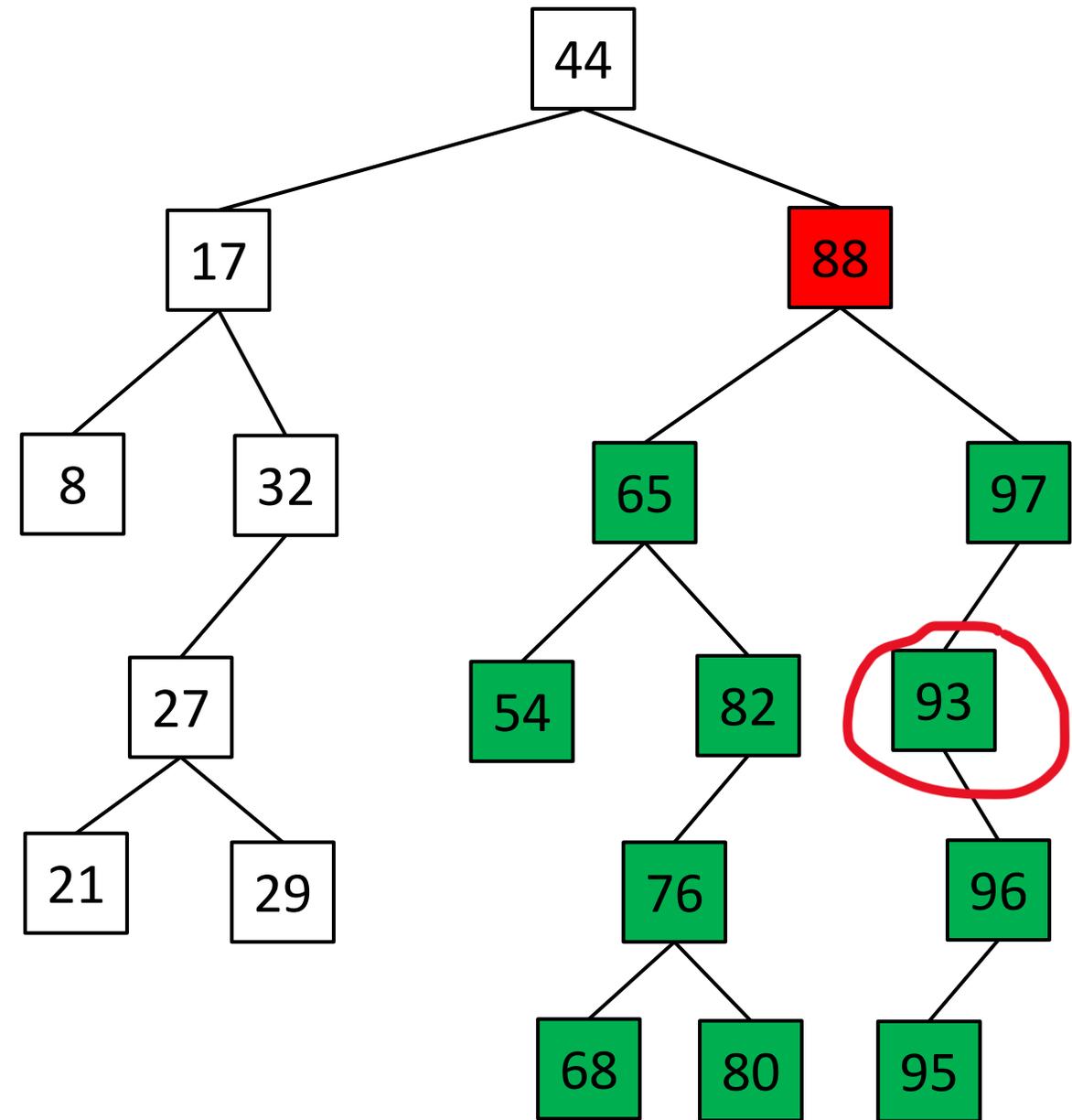
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(88);`

Which child descendant to use?



Binary Search Tree- Removal

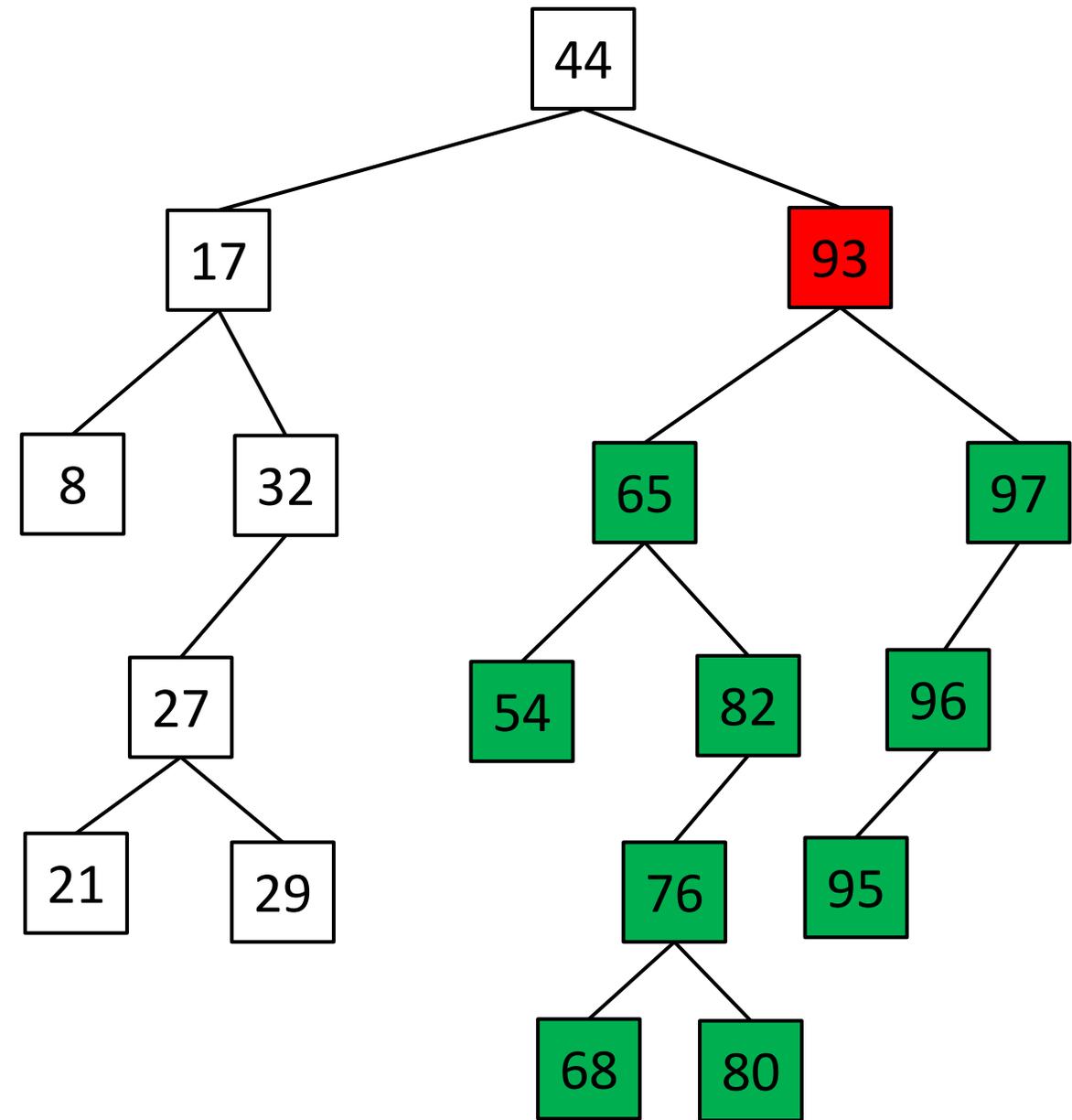
Case 1: Node has no children

Case 2: Node has one child

Case 3: Node has two children

`remove(88);`

Which ~~child~~ descendant to use?



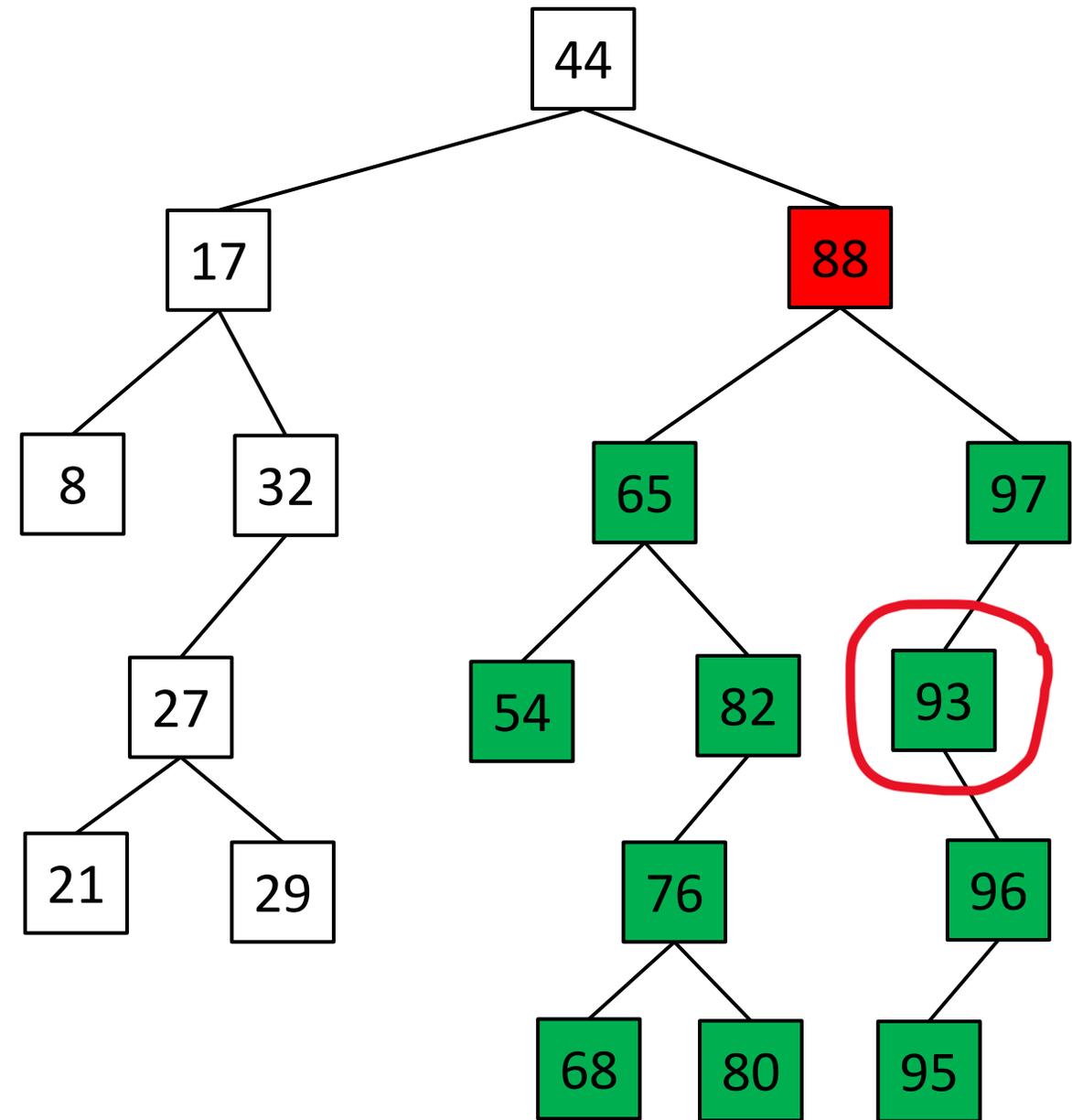
Binary Search Tree- Removal

- Case 1: Node has no children
- Case 2: Node has one child
- Case 3: Node has two children

`remove(88);`

Which child descendant to use?

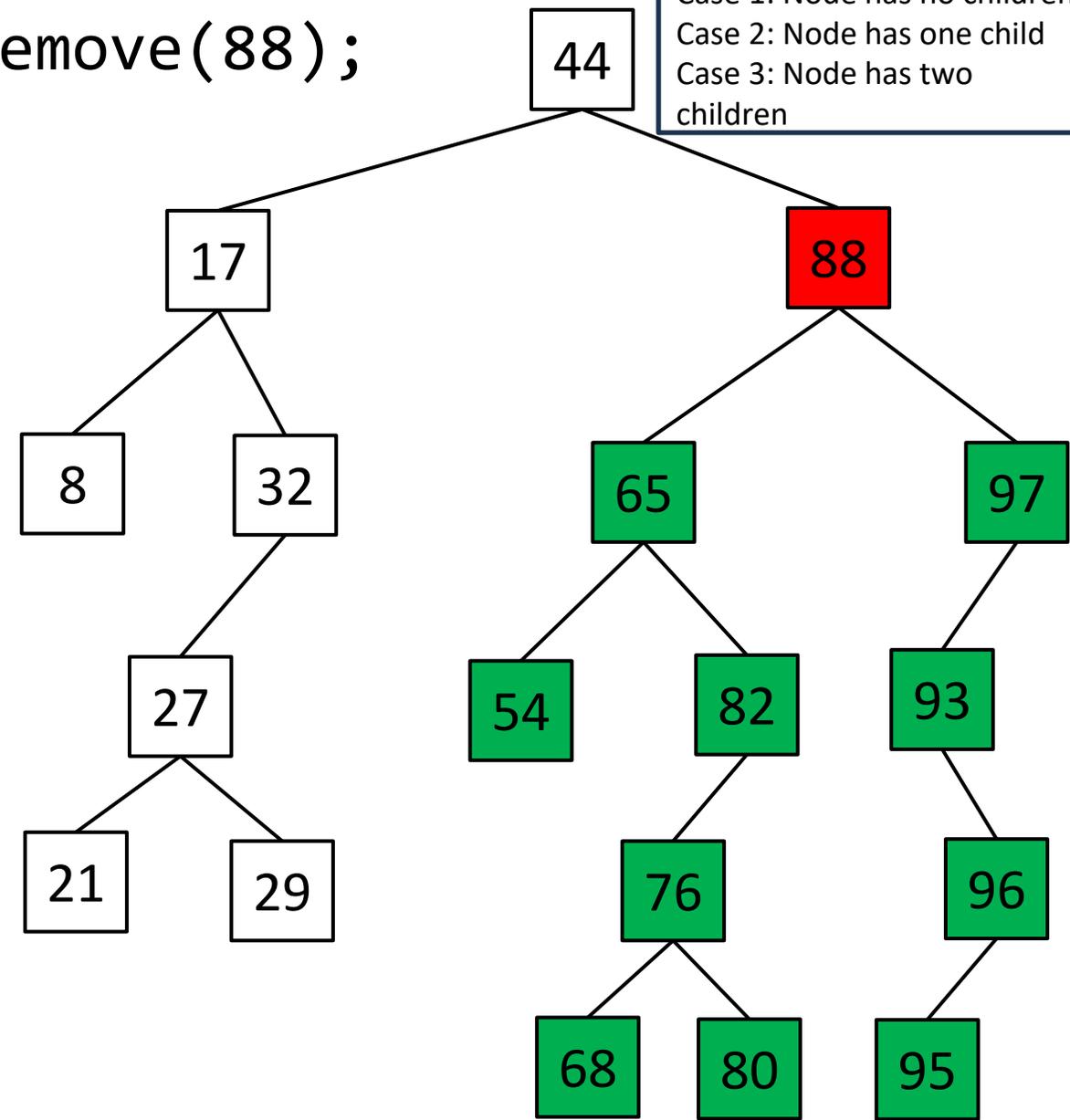
The lowest value in the right subtree
or the highest value in the left subtree



Binary Search Tree- Removal

remove(88);

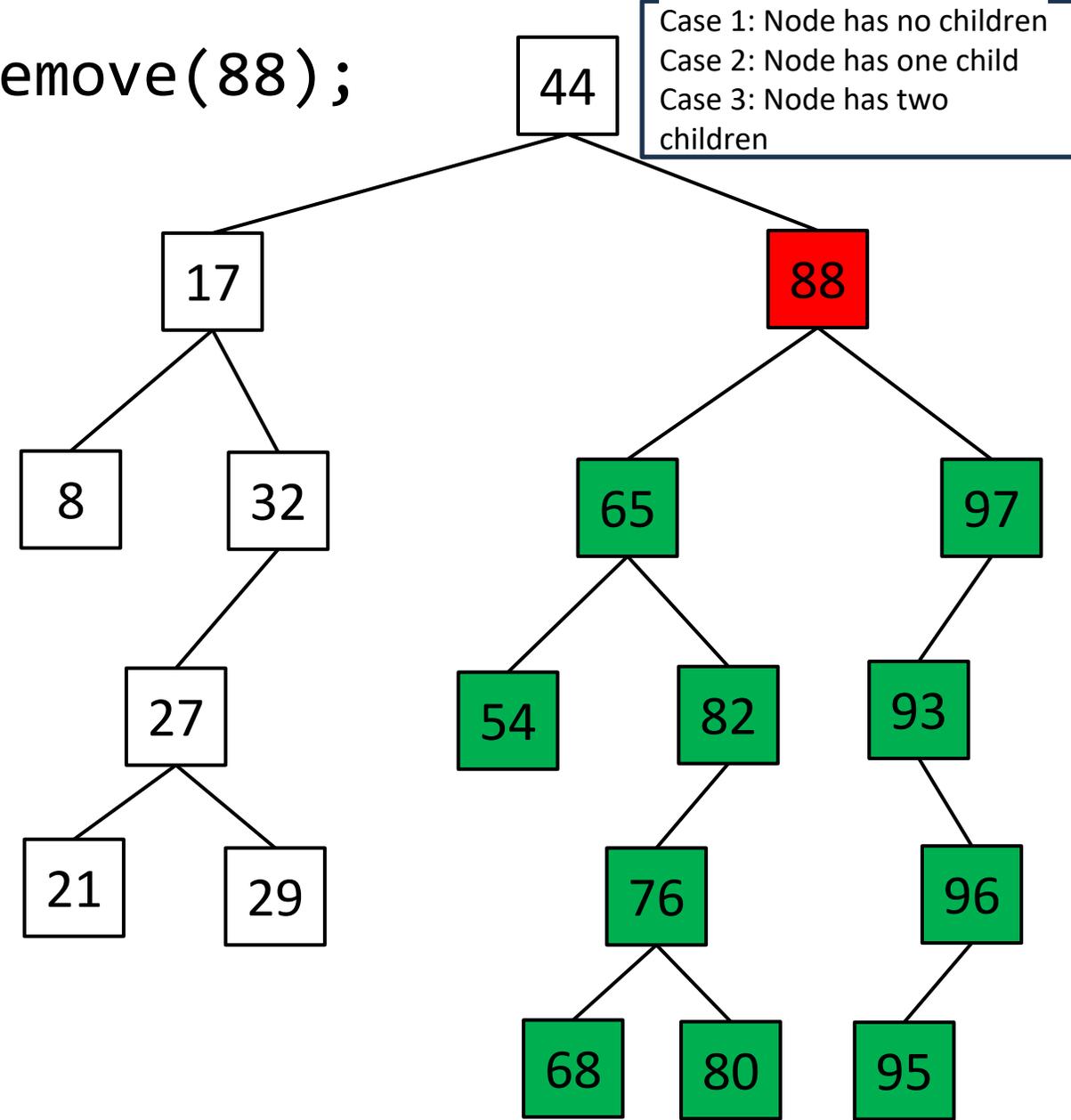
Case 1: Node has no children
Case 2: Node has one child
Case 3: Node has two children



Binary Search Tree- Removal

We will solve this *recursively*

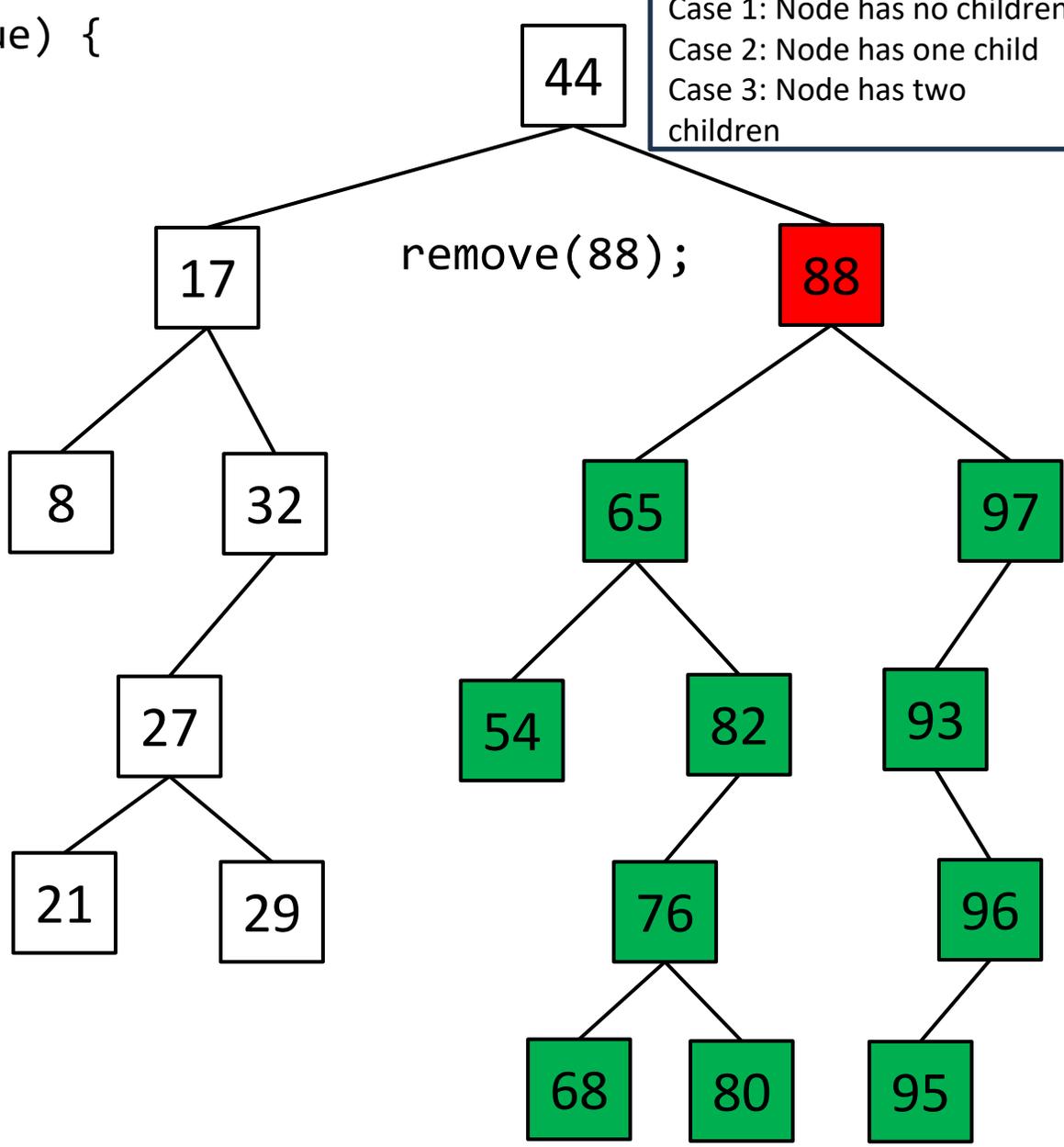
`remove(88);`



- Case 1: Node has no children
- Case 2: Node has one child
- Case 3: Node has two children

```
public Node deleteNode(Node current, int searchValue) {
```

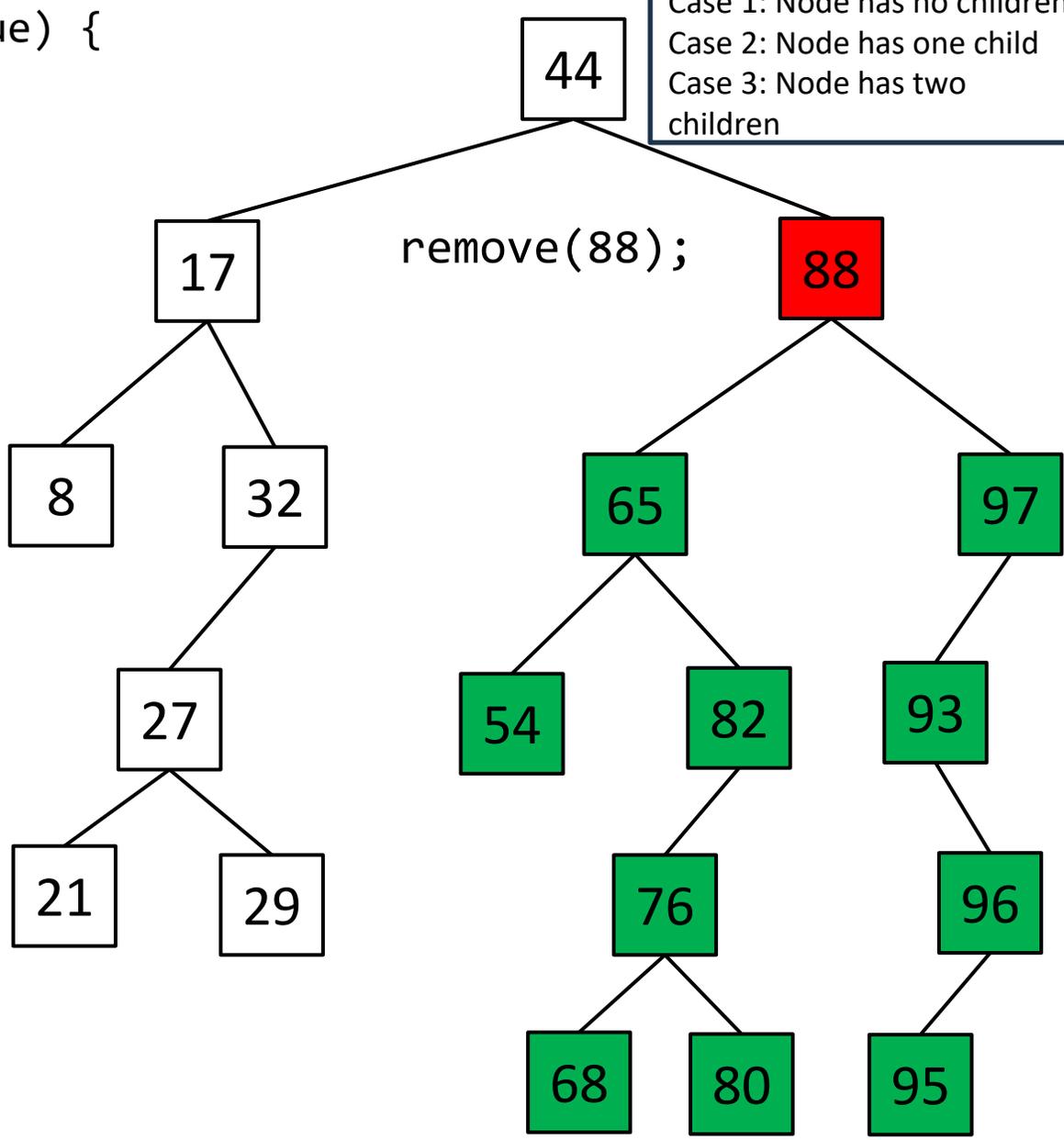
We are going to recursively work our way down the tree, and remove the searchValue, and then return the new *root* of the subtree (if it got modified)



Case 1: Node has no children
Case 2: Node has one child
Case 3: Node has two children

```
public Node deleteNode(Node current, int searchValue) {  
    if (current == null) {  
        return current;  
    }  
}
```

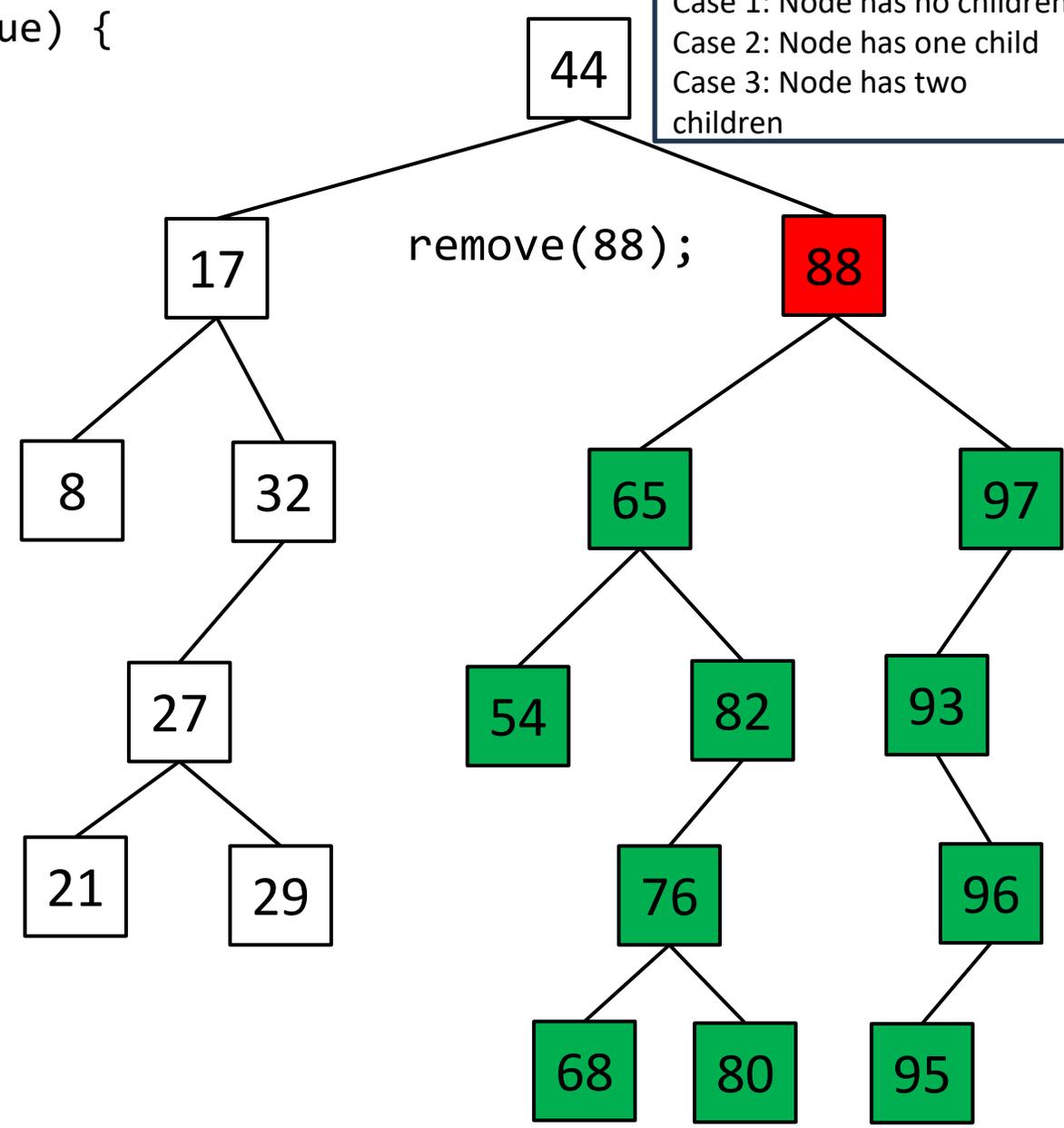
If we ever hit a null value, return



Case 1: Node has no children
Case 2: Node has one child
Case 3: Node has two children

```
public Node deleteNode(Node current, int searchValue) {  
    if (current == null) {  
        return current;  
    }  
}
```

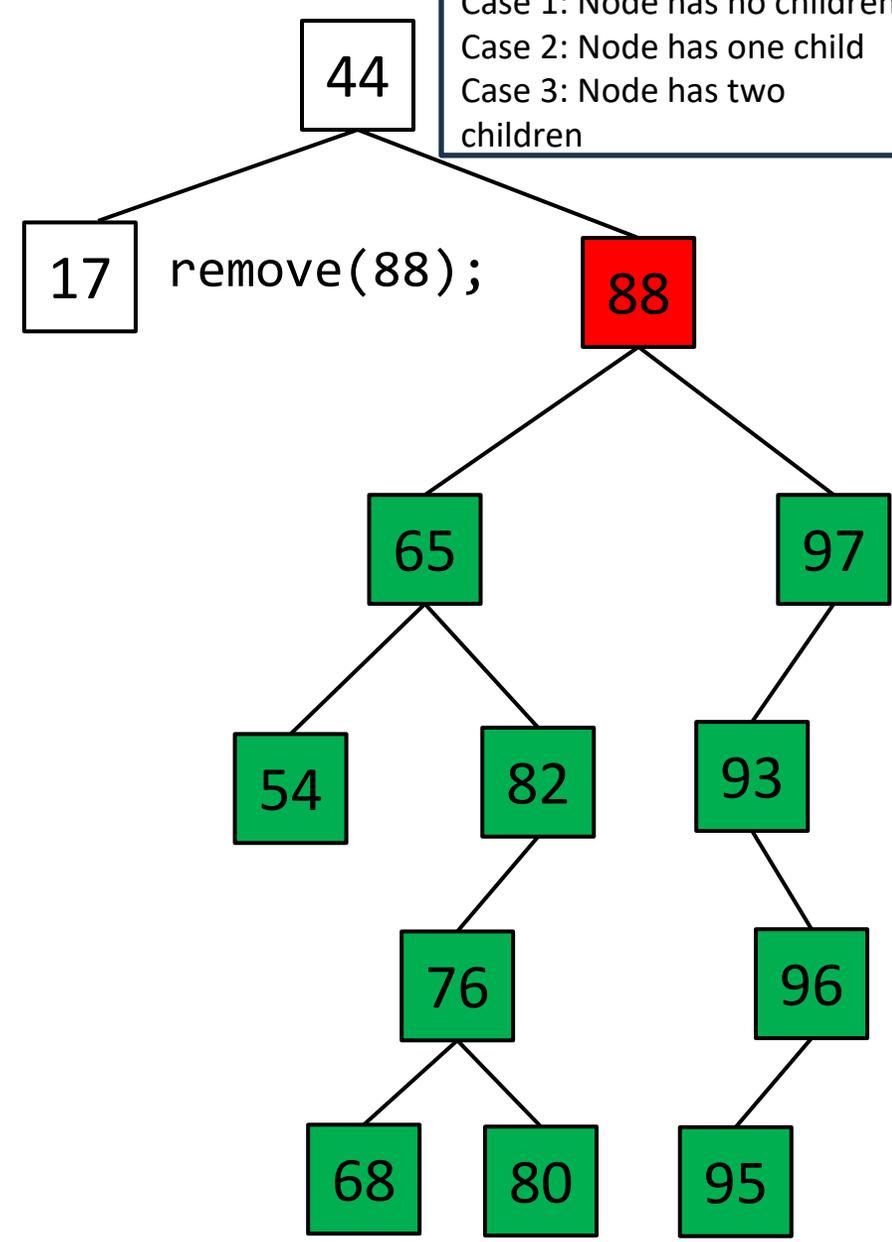
Step 1. Search



Case 1: Node has no children
Case 2: Node has one child
Case 3: Node has two children

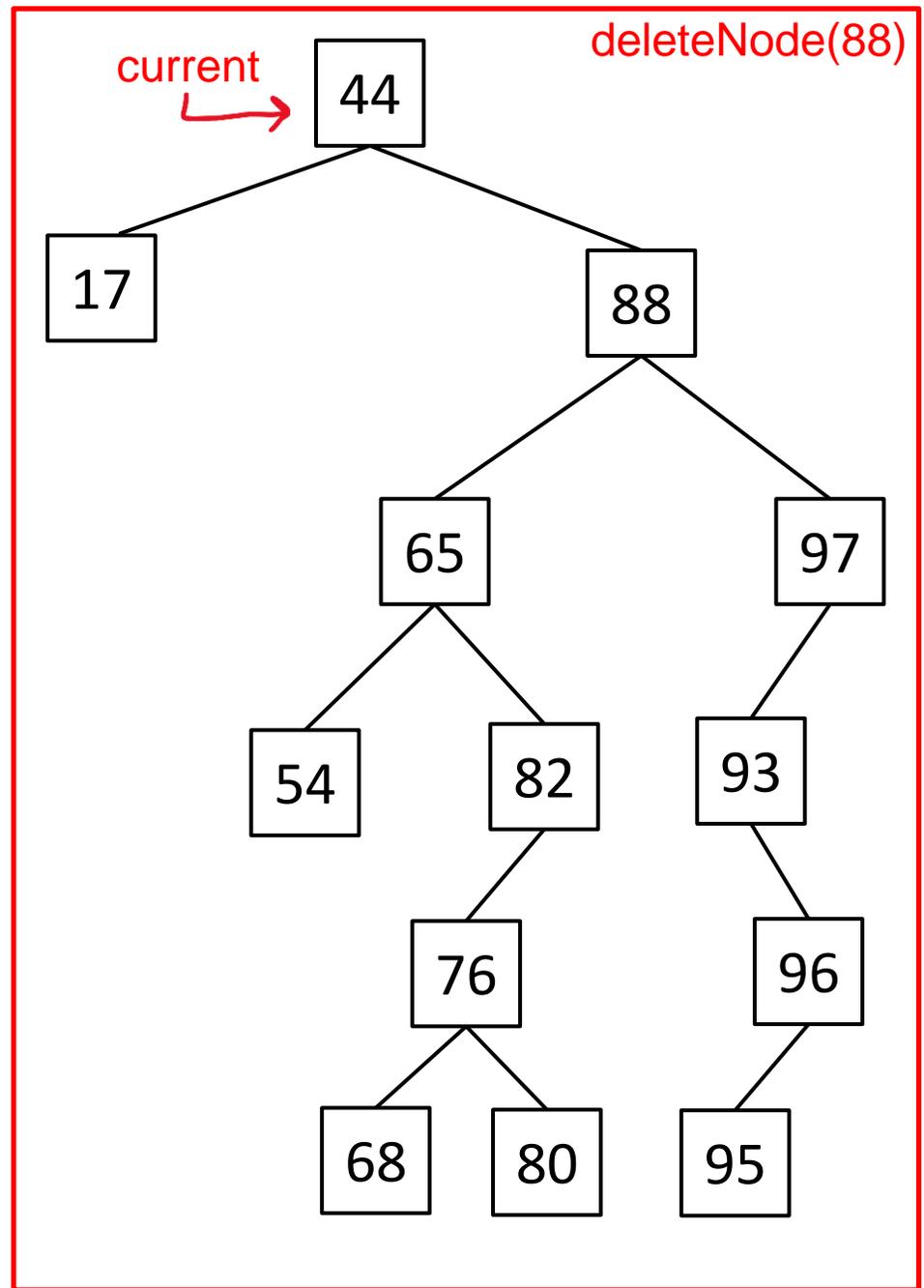
```
public Node deleteNode(Node current, int searchValue) {  
    if (current == null) {  
        return current;  
    }  
    if (current.getValue() > searchValue) {  
        current.setLeft( deleteNode(current.getLeft(), searchValue));  
    }  
    else if (current.getValue() < searchValue) {  
        current.setRight( deleteNode(current.getRight(), searchValue));  
    }  
    else {  

```



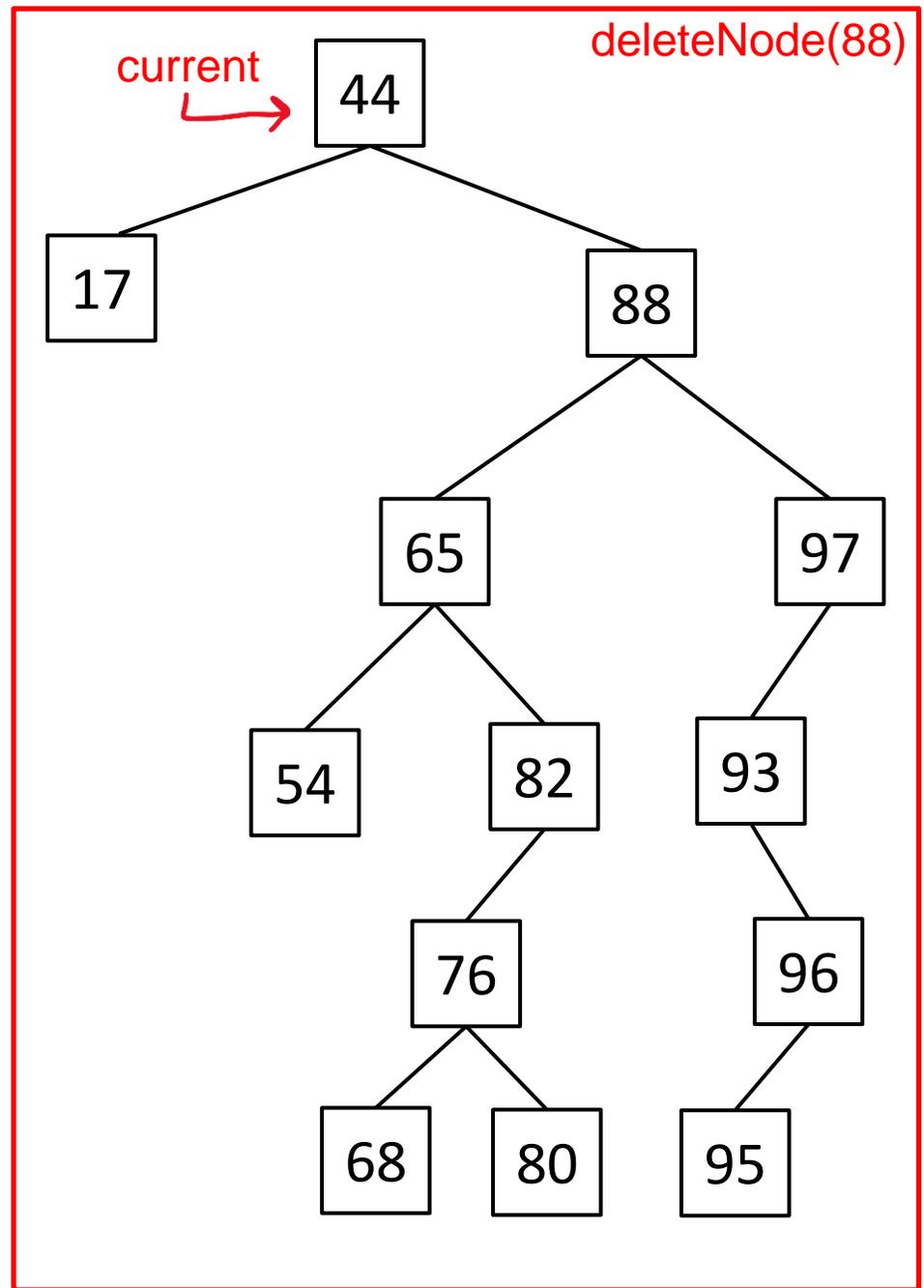
```
public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {

