CSCI 232:
Data Structures and Algorithms

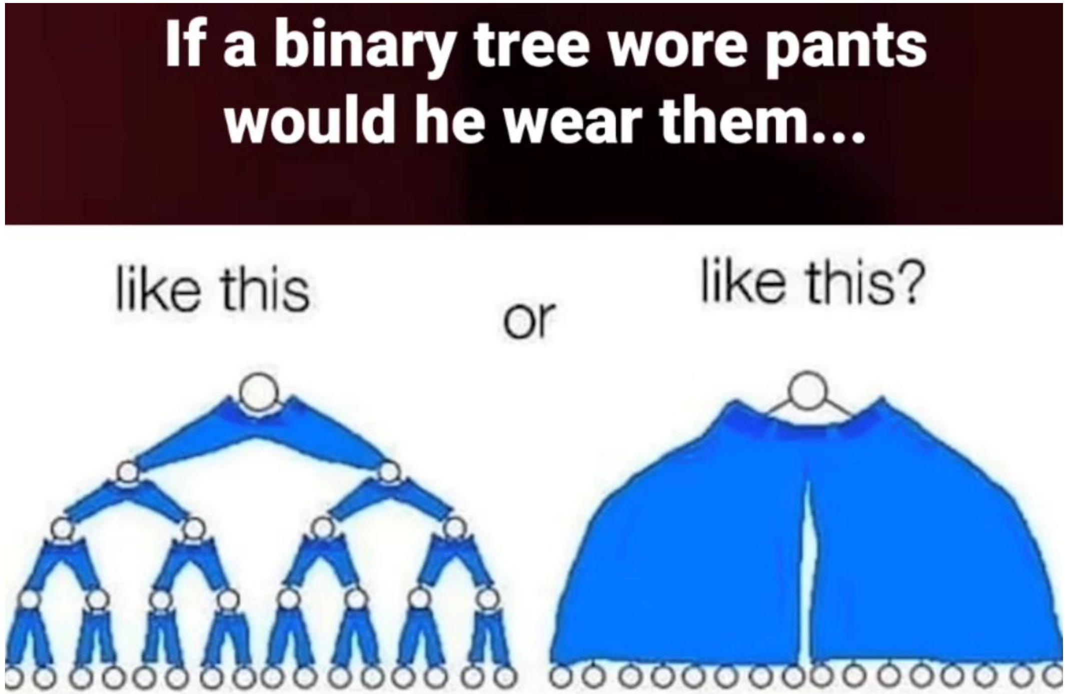
Binary Search Trees (BST) Part 2

Reese Pearsall
Spring 2025

Announcements

Program 1 posted, due Thursday Feb 20th

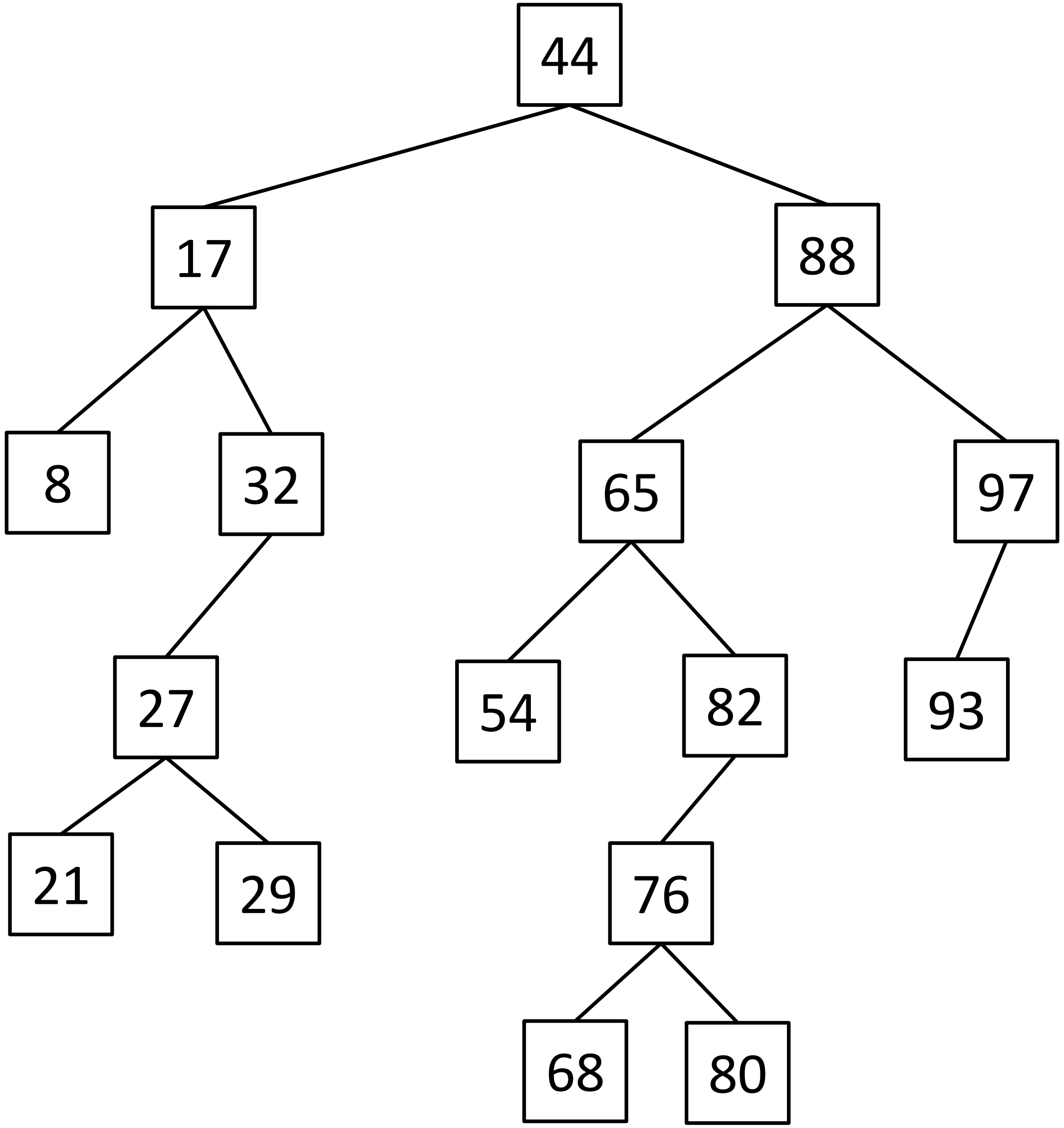
Lab 3 due this Friday
→ After today, you can complete them



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

```
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?

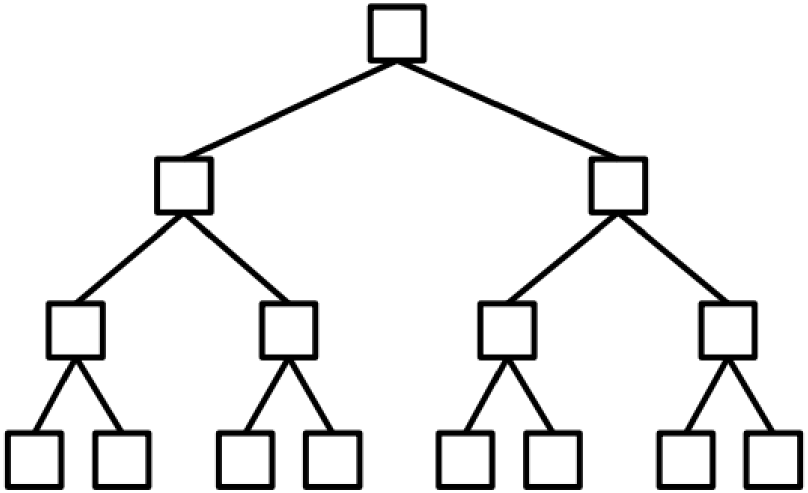
Binary Search Tree - Insertion

```
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



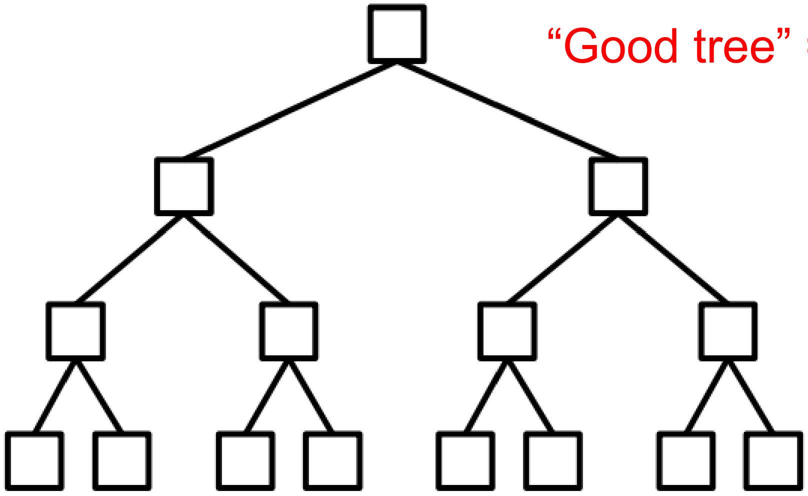
Binary Search Tree - Insertion

```
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



"Good tree" = **O(logn)**

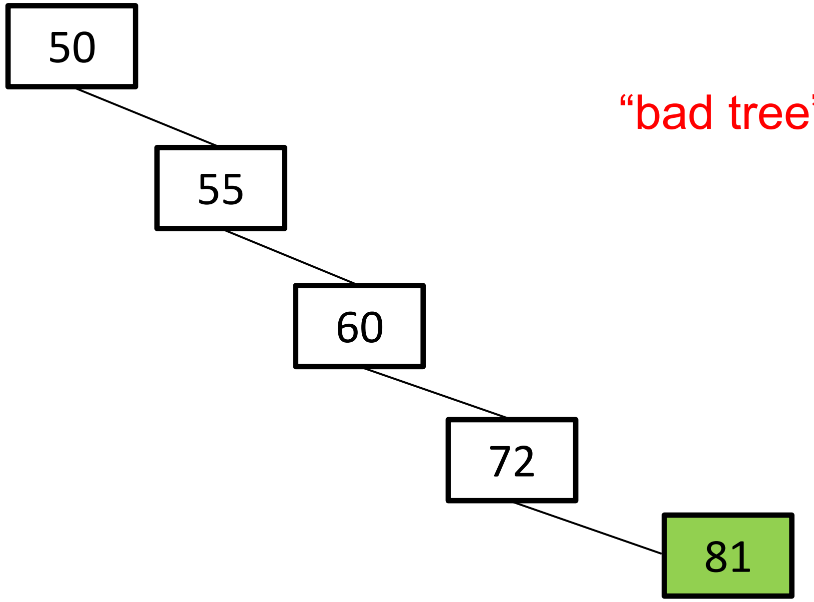
Binary Search Tree - Insertion

```
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



"bad tree" = **O(n)**

Binary Search Tree - Insertion

```java
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 → O(n)
“Good” tree → O(logn)

O(h) → h = height of tree

Running time for adding to an array?

O(n)

Binary Search Tree - Insertion

```
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(logn) insertion time, and O(logn) searching time

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 → O(n)
"Good" tree → O(logn)

O(h) → h = height of tree

Running time for adding to an array?

O(n)

Binary Search Tree - Insertion

```
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(logn)** insertion time, and **O(logn)** searching time

There is a way! Coming soon

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 → O(n)
“Good” tree → O(logn)

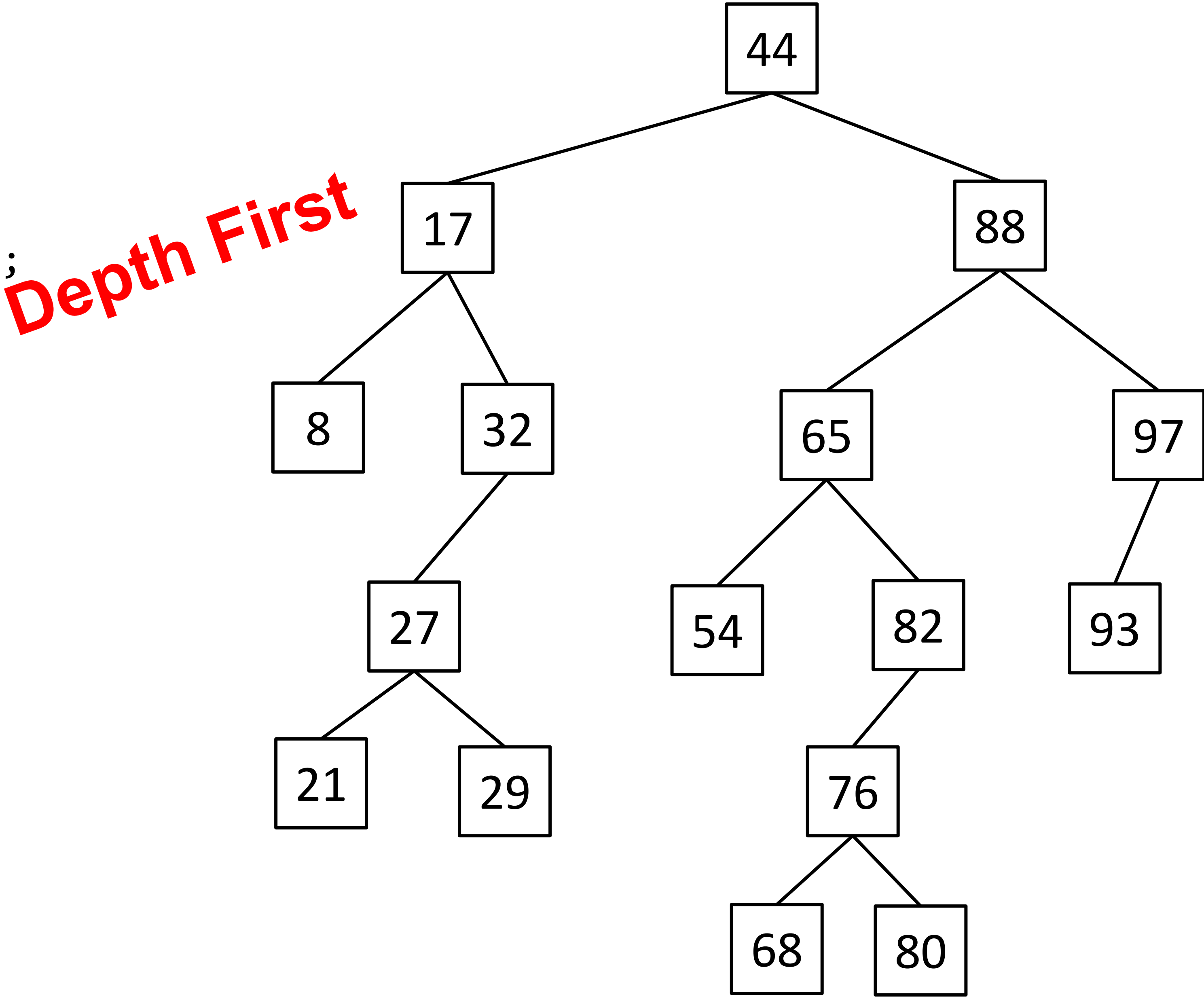
O(h) → h = height of tree

Running time for adding to an array?

O(n) 🤔

Binary Search Tree- Traversal

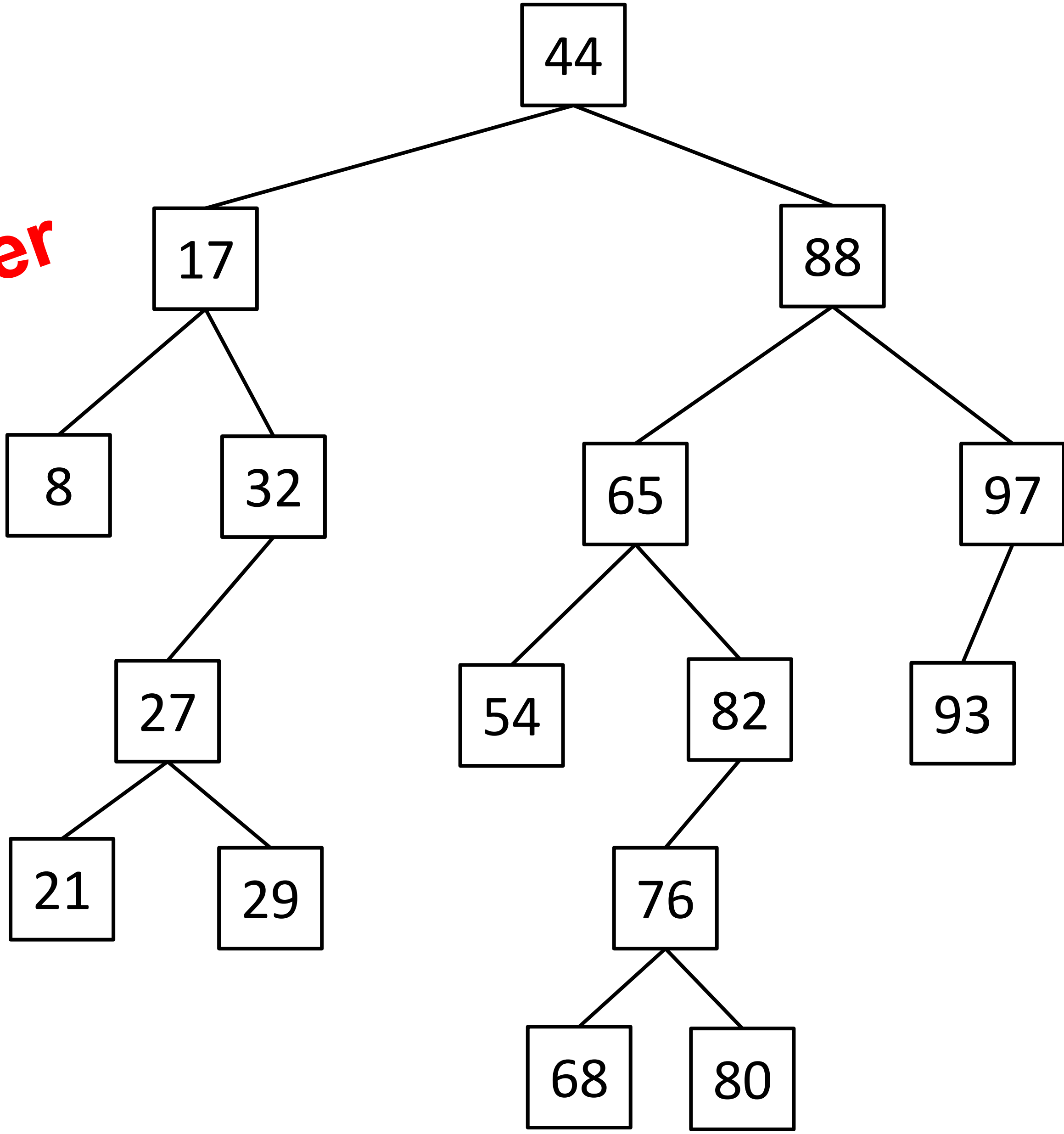
```
public void depthFirst(Node n) {
  if(n != null) {
     System.out.println(n.getValue());
     depthFirst(n.getLeft());
     depthFirst(n.getRight());
  }
}
```



Binary Search Tree- Traversal

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

Preorder

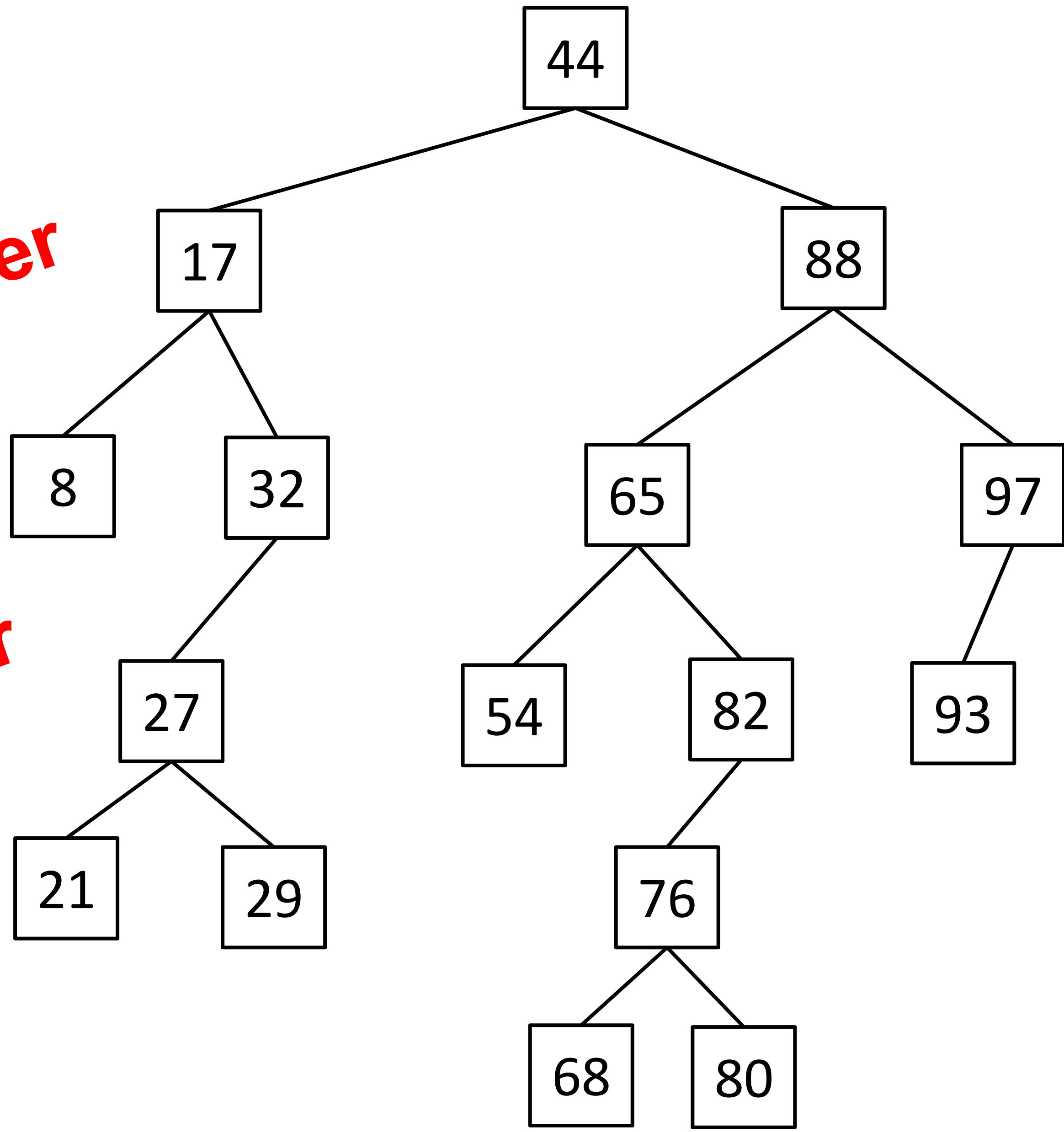


Binary Search Tree- Traversal