```



```
public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {

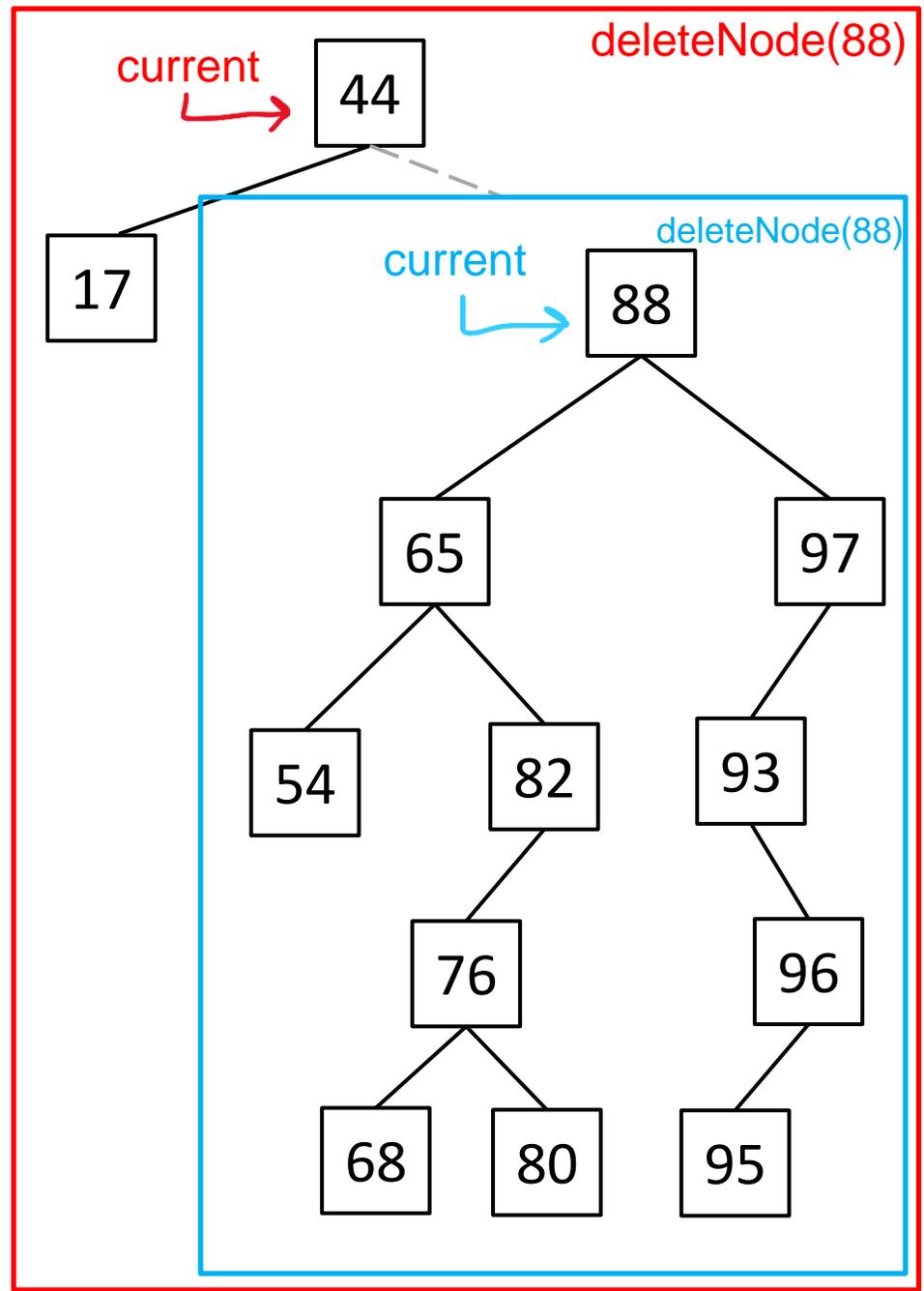
```



```

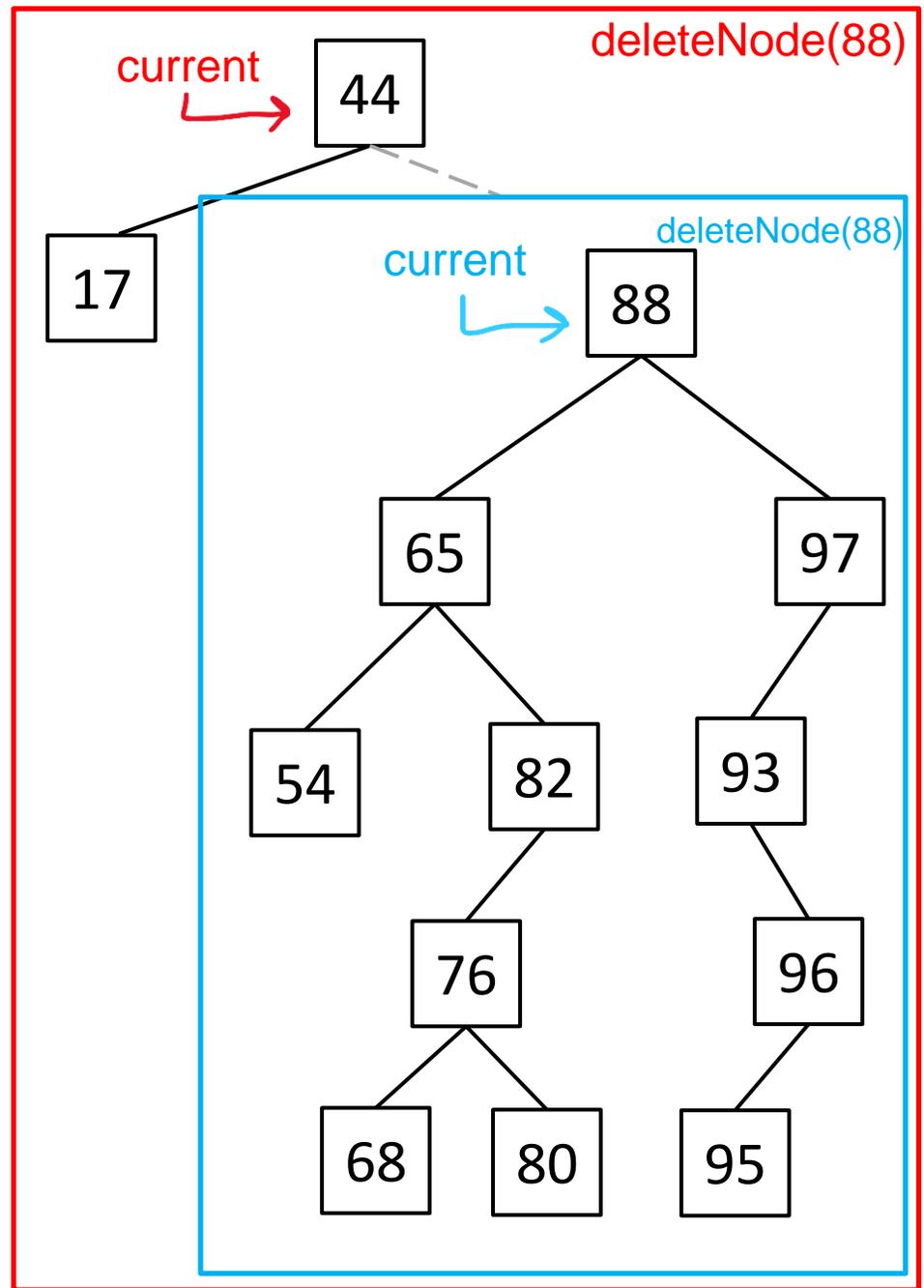
public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        Current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {

```



```
public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {

```

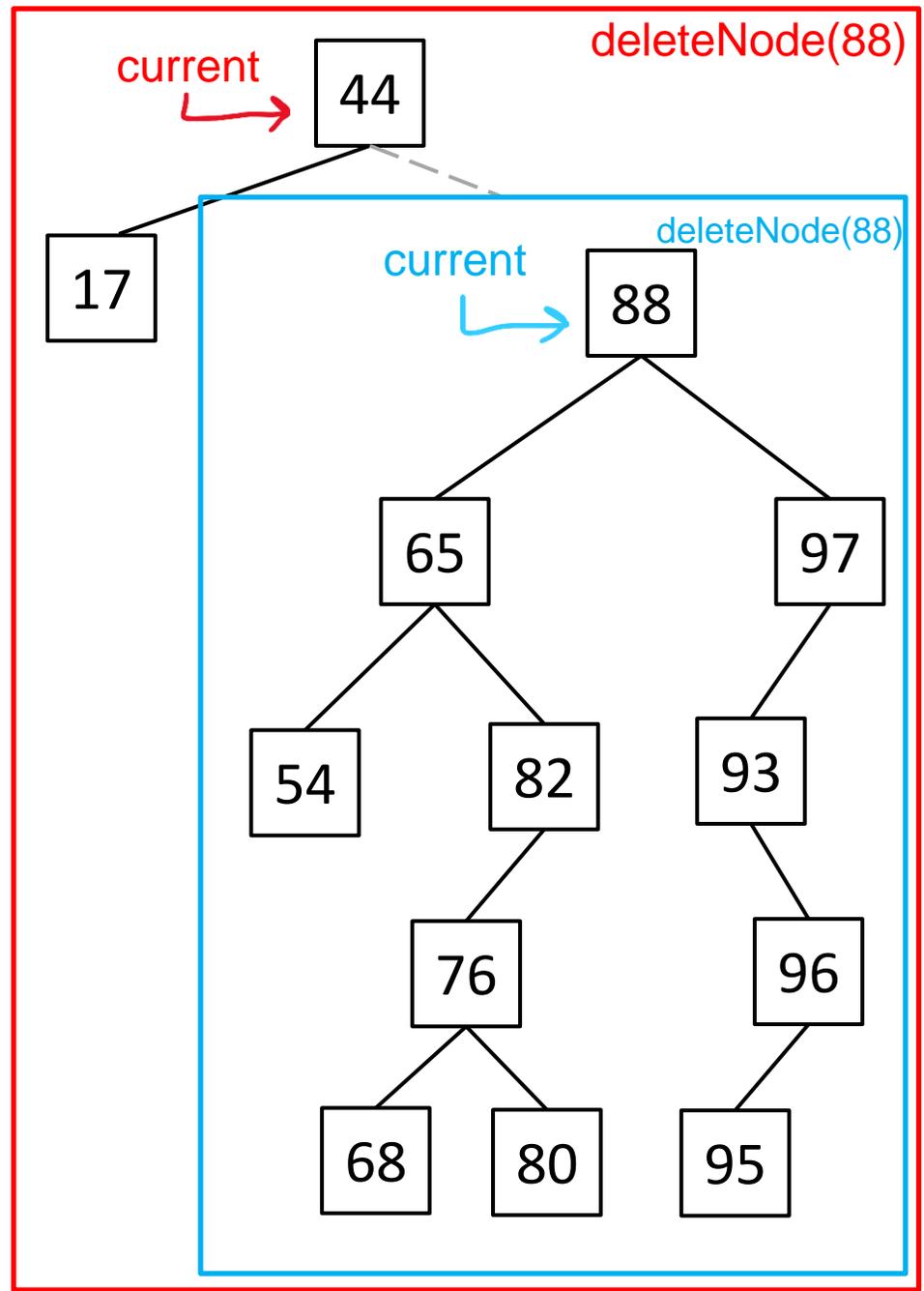


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {

```

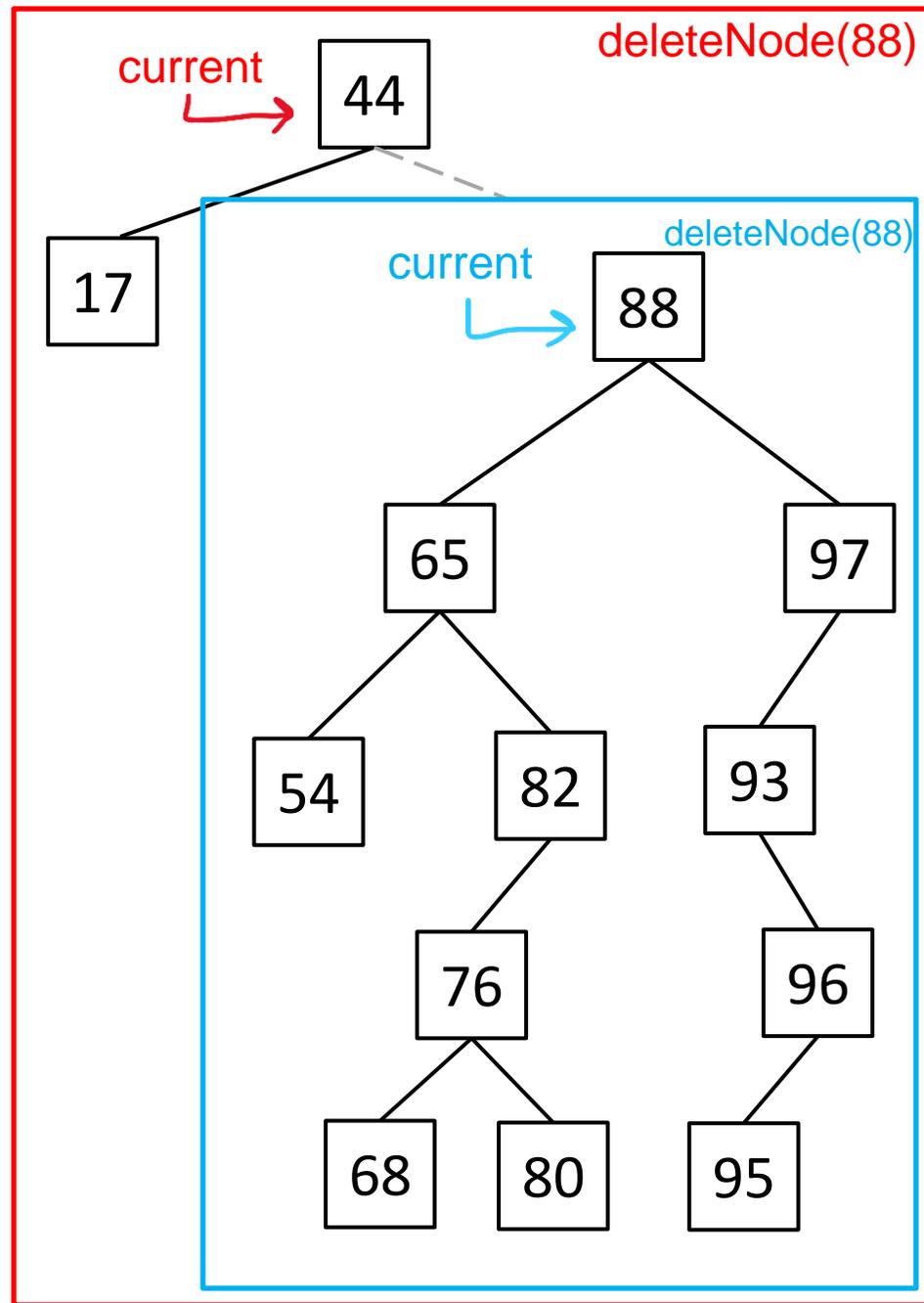
Case 1: Node has no children
 Case 2: Node has one child
 Case 3: Node has two children



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
    }
}

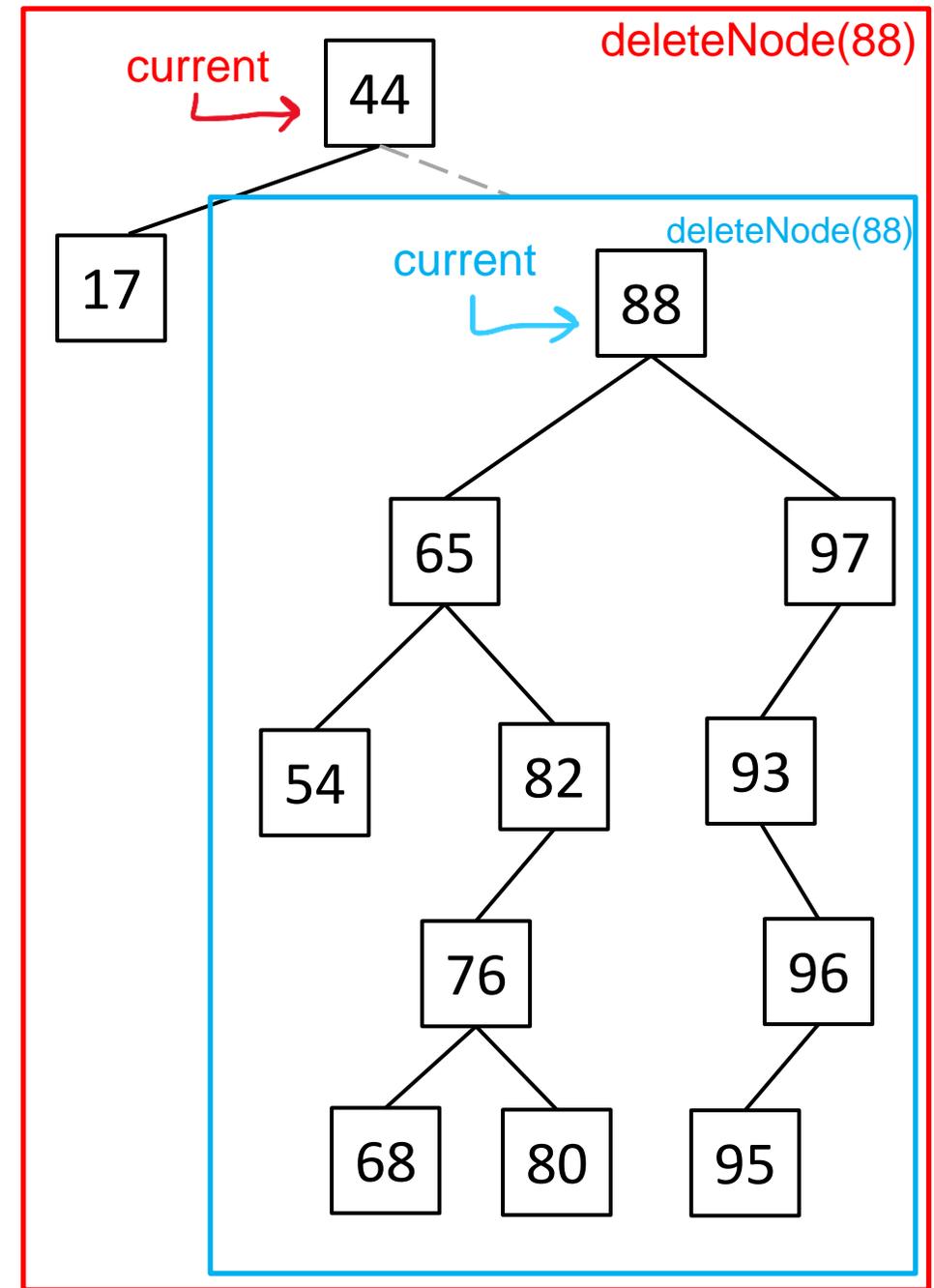
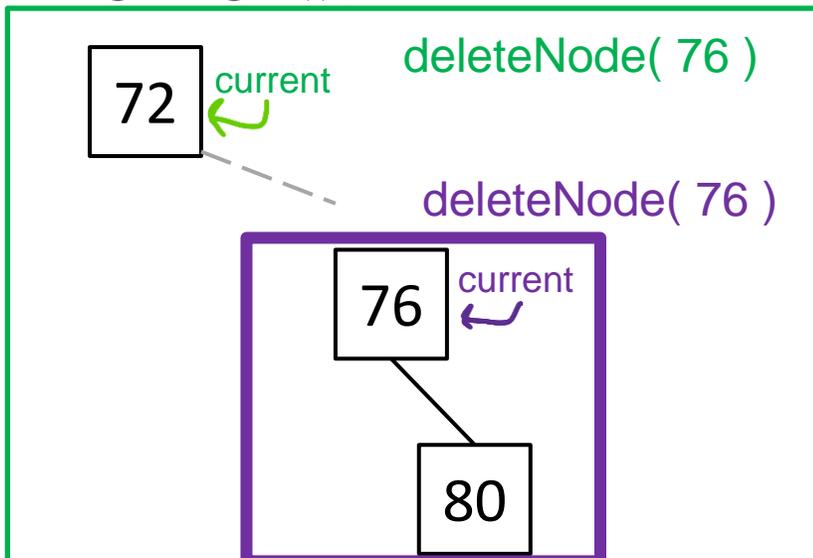
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        Current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
    }
}

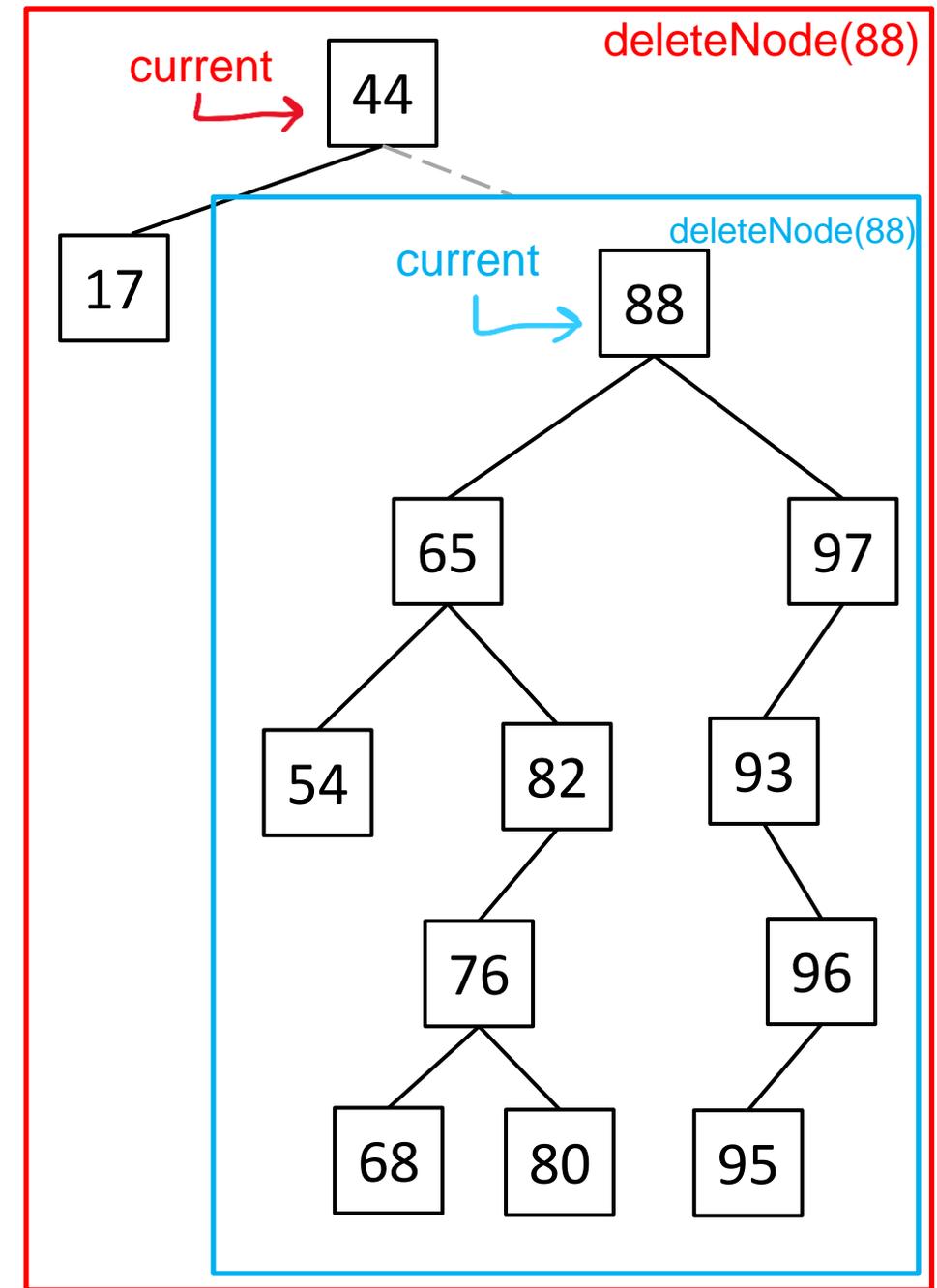
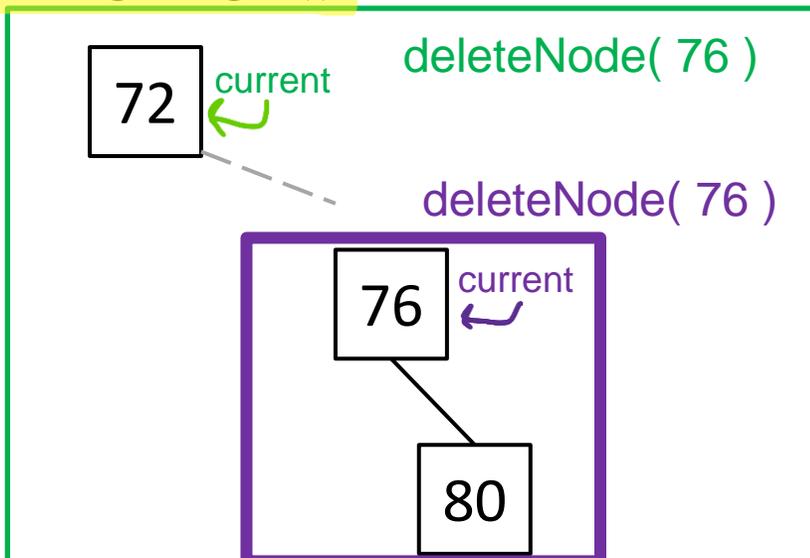
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
    }
}

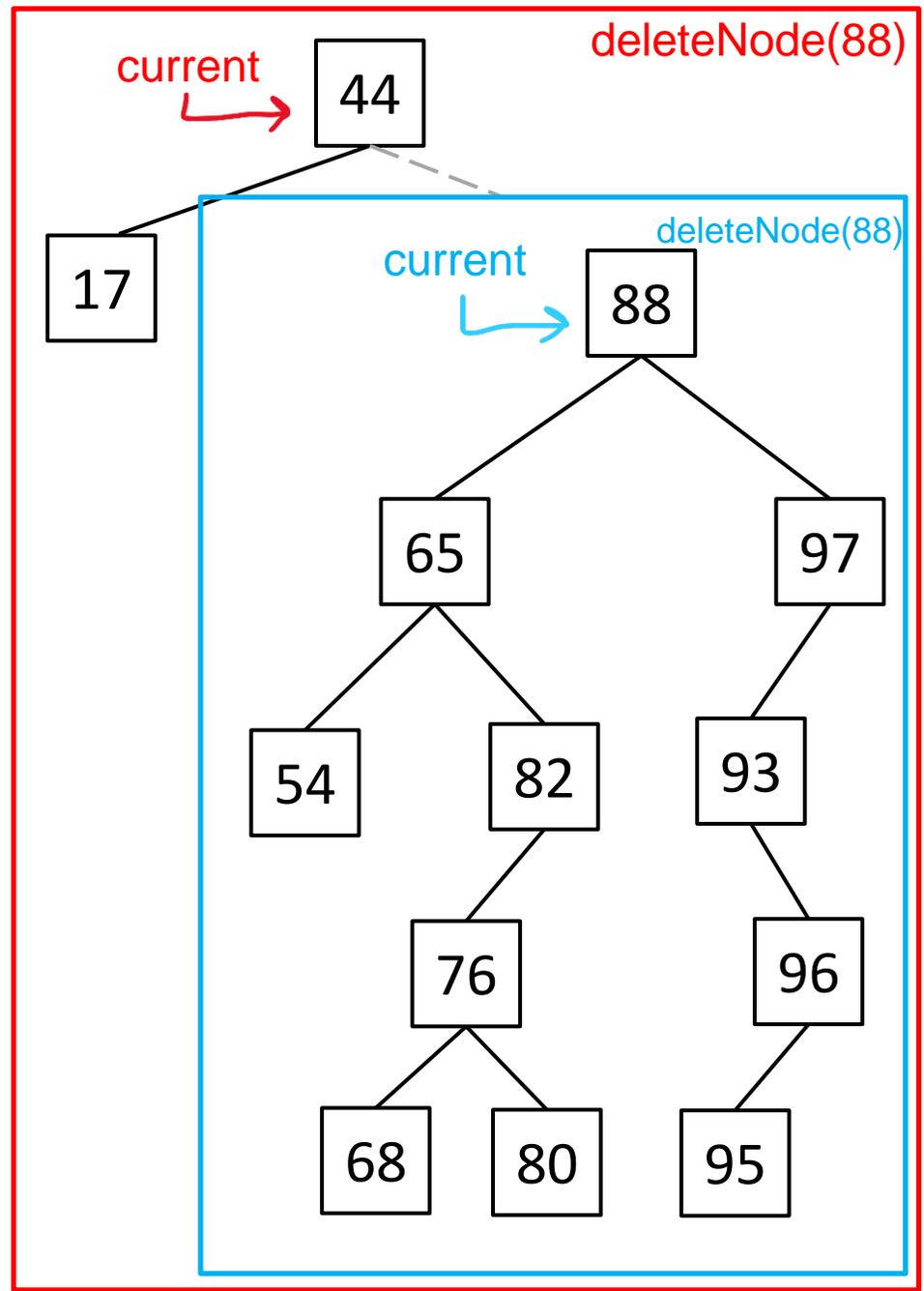
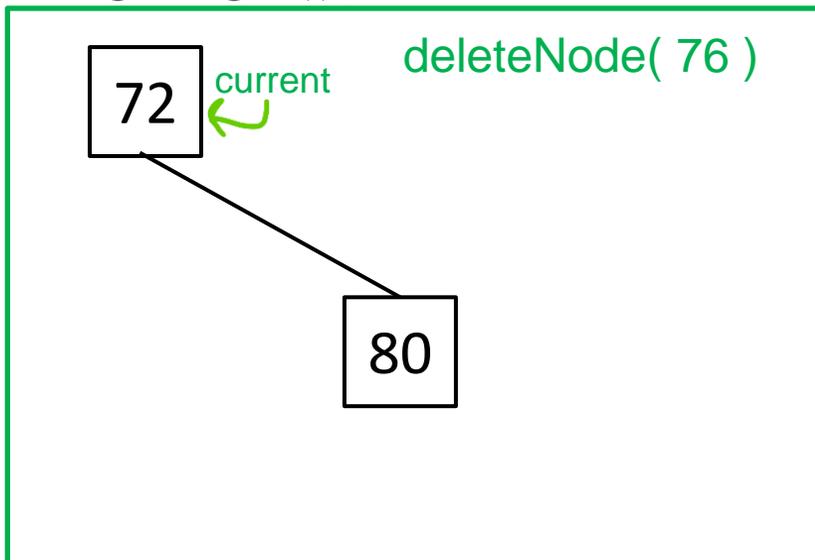
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        Current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
    }
}

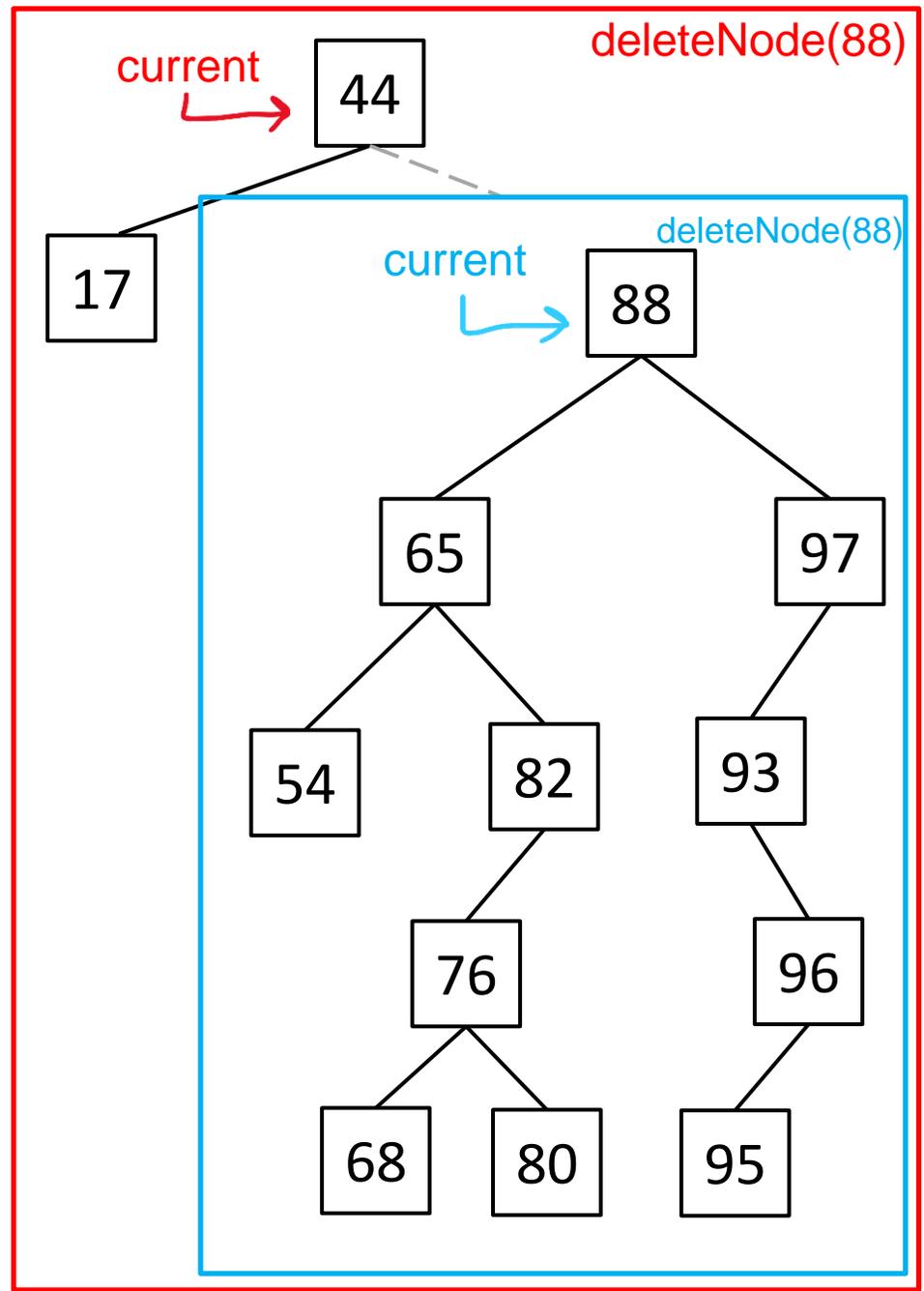
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
    }
}

```

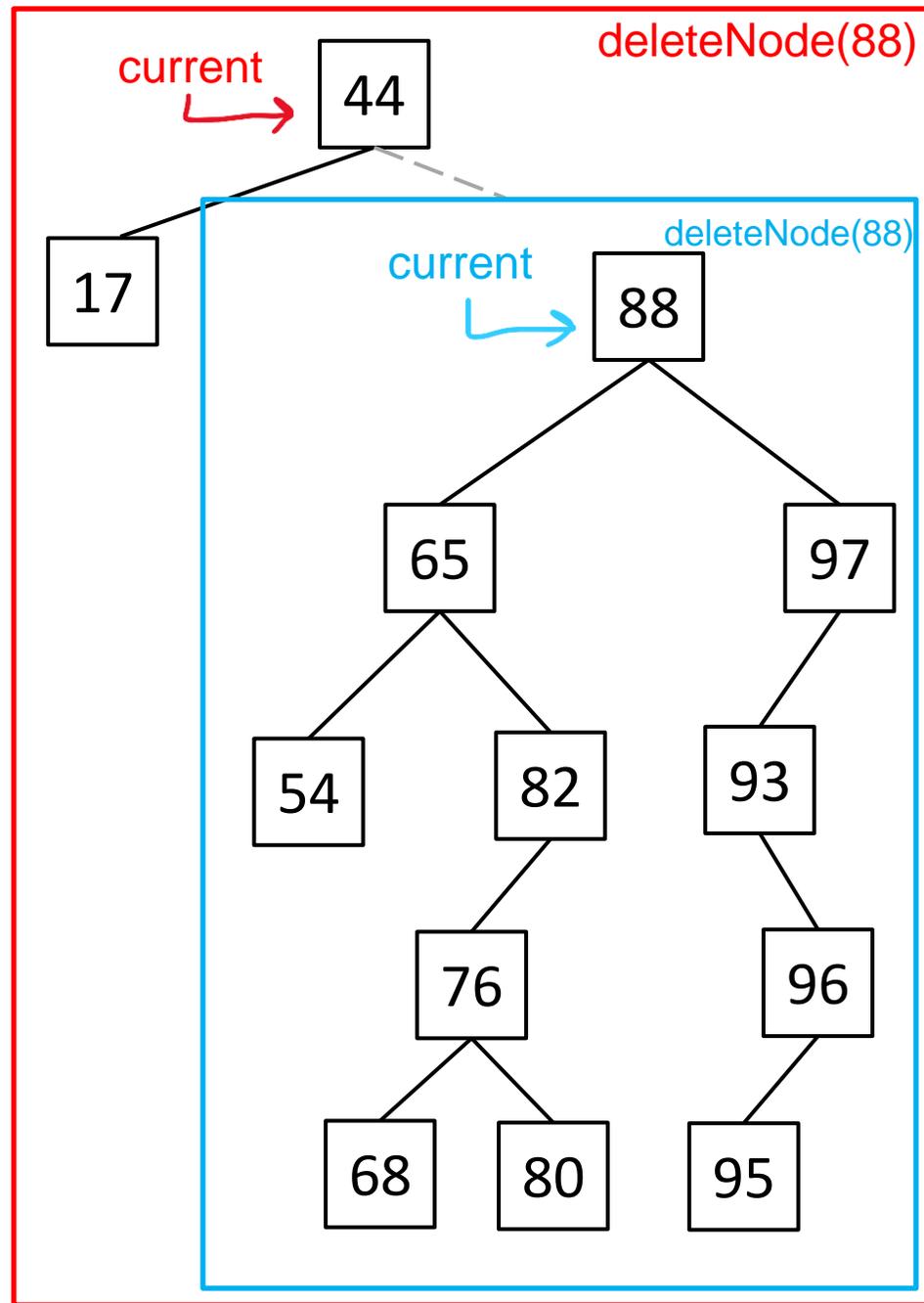


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) { ✗
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) { ✗
            return current.getLeft();
        }
    }
}

```

In our current recursive call (blue square), the node we are currently at has two children

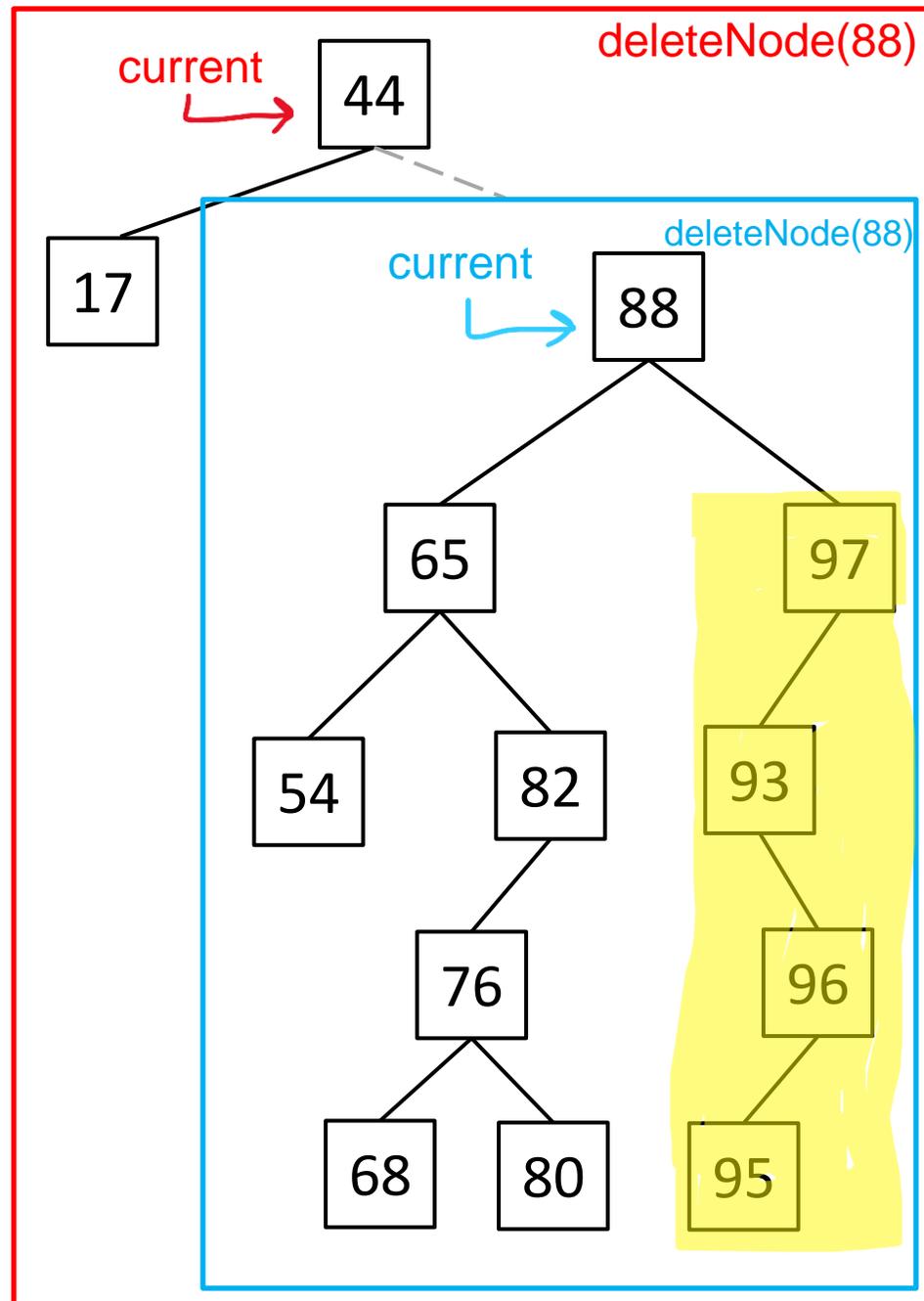


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
    }
}