```
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



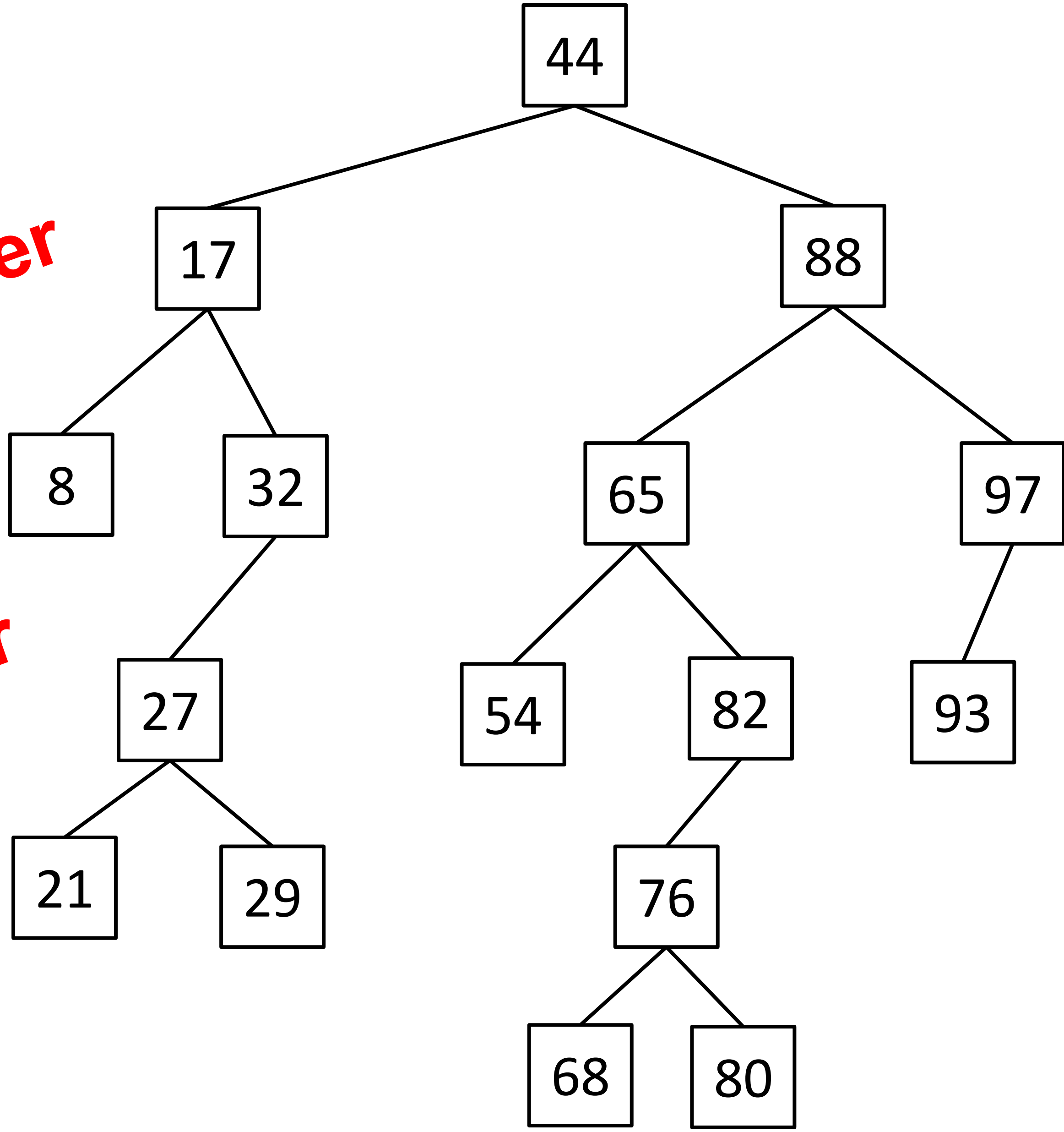
Binary Search Tree- Traversal

```java
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

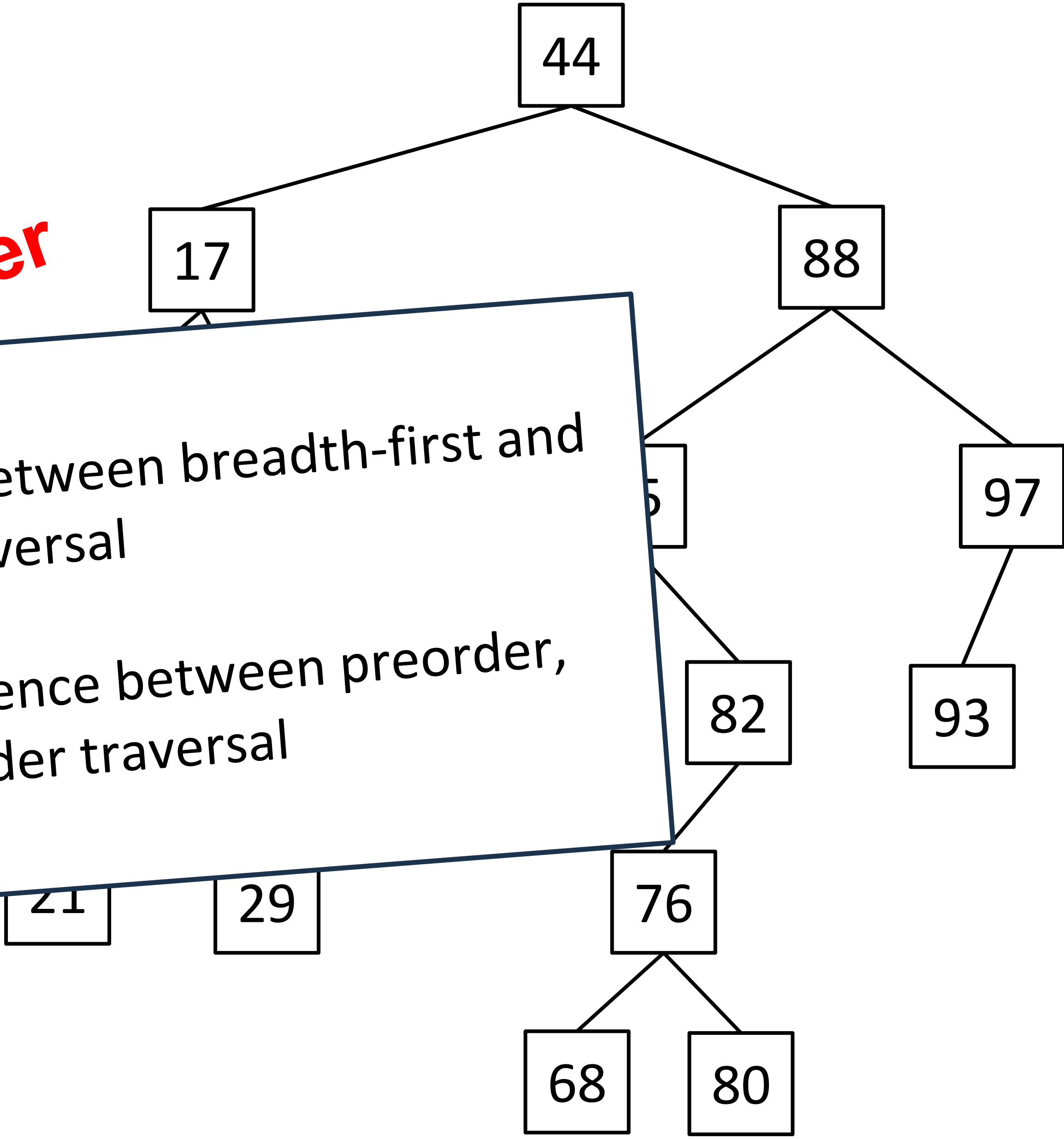
Postorder



Binary Search Tree- Traversal

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

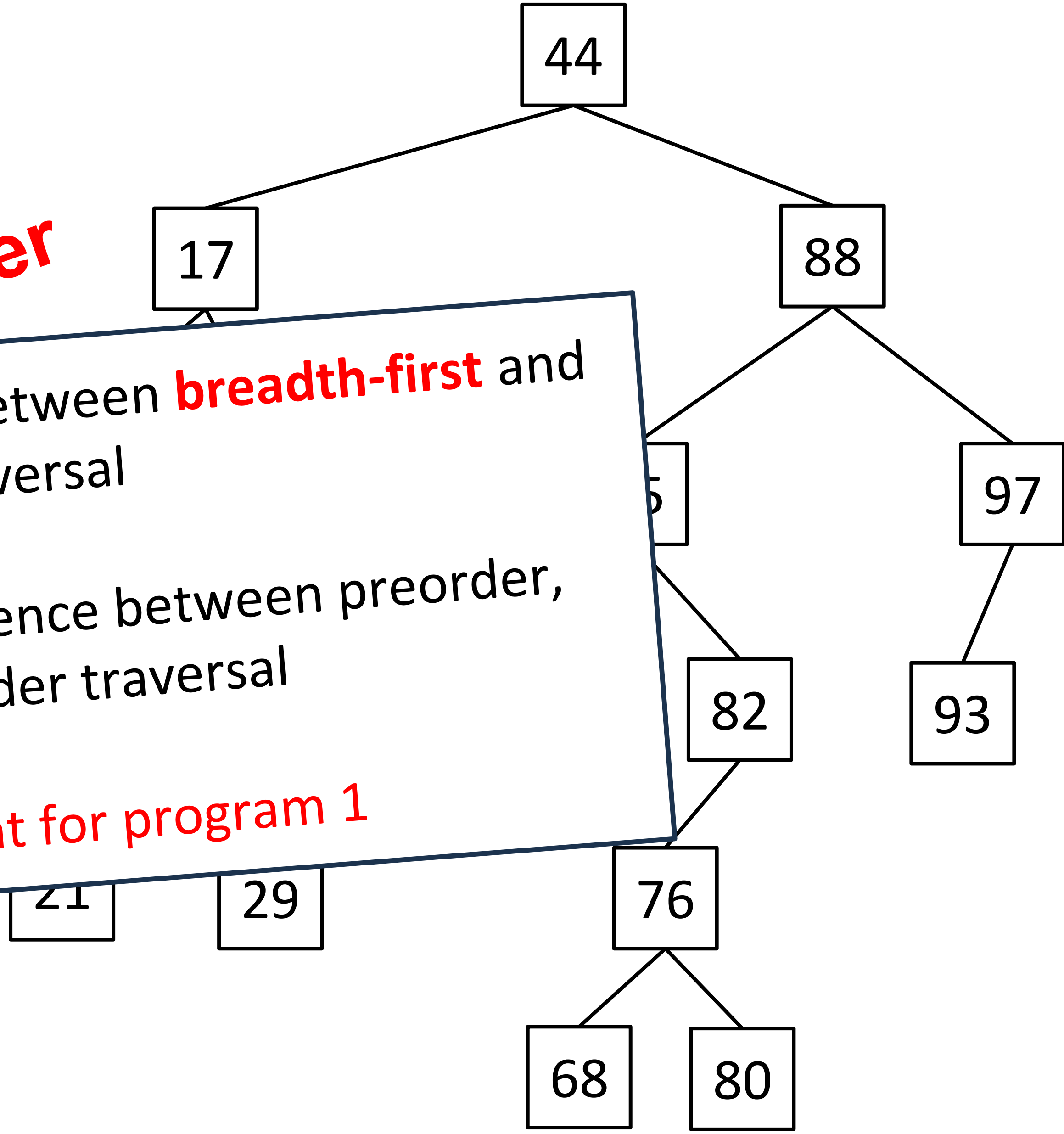
Preorder

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

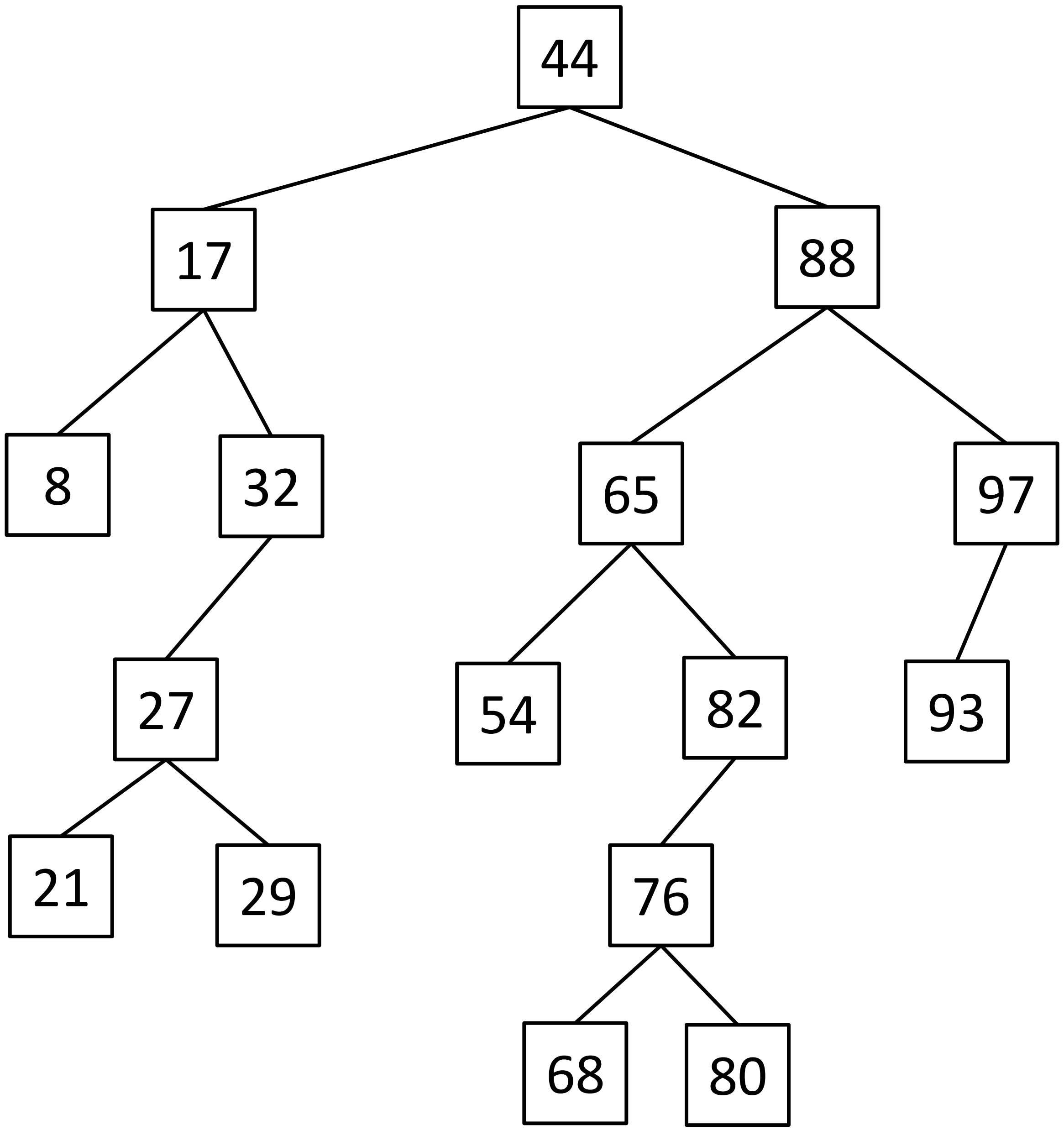
You should also know the difference between preorder, **inorder**, and postorder traversal

These will be important for program 1

Postorder



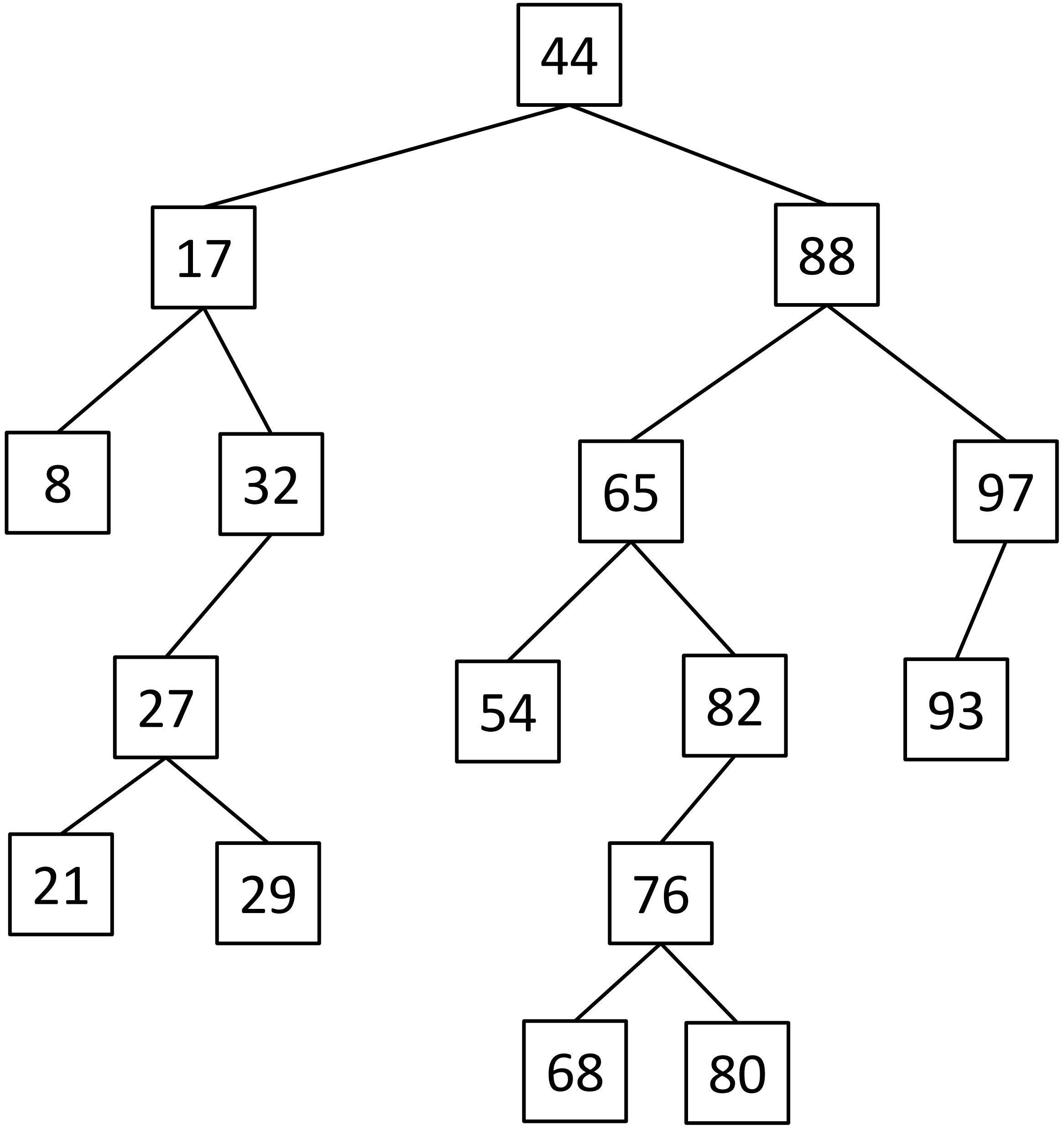
Binary Search Tree- Removal



Binary Search Tree- Removal