```

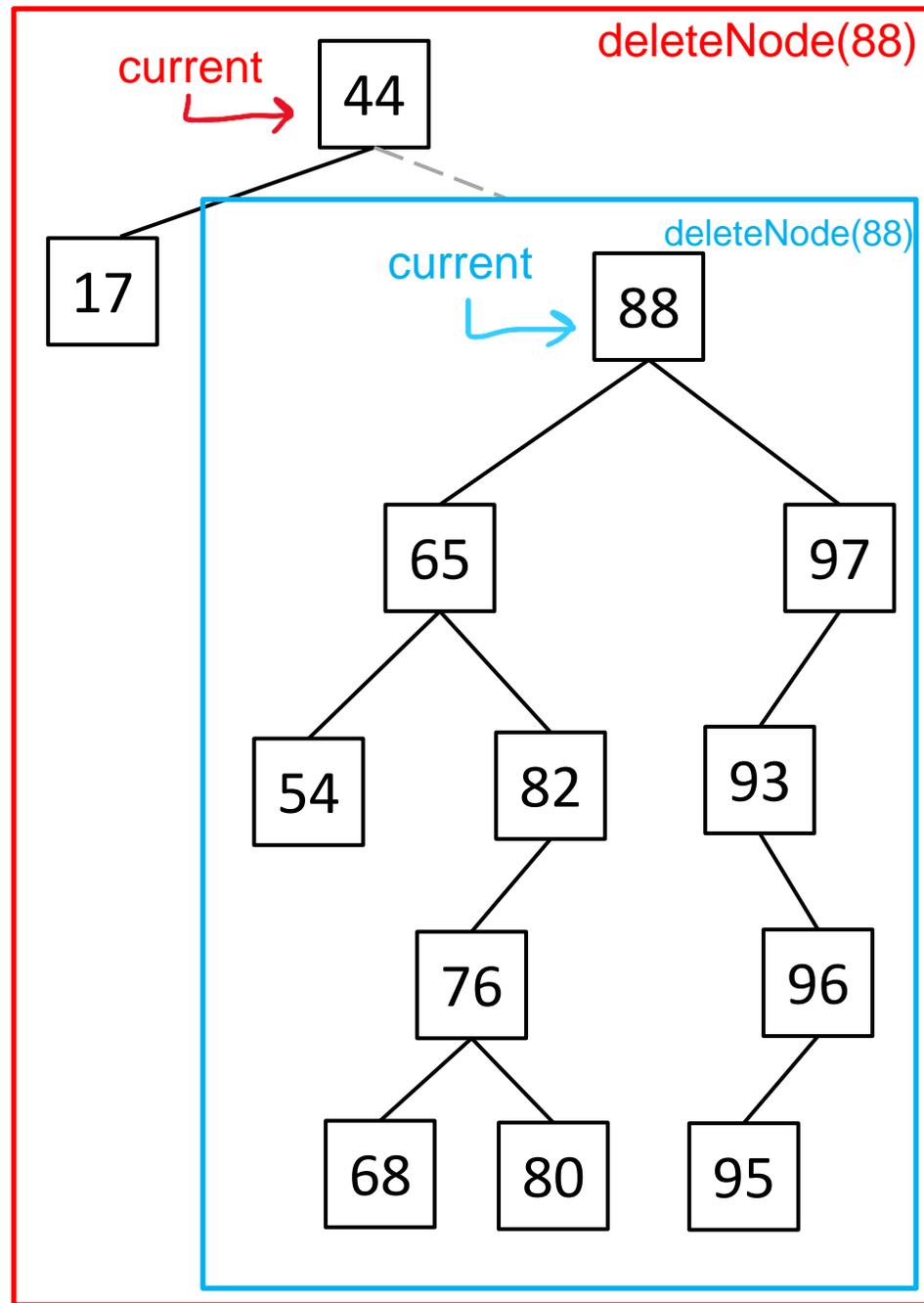
Helper method to find replacement (smallest value in right subtree)



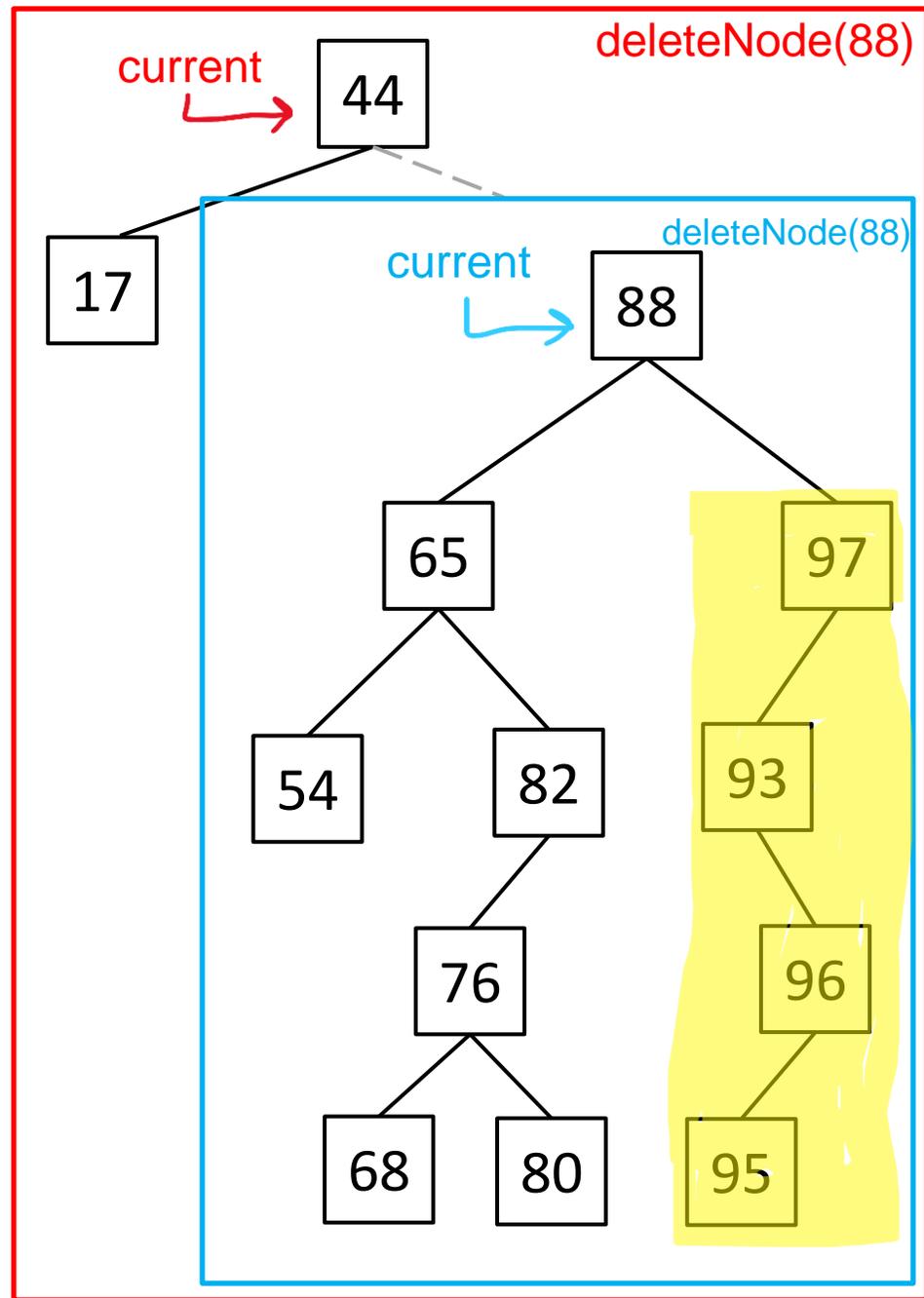
```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
    }
}

```

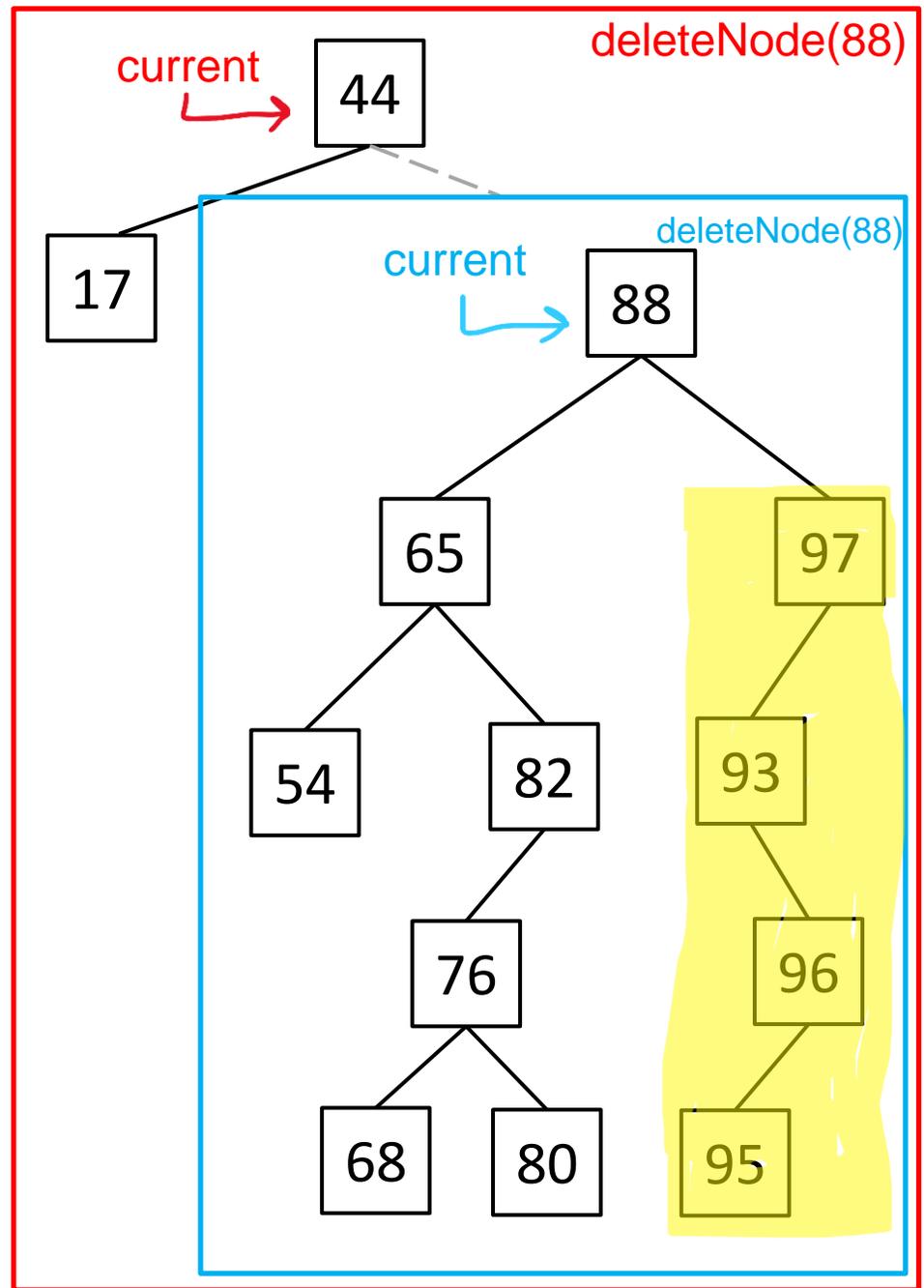


```
public Node findReplacement(Node current) {
    current = current.getRight();
    while(current != null && current.getLeft() != null) {
        current = current.getLeft();
    }
    return current;
}
```



```
public Node findReplacement(Node current) {  
    current = current.getRight();  
    while(current != null && current.getLeft() != null) {  
        current = current.getLeft();  
    }  
    return current;  
}
```

1. Go into right subtree

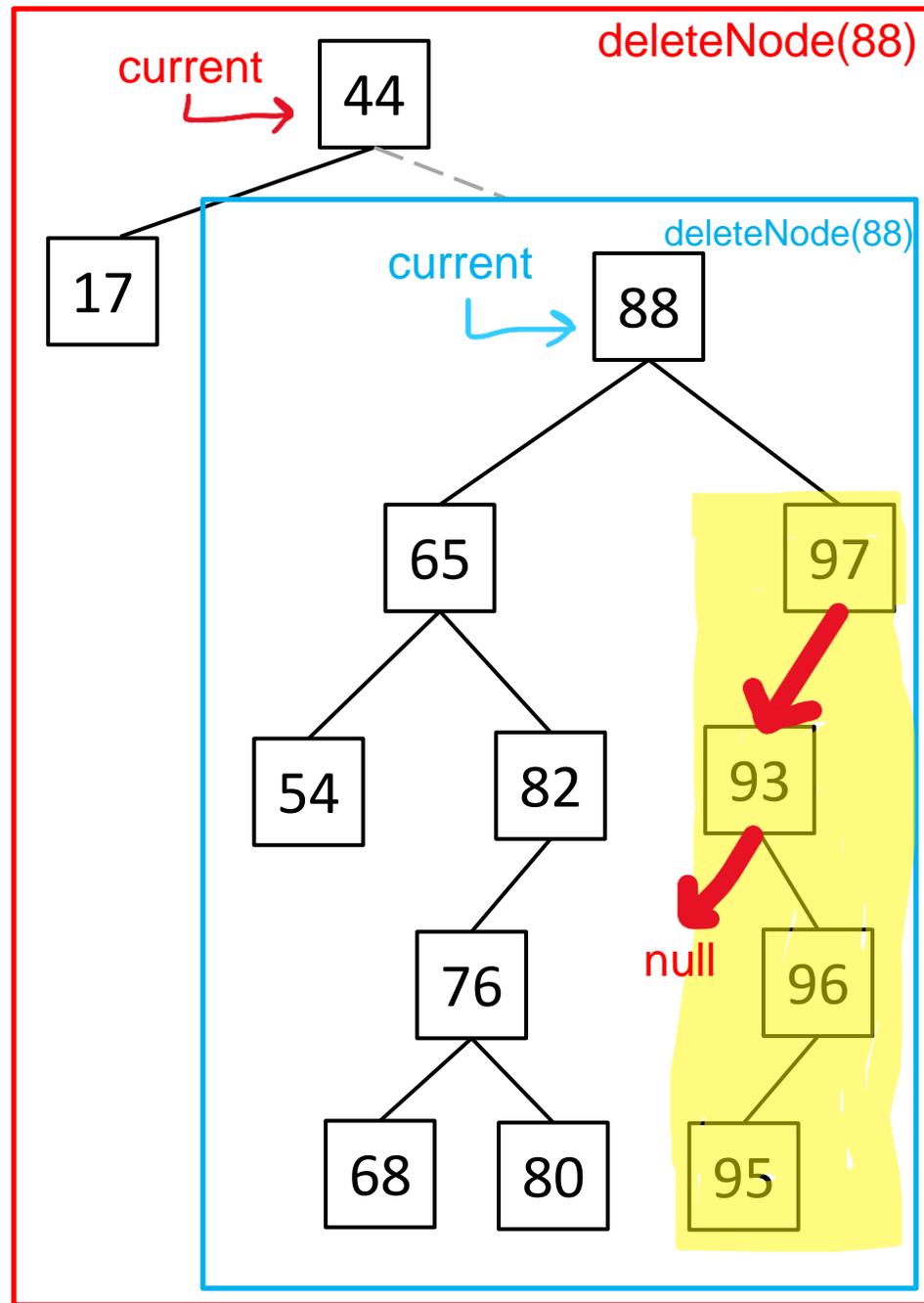


```

public Node findReplacement(Node current) {
    current = current.getRight();
    while(current != null && current.getLeft() != null) {
        current = current.getLeft();
    }
    return current;
}

```

1. Go into right subtree
2. Keep going left in the subtree

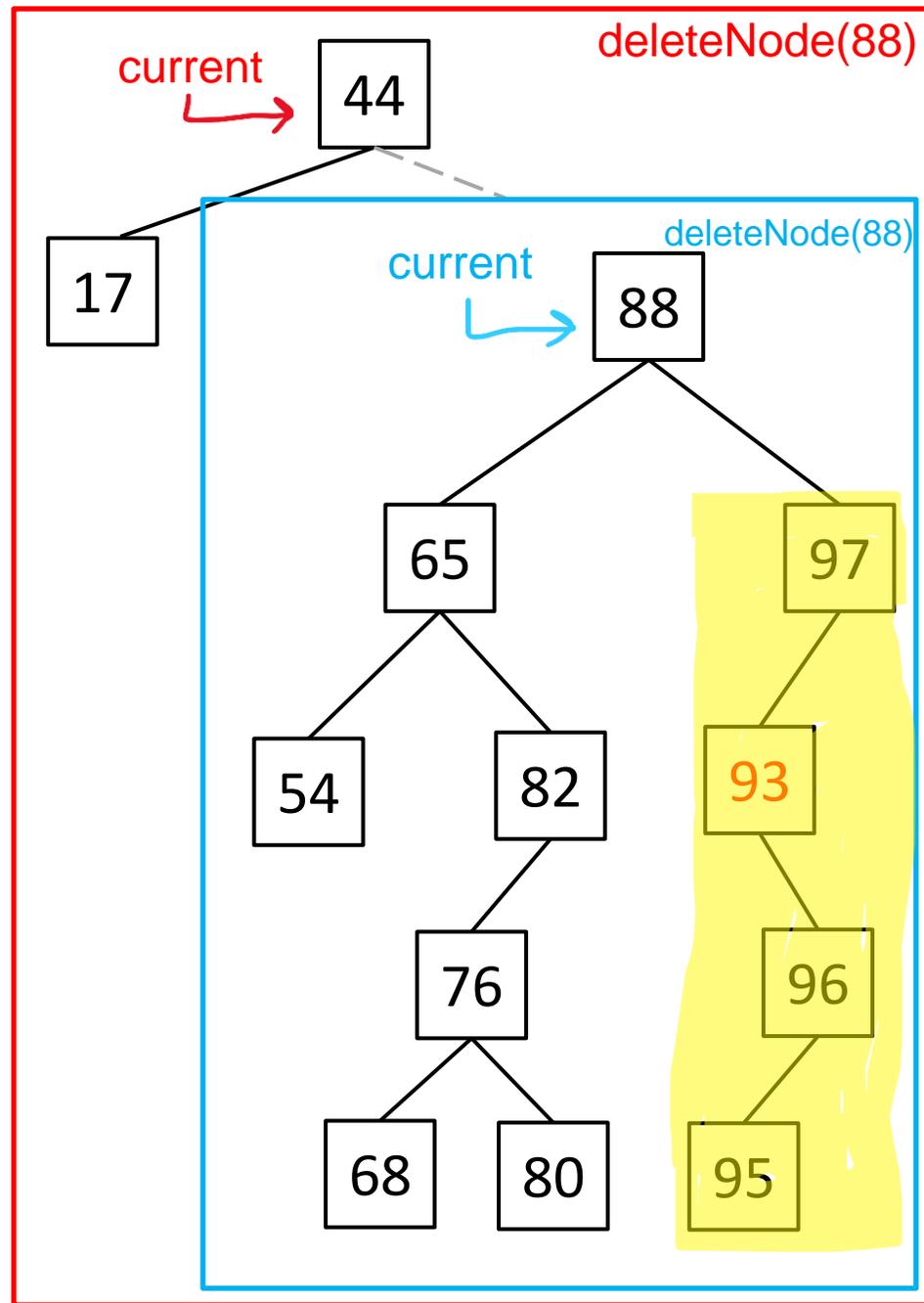


```

public Node findReplacement(Node current) {
    current = current.getRight();
    while(current != null && current.getLeft() != null) {
        current = current.getLeft();
    }
    return current;
}

```

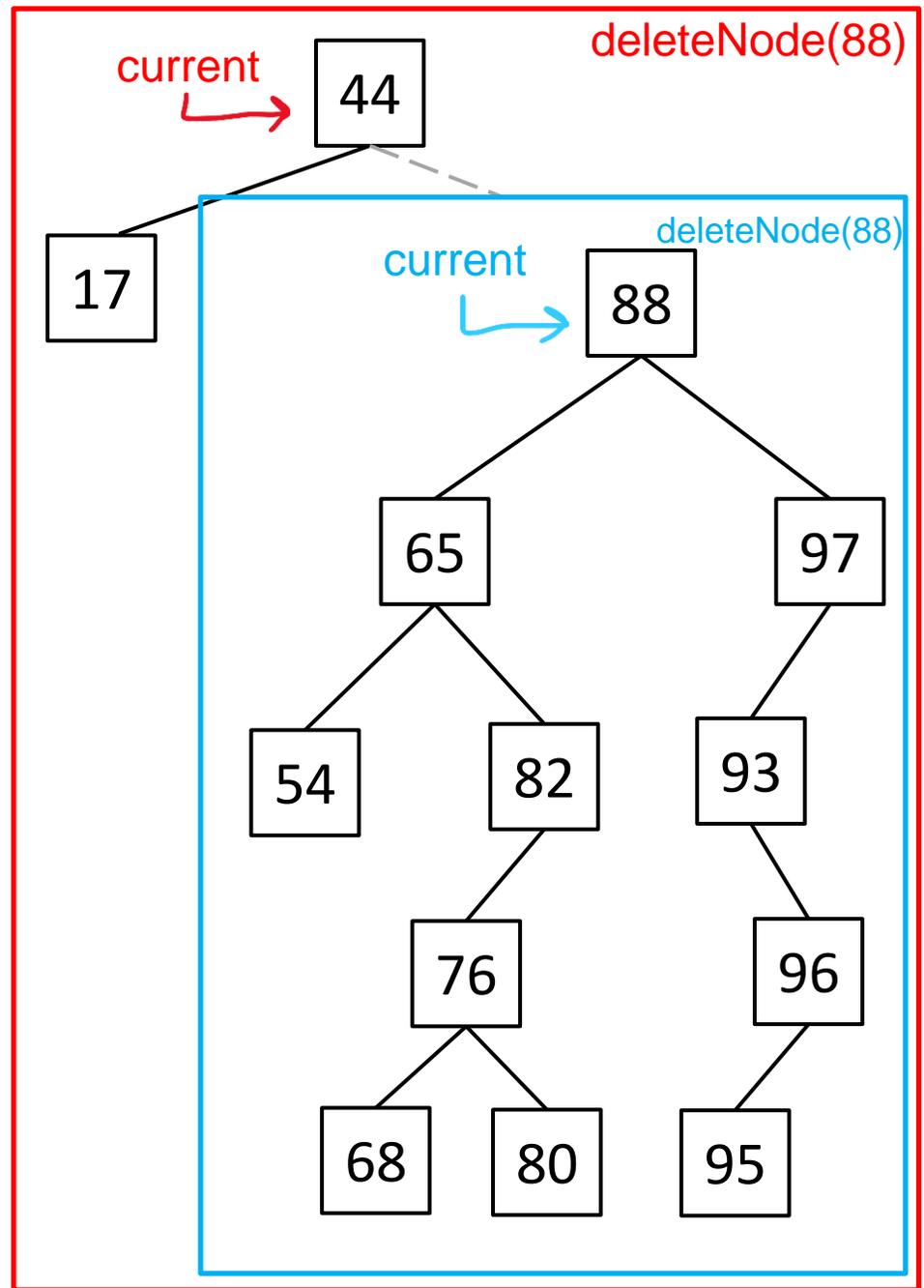
1. Go into right subtree
2. Keep going left in the subtree
3. When we cant go left anymore, return node we are at



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
    }
}

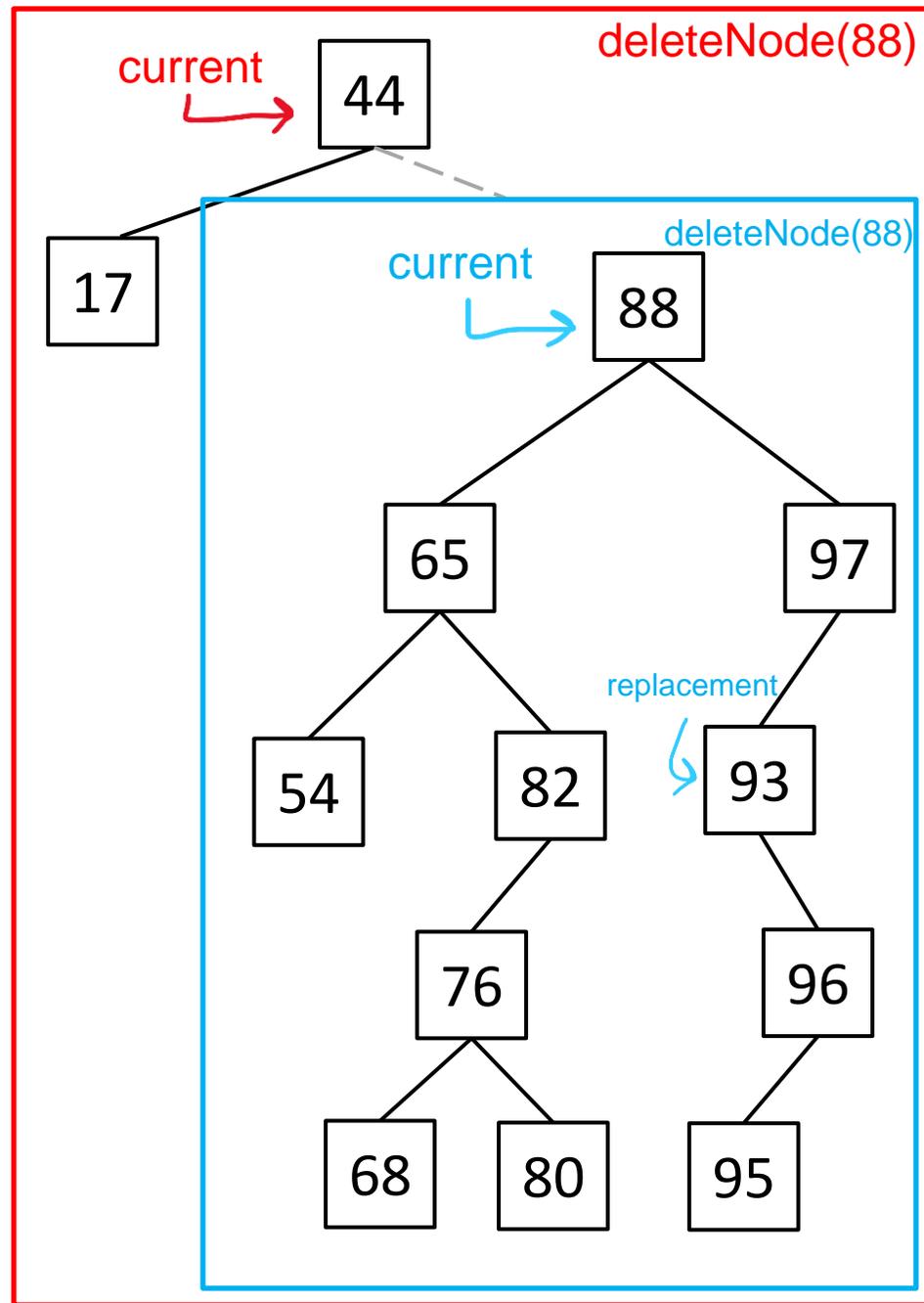
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        Current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
    }
}

```

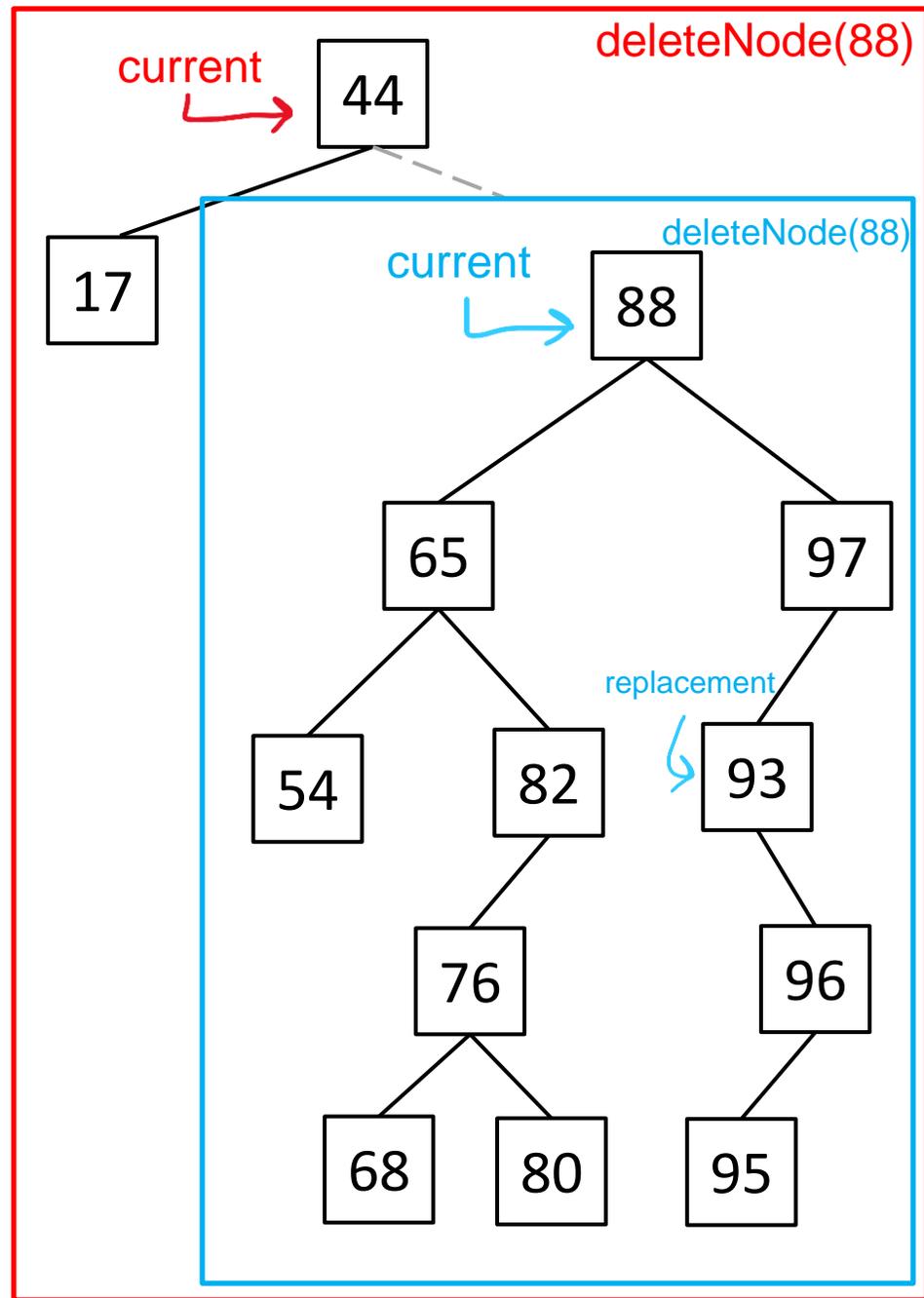


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
    }
}

```

Do our swap !

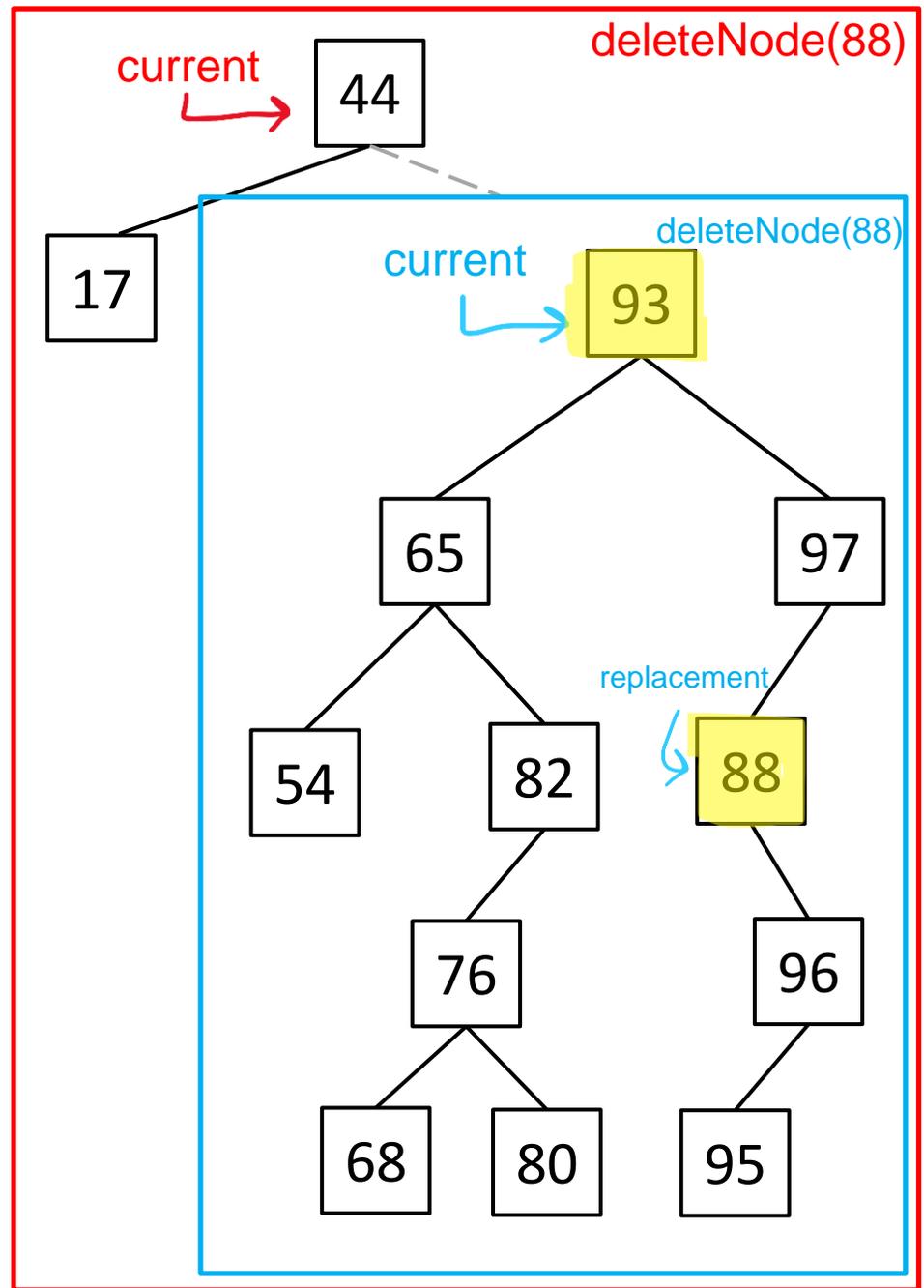


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
    }
}

```

Do the replacement!

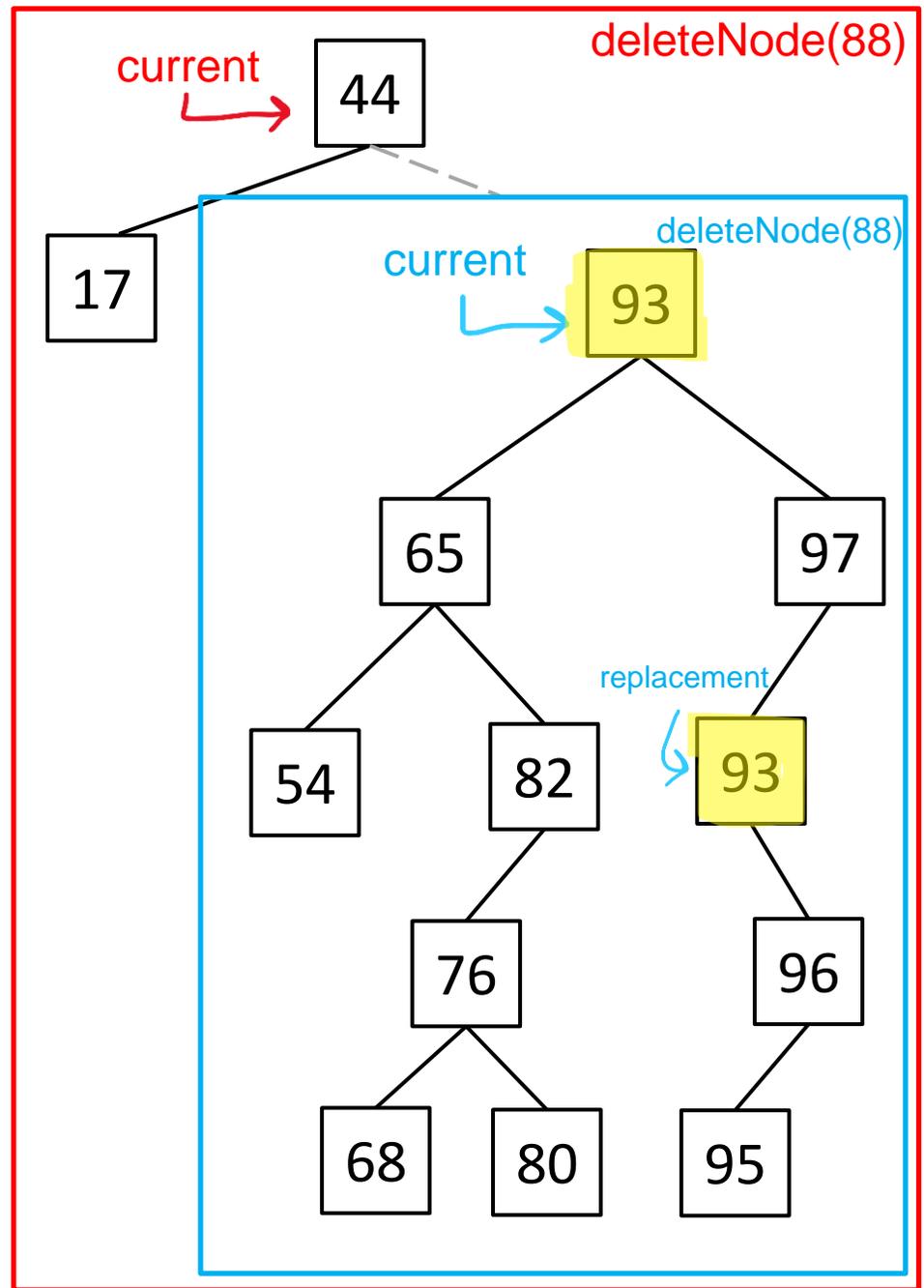


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
    }
}

```

93 is now a duplicate!

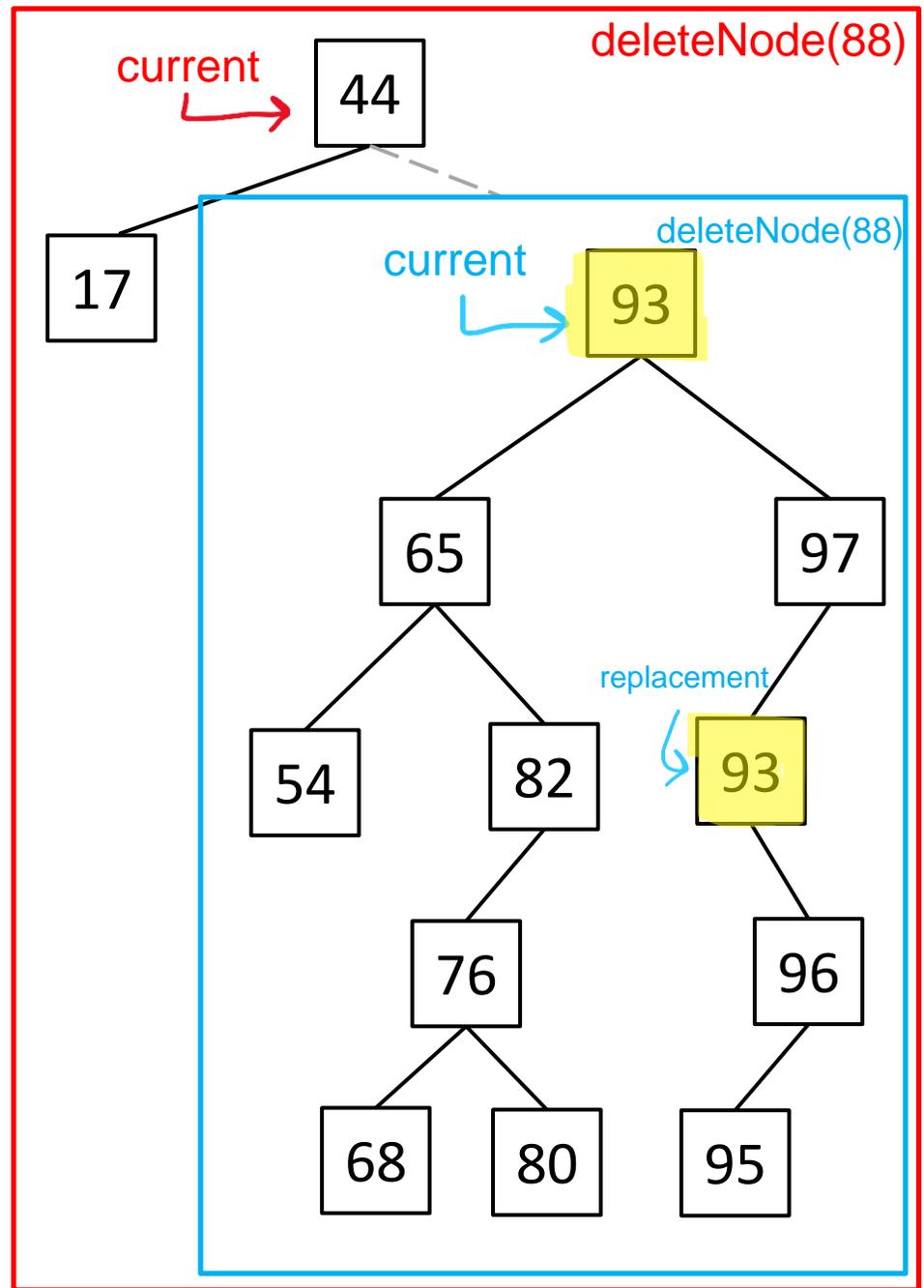


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
    }
}

```

How to remove a duplicate in a BST?

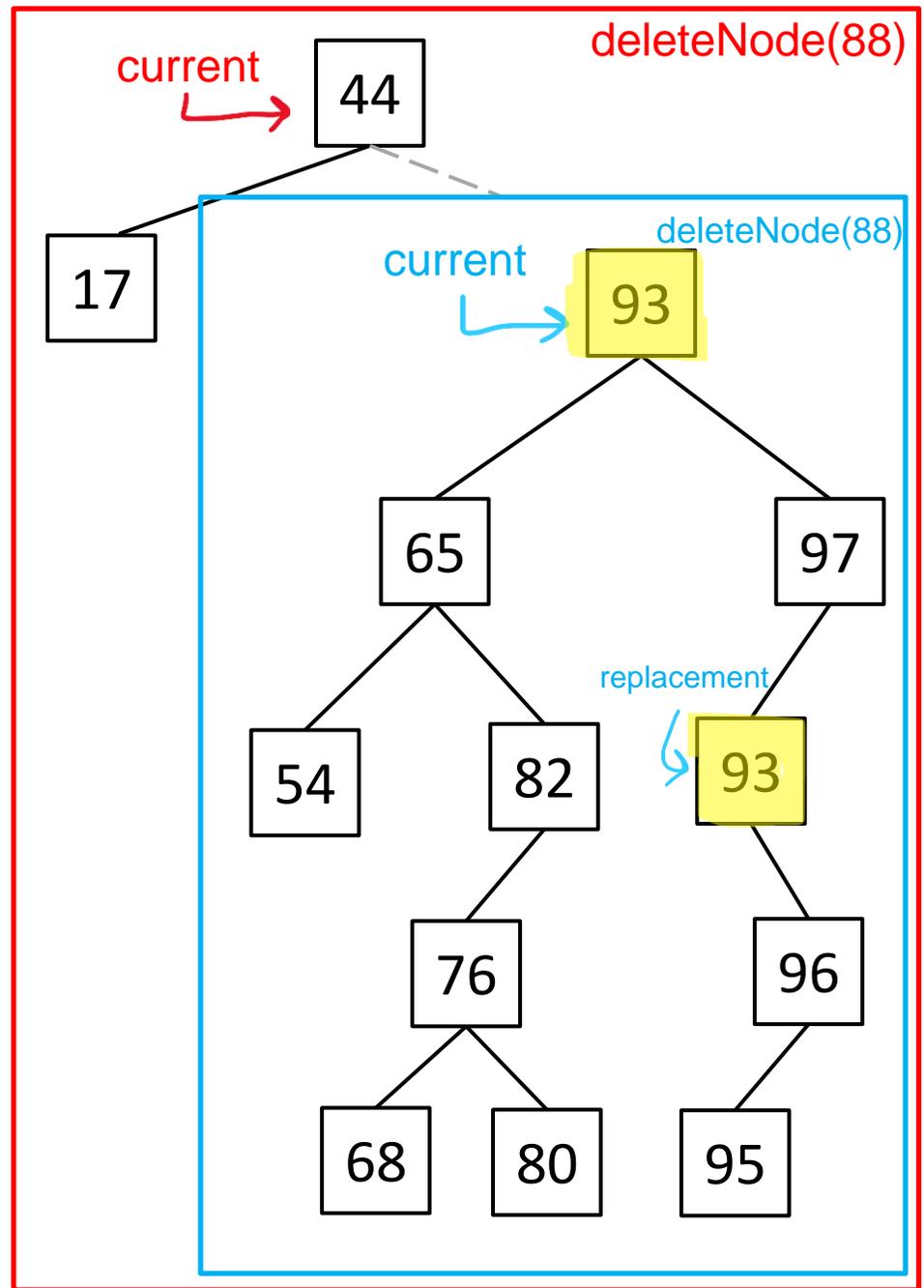


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
    }
}

```

How to remove a ~~duplicate~~ ^{node} in a BST?



```

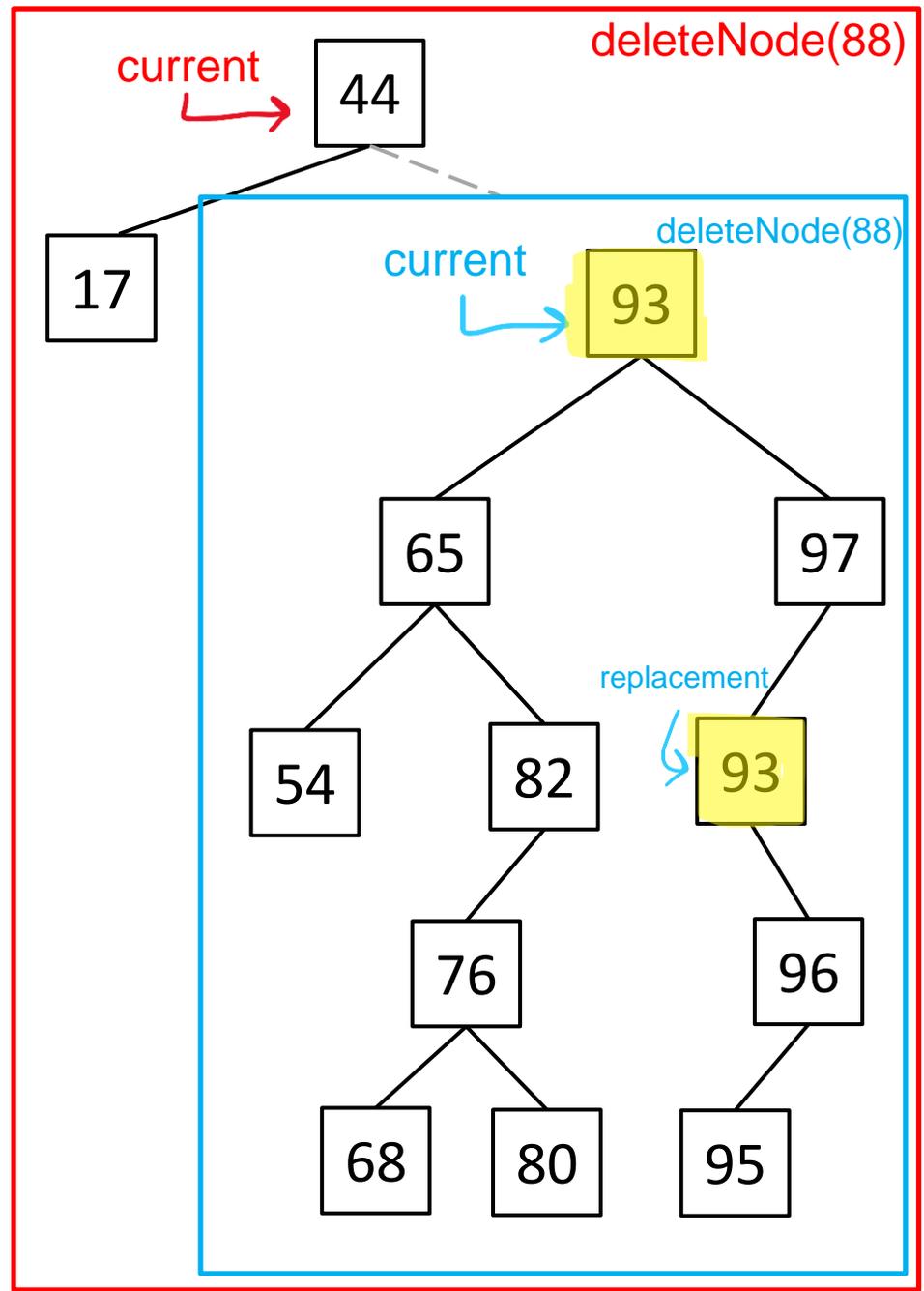
public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
    }
}

```

It's the method we are currently writing



How to remove a node in a BST? We have a method to do that

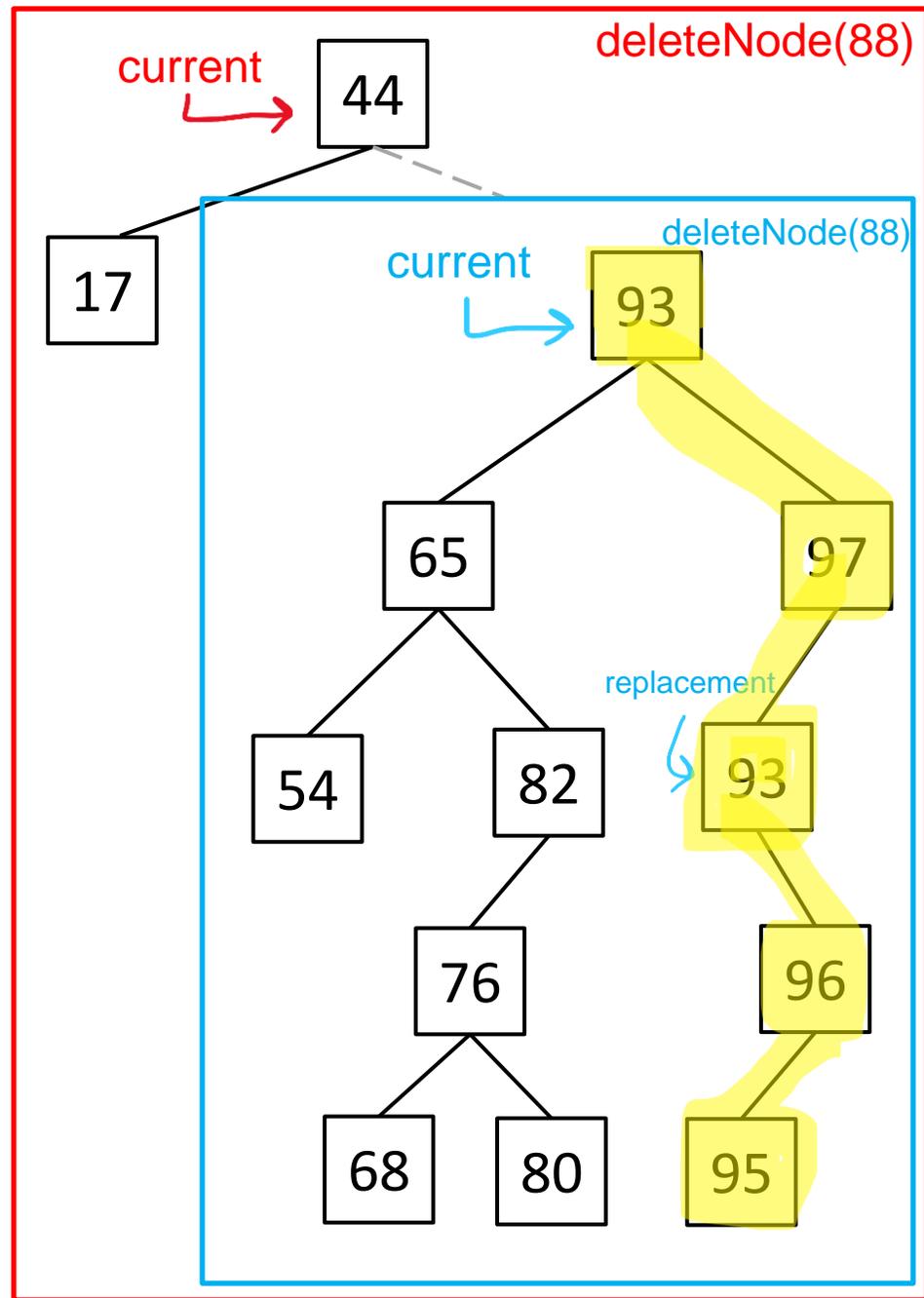


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
    }
}

```

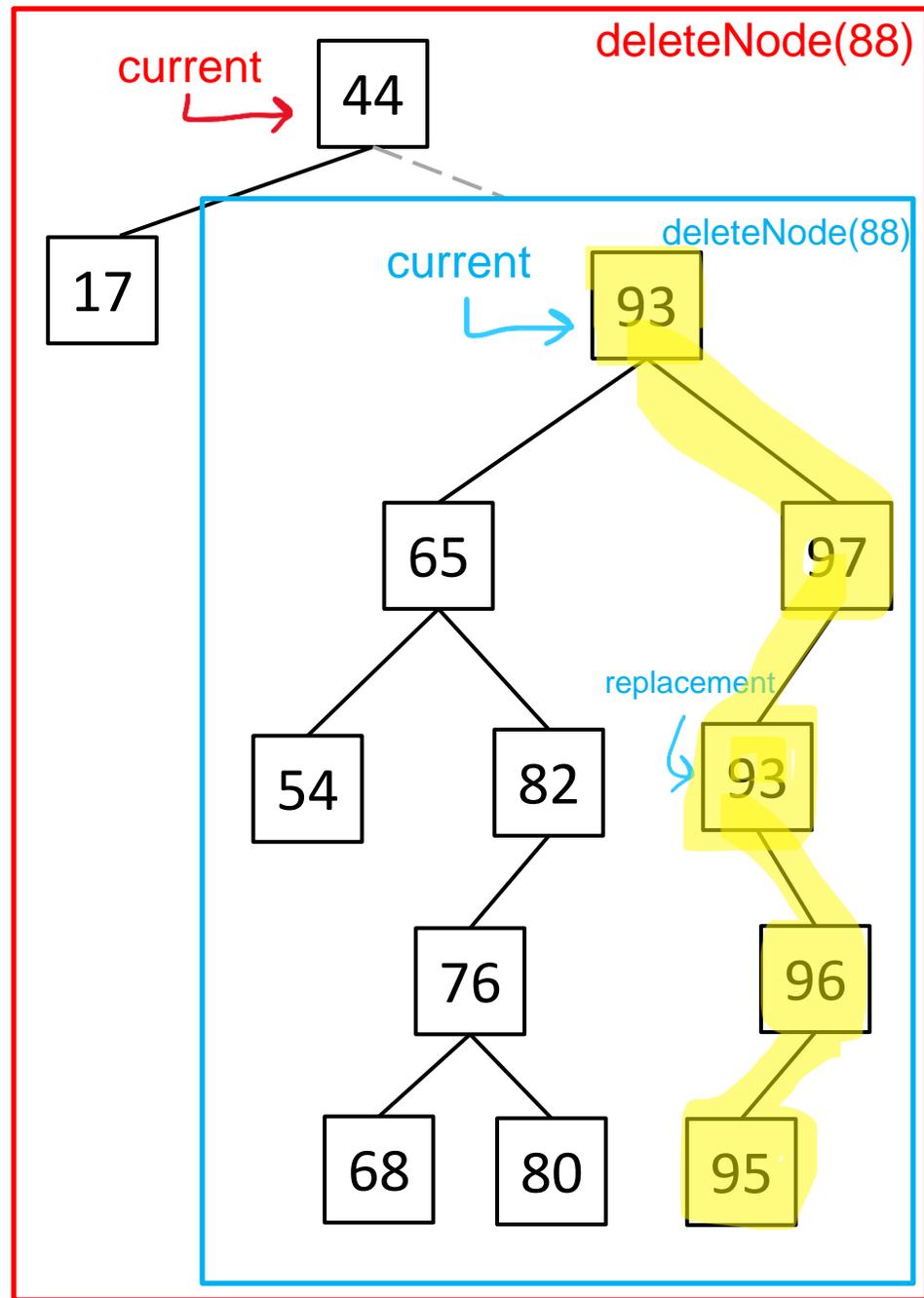
When the replacement was done, the only thing that was affected was the **right subtree**



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight( ??? );
    }
}

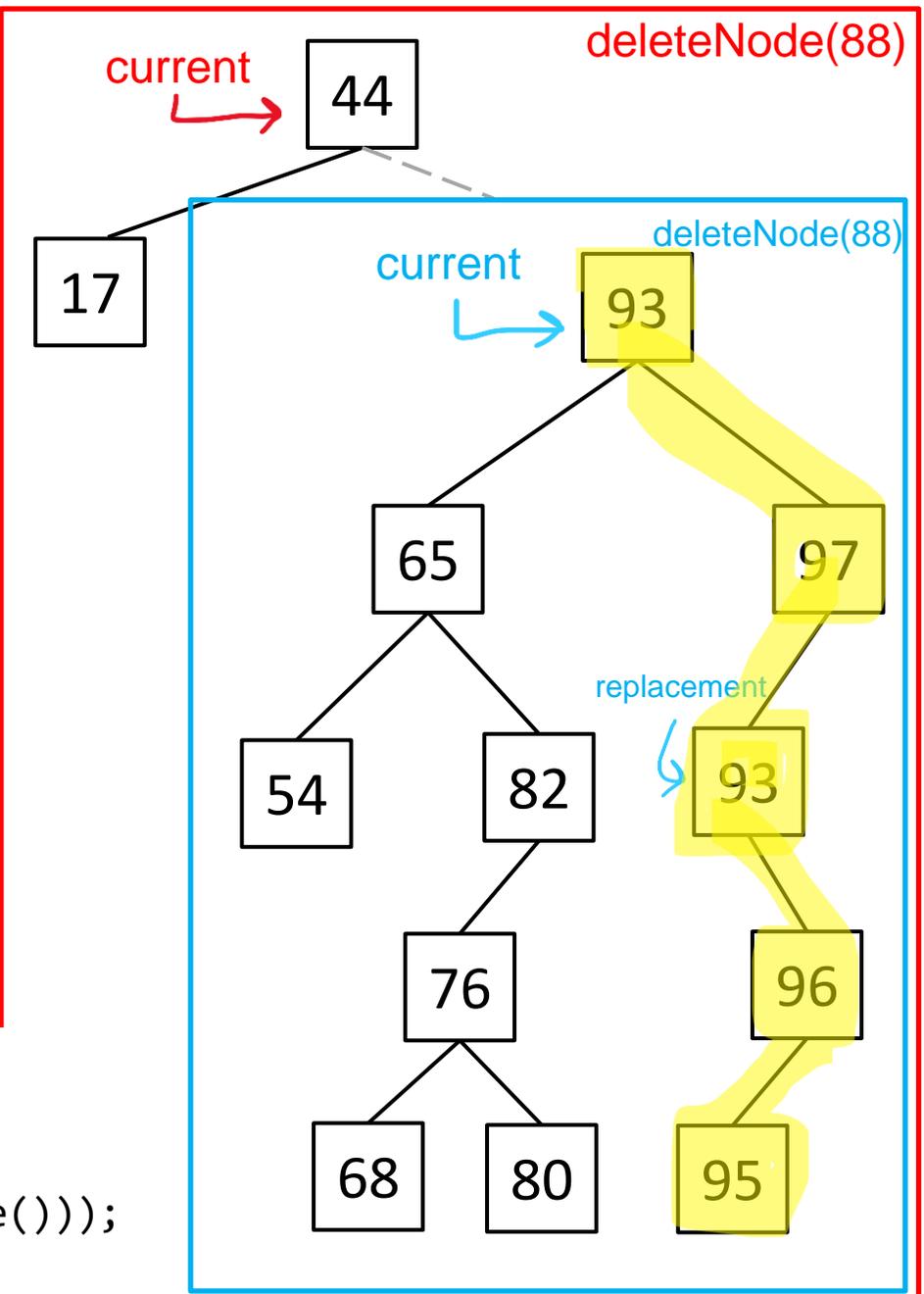
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacement.getValue()));
    }
}

```

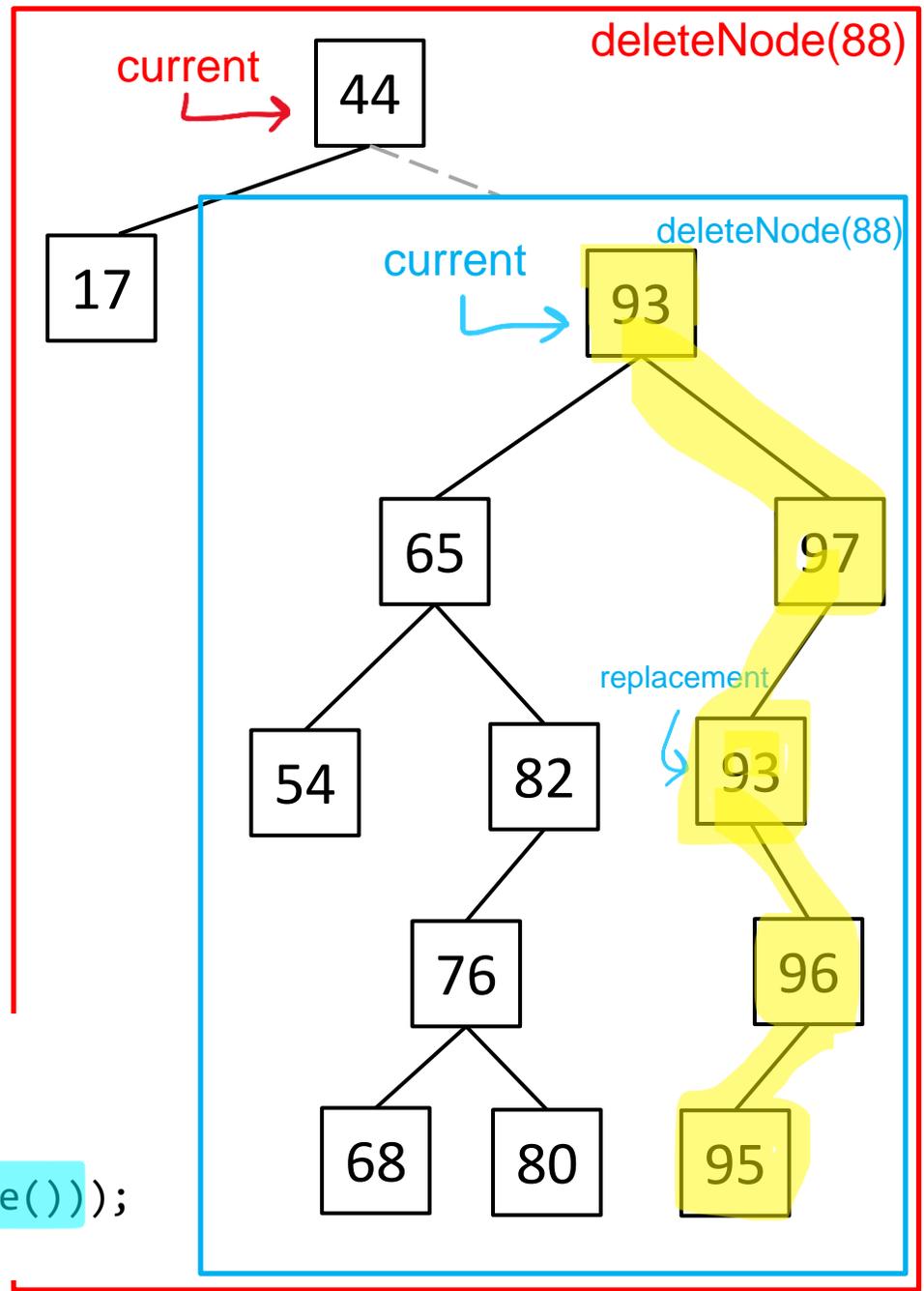


```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacement.getValue()));
    }
}

```

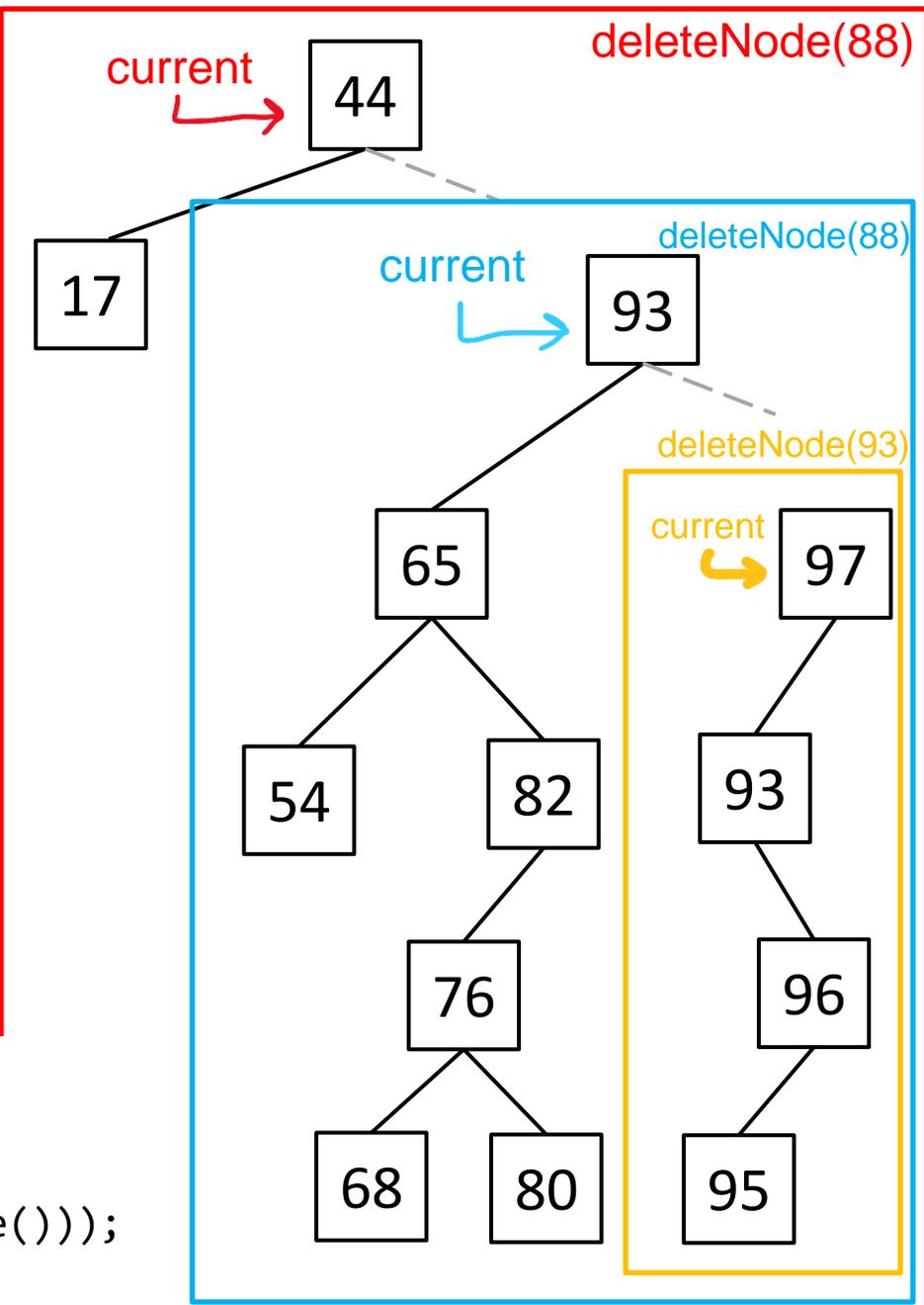
Recursively go through the right subtree and remove the duplicate value



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
}

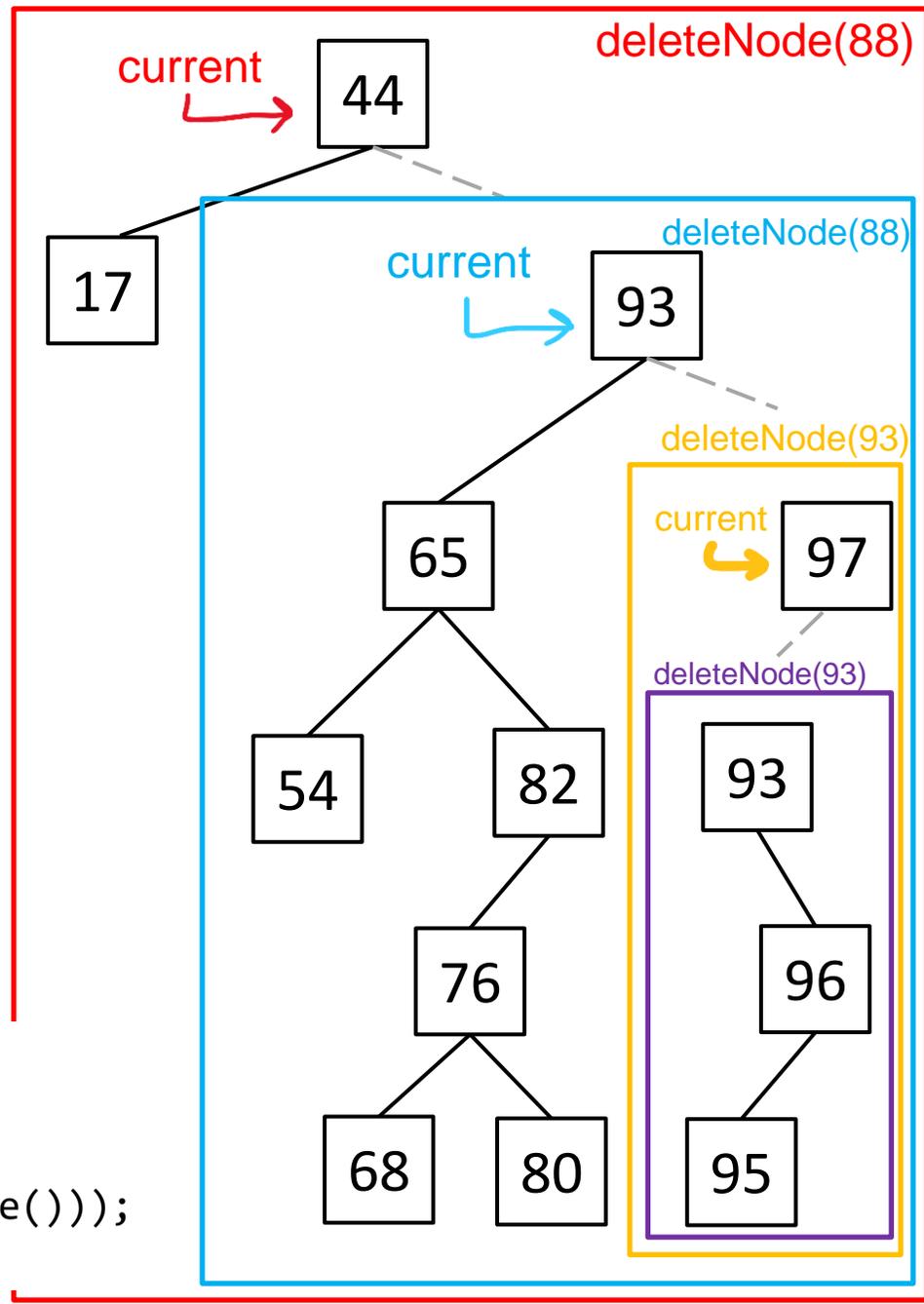
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
}

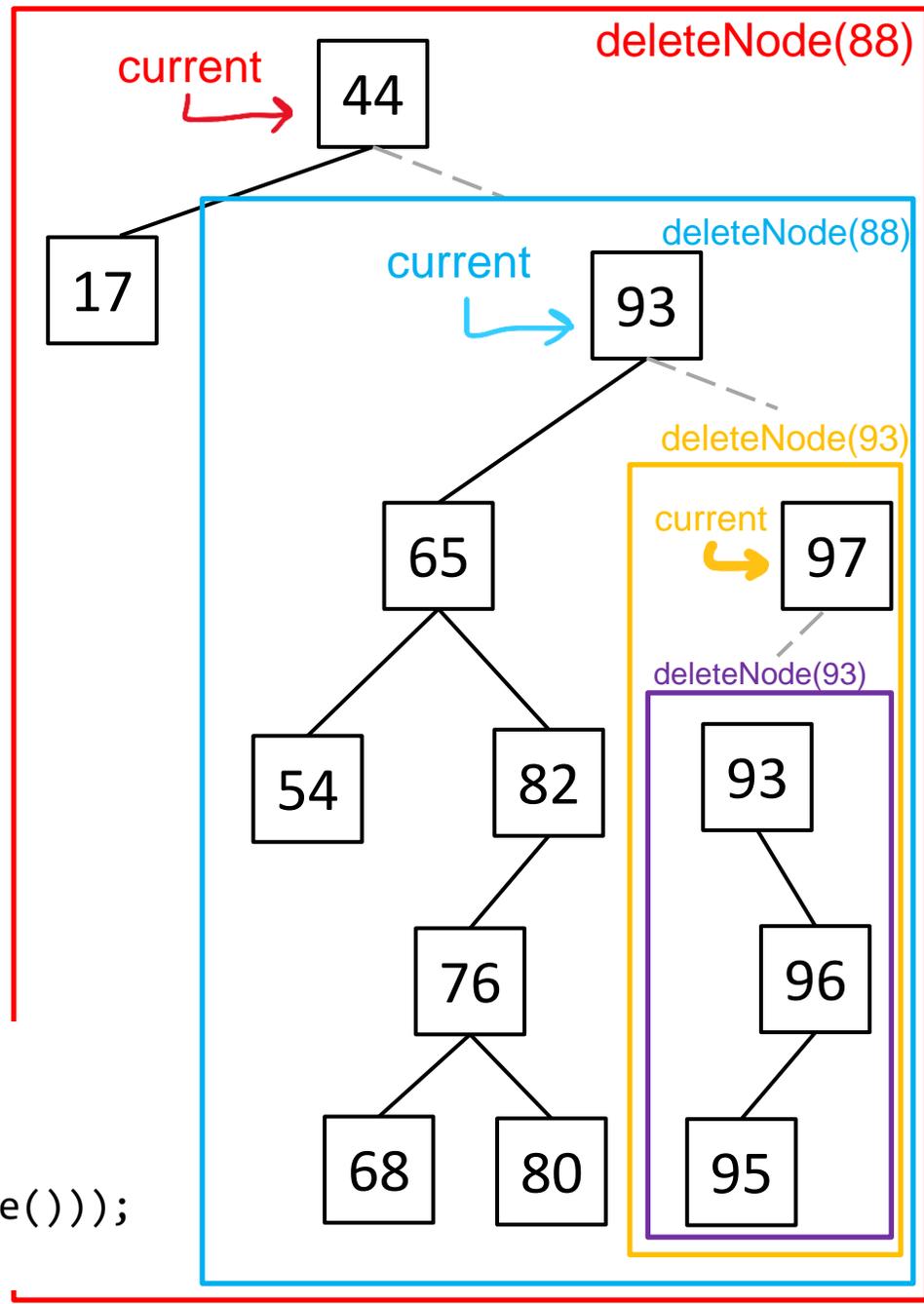
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
}

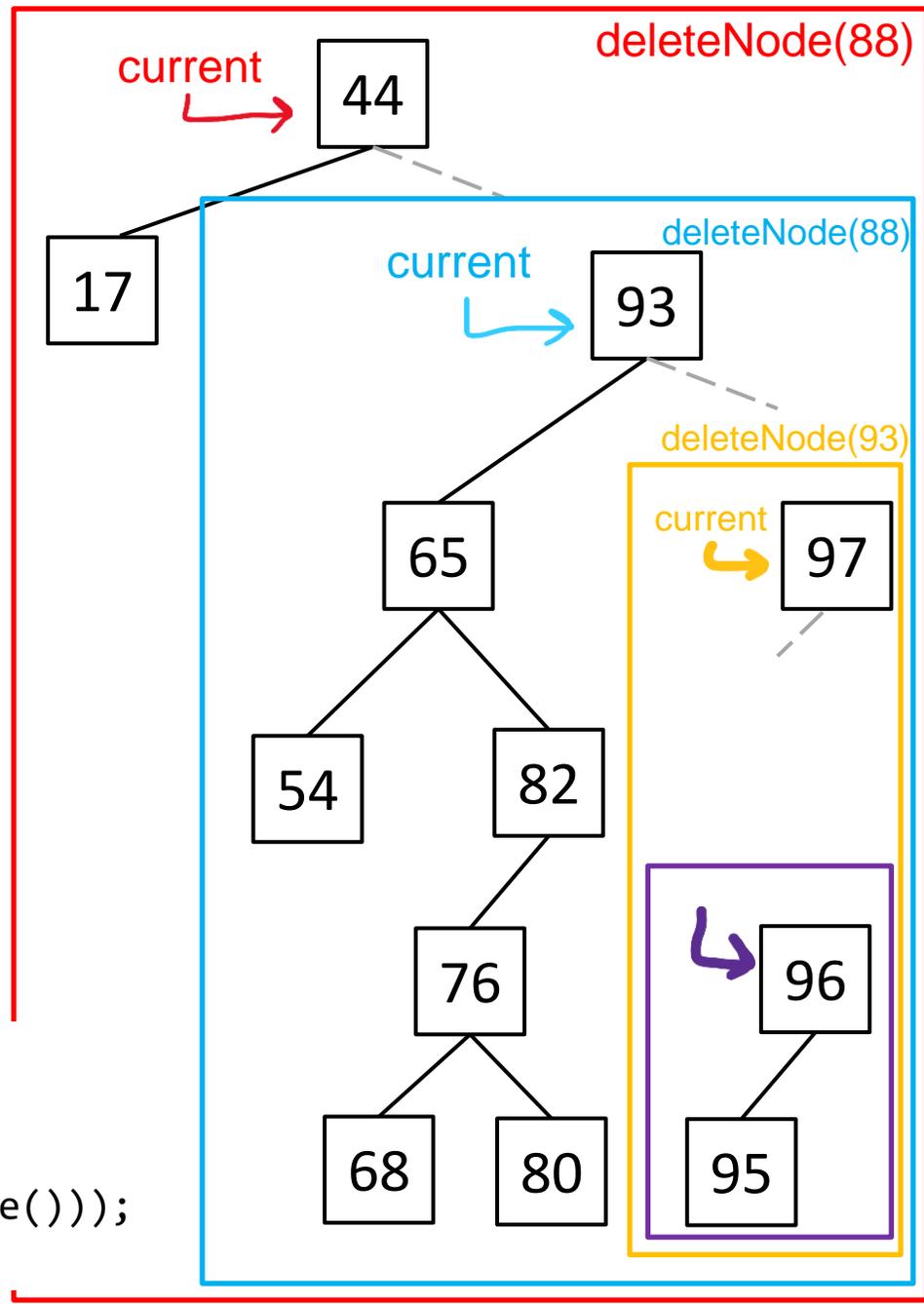
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
}

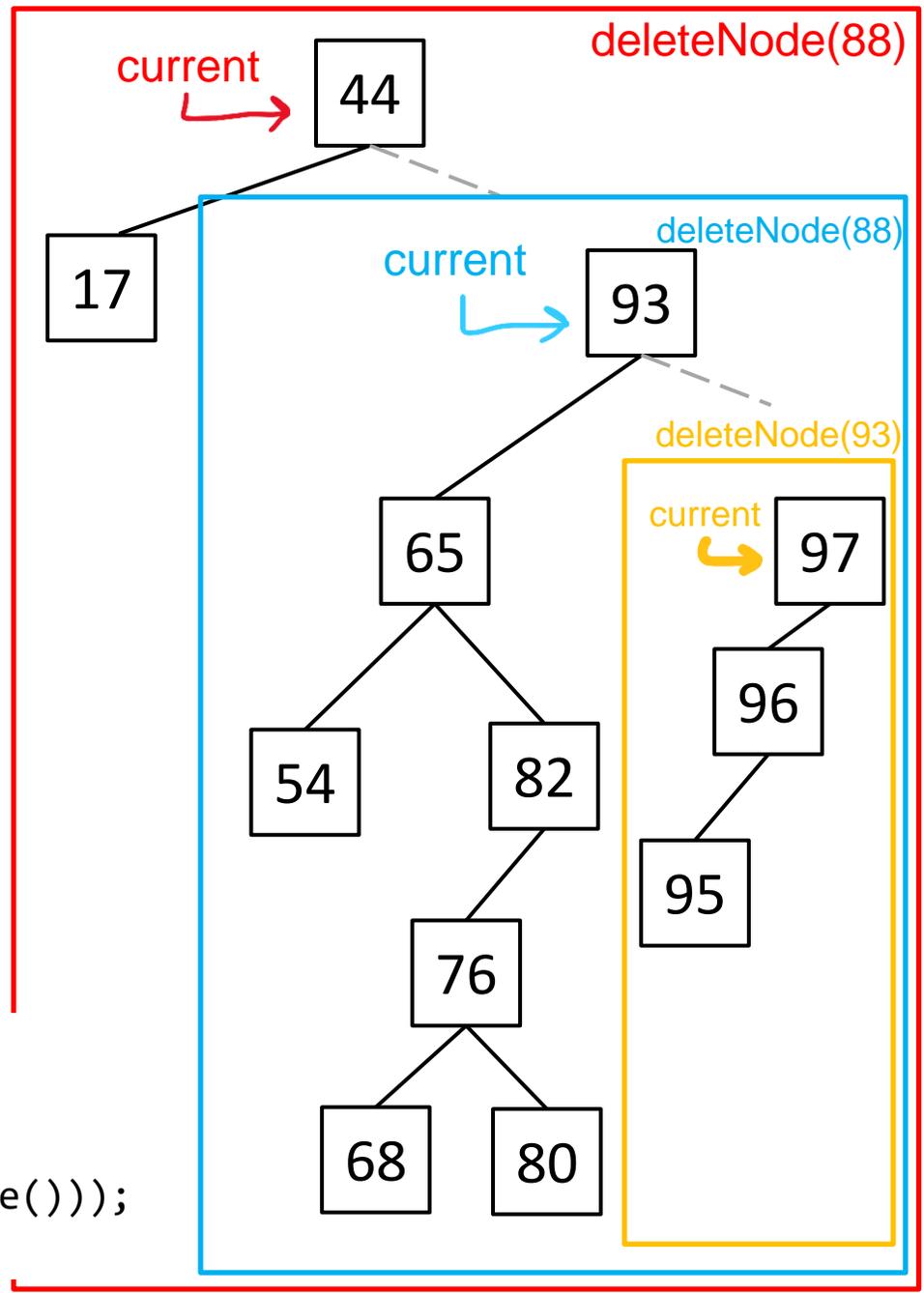
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        Current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
}