```
remove(68);
```

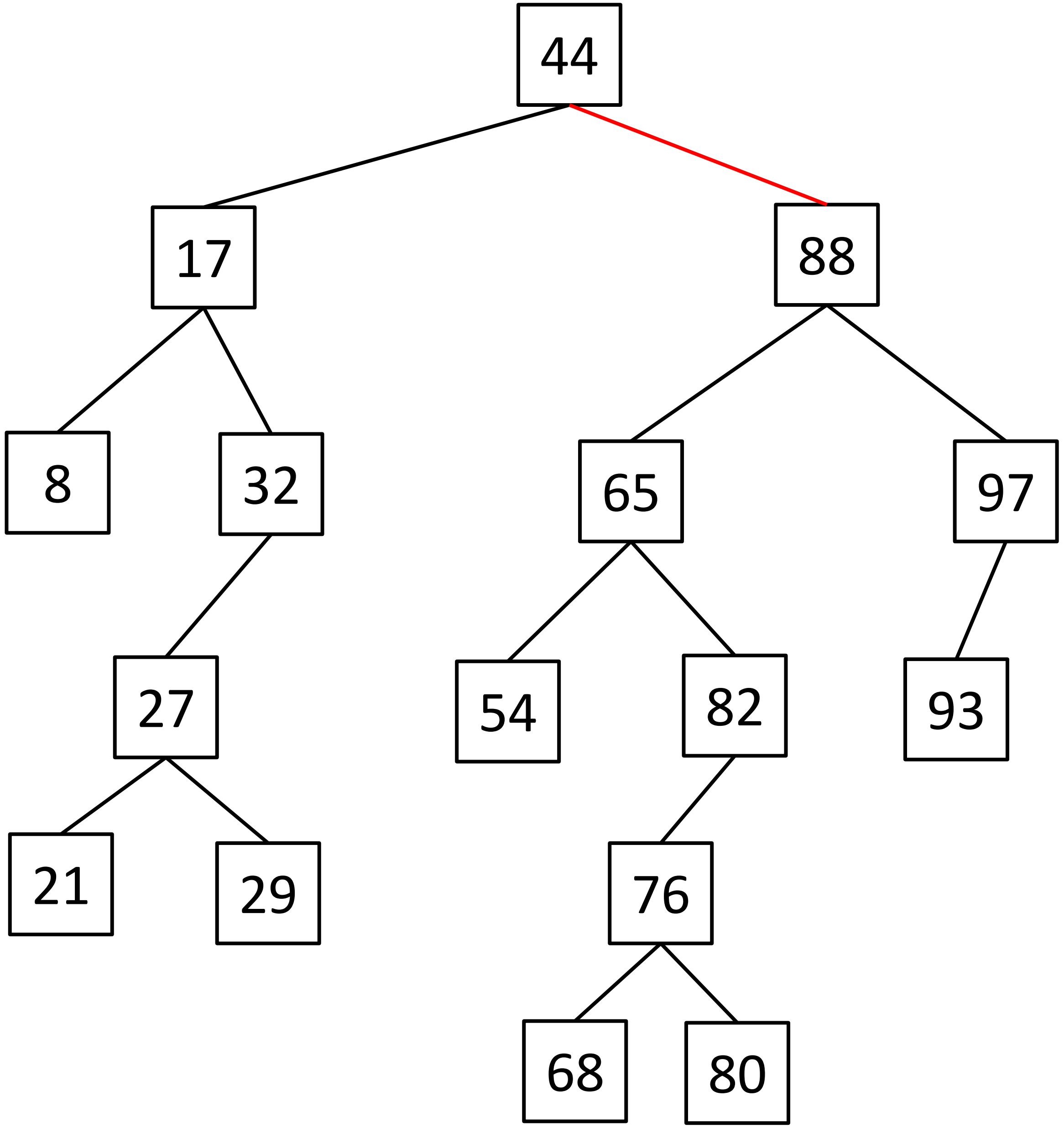
Step 1: Find the node in the tree



Binary Search Tree- Removal

```
remove(68);
```

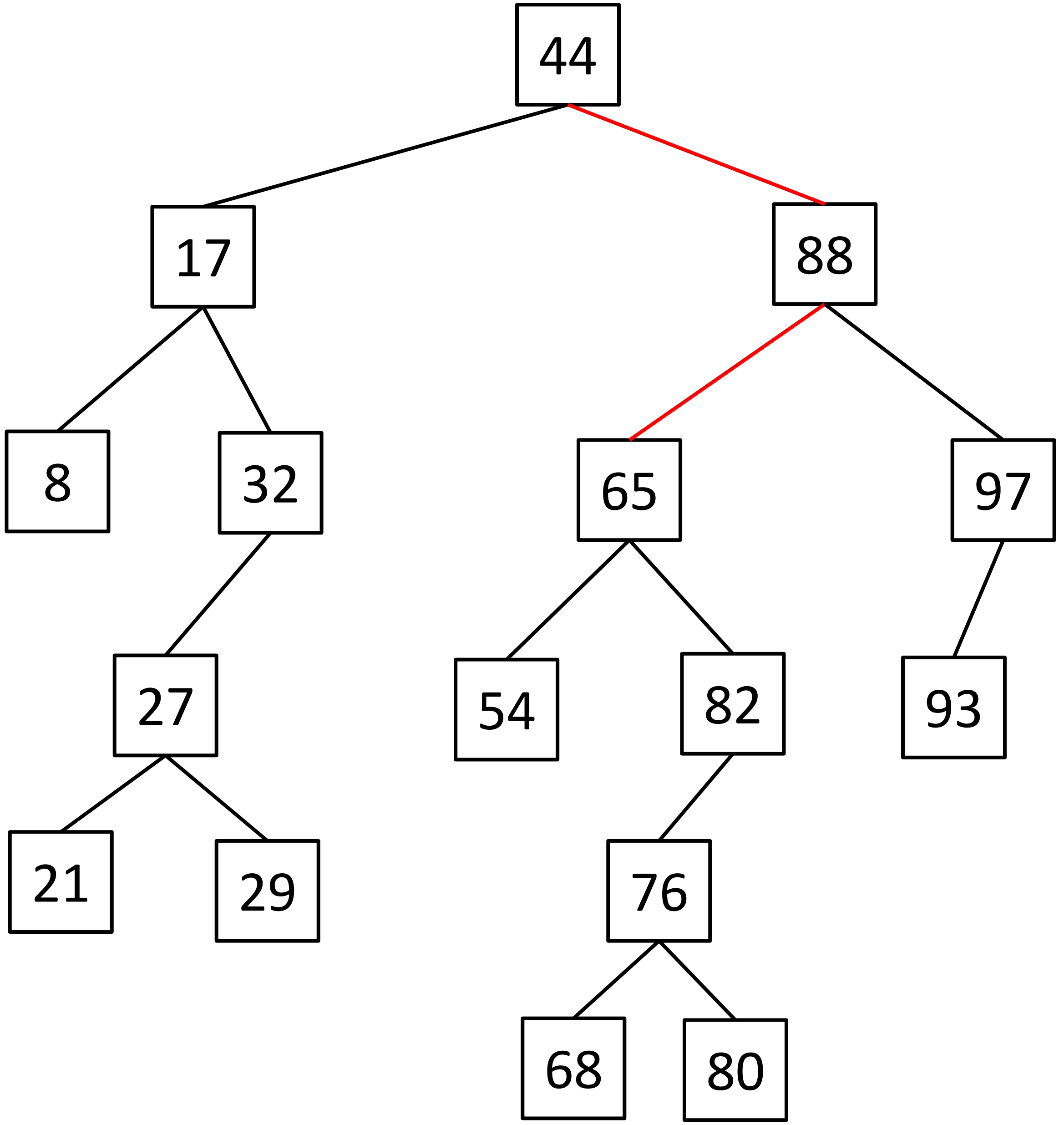
Step 1: Find the node in the tree



Binary Search Tree- Removal

`remove(68);`

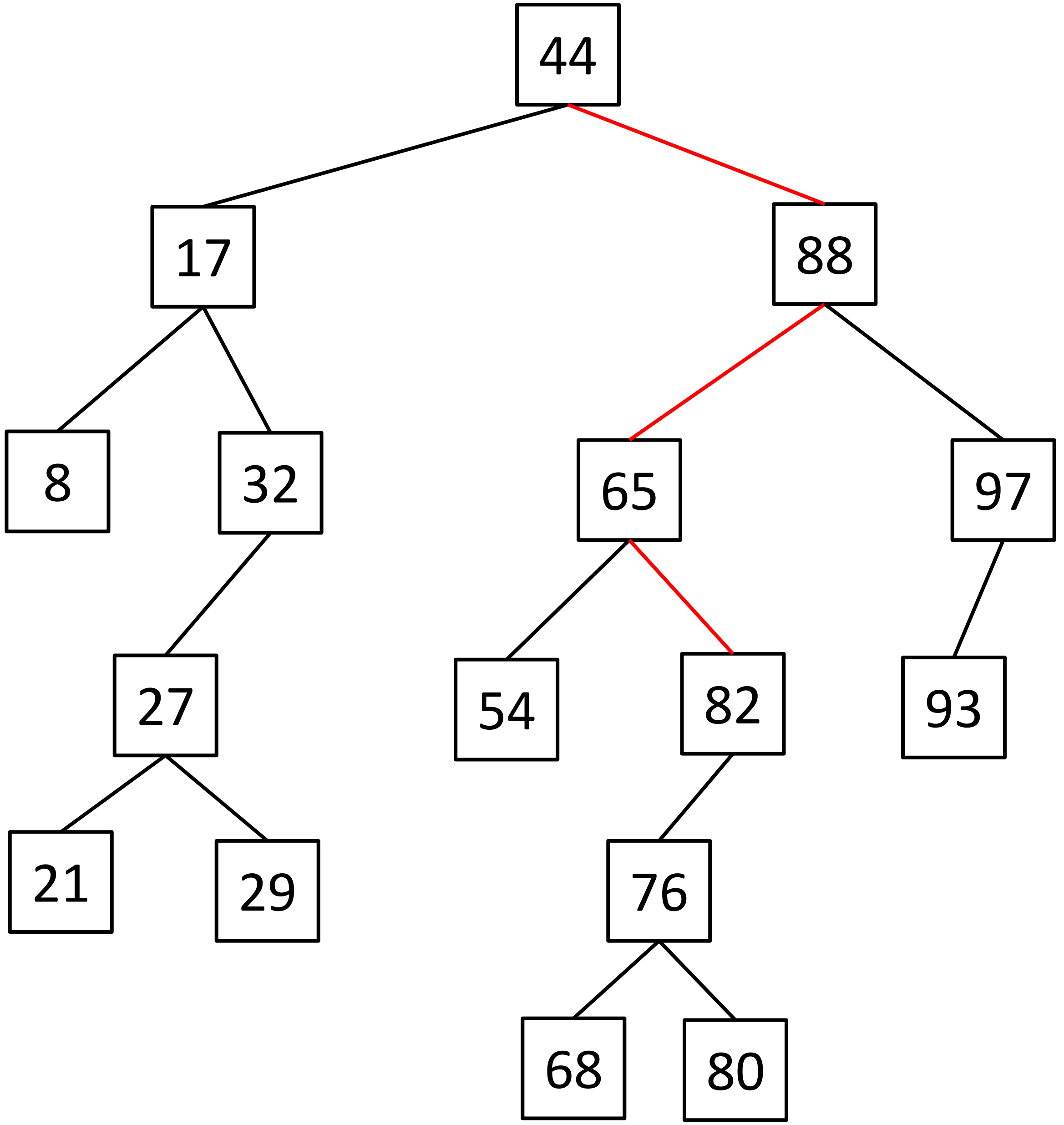
Step 1: Find the node in the tree



Binary Search Tree- Removal

```
remove(68);
```

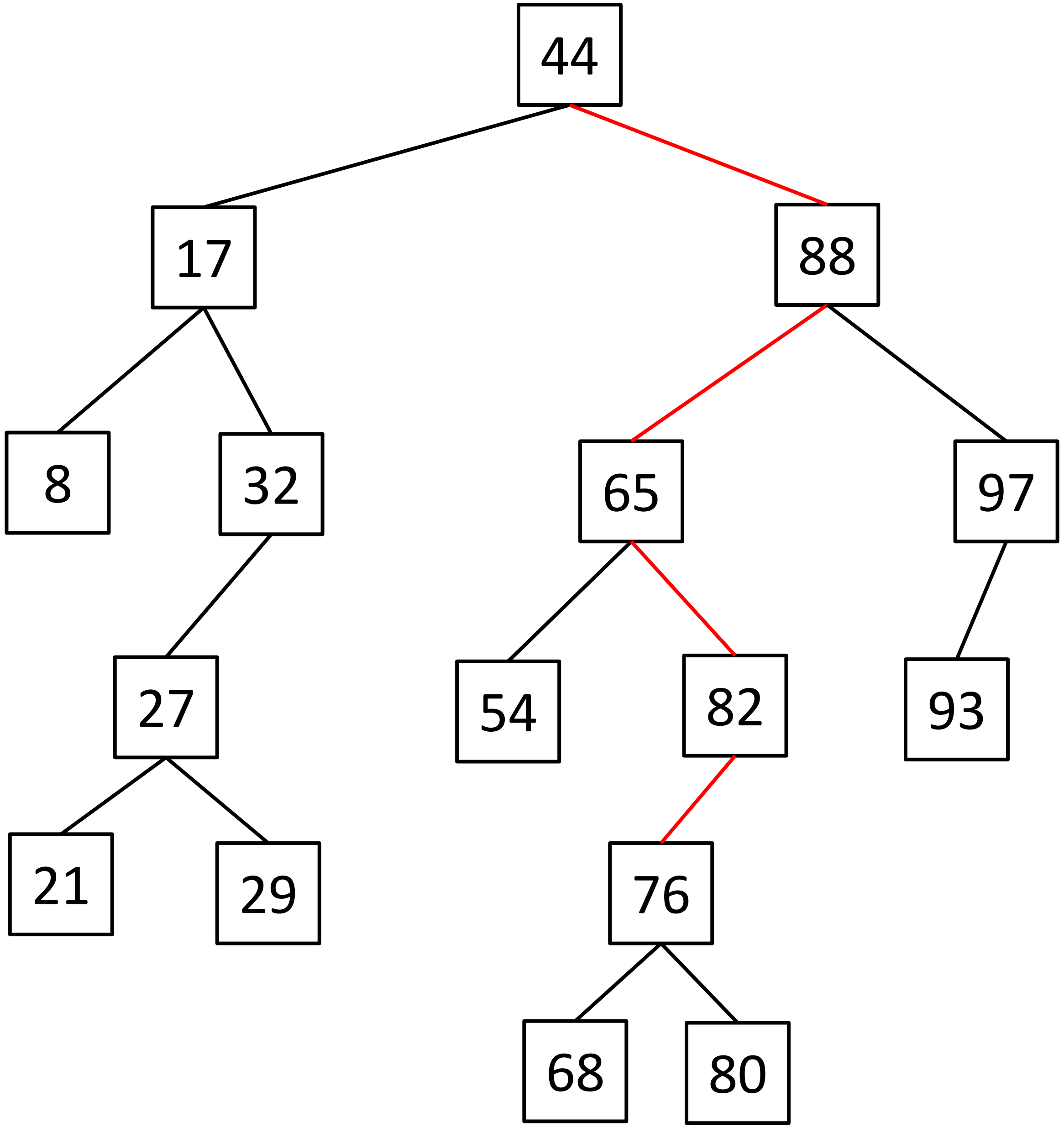
Step 1: Find the node in the tree



Binary Search Tree- Removal

```
remove(68);
```

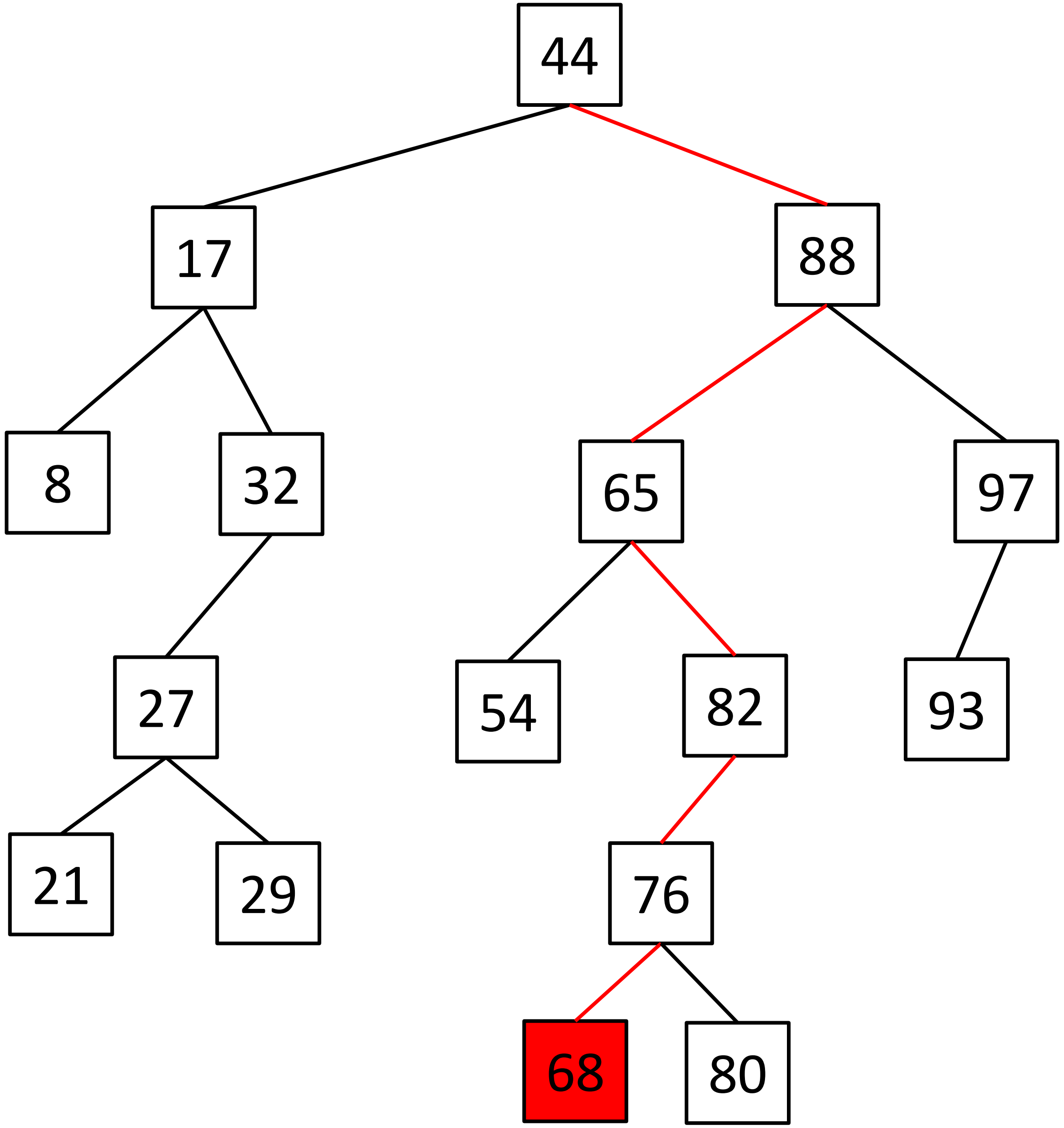
Step 1: Find the node in the tree



Binary Search Tree- Removal

```
remove(68);
```

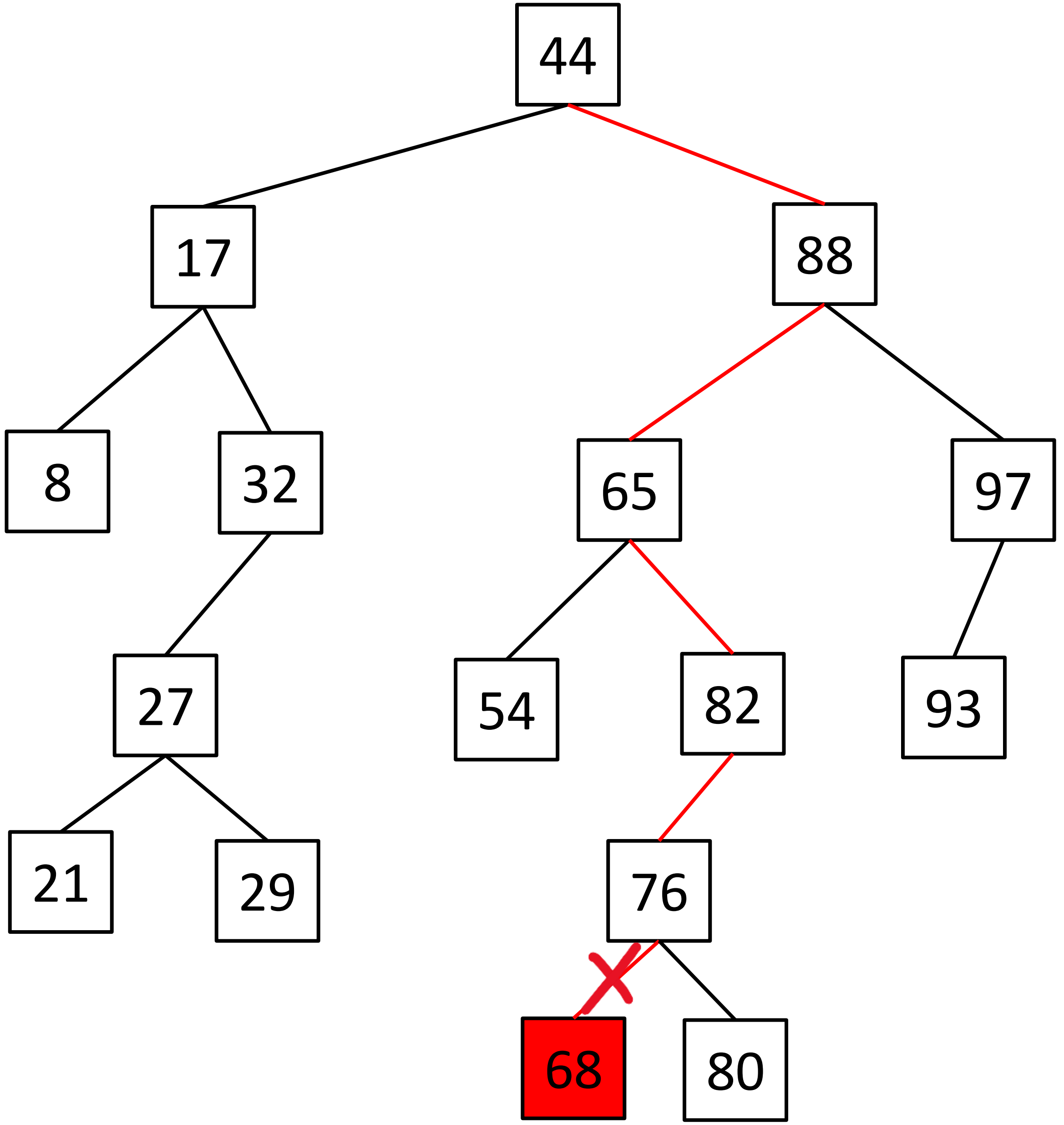
Step 1: Find the node in the tree



Binary Search Tree- Removal