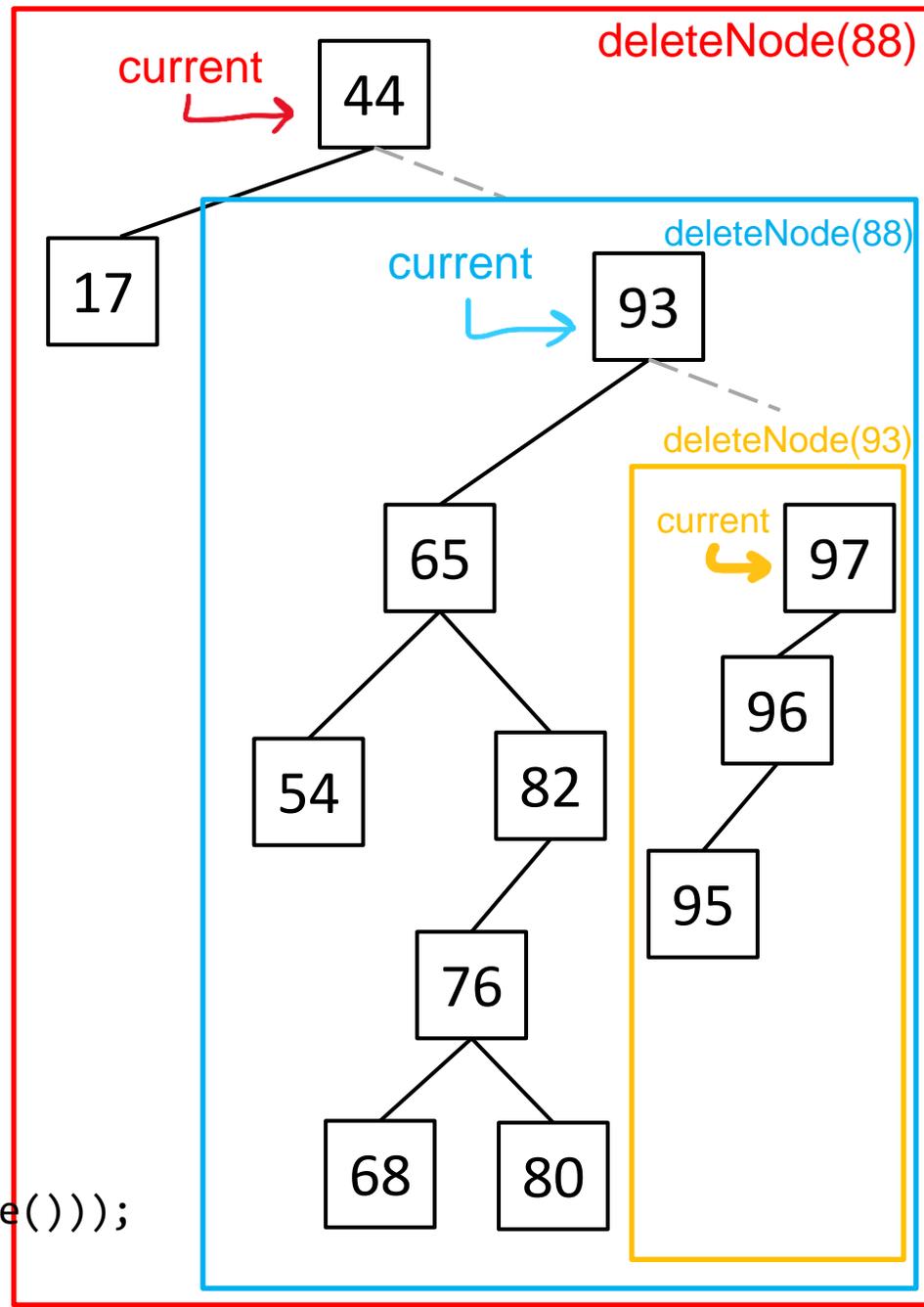
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacement.getValue()));
    }
    return current;
}

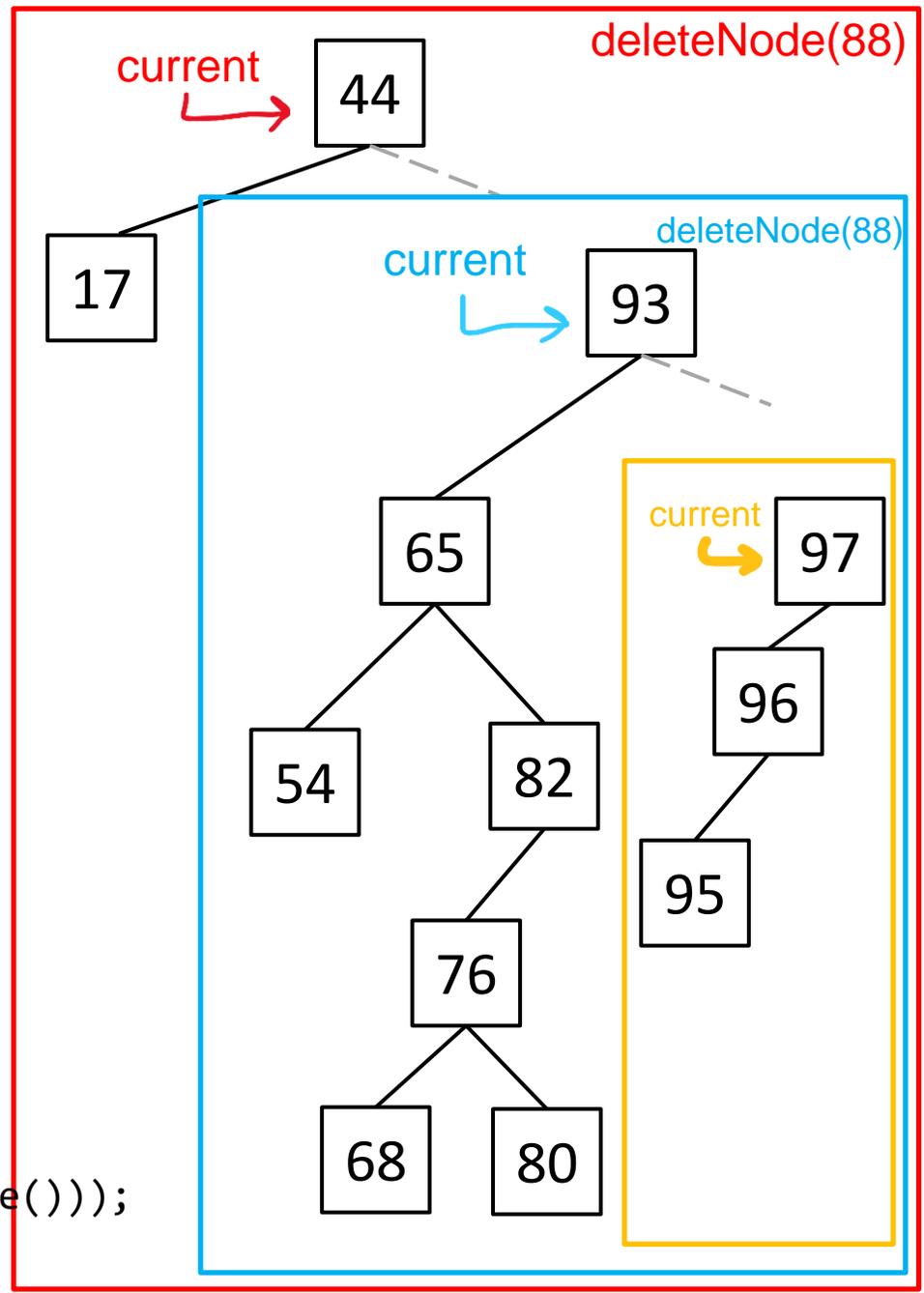
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
    return current;
}

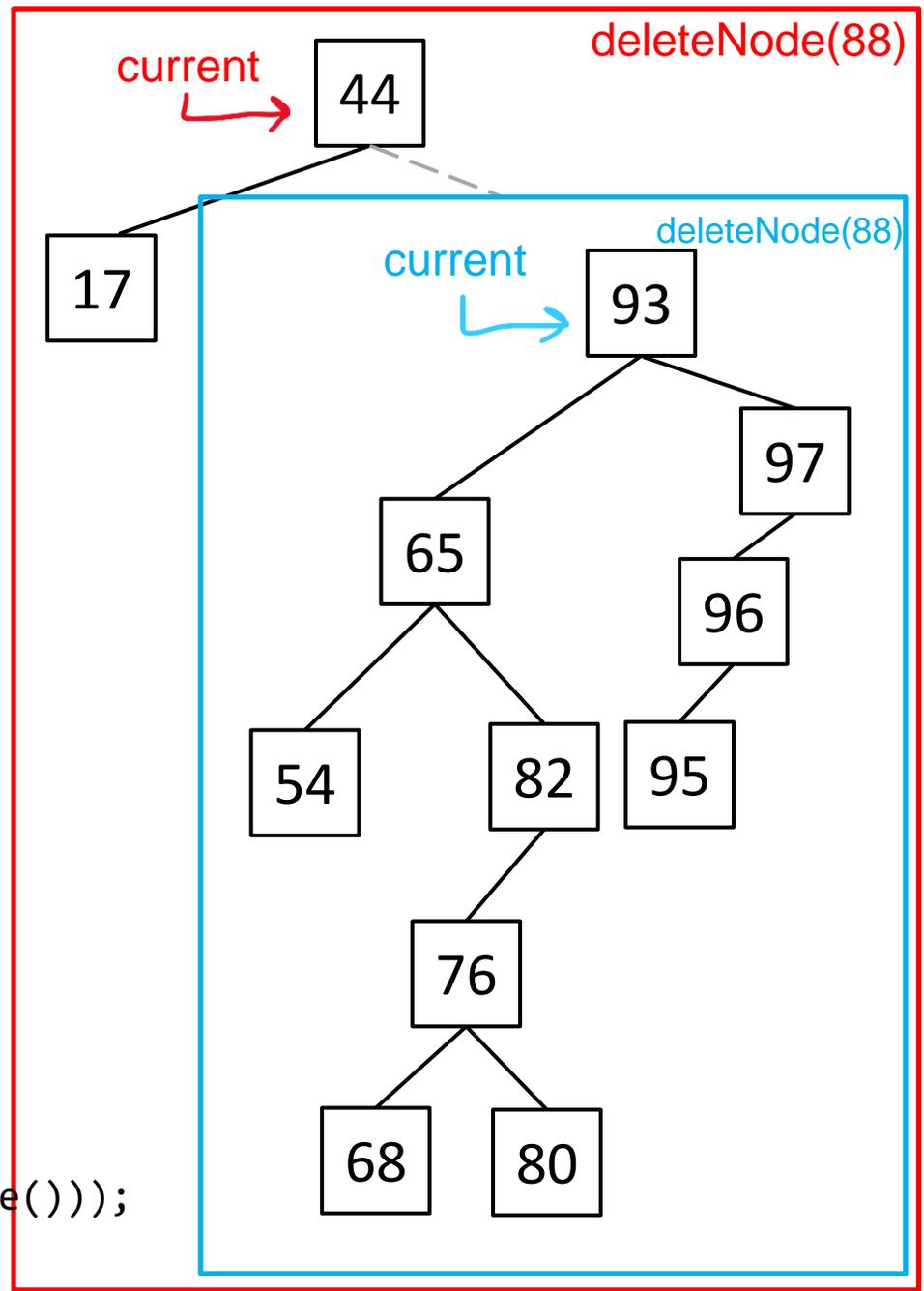
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        Current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
    return current;
}

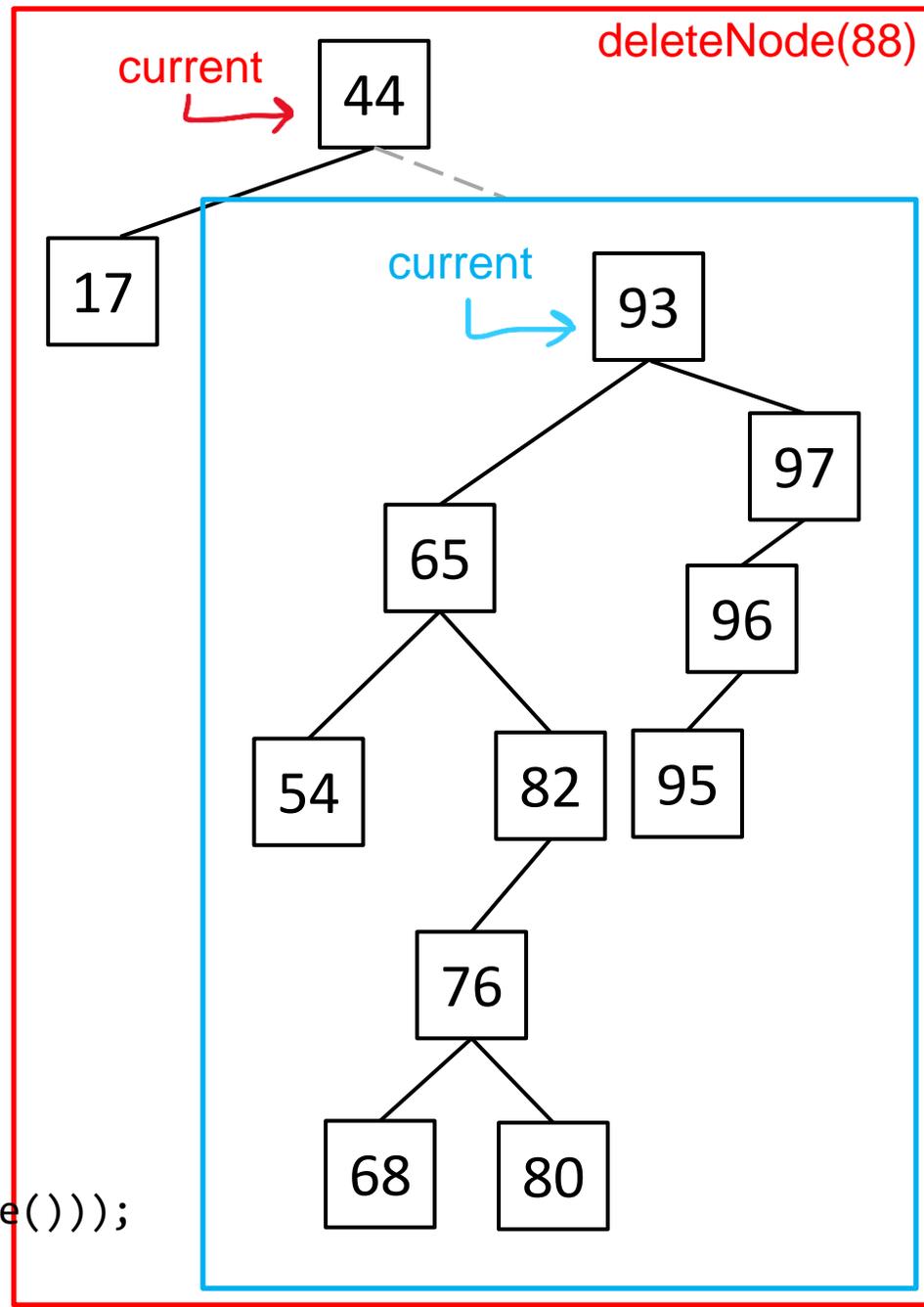
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
    return current;
}

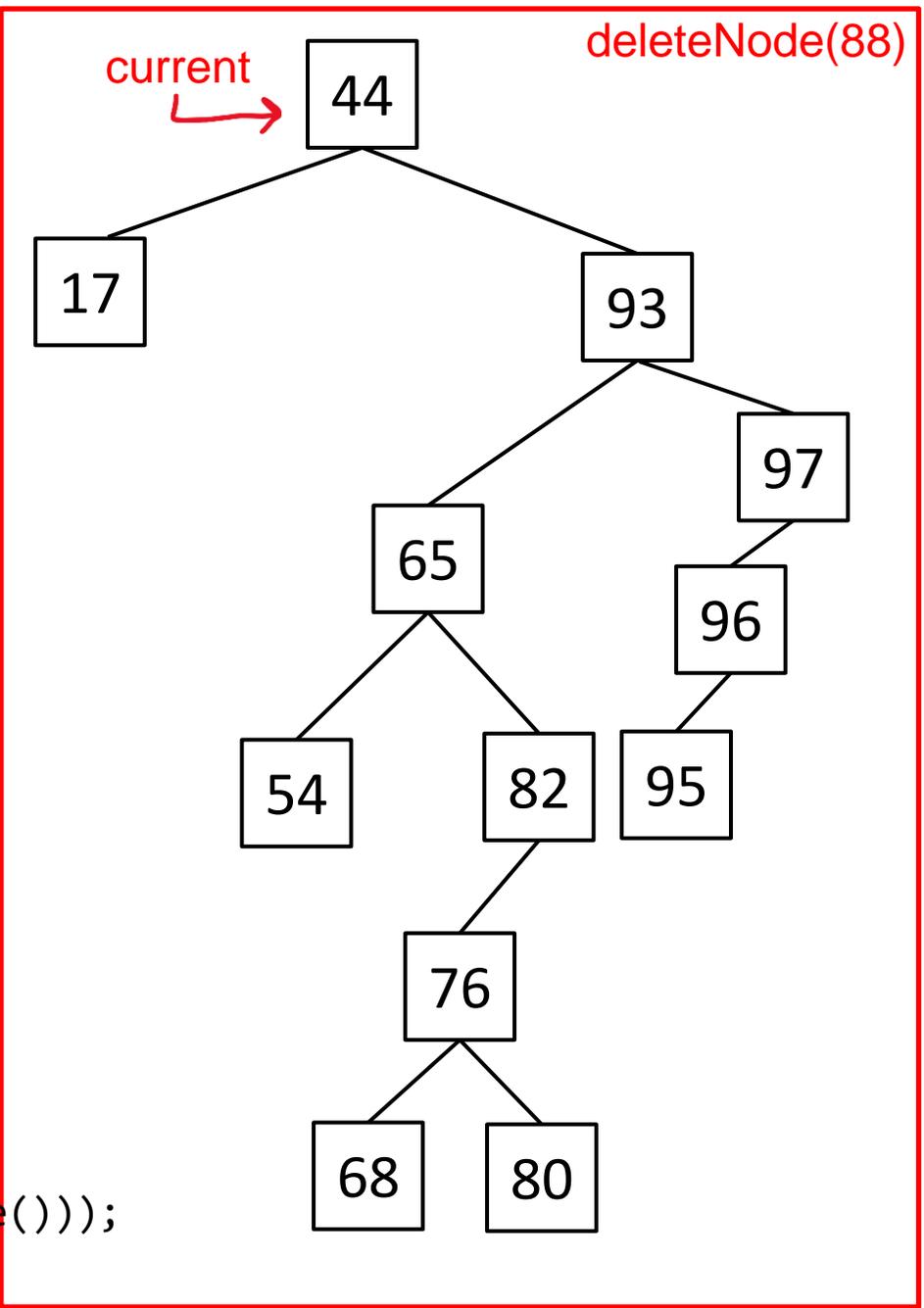
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
    return current;
}

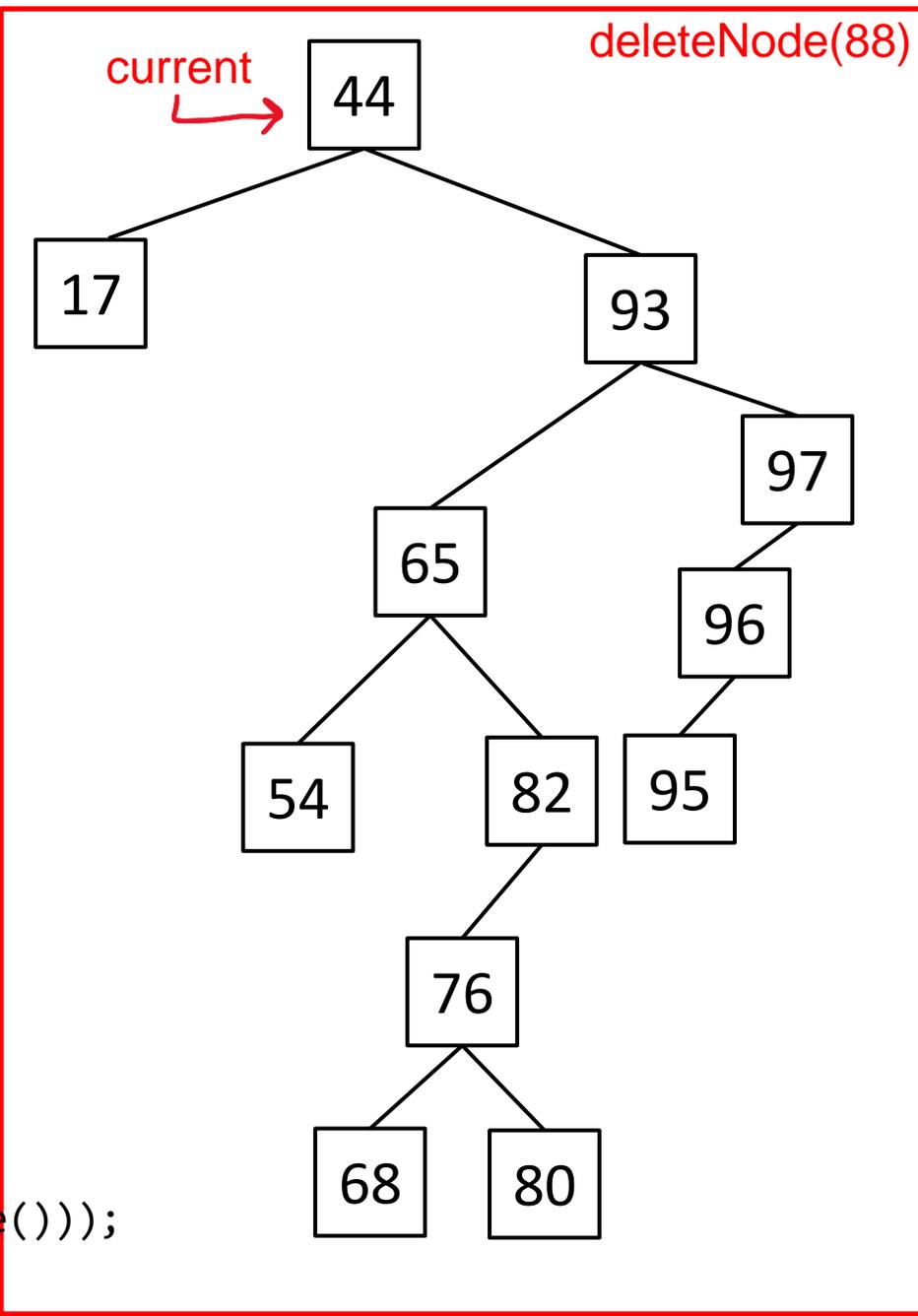
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
    return current;
}

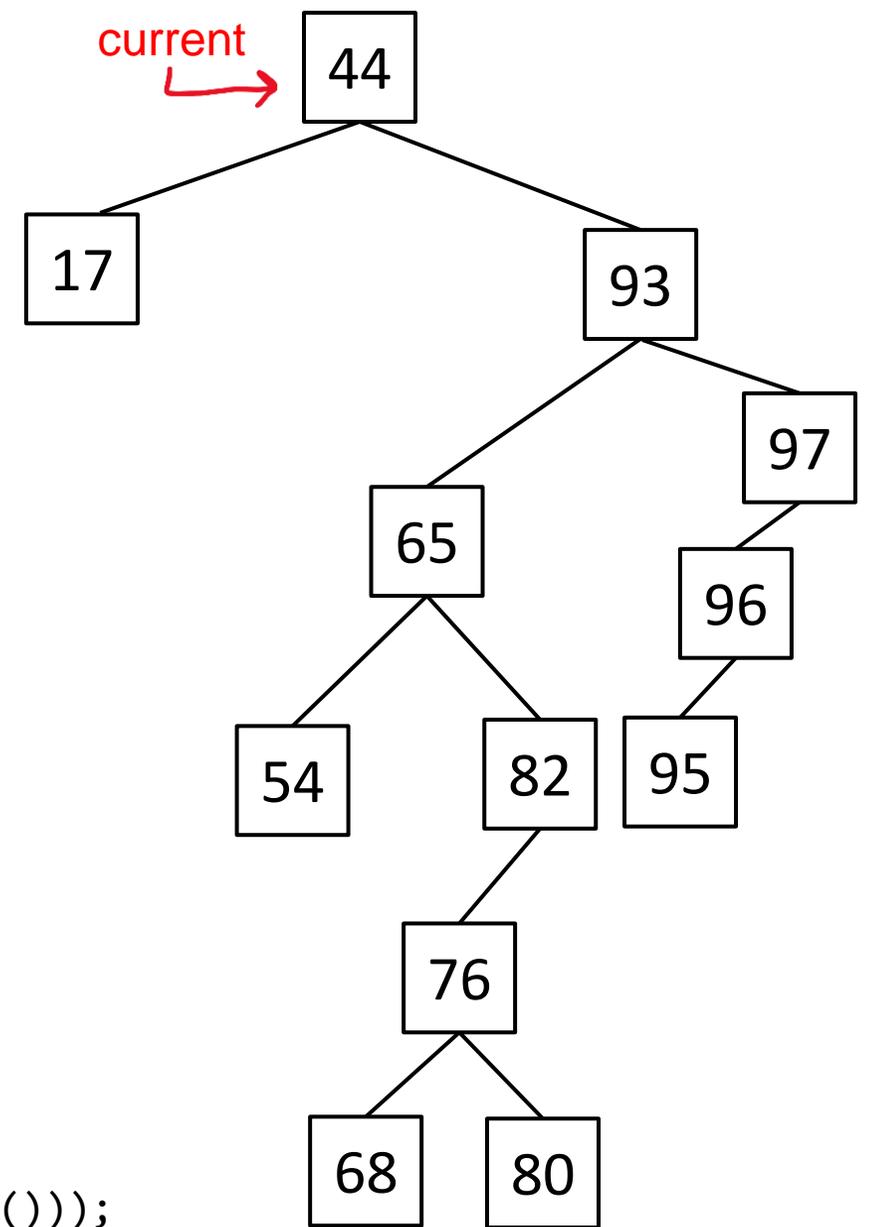
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
    return current;
}

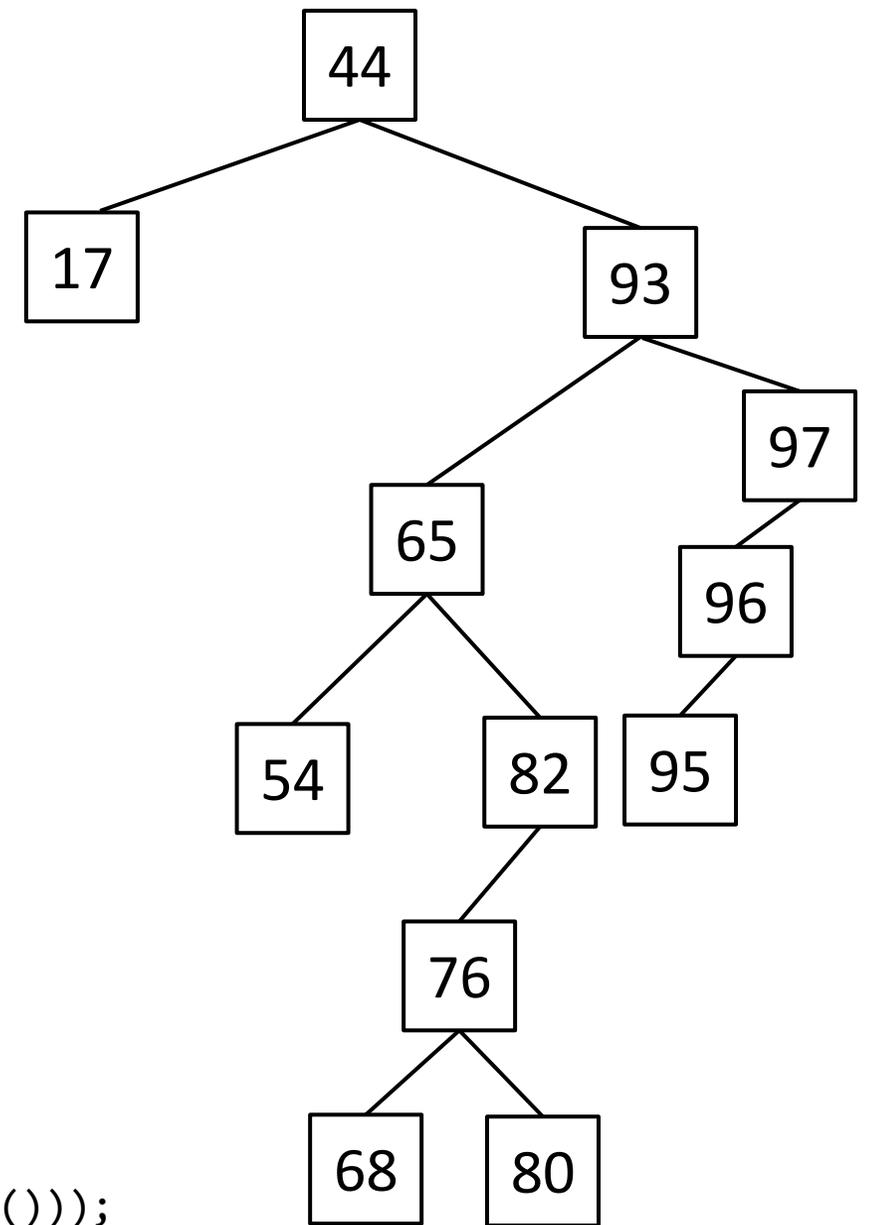
```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
    return current;
}

```



```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }

        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacement.getValue()));
    }
    return current;
}

```

1. Find value we are searching for (recursively)

```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacement.getValue()));
    }
    return current;
}

```

1. Find value we are searching for (recursively)

2. Check their children to determine how to find replacement

```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacment.getValue()));
    }
    return current;
}

```

1. Find value we are searching for (recursively)

2. Check their children to determine how to find replacement

3. (In case of 2 children) Find replacement/successor of removed node

```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }

        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacement.getValue()));
    }
    return current;
}

```

1. Find value we are searching for (recursively)

2. Check their children to determine how to find replacement

3. (In case of 2 children) Find replacement/successor of removed node

4. Copy replacement value into removed node

```

public Node deleteNode(Node current, int searchValue) {
    if (current == null) {
        return current;
    }
    if (current.getValue() > searchValue) {
        current.setLeft( deleteNode(current.getLeft(), searchValue));
    }
    else if (current.getValue() < searchValue) {
        current.setRight( deleteNode(current.getRight(), searchValue));
    }
    else {
        // only right child
        if (current.getLeft() == null) {
            return current.getRight();
        }
        // only left child
        if (current.getRight() == null) {
            return current.getLeft();
        }
        // When both children are present
        Node replacement = findReplacement(current);
        current.setValue(replacement.getValue());
        current.setRight(deleteNode(current.getRight(), replacement.getValue()));
    }
    return current;
}

```

1. Find value we are searching for (recursively)

2. Check their children to determine how to find replacement

3. (In case of 2 children) Find replacement/successor of removed node

4. Copy replacement value into removed node

5. Remove duplicate

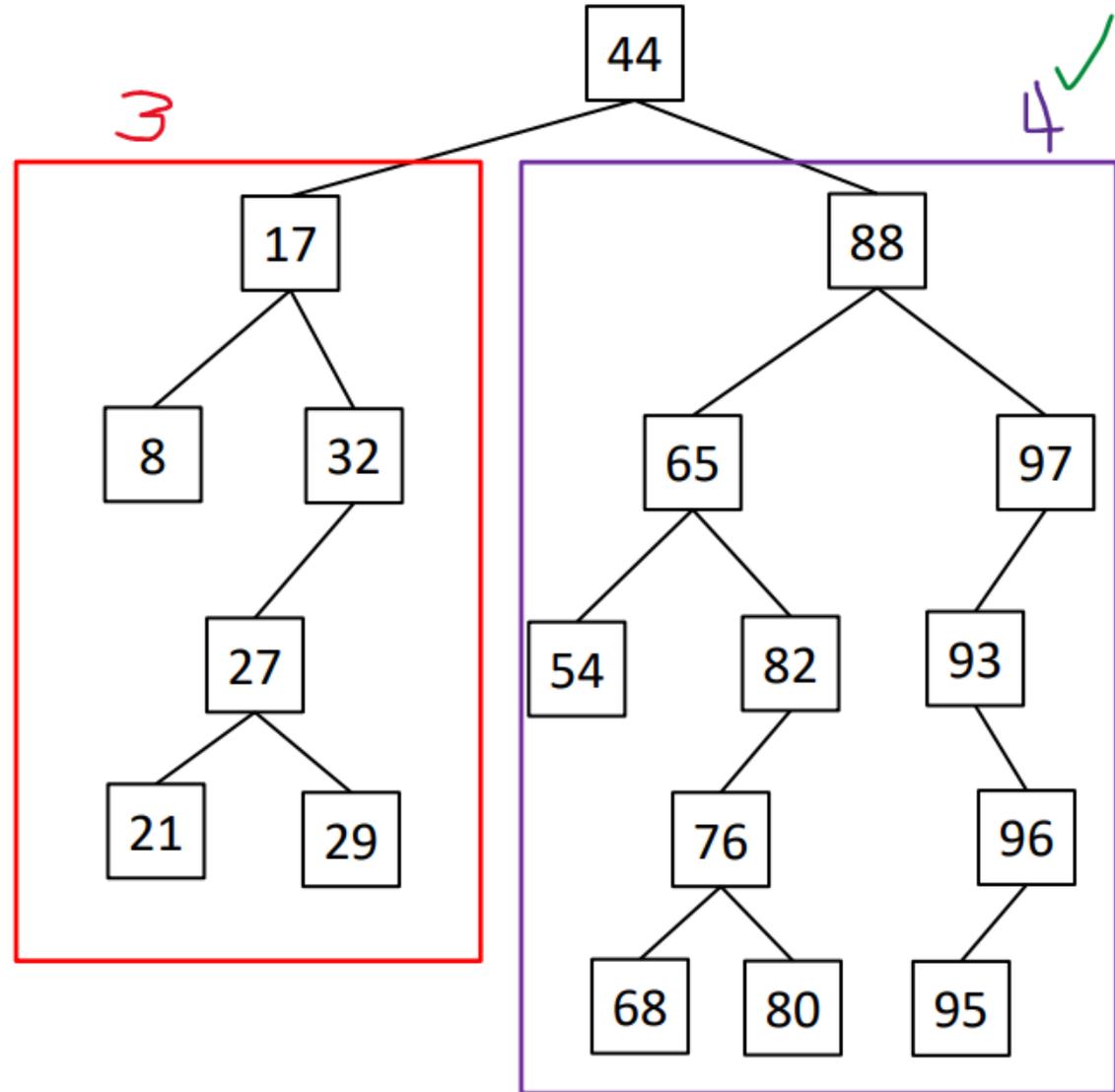
Find height of tree

1. Recursively find height of left subtree and right subtree

2. Select maximum value

3. Return 1 + maximum value

```
public int findHeight(Node current) {  
    if(current == null) {  
        return -1;  
    }  
    else {  
        int rightHeight = findHeight(current.getRight());  
        int leftHeight = findHeight(current.getLeft());  
        return 1 + Math.max(rightHeight, leftHeight);  
    }  
}
```



Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

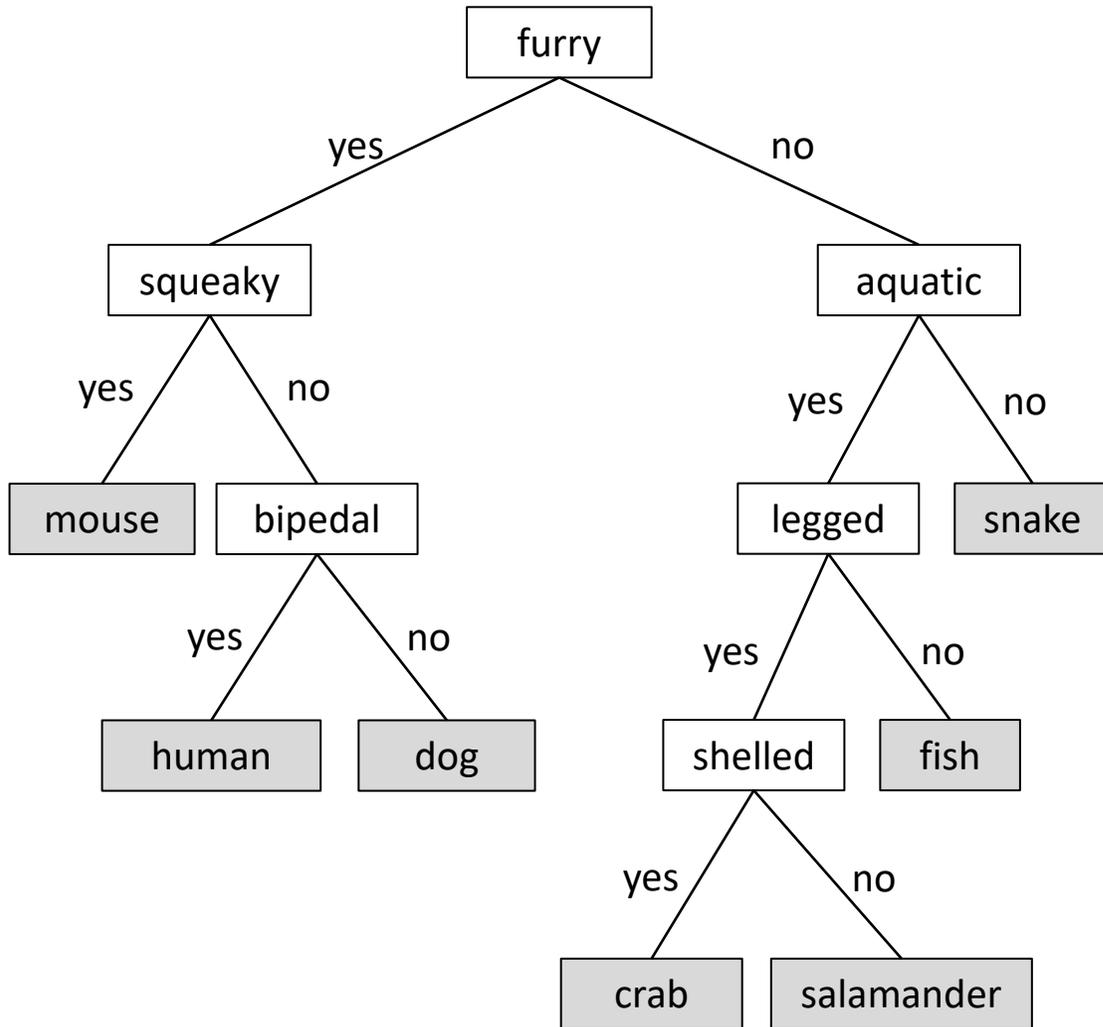
I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive

Program Execution:

1.



Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

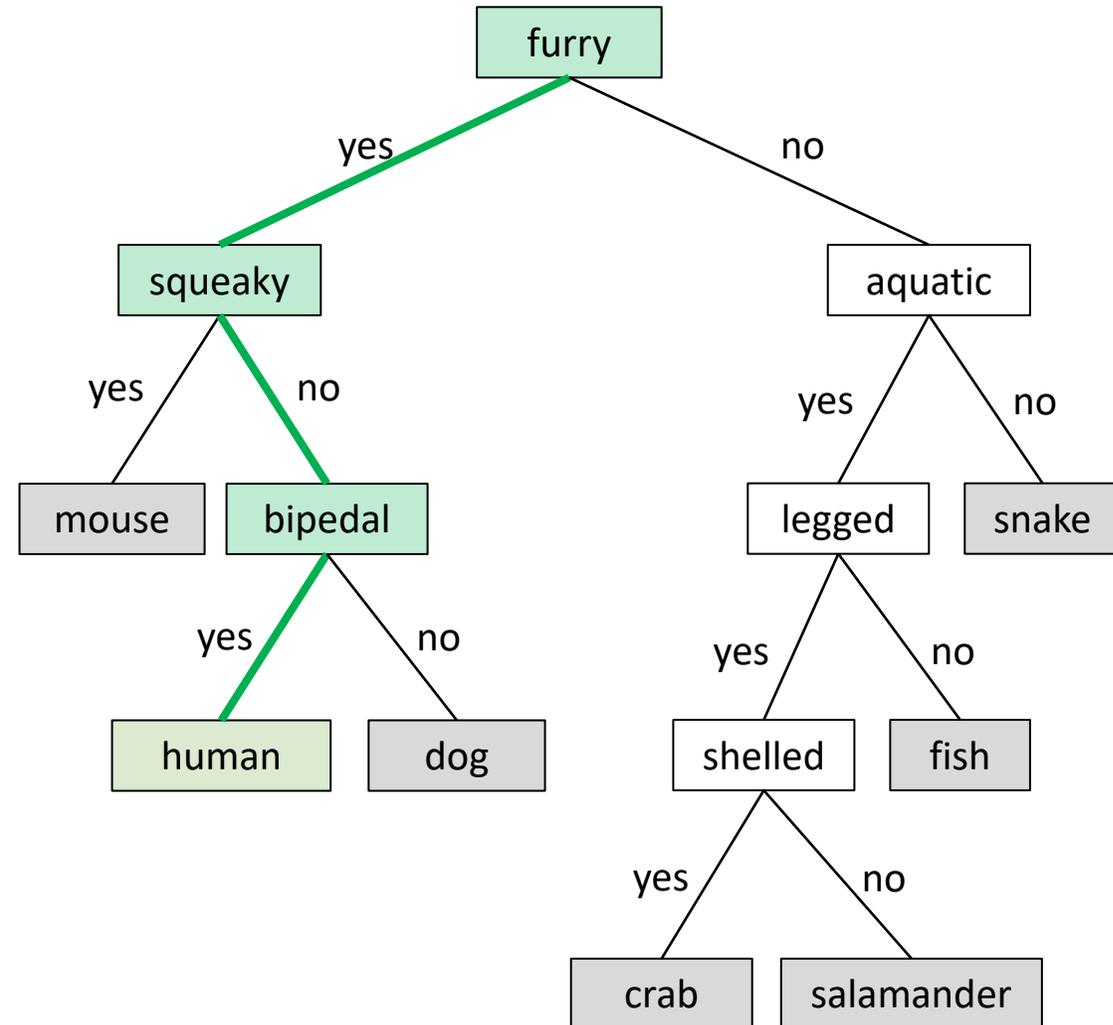
Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive



Program Execution:

1. Yes/No questions to navigate to a leaf (animal).
2. Is animal correct?
- 3.

Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

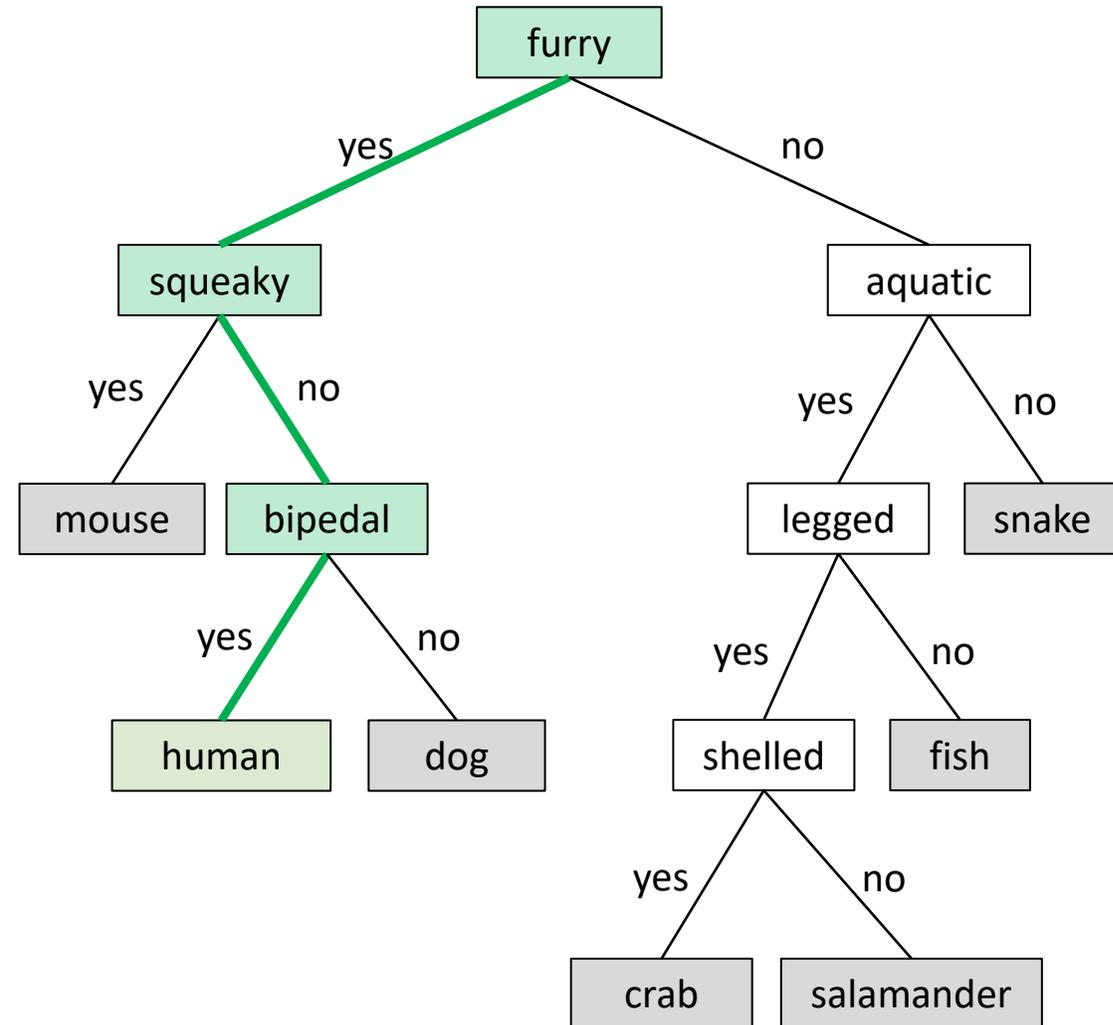
Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive



Program Execution:

1. Yes/No questions to navigate to a leaf (animal).

2. Is animal correct?

3. If not:

3.1. Print location in tree.

3.2.

Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

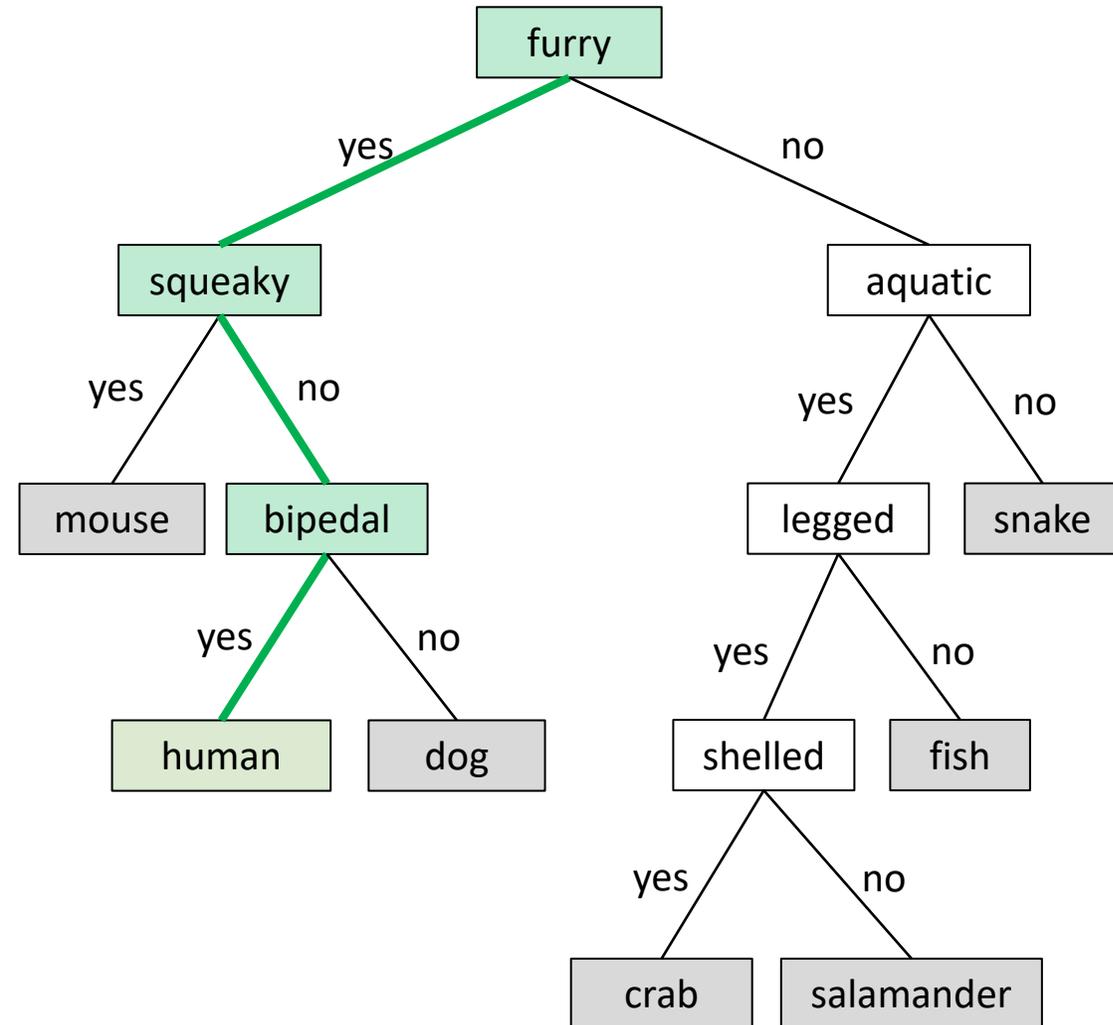
Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive



Program Execution:

1. Yes/No questions to navigate to a leaf (animal).

2. Is animal correct?

3. If not:

3.1. Print location in tree.

3.2. Get name of new animal.

3.3. Get distinguishing characteristic.

3.4.

Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

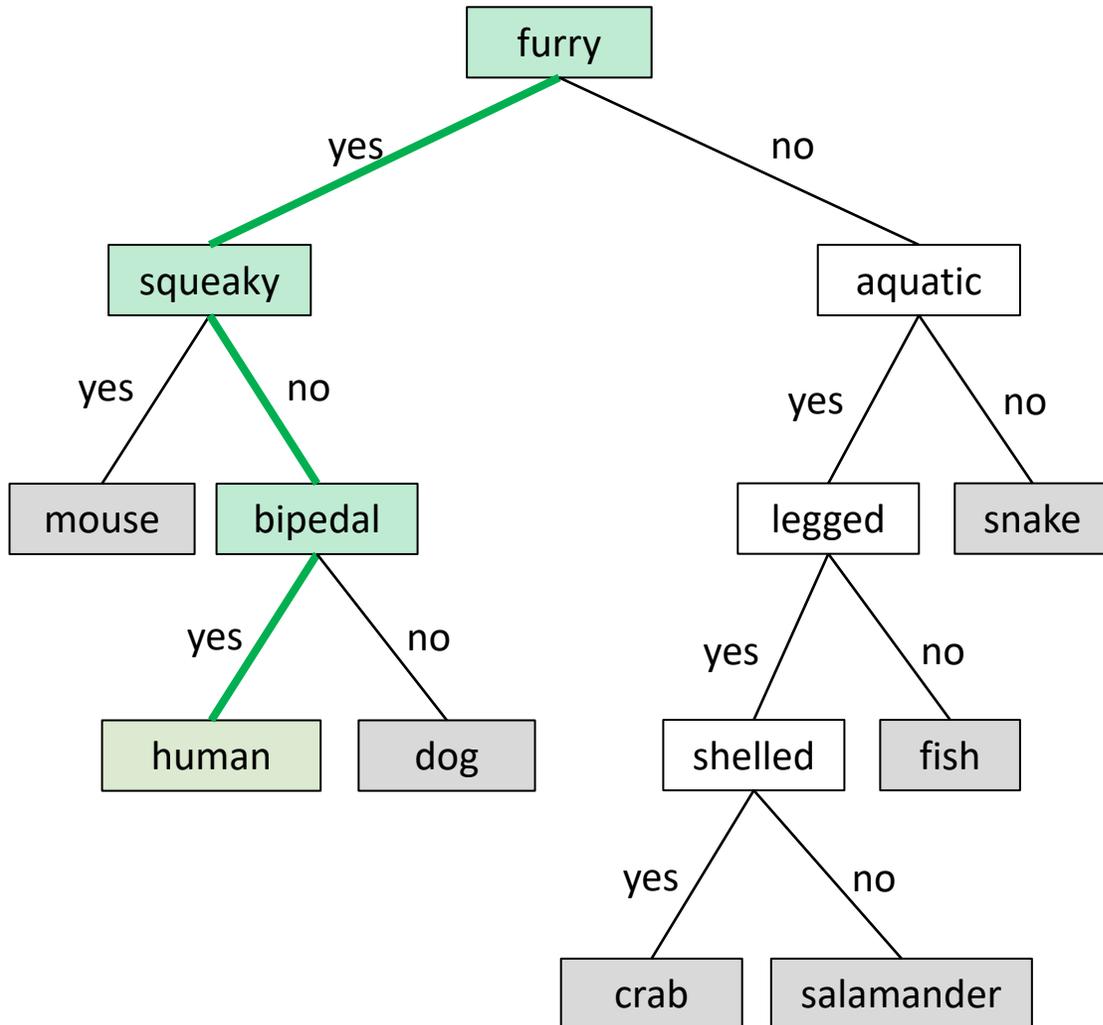
Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive



Program Execution:

1. Yes/No questions to navigate to a leaf (animal).

2. Is animal correct?

3. If not:

3.1. Print location in tree.

3.2. Get name of new animal.

3.3. Get distinguishing characteristic.

3.4. Modify tree:

3.4.1.

Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

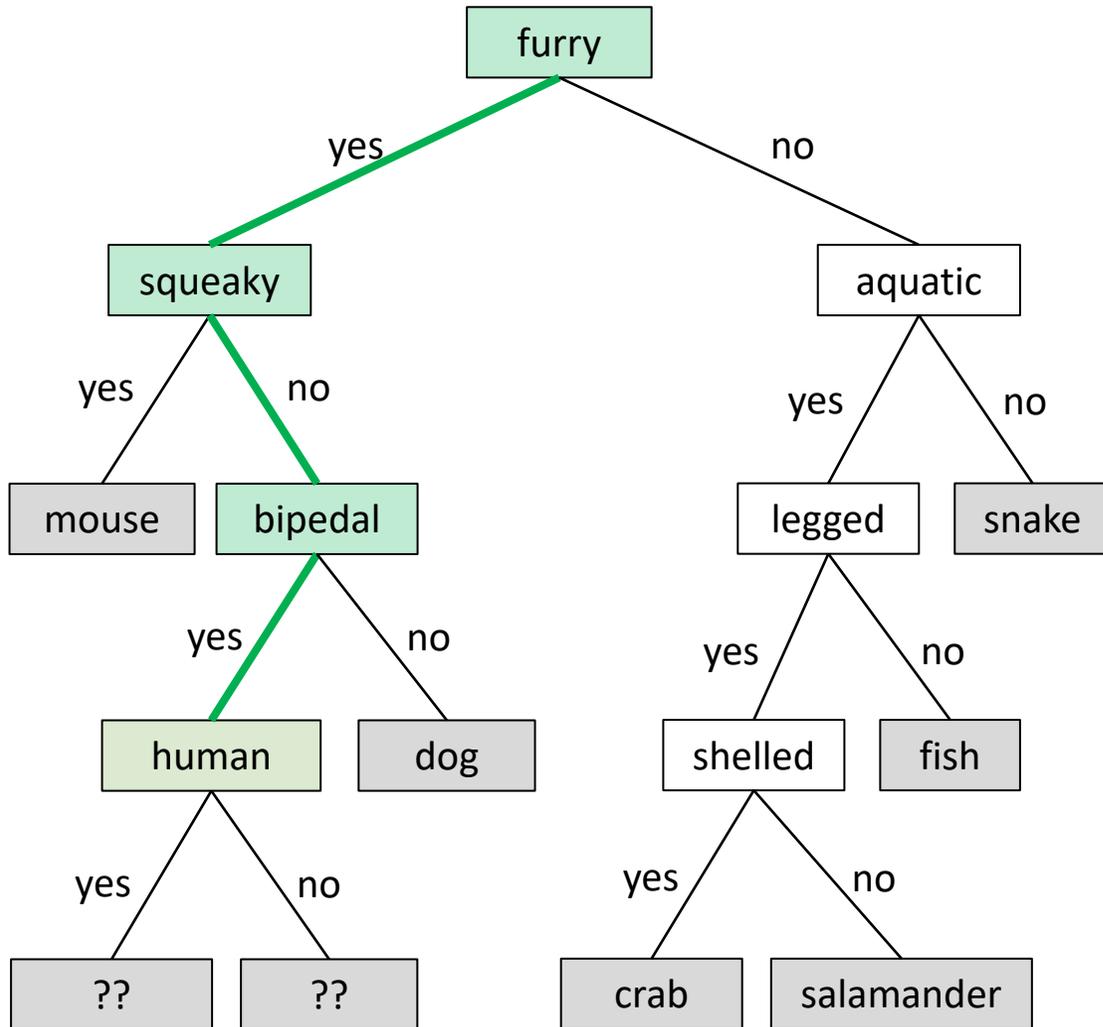
Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive



Program Execution:

1. Yes/No questions to navigate to a leaf (animal).

2. Is animal correct?

3. If not:

3.1. Print location in tree.

3.2. Get name of new animal.

3.3. Get distinguishing characteristic.

3.4. Modify tree:

3.4.1. Create two new child nodes at current leaf.

3.4.2.

Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

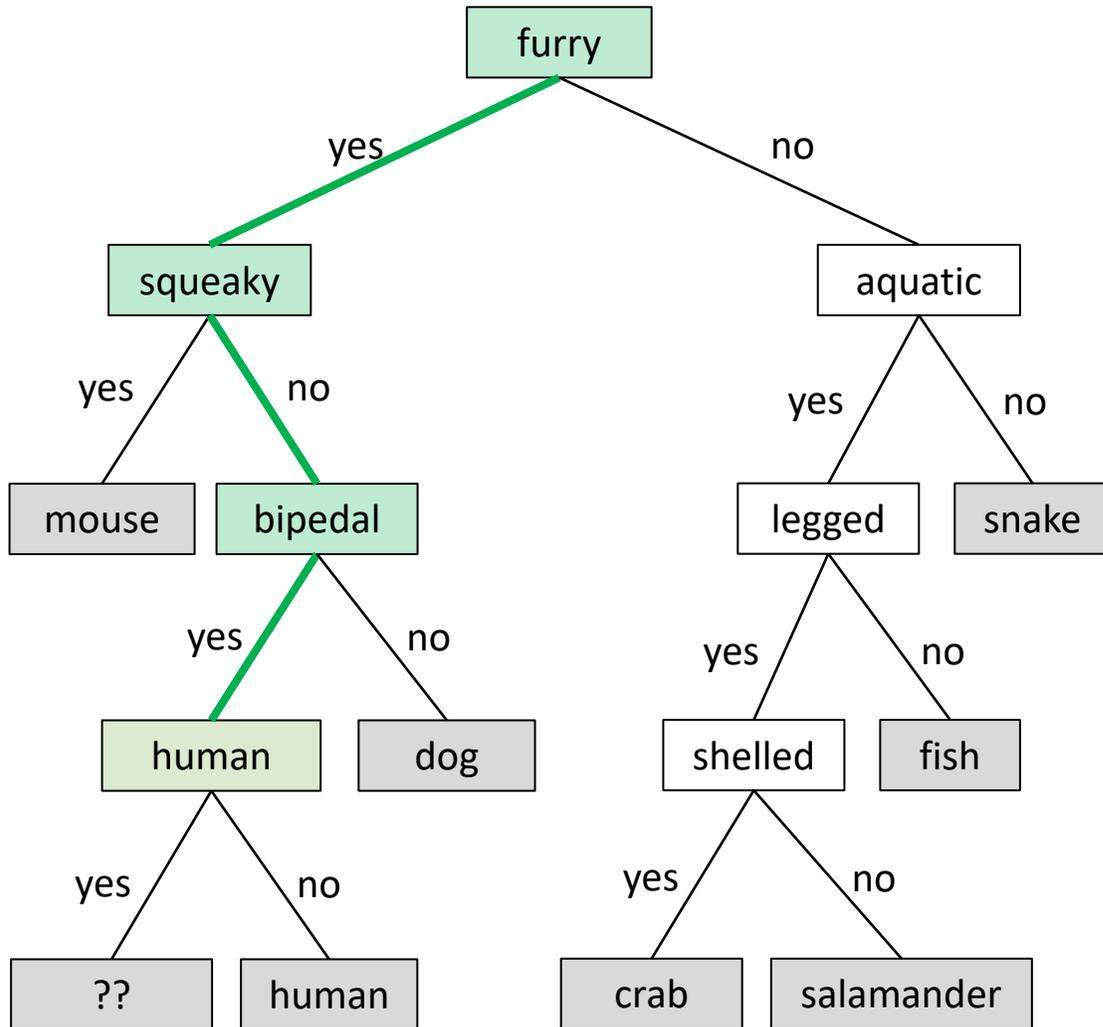
Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive



Program Execution:

1. Yes/No questions to navigate to a leaf (animal).

2. Is animal correct?

3. If not:

3.1. Print location in tree.

3.2. Get name of new animal.

3.3. Get distinguishing characteristic.

3.4. Modify tree:

3.4.1. Create two new child nodes at current leaf.

3.4.2. Make "no" child node animal be old leaf.

3.4.3.

Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

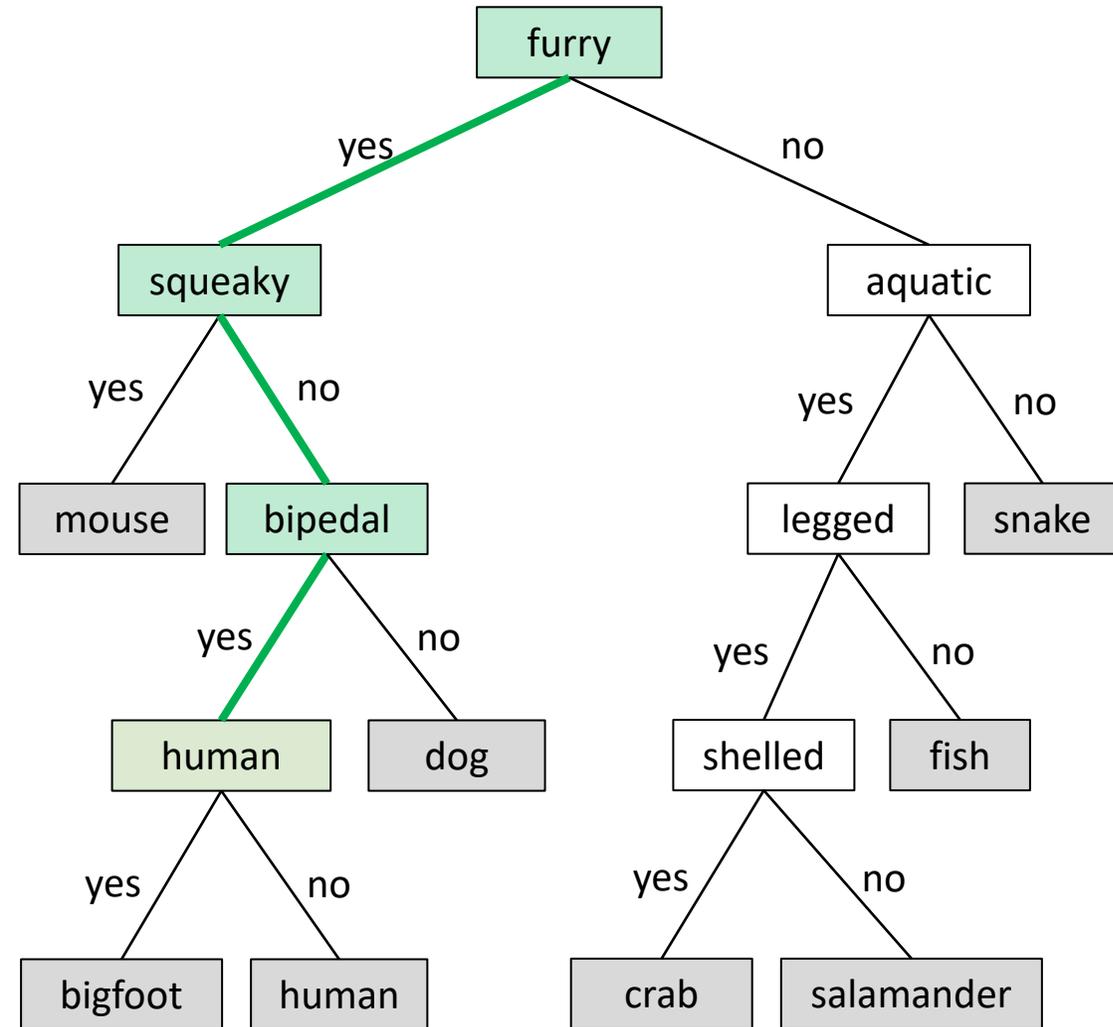
Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive



Program Execution:

1. Yes/No questions to navigate to a leaf (animal).

2. Is animal correct?

3. If not:

3.1. Print location in tree.

3.2. Get name of new animal.

3.3. Get distinguishing characteristic.

3.4. Modify tree:

3.4.1. Create two new child nodes at current leaf.

3.4.2. Make "no" child node animal be old leaf.

3.4.3. Make "yes" child node animal be new animal.

3.4.4.

Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

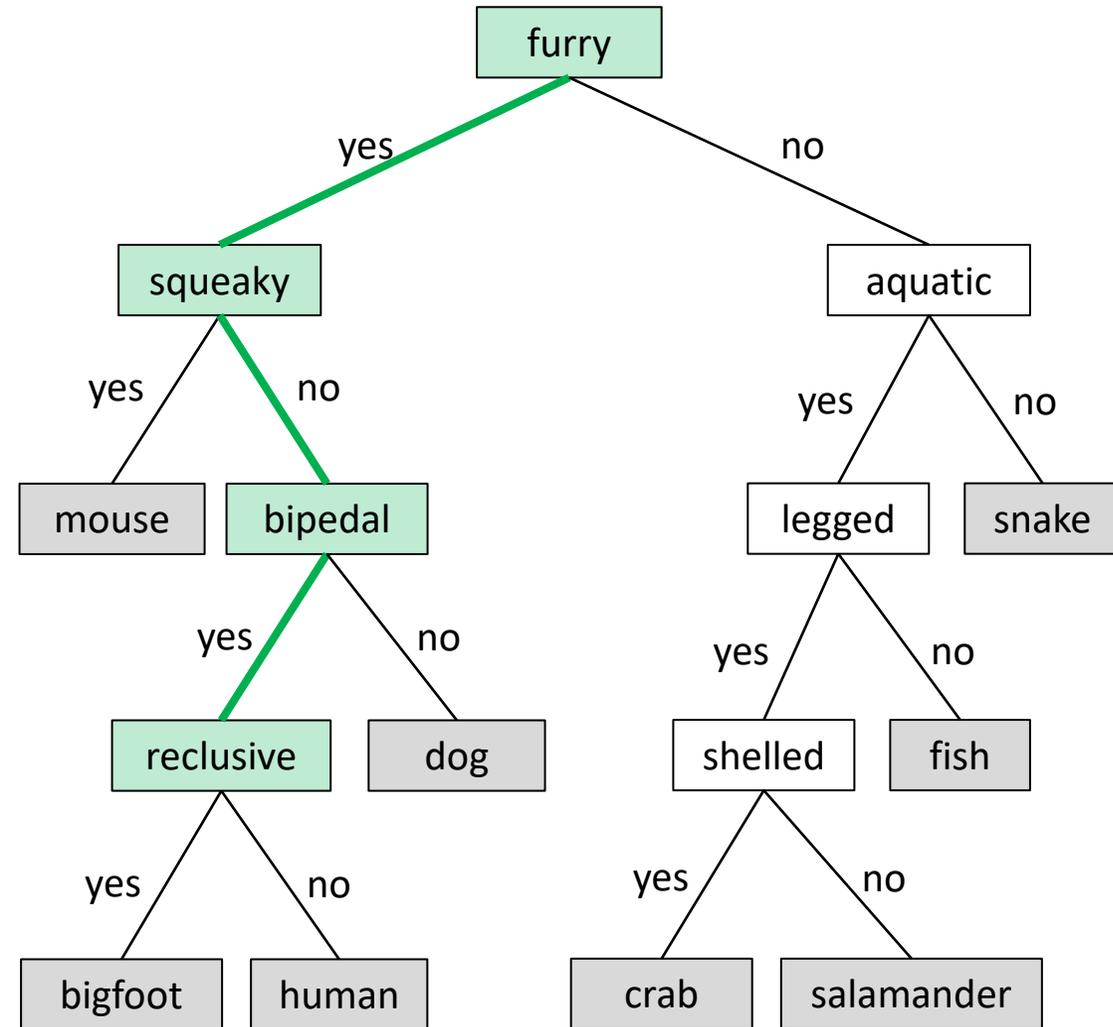
Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive



Program Execution:

1. Yes/No questions to navigate to a leaf (animal).

2. Is animal correct?

3. If not:

3.1. Print location in tree.

3.2. Get name of new animal.

3.3. Get distinguishing characteristic.

3.4. Modify tree:

3.4.1. Create two new child nodes at current leaf.

3.4.2. Make "no" child node animal be old leaf.

3.4.3. Make "yes" child node animal be new animal.

3.4.4. Make old leaf be distinguishing characteristic.

Do you have another animal to identify? (Y/N) > Y

Is this animal furry? (Y/N) > Y

Is this animal squeaky? (Y/N) > N

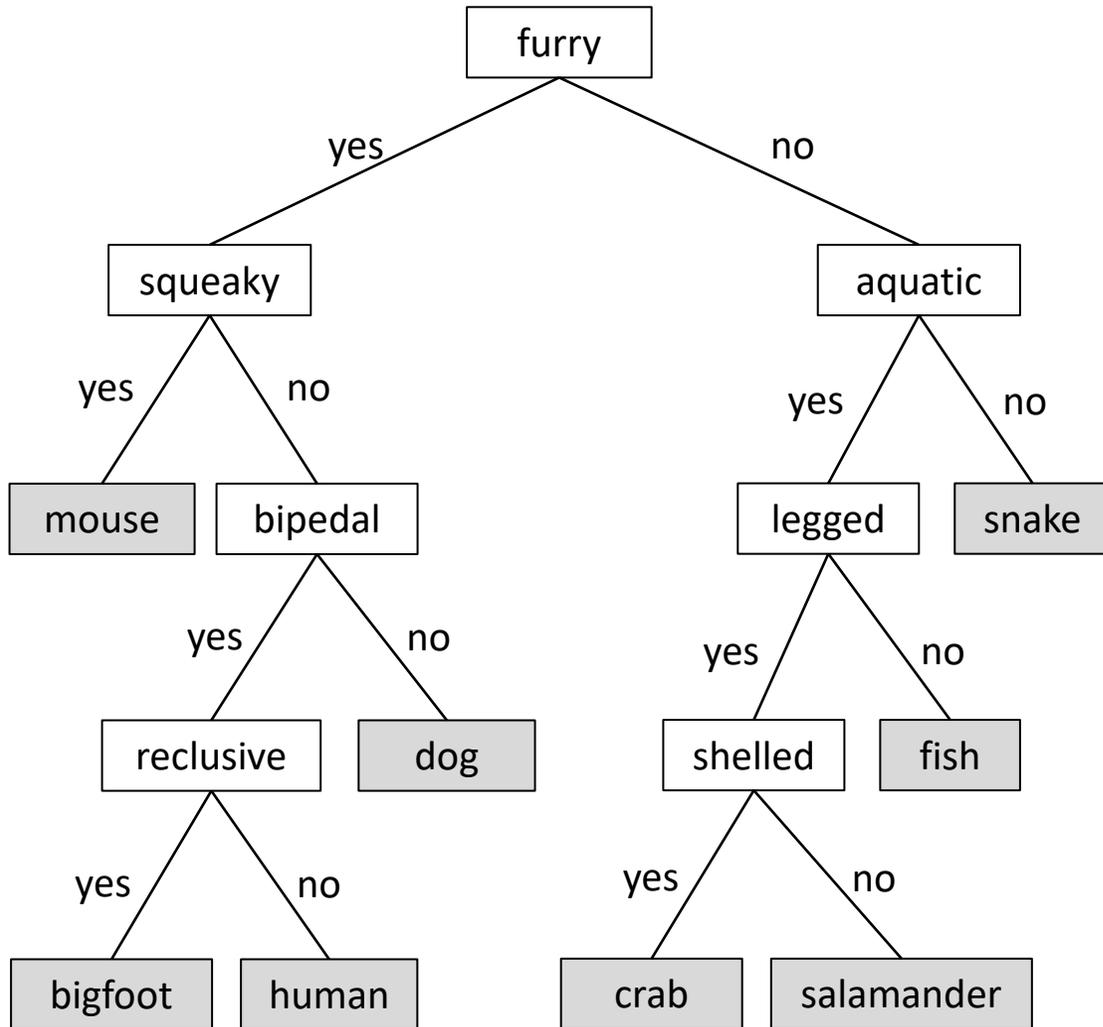
Is this animal bipedal? (Y/N) > Y

Is this animal a human? (Y/N) > N

I don't know any furry, not squeaky, bipedal animals that aren't a human.

What is the new animal? > bigfoot

What characteristic does a bigfoot have that a human does not? > reclusive



Program Execution:

1. Yes/No questions to navigate to a leaf (animal).

2. Is animal correct?

3. If not:

3.1. Print location in tree.

3.2. Get name of new animal.

3.3. Get distinguishing characteristic.

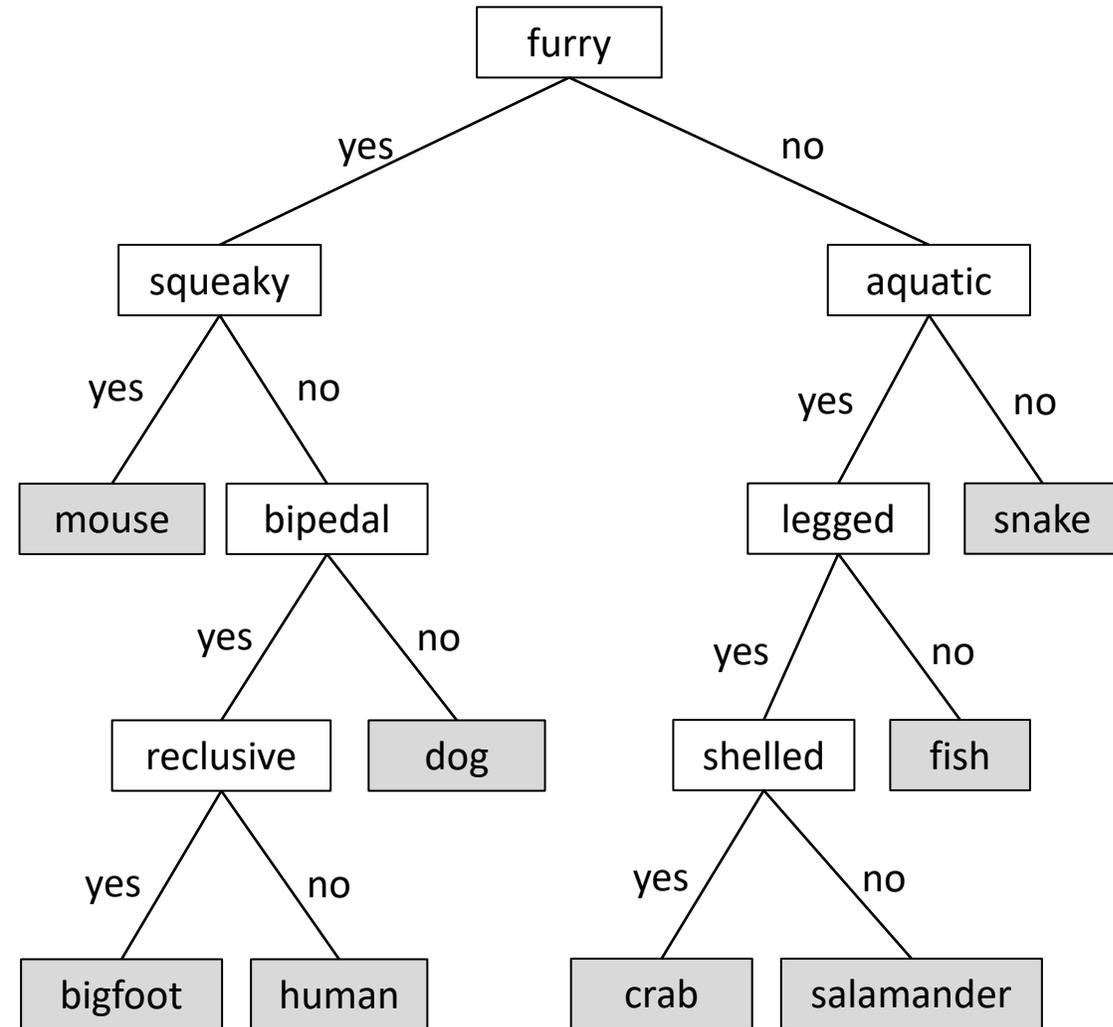
3.4. Modify tree:

3.4.1. Create two new child nodes at current leaf.

3.4.2. Make "no" child node animal be old leaf.

3.4.3. Make "yes" child node animal be new animal.

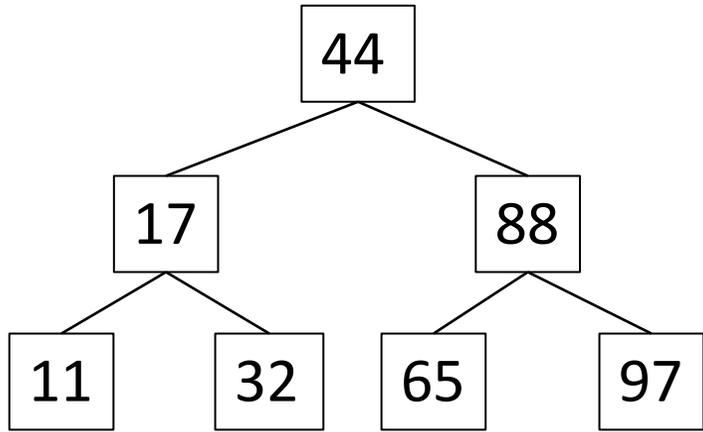
3.4.4. Make old leaf be distinguishing characteristic.



```
public class Node {  
    private String text;  
    private Node yesChild;  
    private Node noChild;  
    private Node parent;  
  
    ...  
}
```

File read/writing

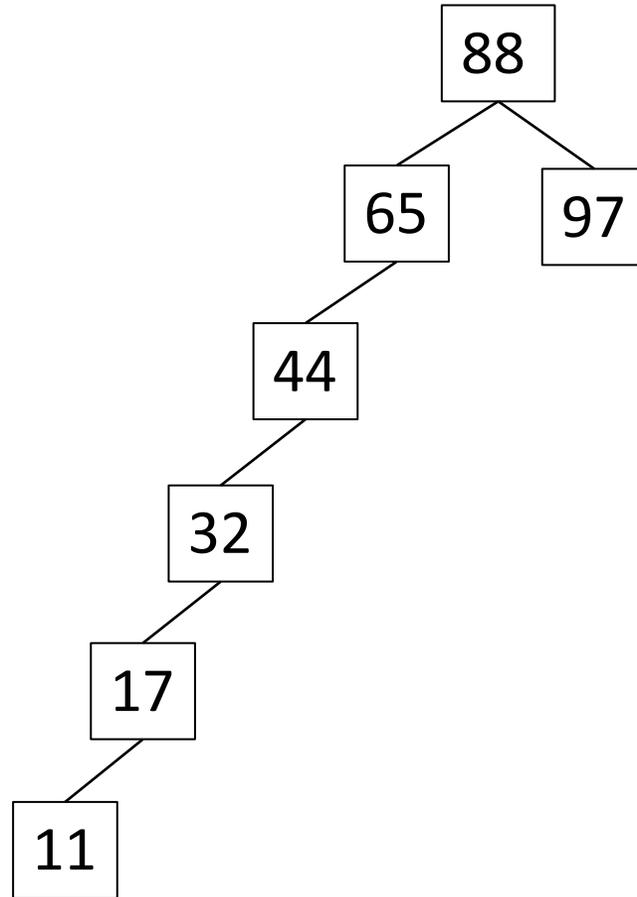
Order Matters



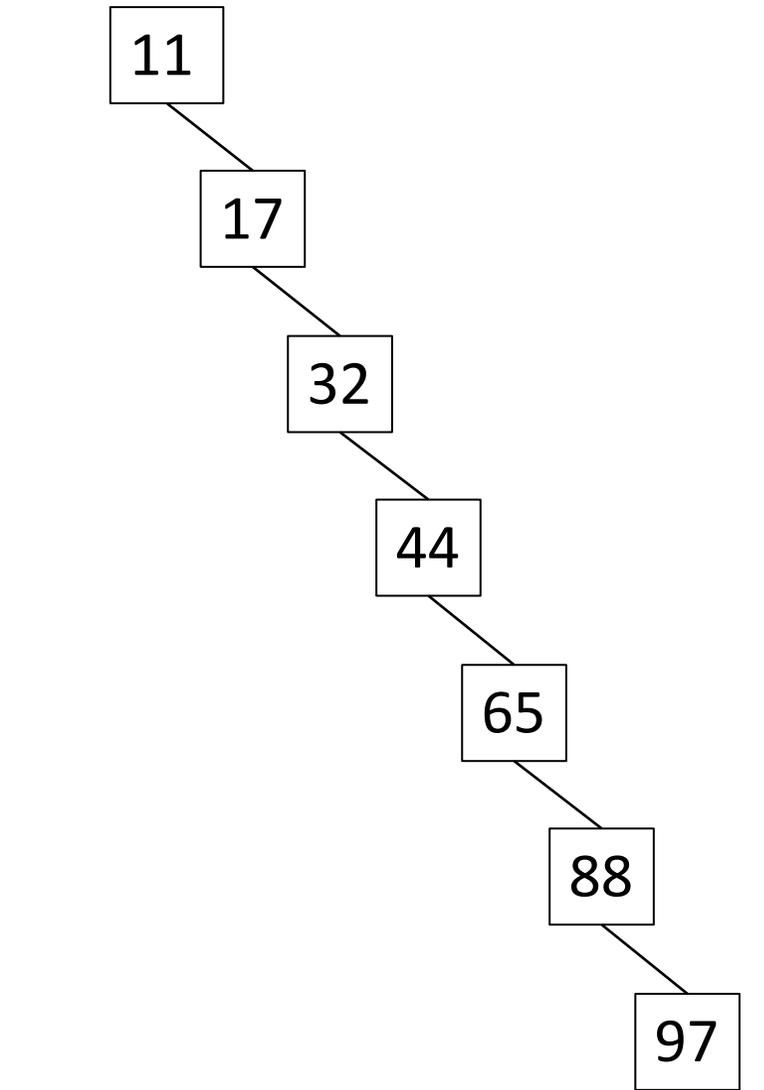
44, 17, 88, 11, 32, 65, 97

44, 17, 32, 88, 11, 97, 65

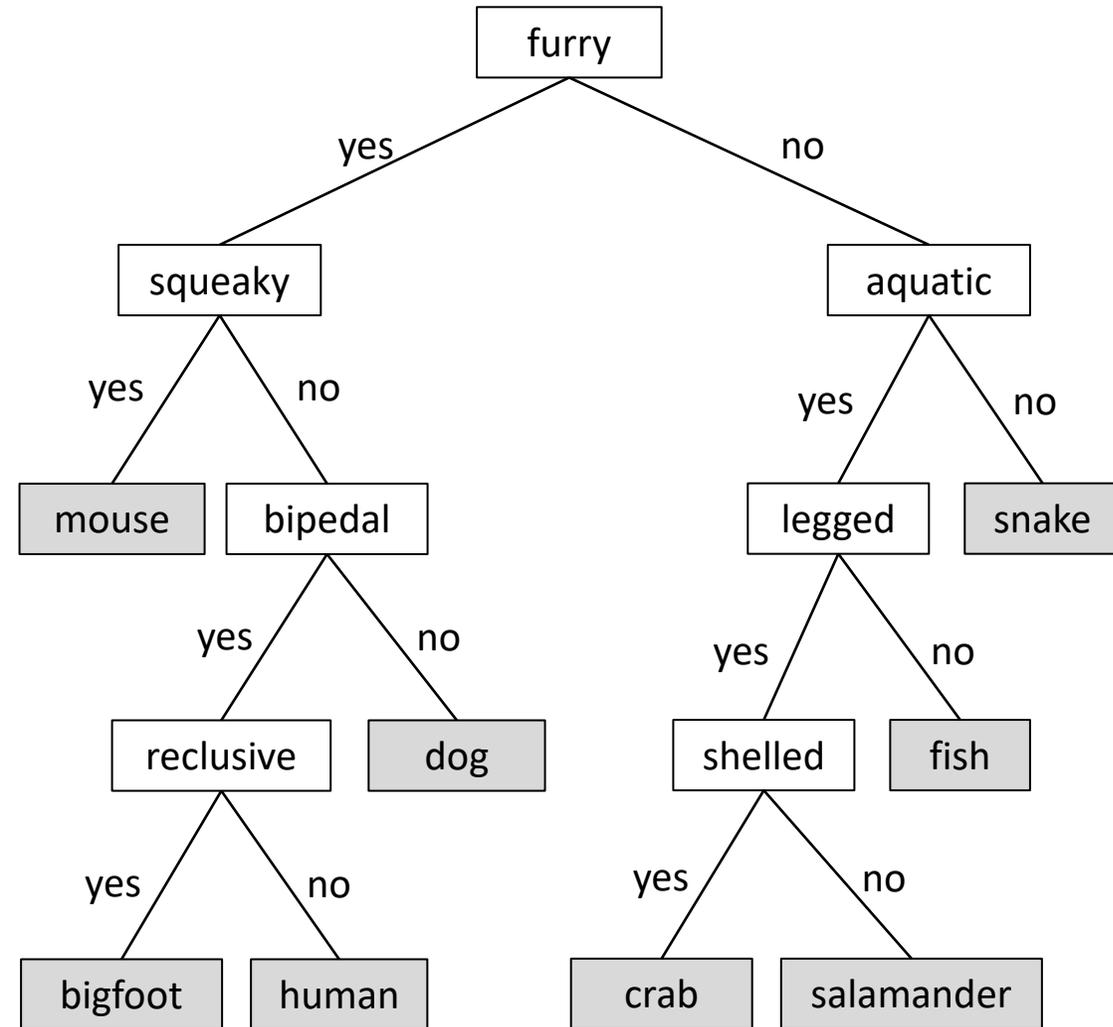
44, 88, 65, 97, 17, 32, 11



88, 65, 44, 32, 97, 17, 11

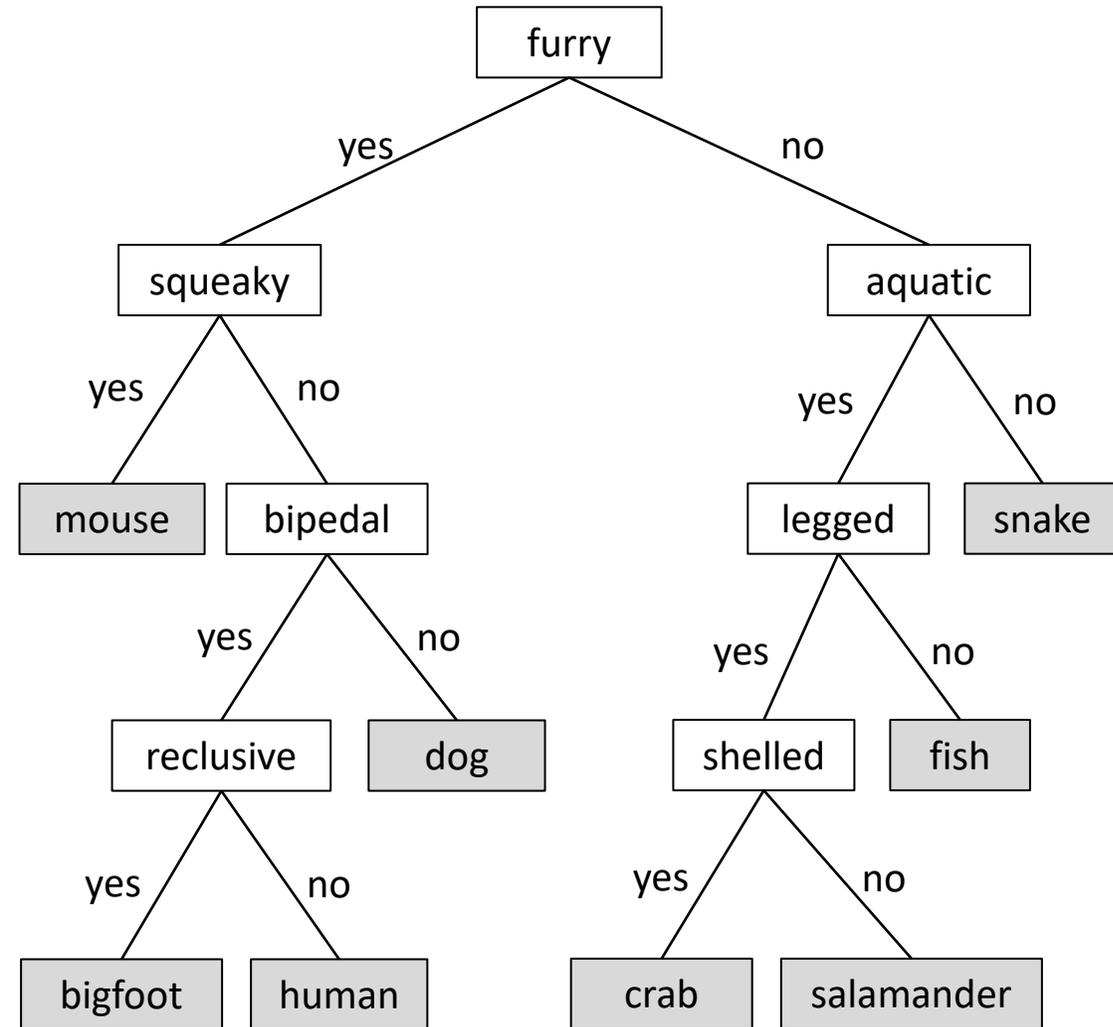


11, 17, 32, 44, 65, 88, 97



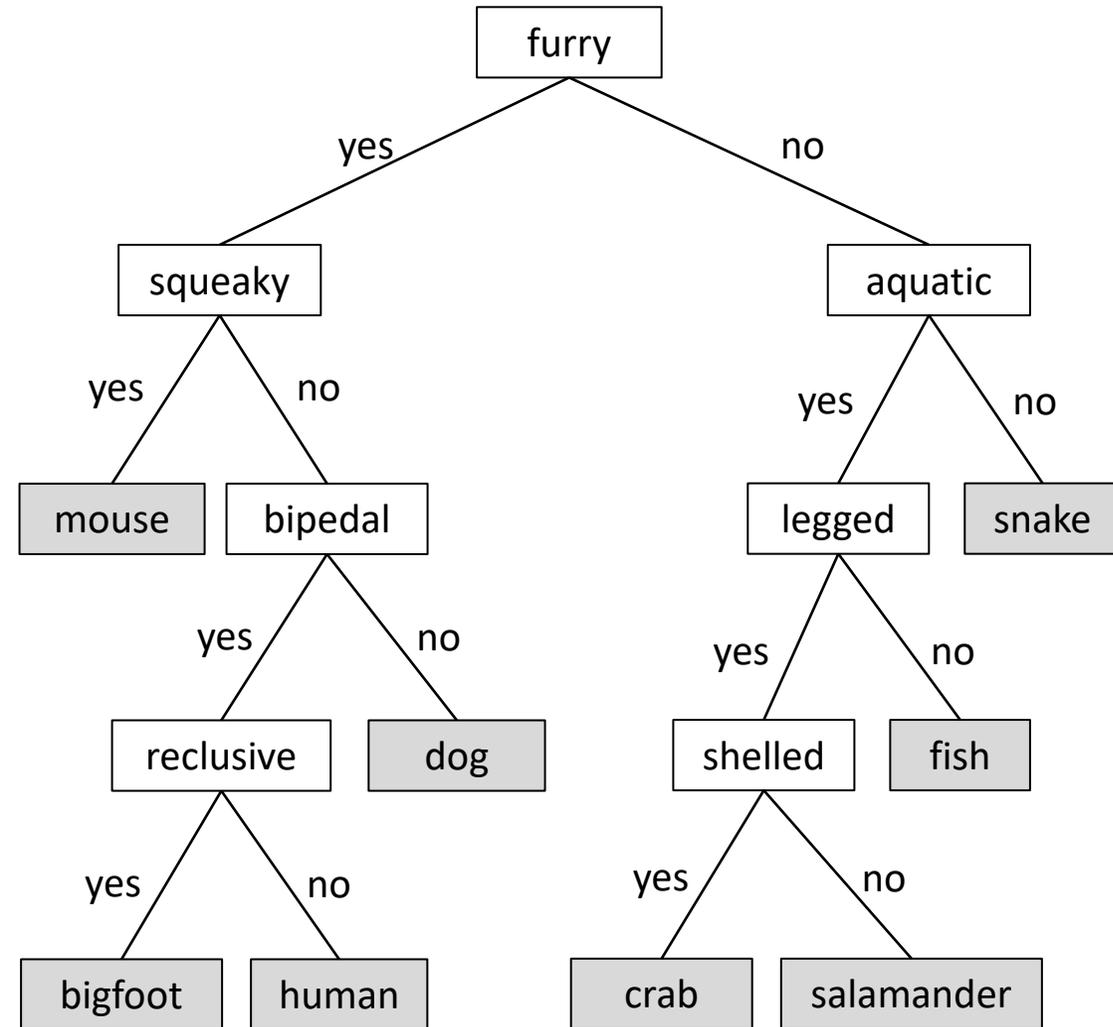
```
public class Node {  
    private String text;  
    private Node yesChild;  
    private Node noChild;  
    private Node parent;  
  
    ...  
}
```

File read/writing



```
public class Node {  
    private String text;  
    private Node yesChild;  
    private Node noChild;  
    private Node parent;  
    private int tag;  
    ...  
}
```

File read/writing

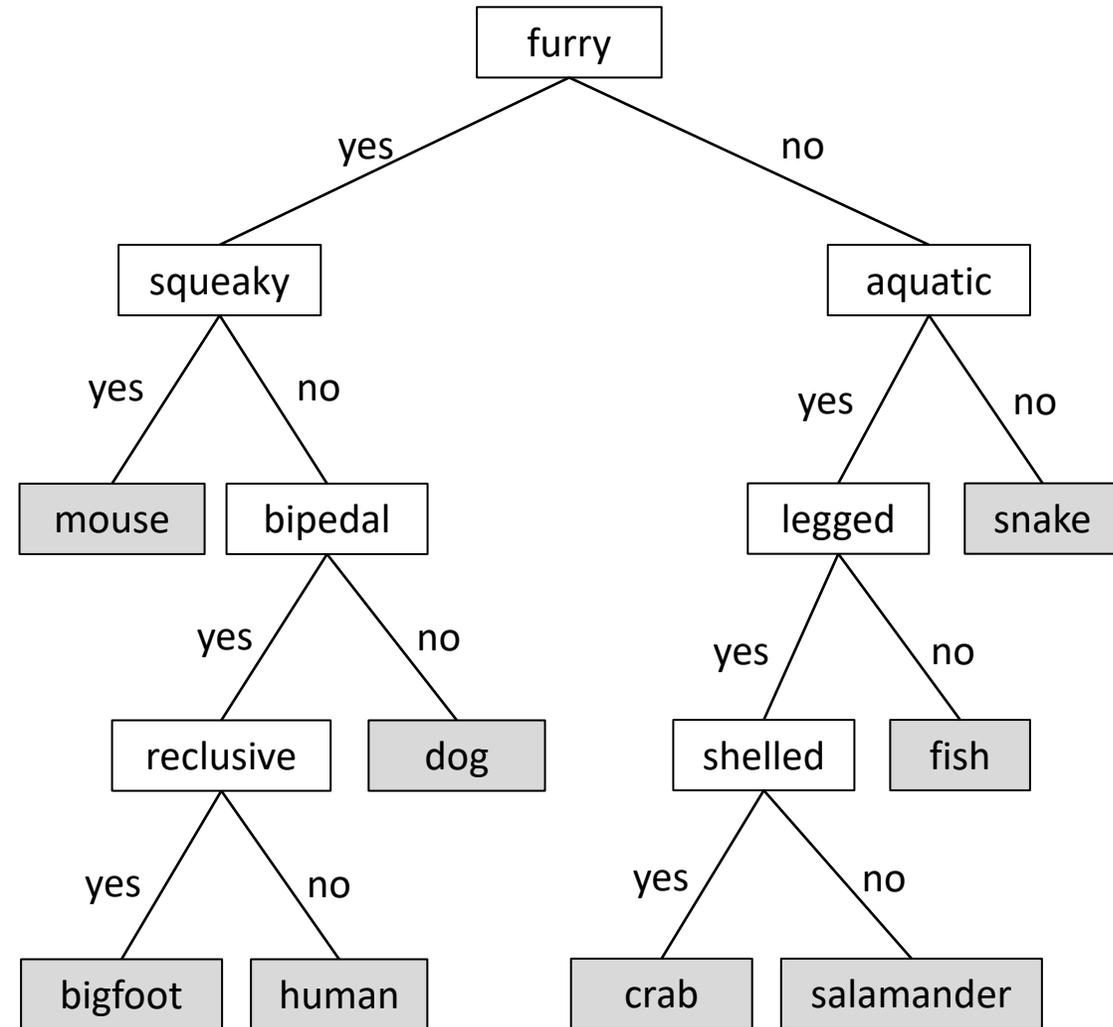


```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

File read/writing

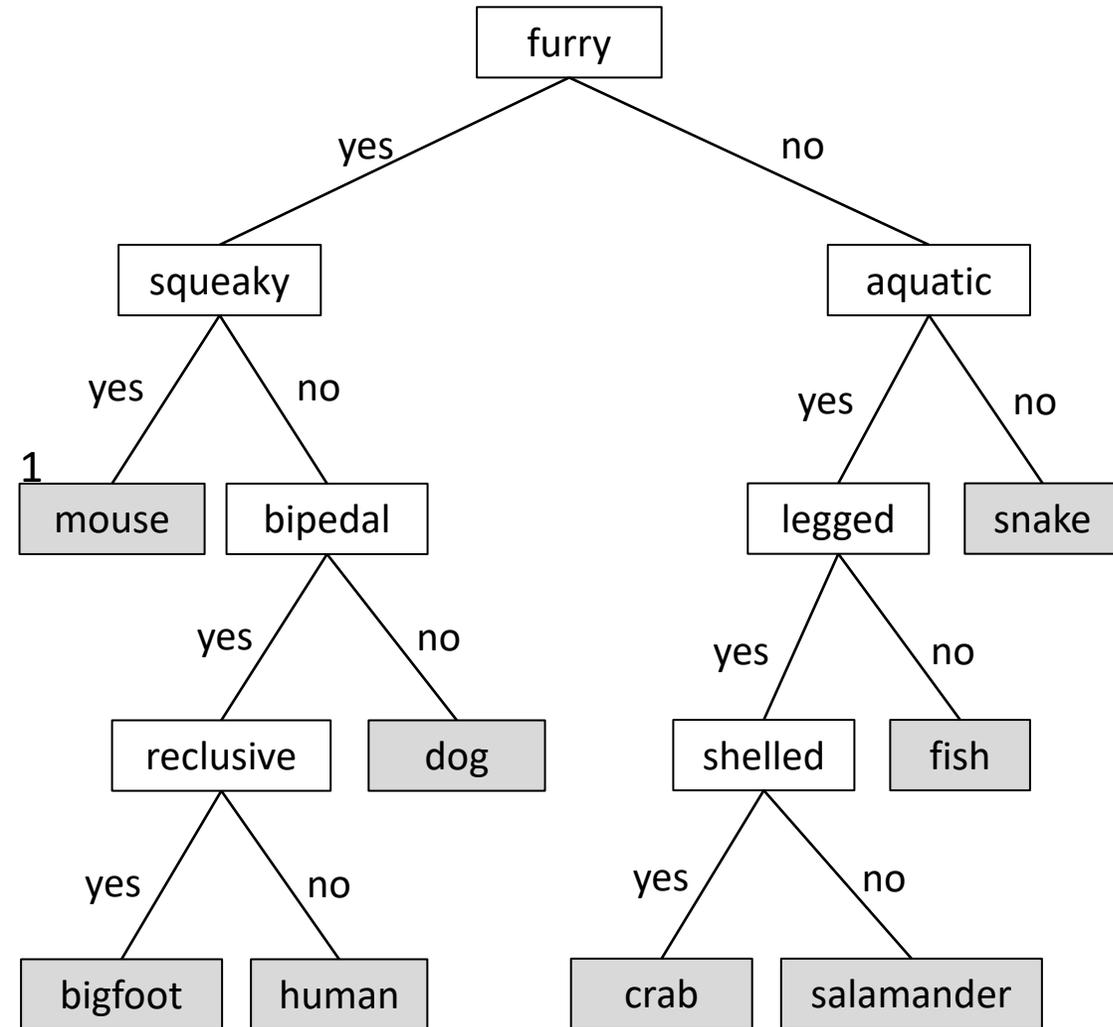


```
public class Node {  
    private String text;  
    private Node yesChild;  
    private Node noChild;  
    private Node parent;  
    private int tag;  
    ...  
}
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



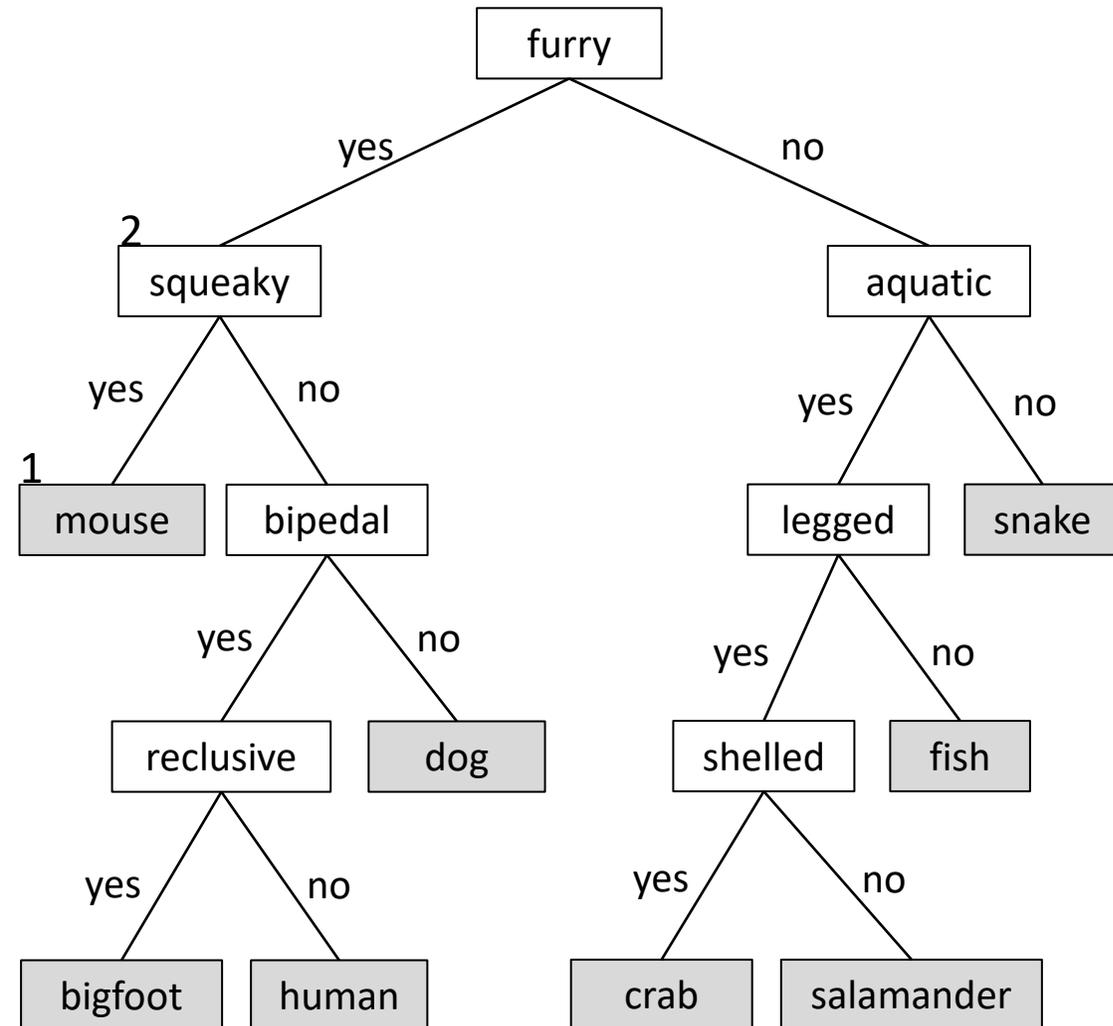
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



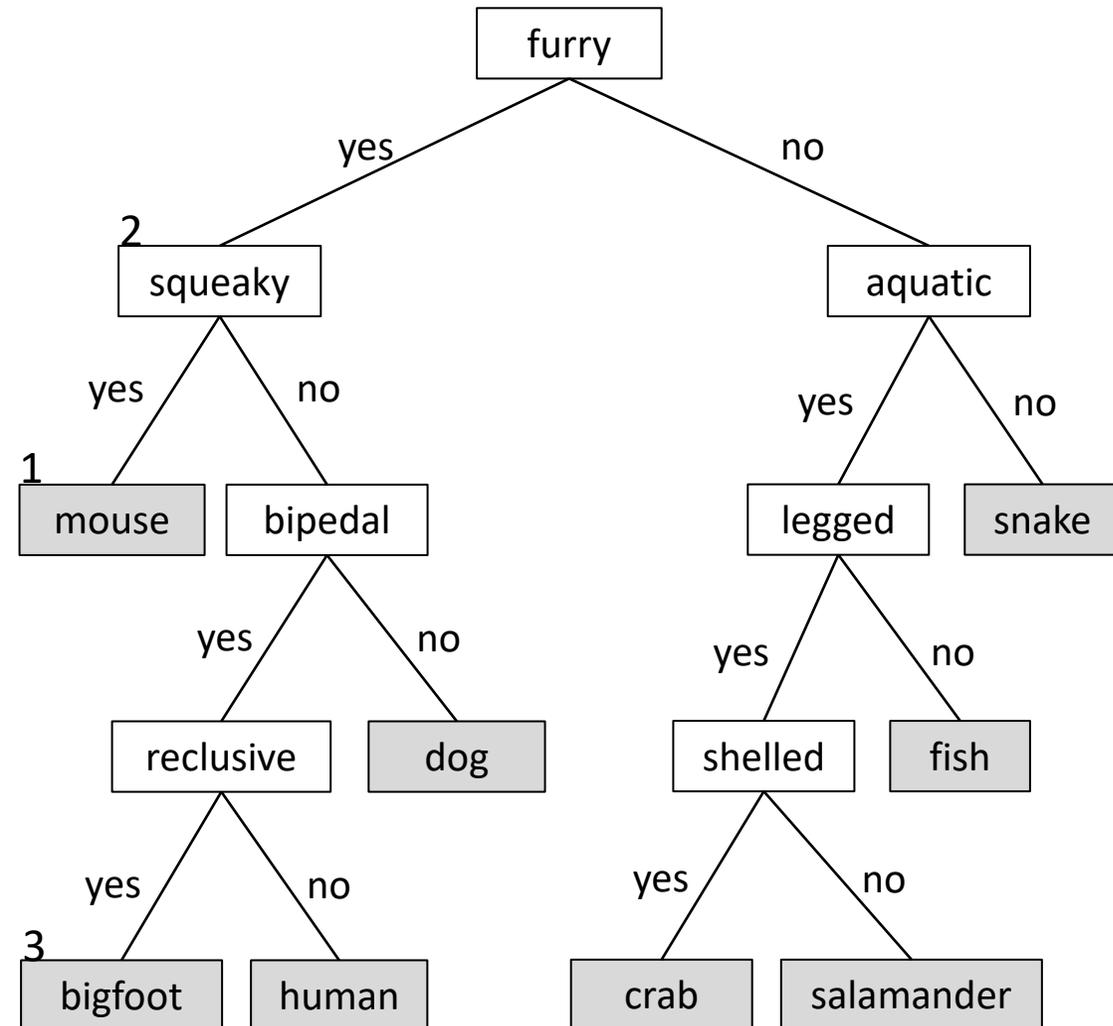
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



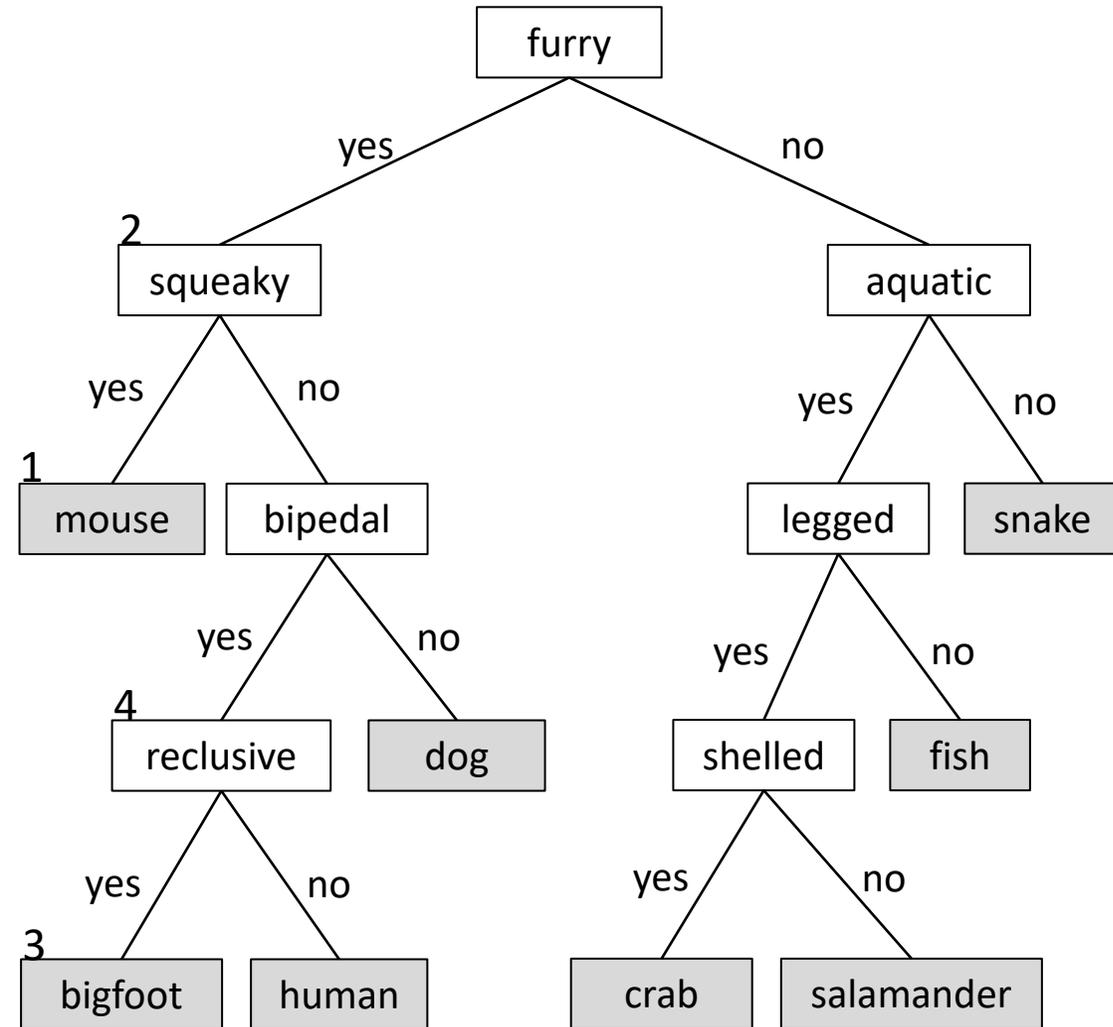
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



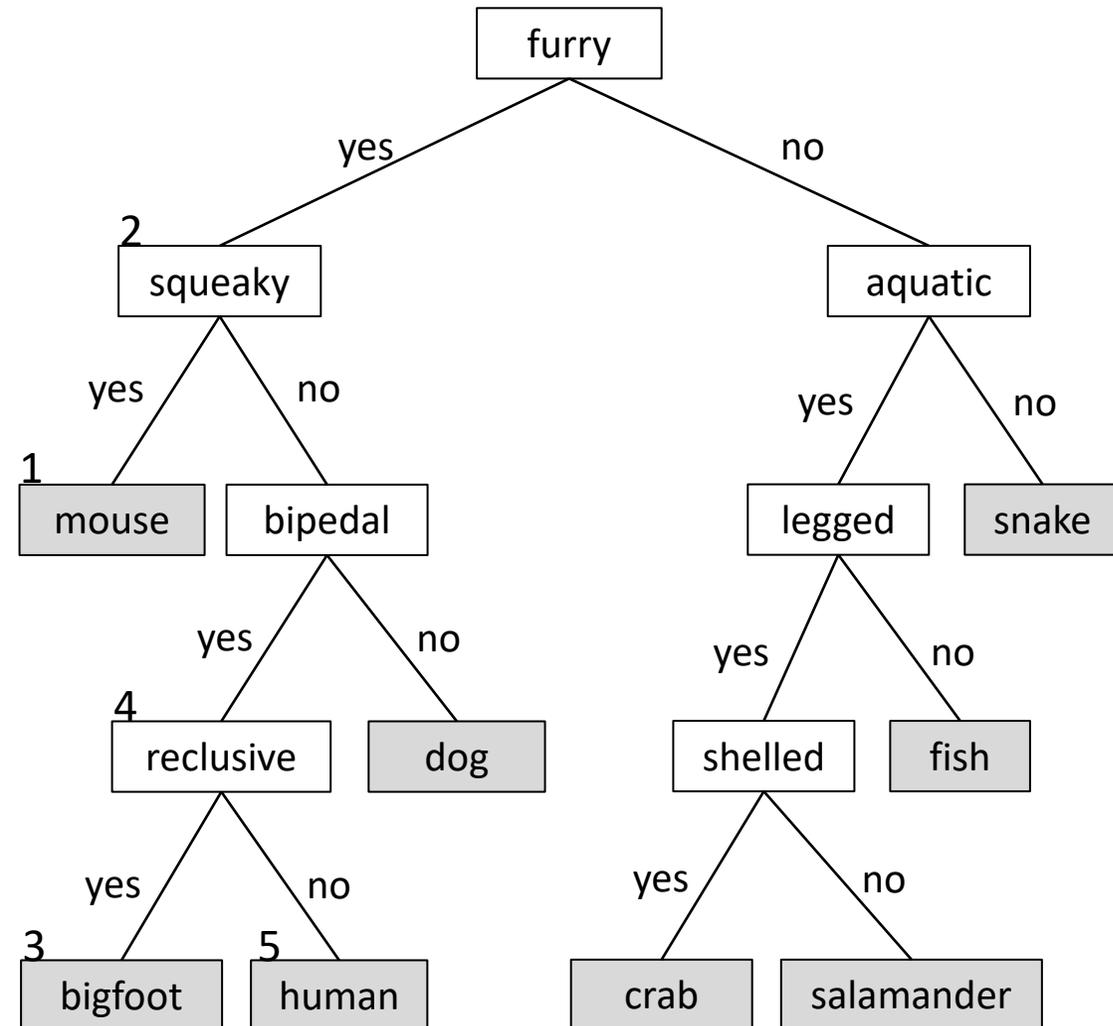
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



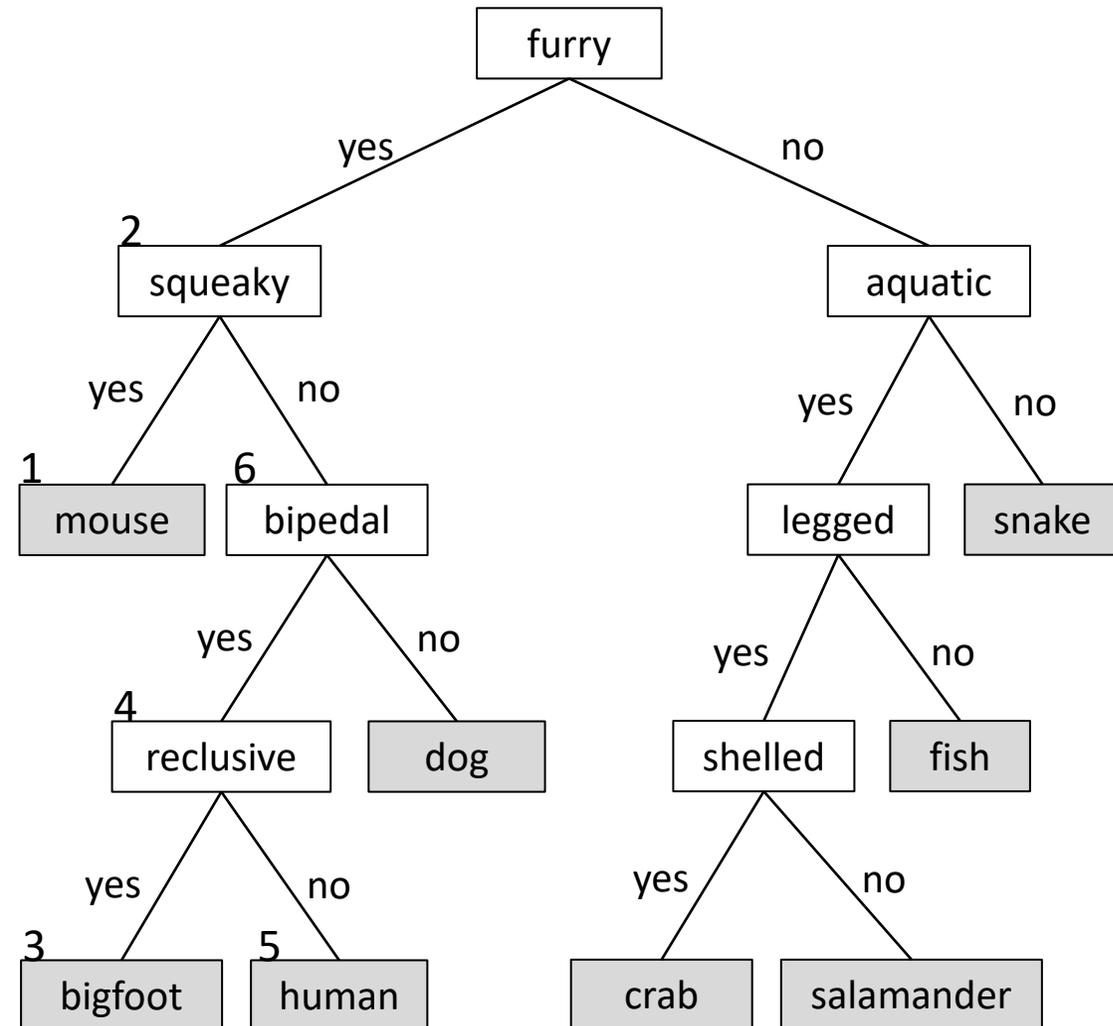
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



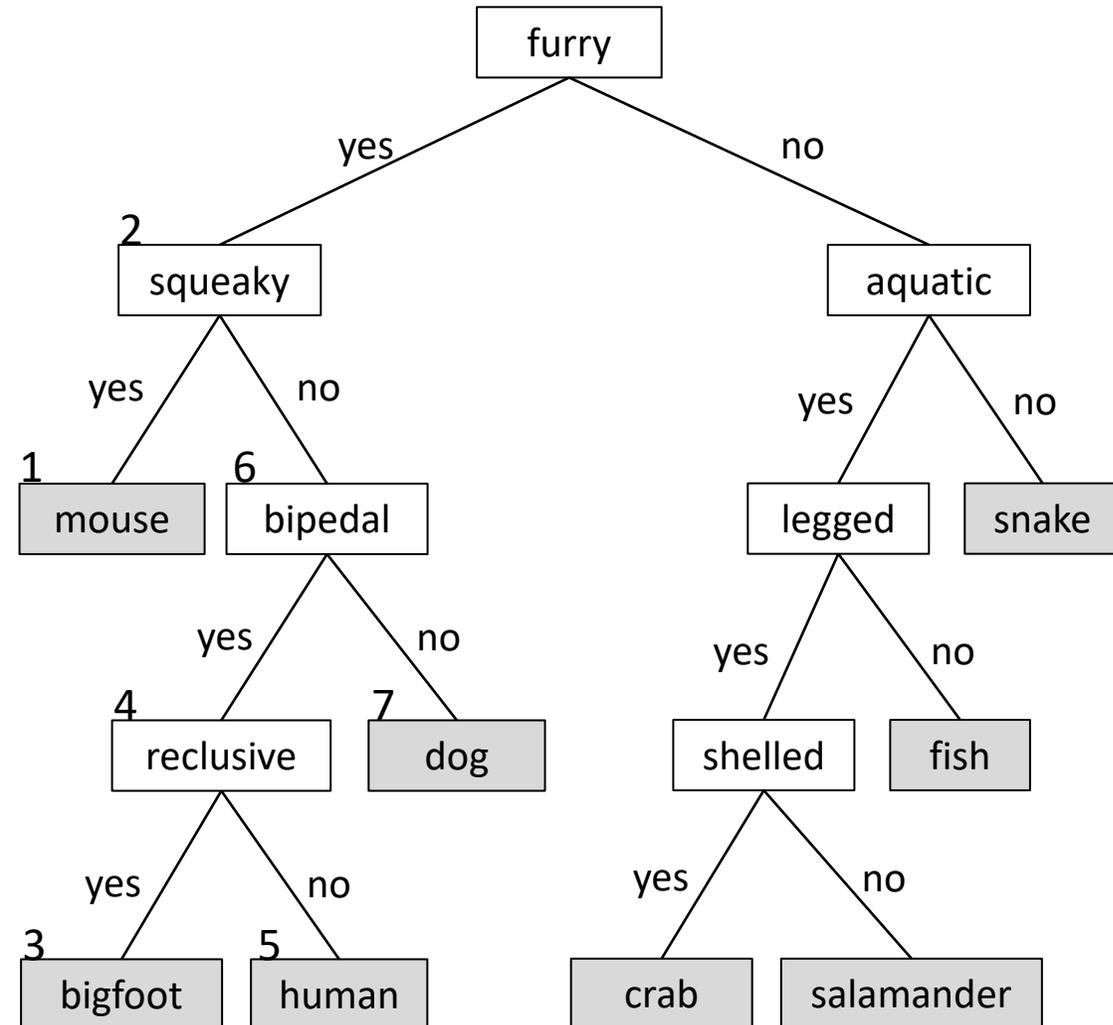
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



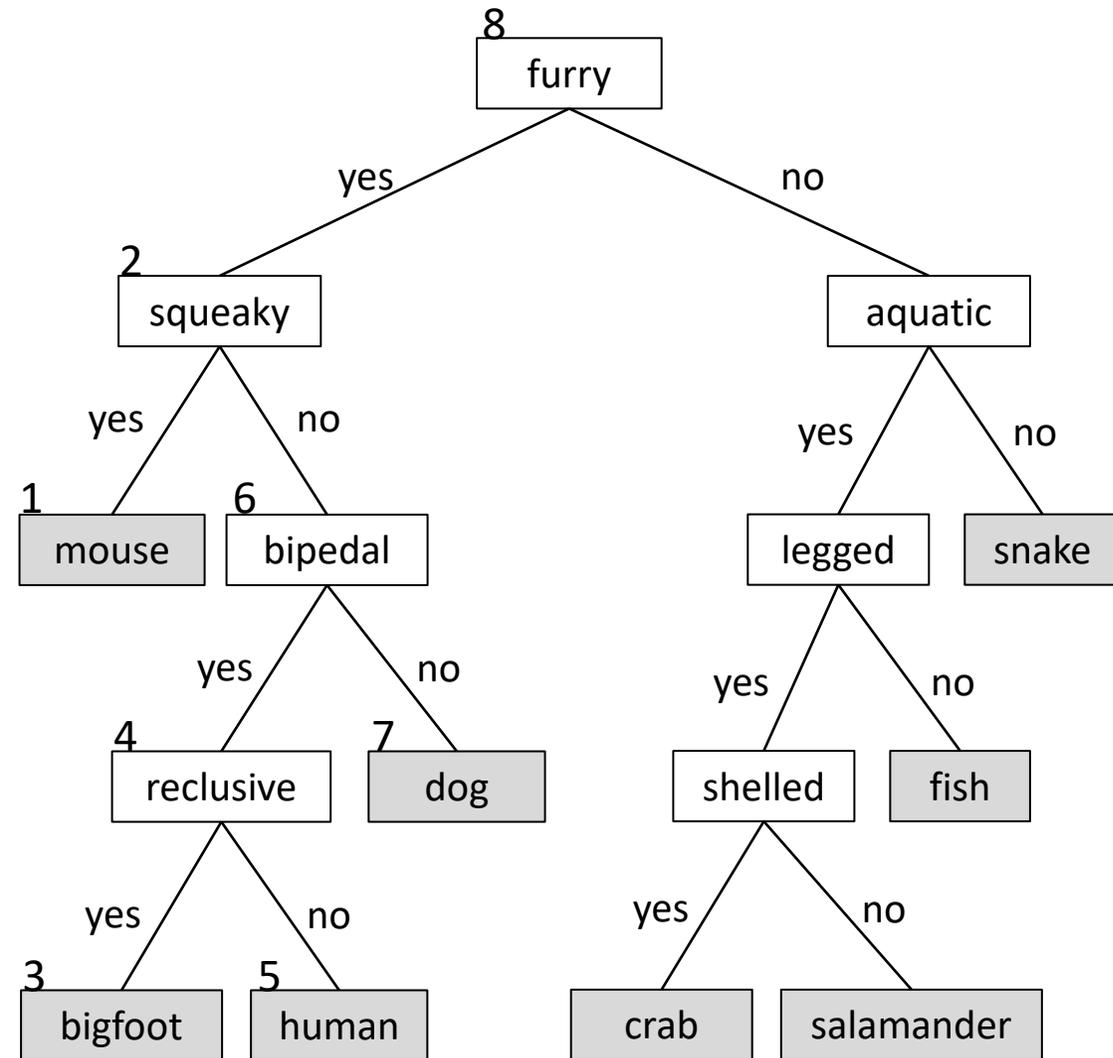
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