```
remove(68);
```

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

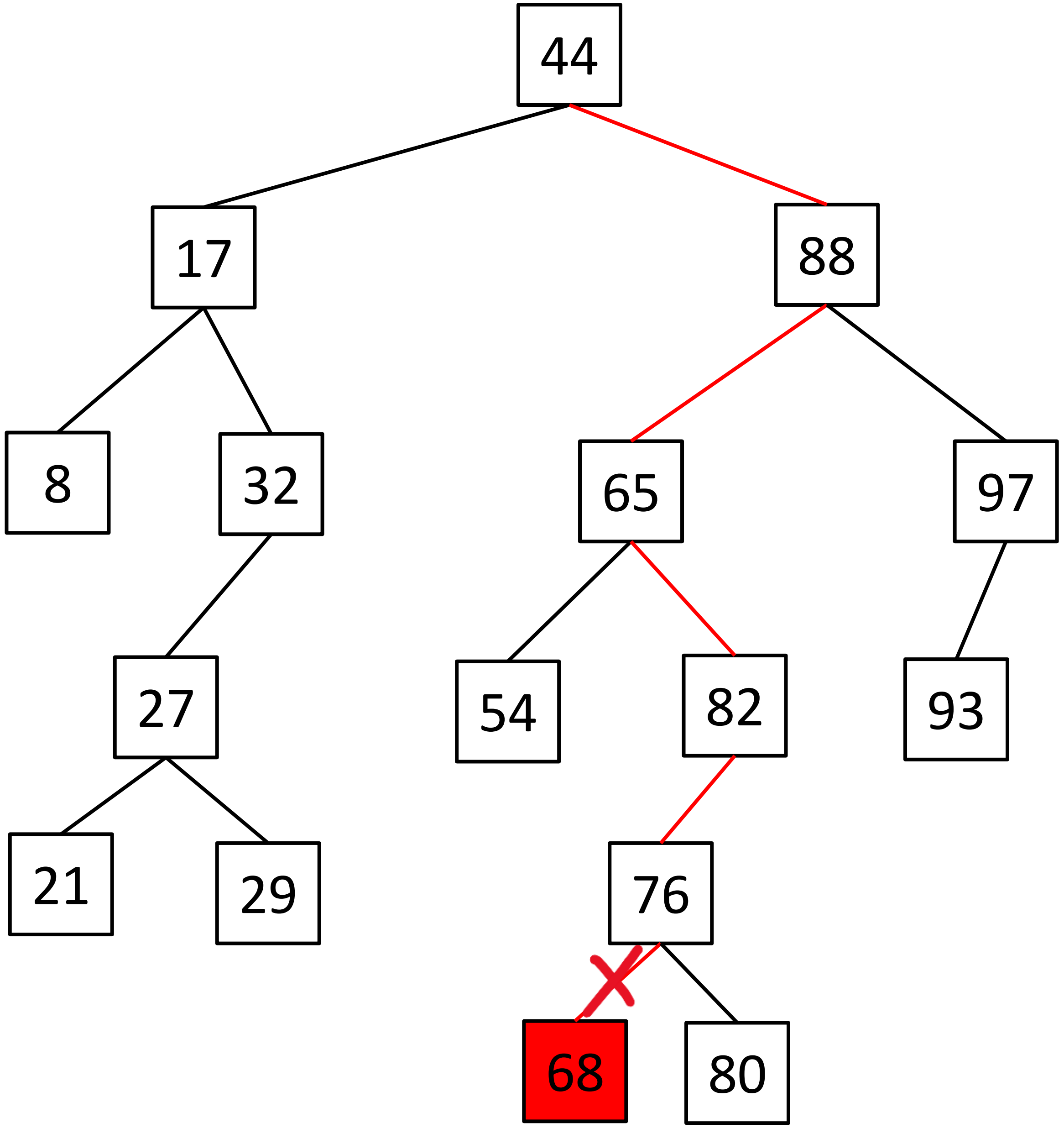


Binary Search Tree- Removal