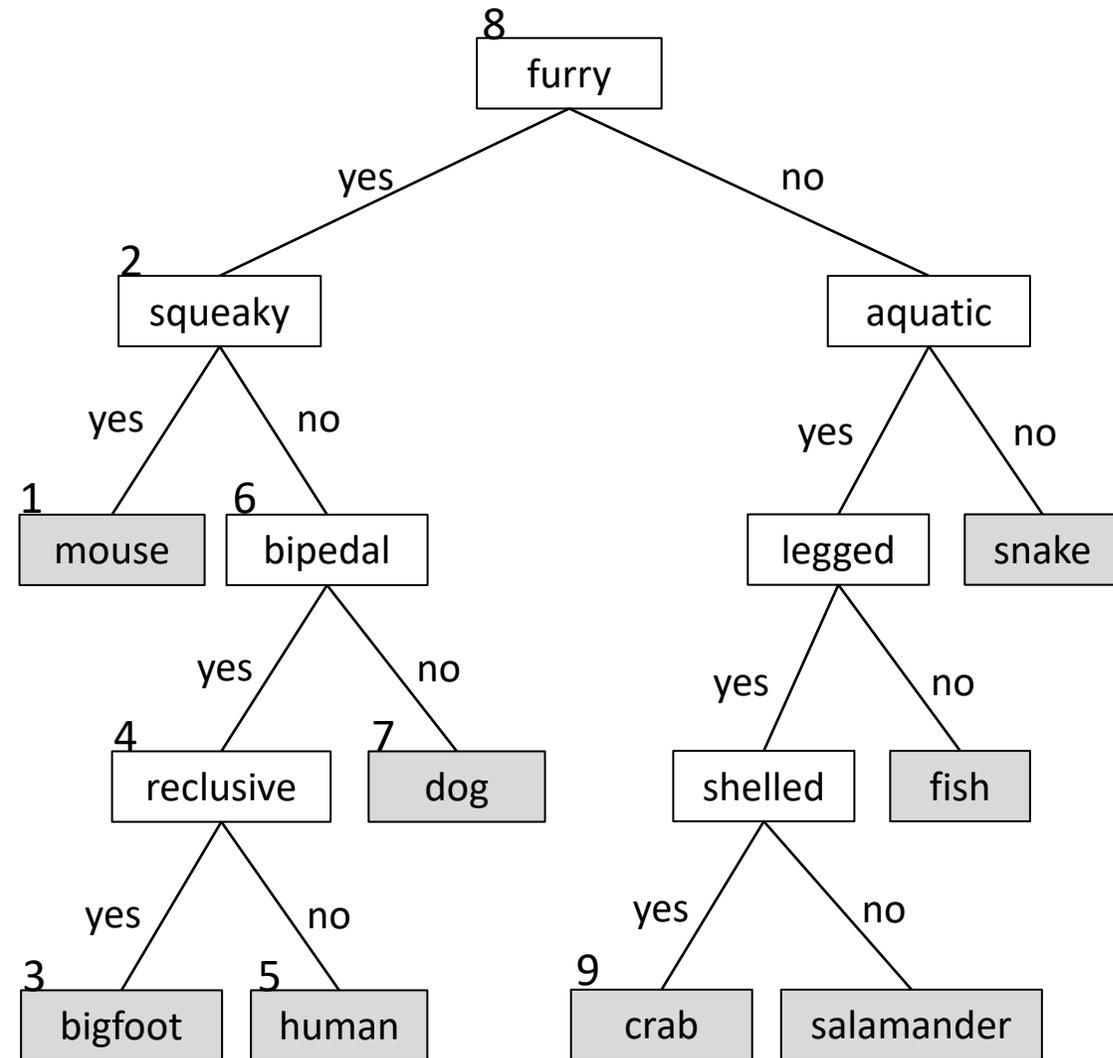
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



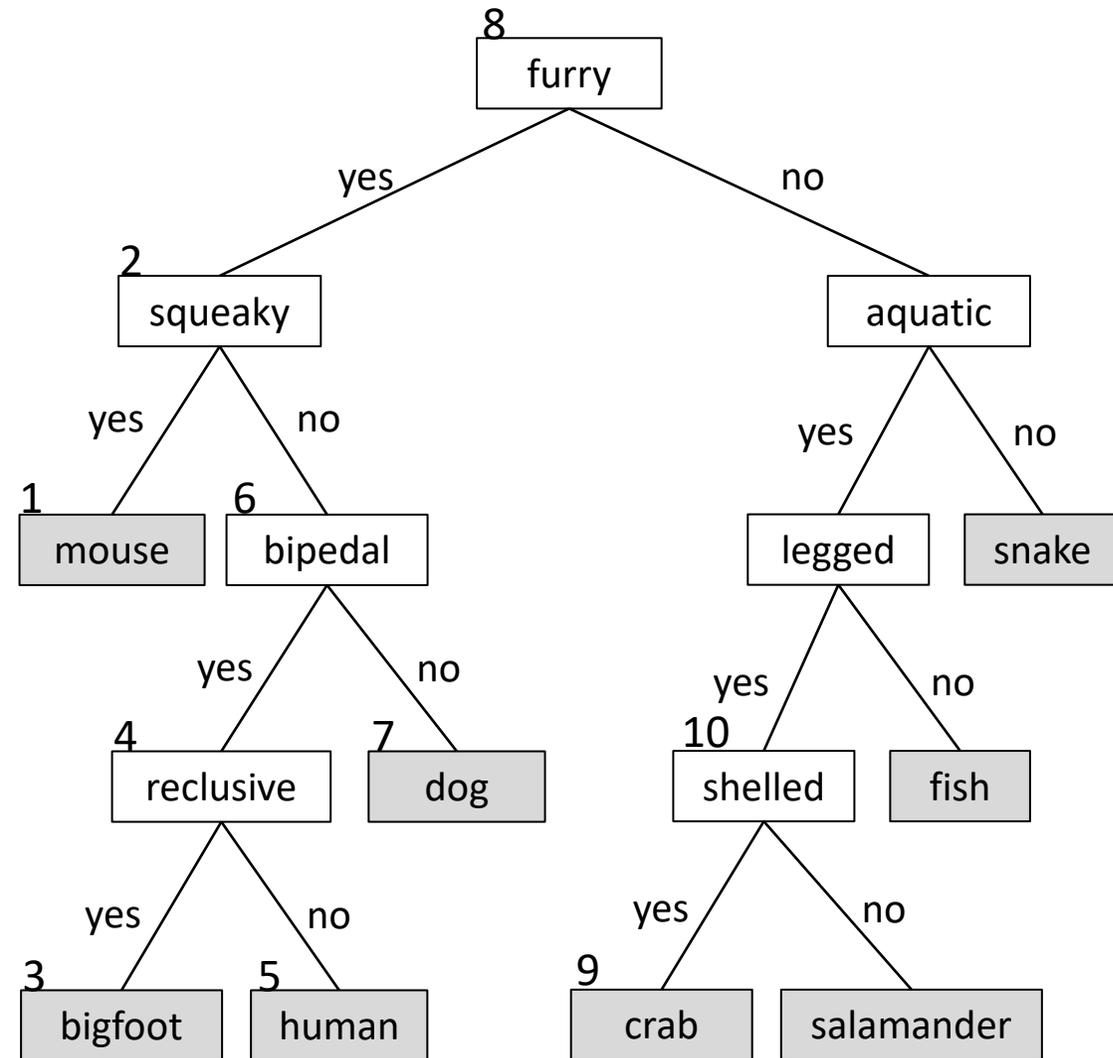
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



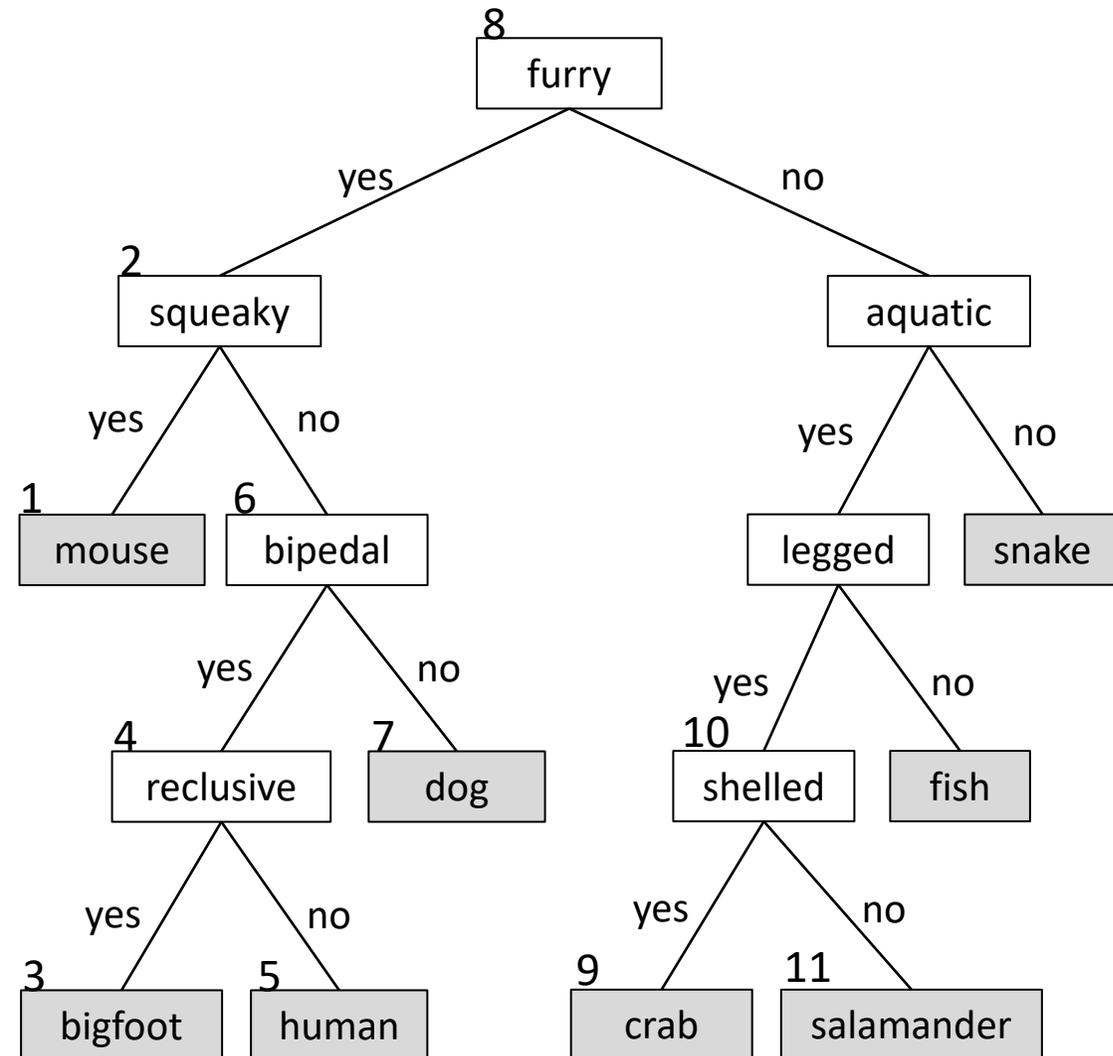
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



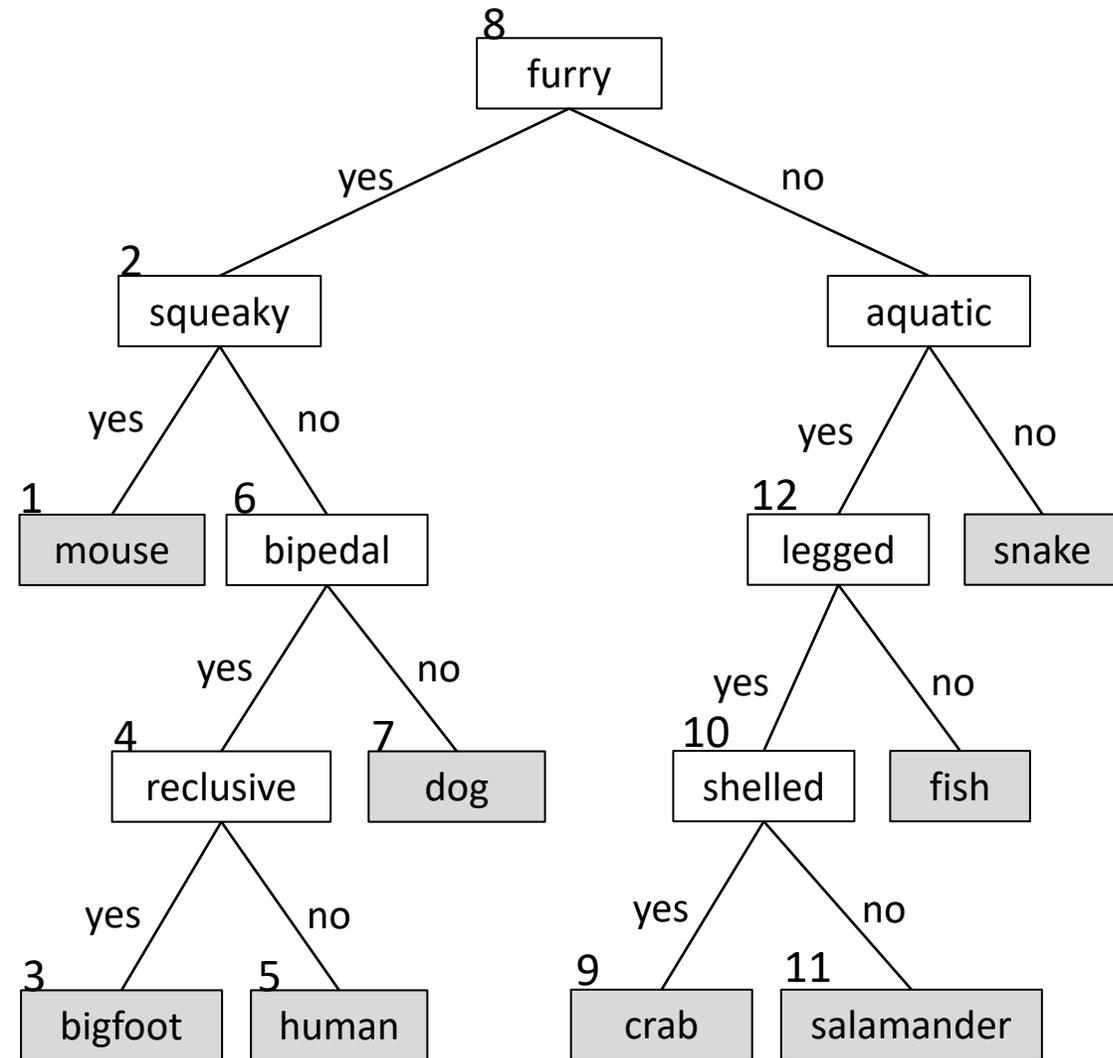
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



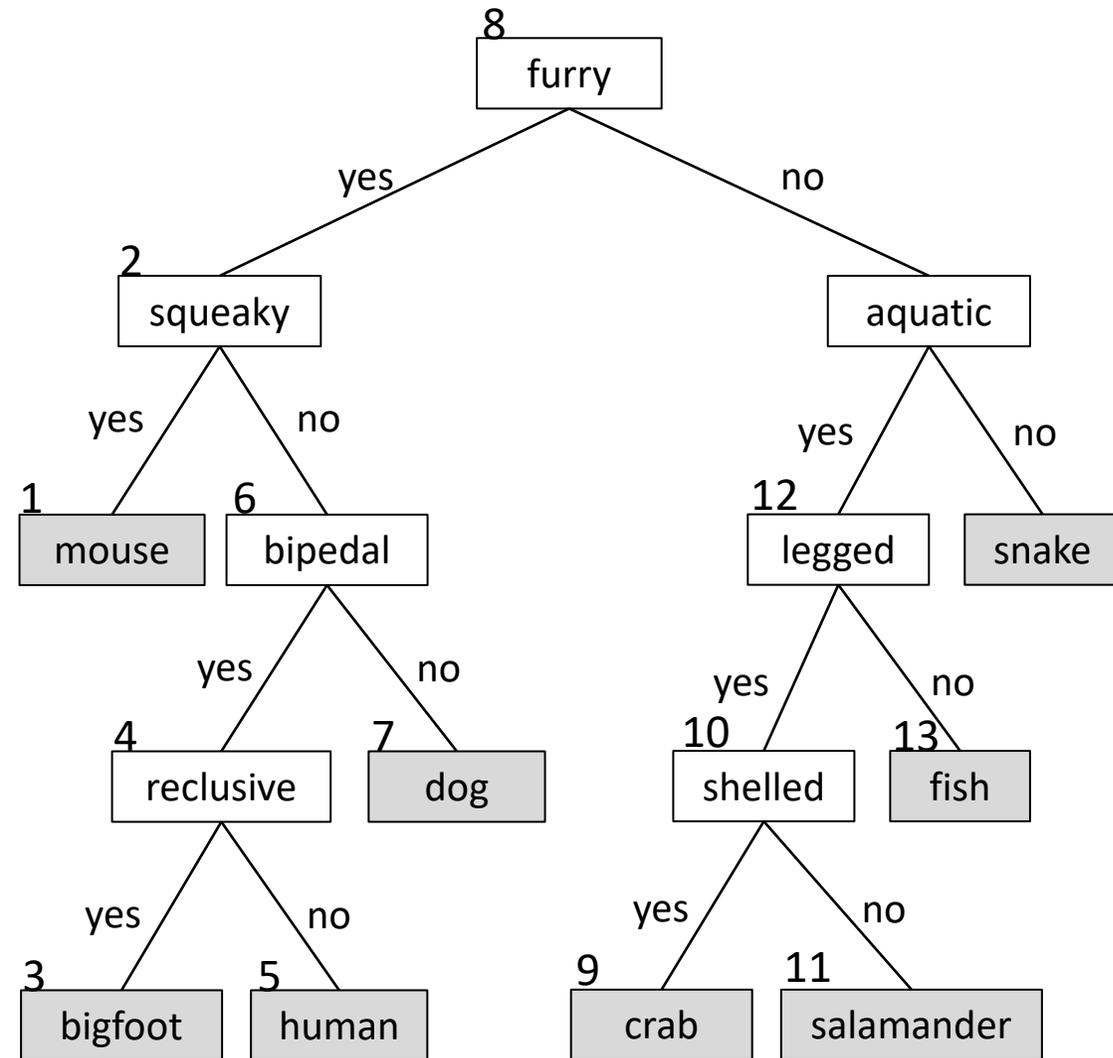
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



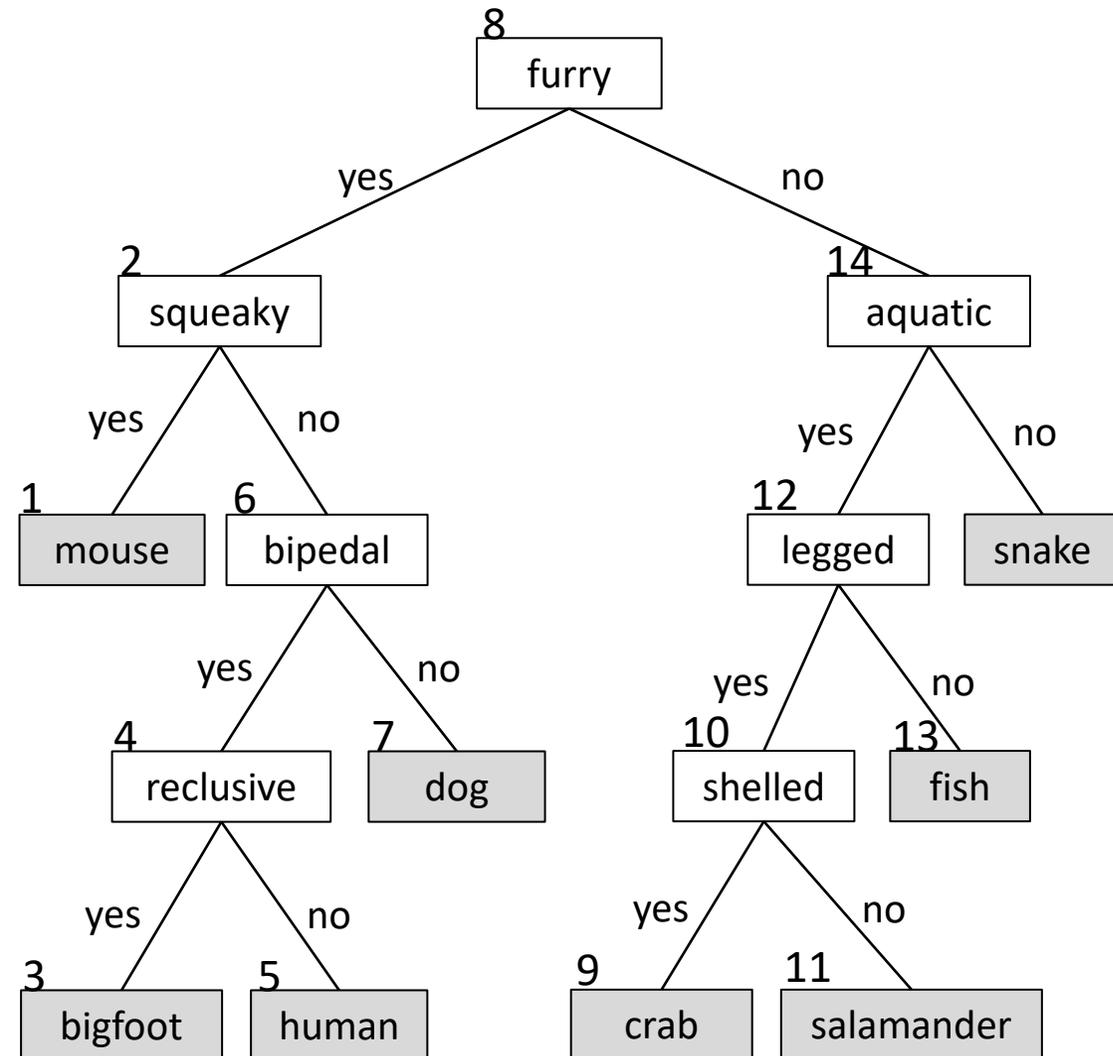
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



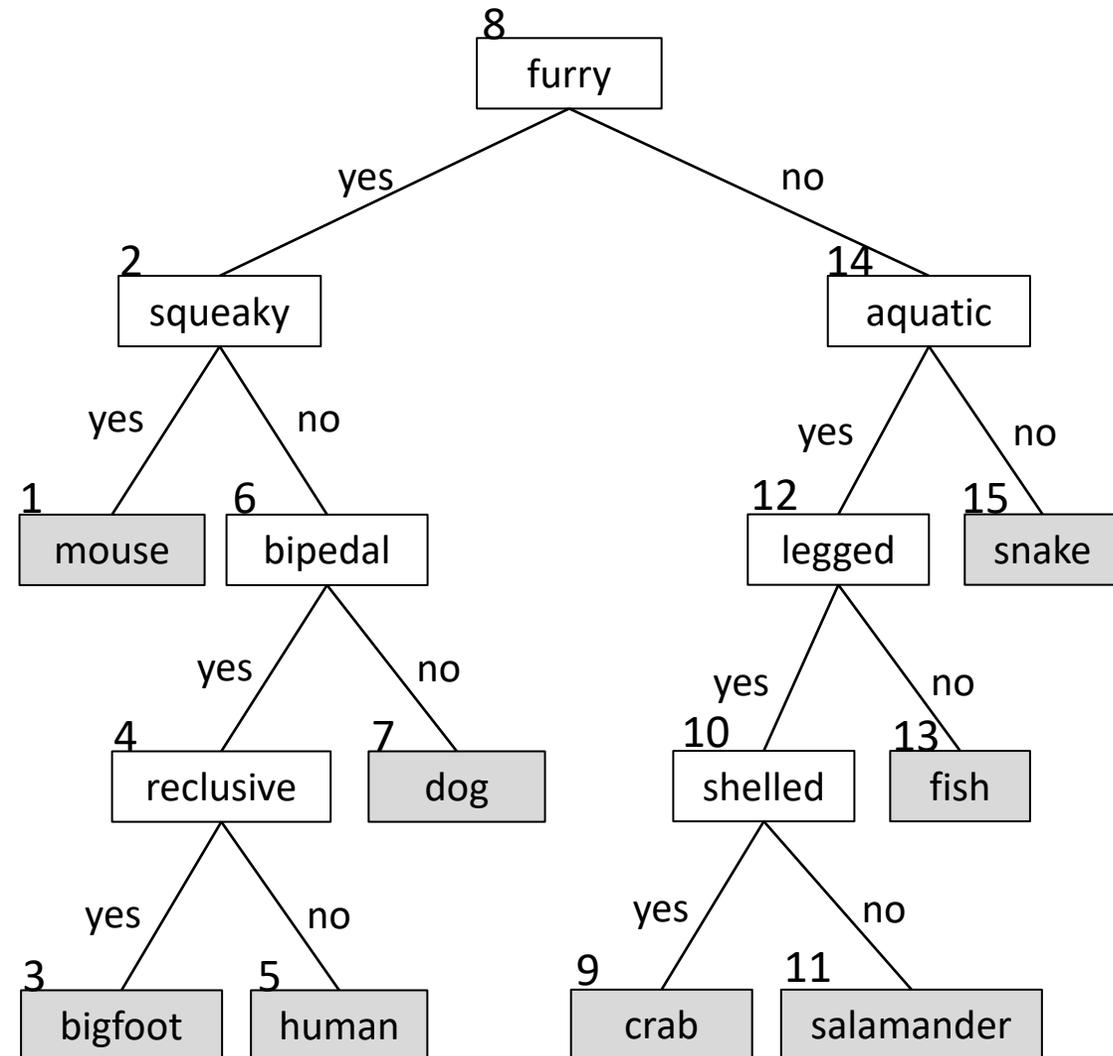
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



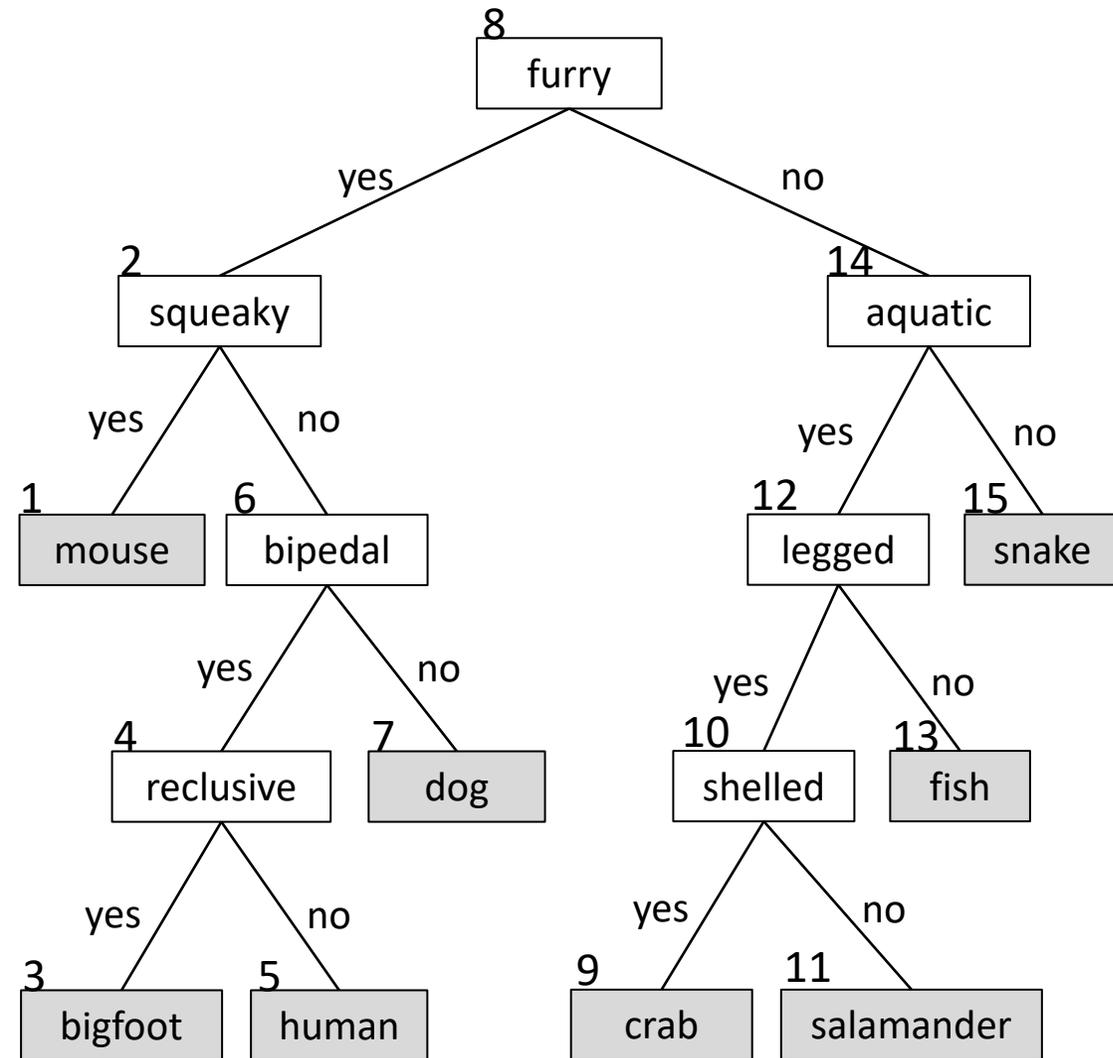
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.

File read/writing



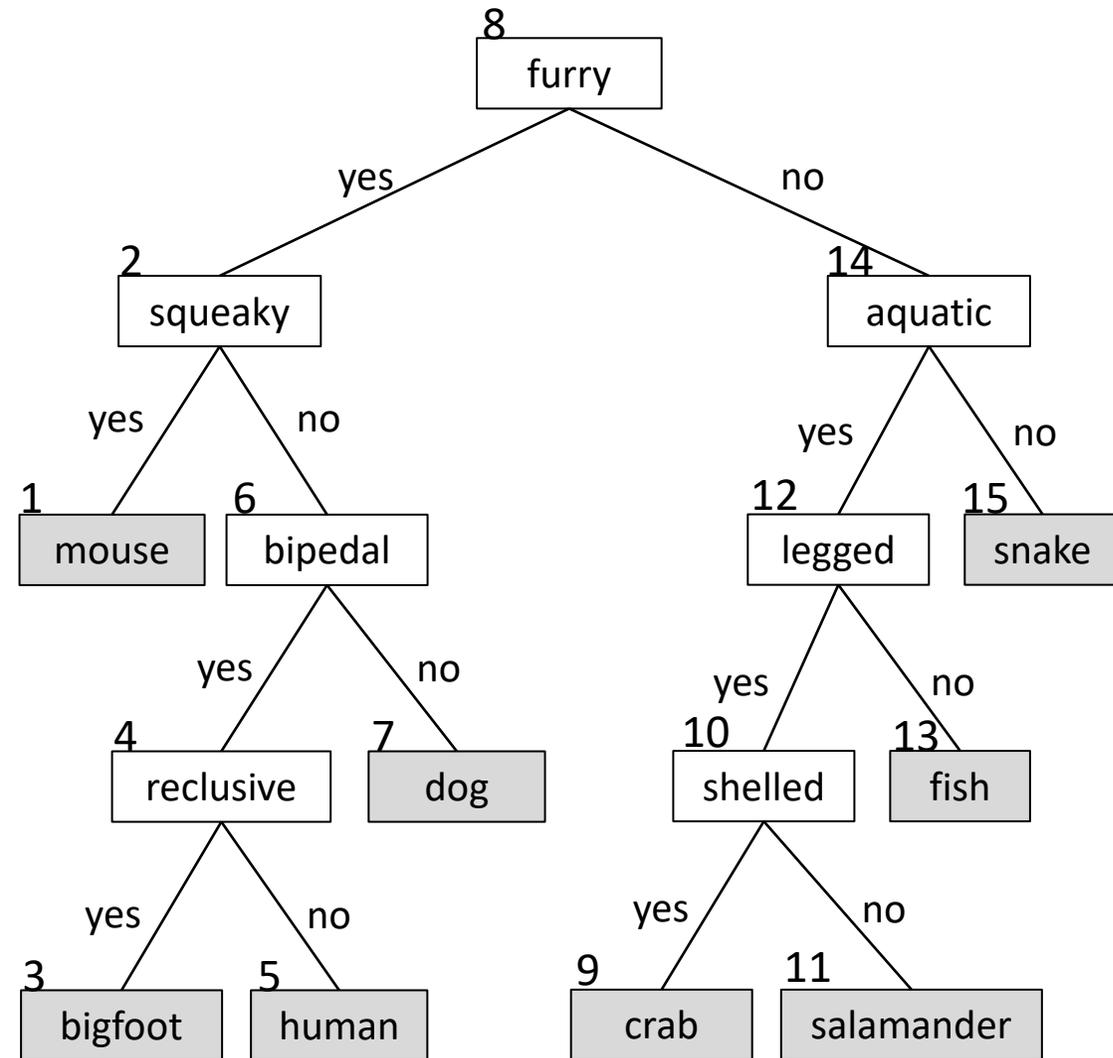
```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.
2. Do breadth first traversal and write tag and text values to file. E.g. 8-furry,2-squeaky,14-aquatic,1-mouse,6-bipedal,...

File read/writing



```

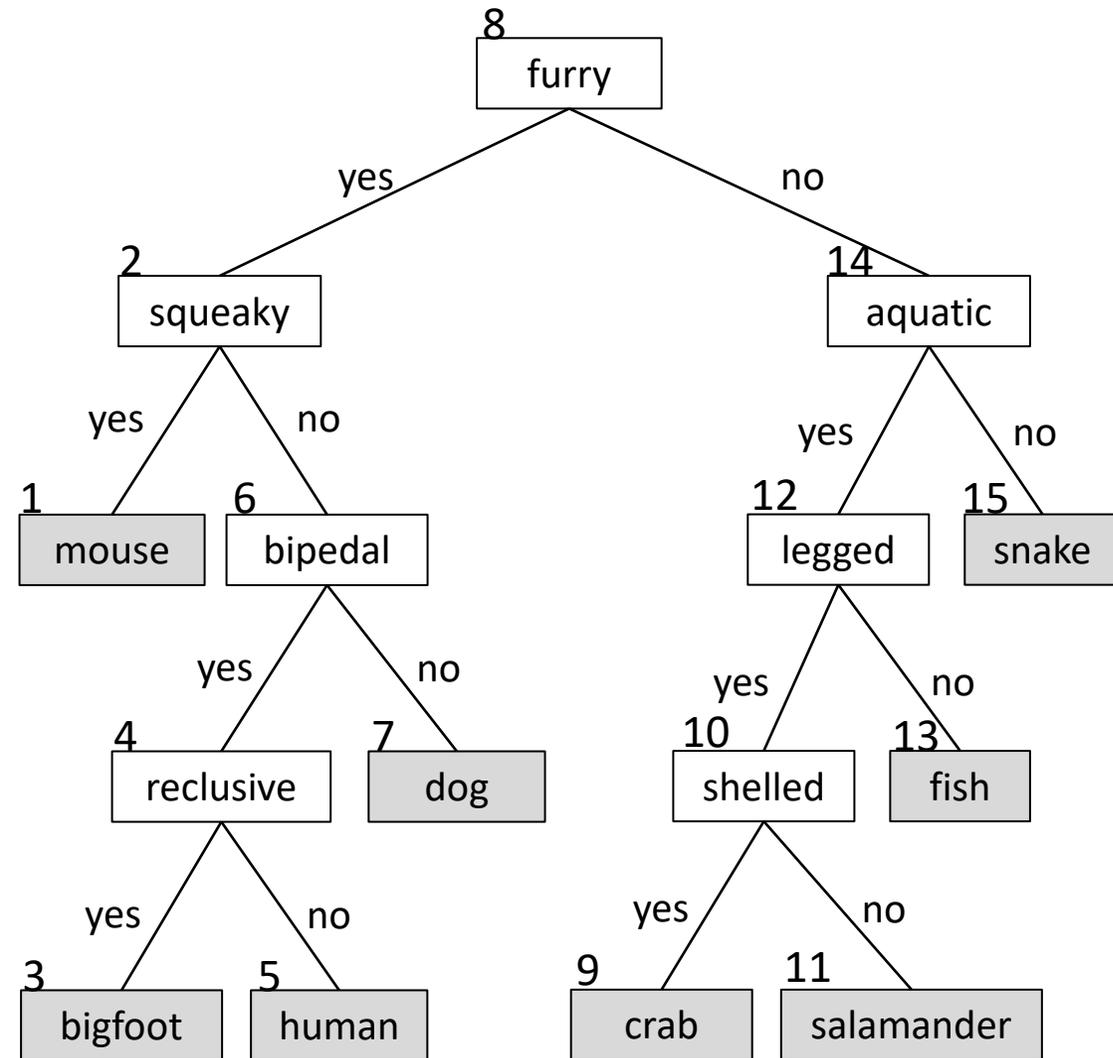
public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.
2. Do breadth first traversal and write tag and text values to file. E.g. 8-furry,2-squeaky,14-aquatic,1-mouse,6-bipedal,...

Build from file:

File read/writing



File read/writing

```

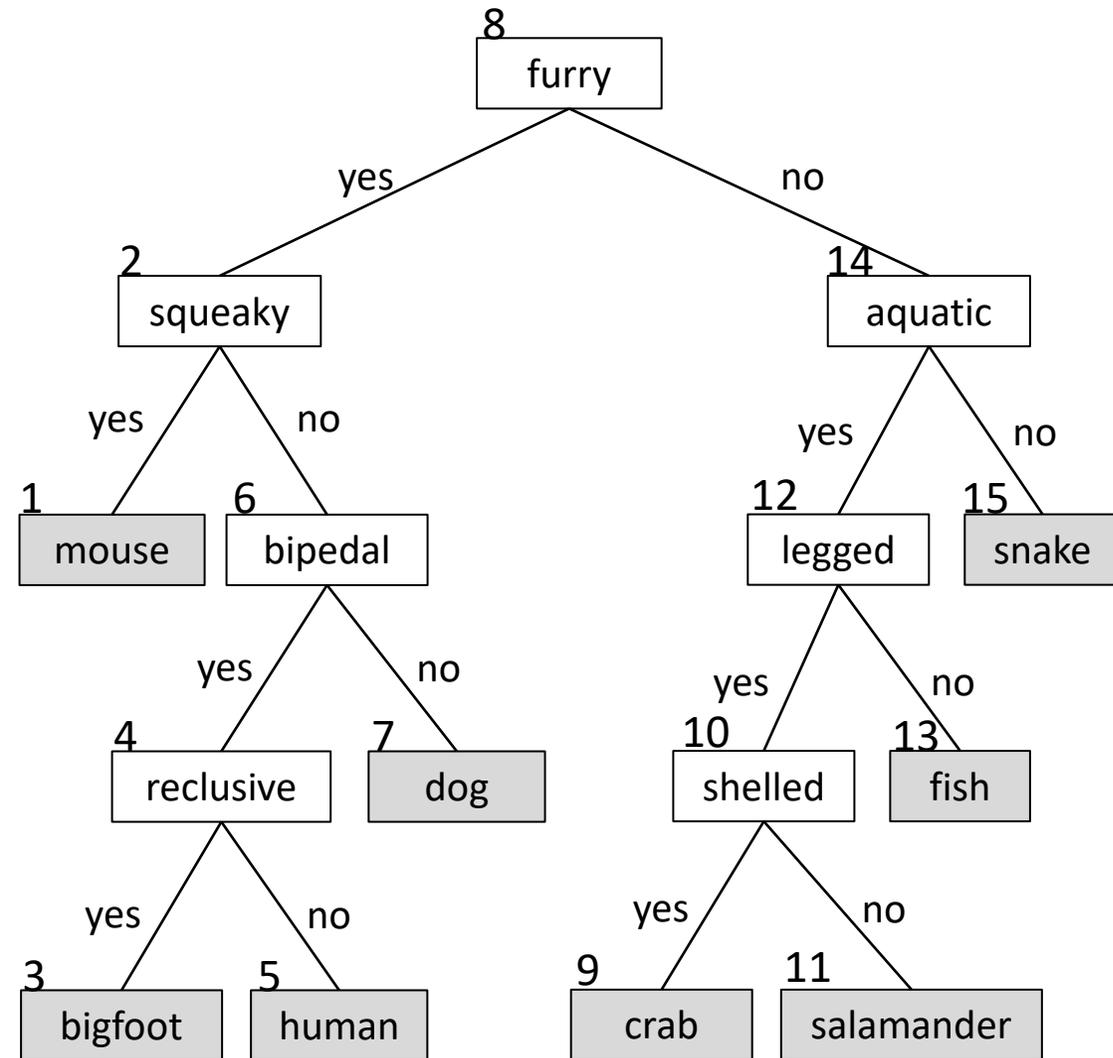
public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.
2. Do breadth first traversal and write tag and text values to file. E.g. 8-furry,2-squeaky,14-aquatic,1-mouse,6-bipedal,...

Build from file:

1. Parse input on commas to get each entry.
2. Parse each entry on dash to get tag value and text value.



File read/writing

```

public class Node {
    private String text;
    private Node yesChild;
    private Node noChild;
    private Node parent;
    private int tag;
    ...
}
  
```

Save to file:

1. Do inorder traversal of tree and assign sequential integer tag values.
2. Do breadth first traversal and write tag and text values to file. E.g. 8-furry,2-squeaky,14-aquatic,1-mouse,6-bipedal,...

Build from file:

1. Parse input on commas to get each entry.
2. Parse each entry on dash to get tag value and text value.
3. Use BST insert method to put tag/text where it should be.