```
remove(68);
```

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

Does this always work?

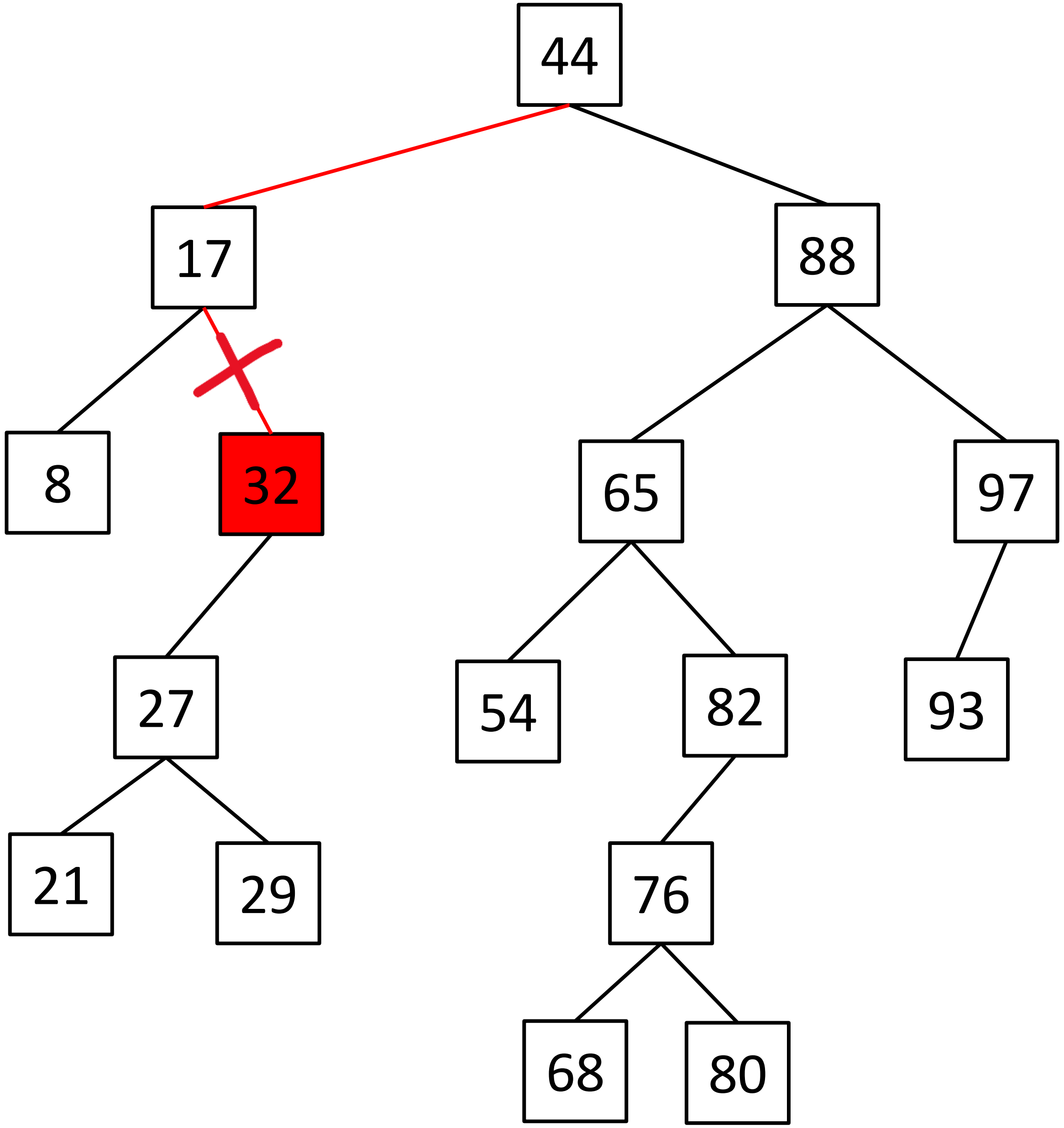


Binary Search Tree- Removal

`remove(32);`

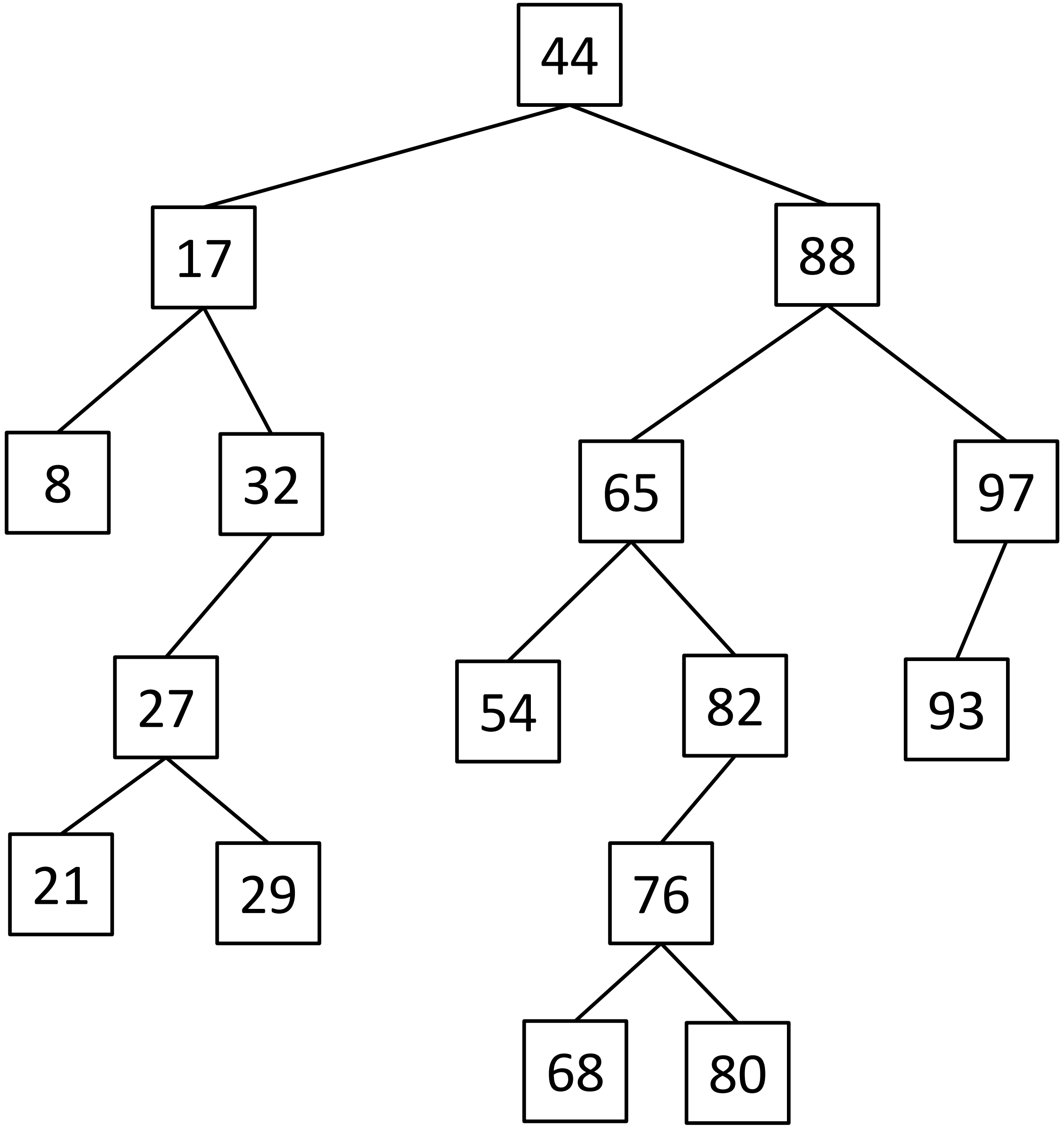
Step 1: Find the node in the tree
Step 2: Change parent to point to `null`

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

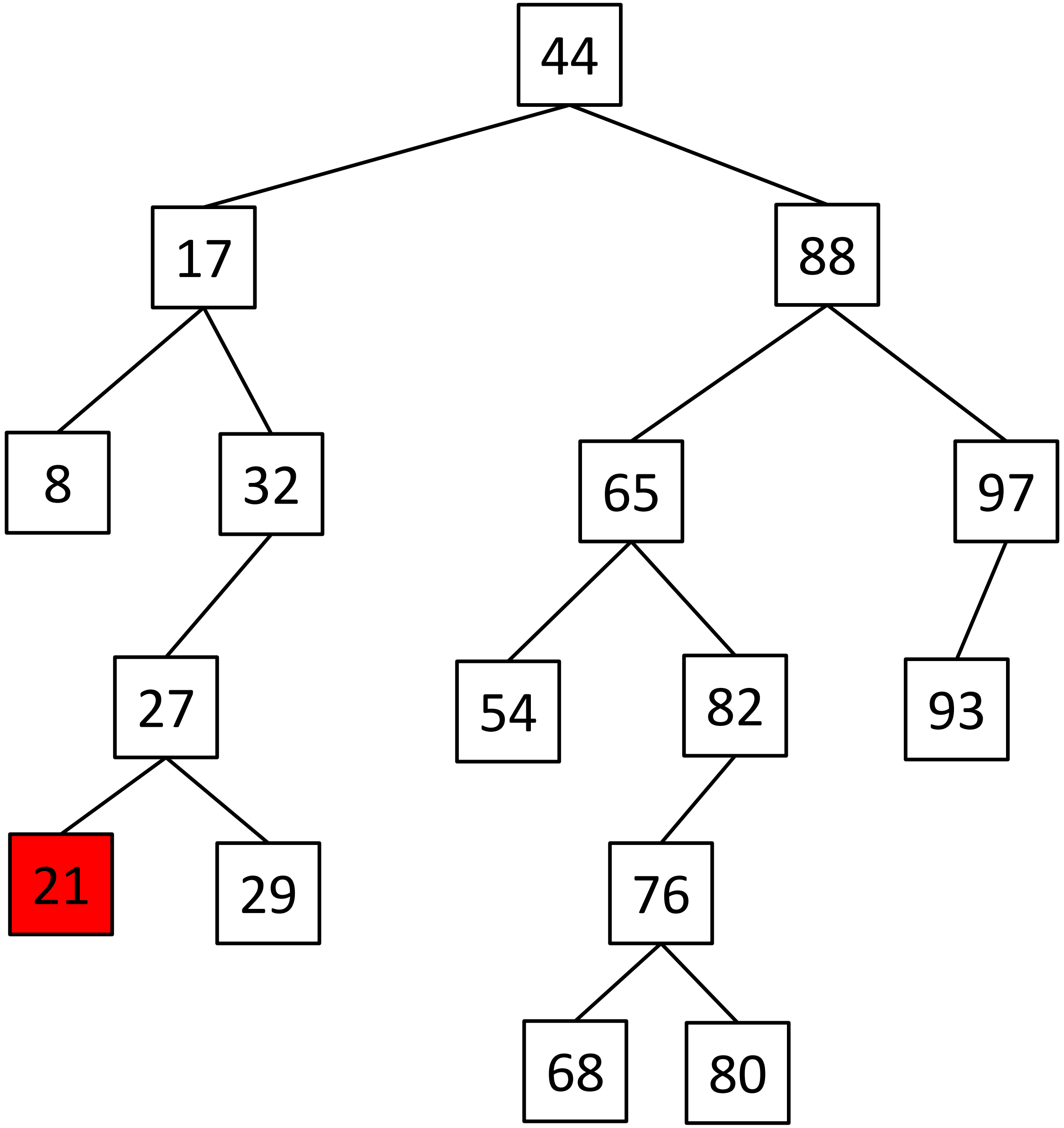


Binary Search Tree- Removal

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?

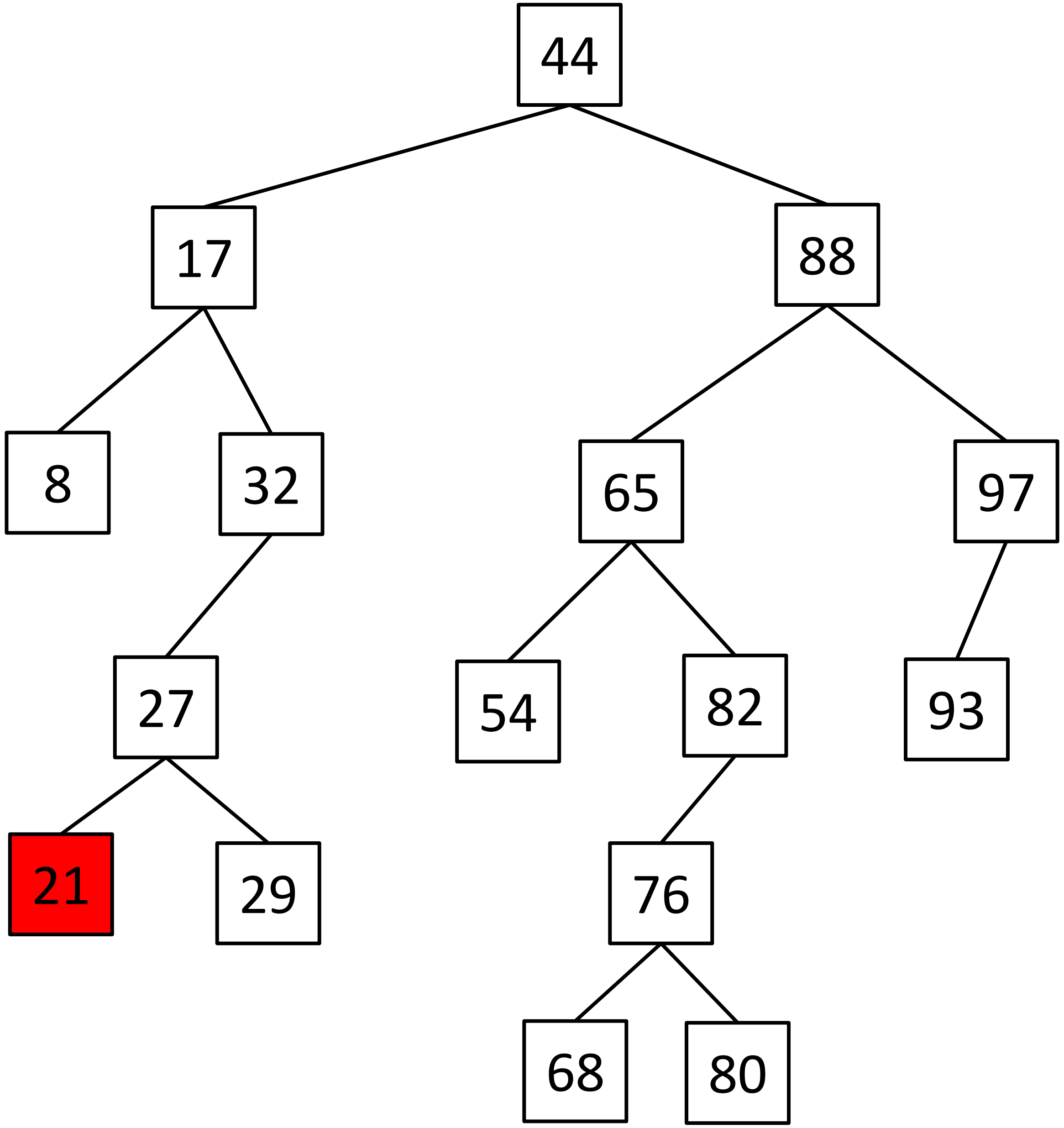


Binary Search Tree- Removal

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

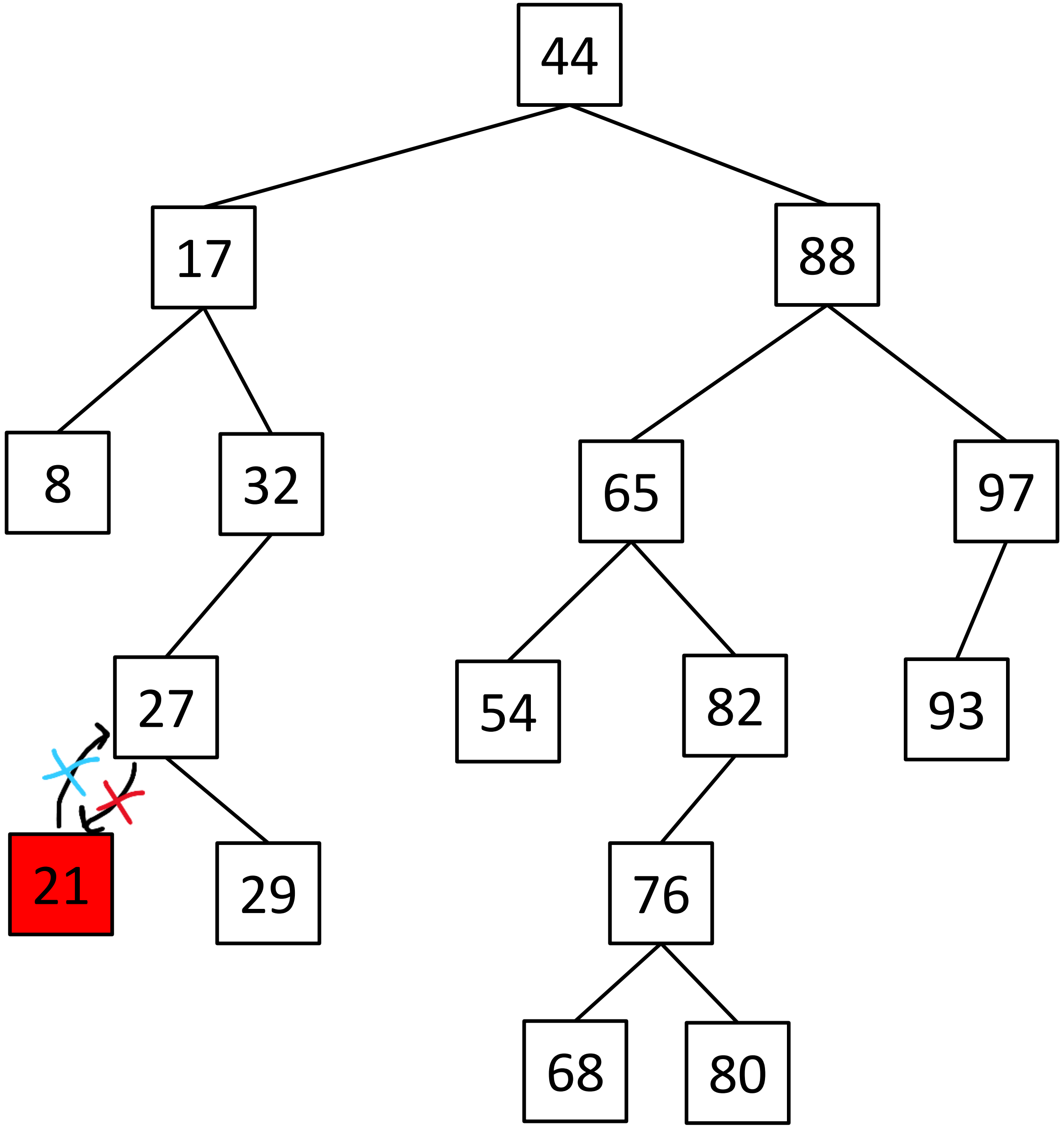


Binary Search Tree- Removal

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`**

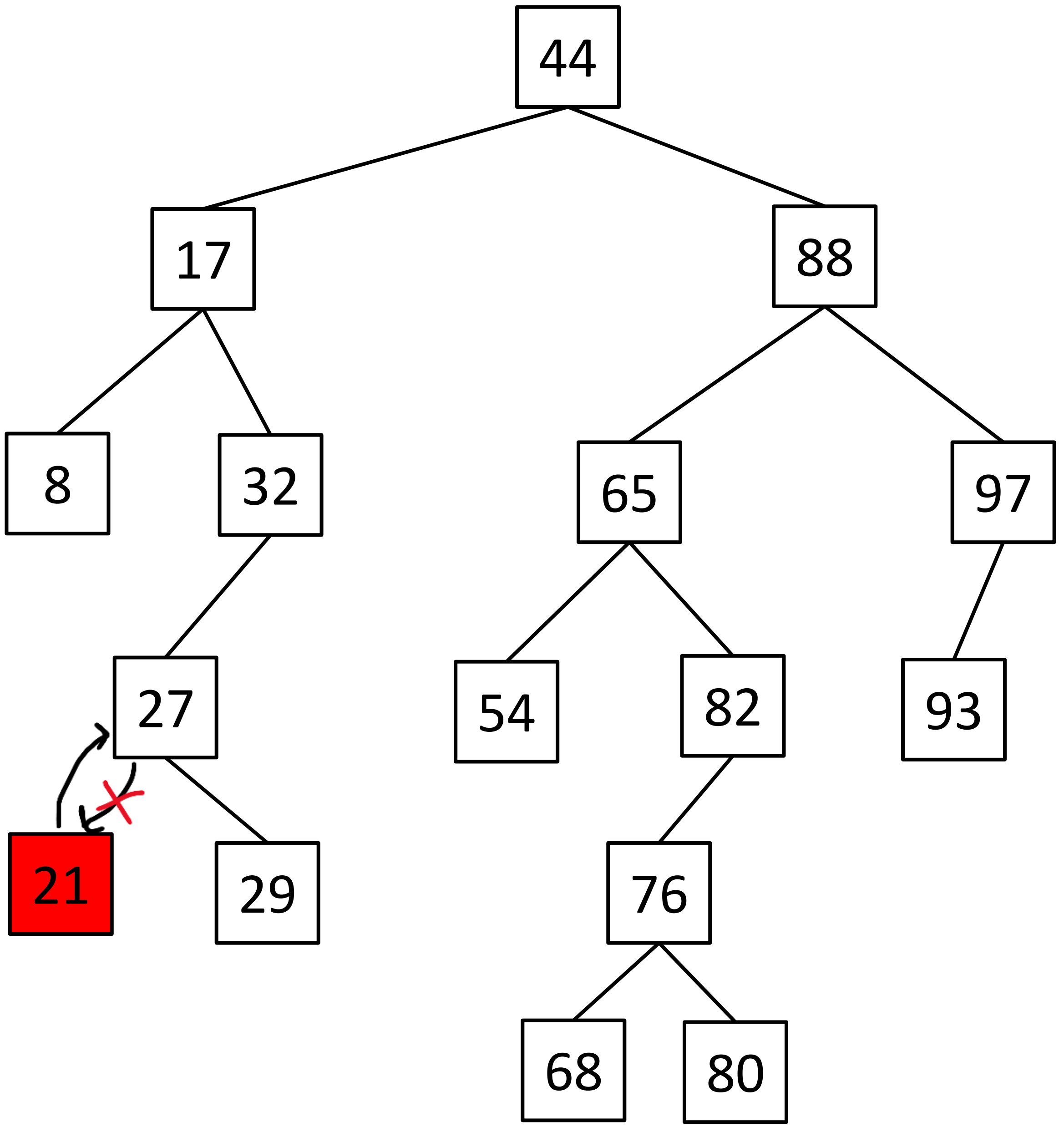


Binary Search Tree- Removal

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`**~~

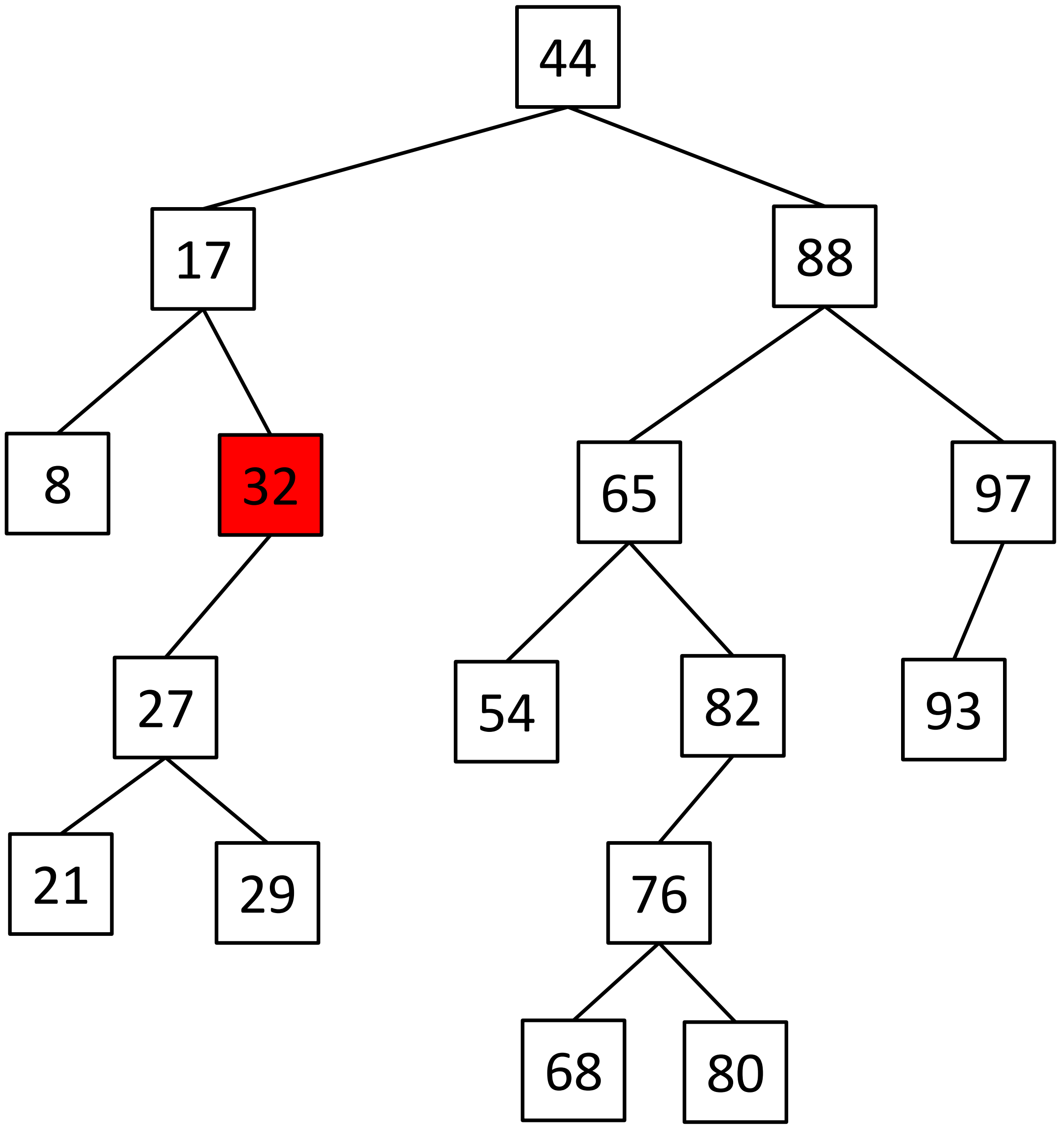


Binary Search Tree- Removal

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

```
remove(32);
```

???



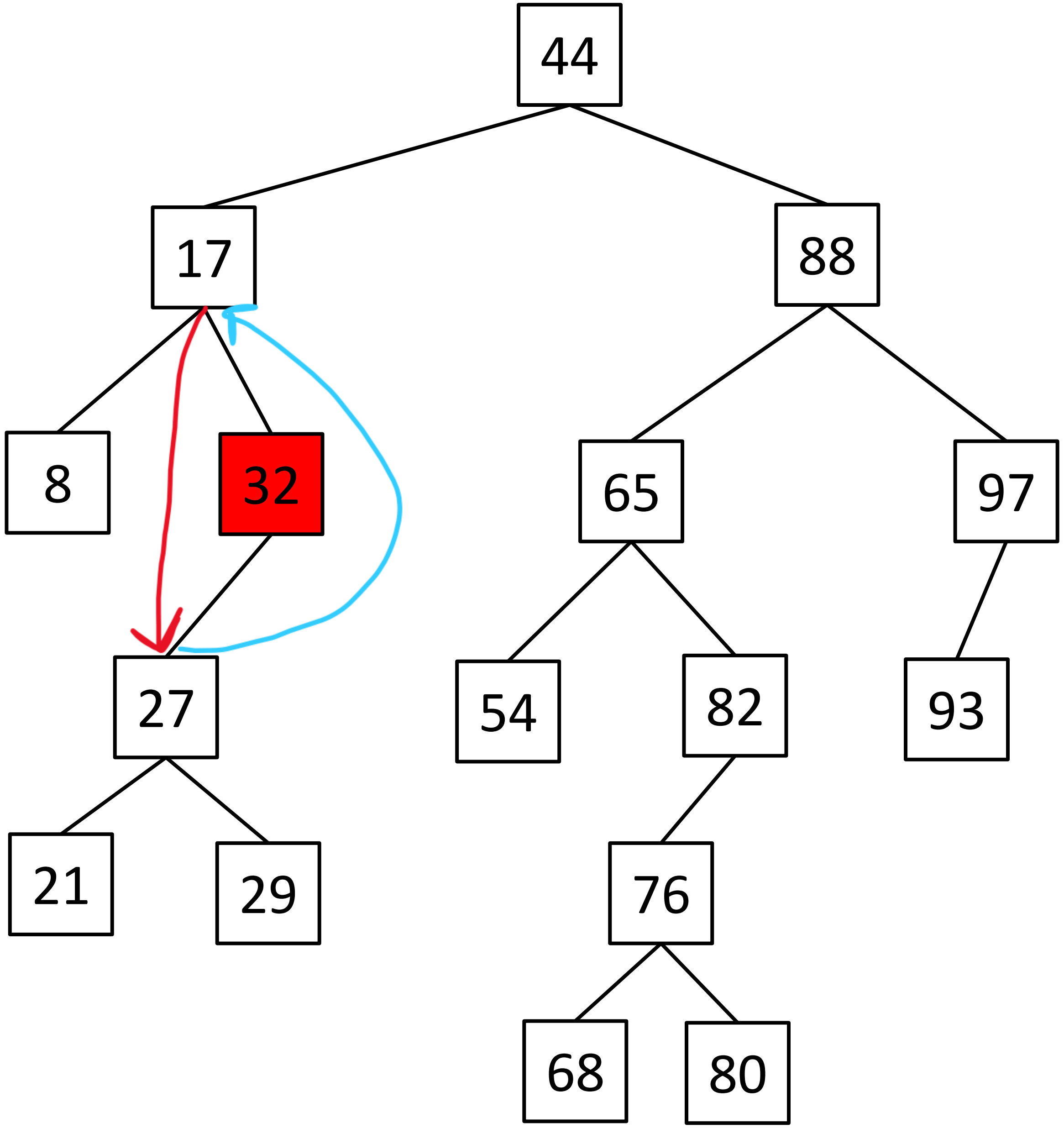
Binary Search Tree- Removal

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

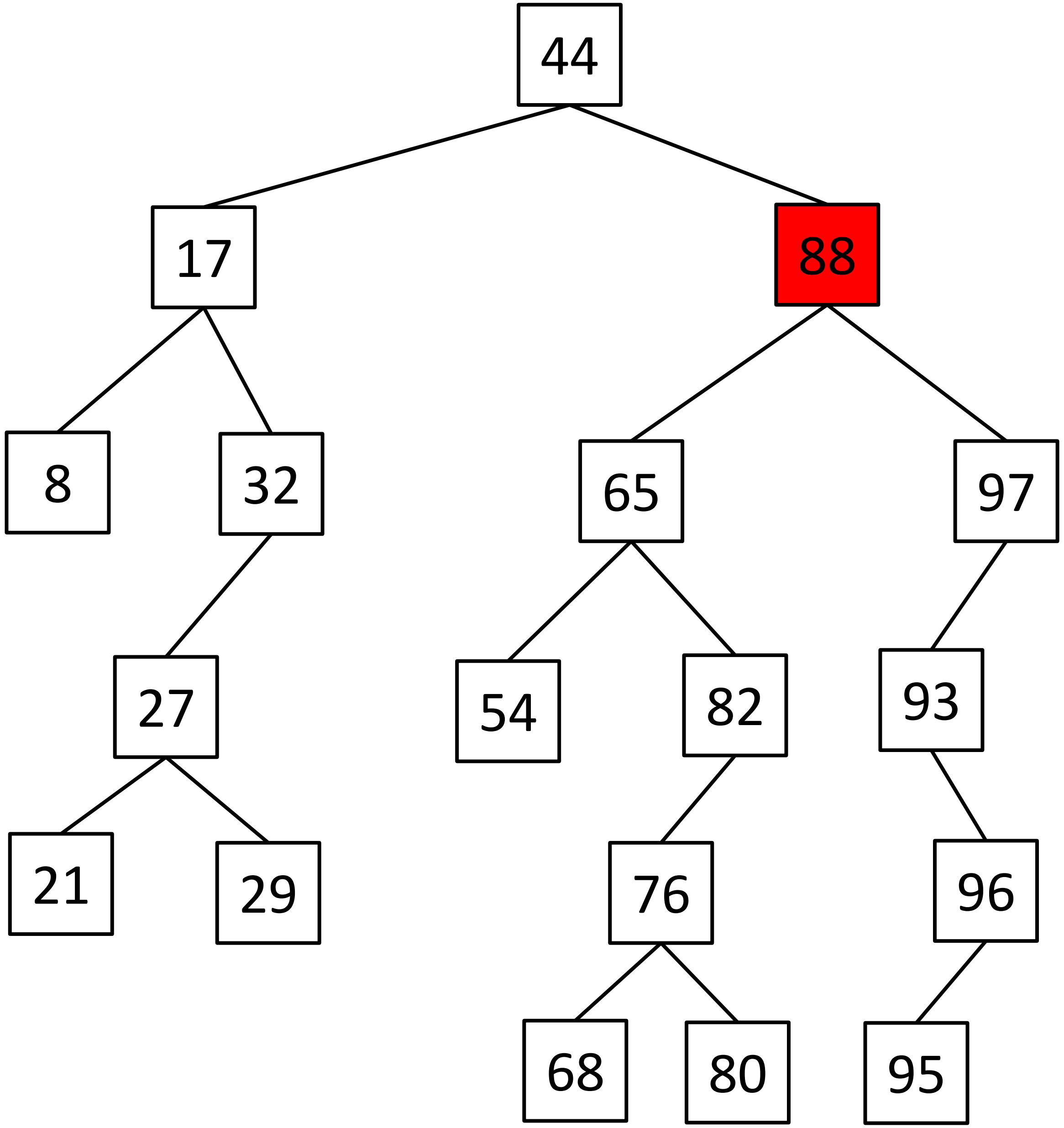
Update the Node's only child `parent` to point to the Node's Parent



Binary Search Tree- Removal

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

```
remove(88);
```

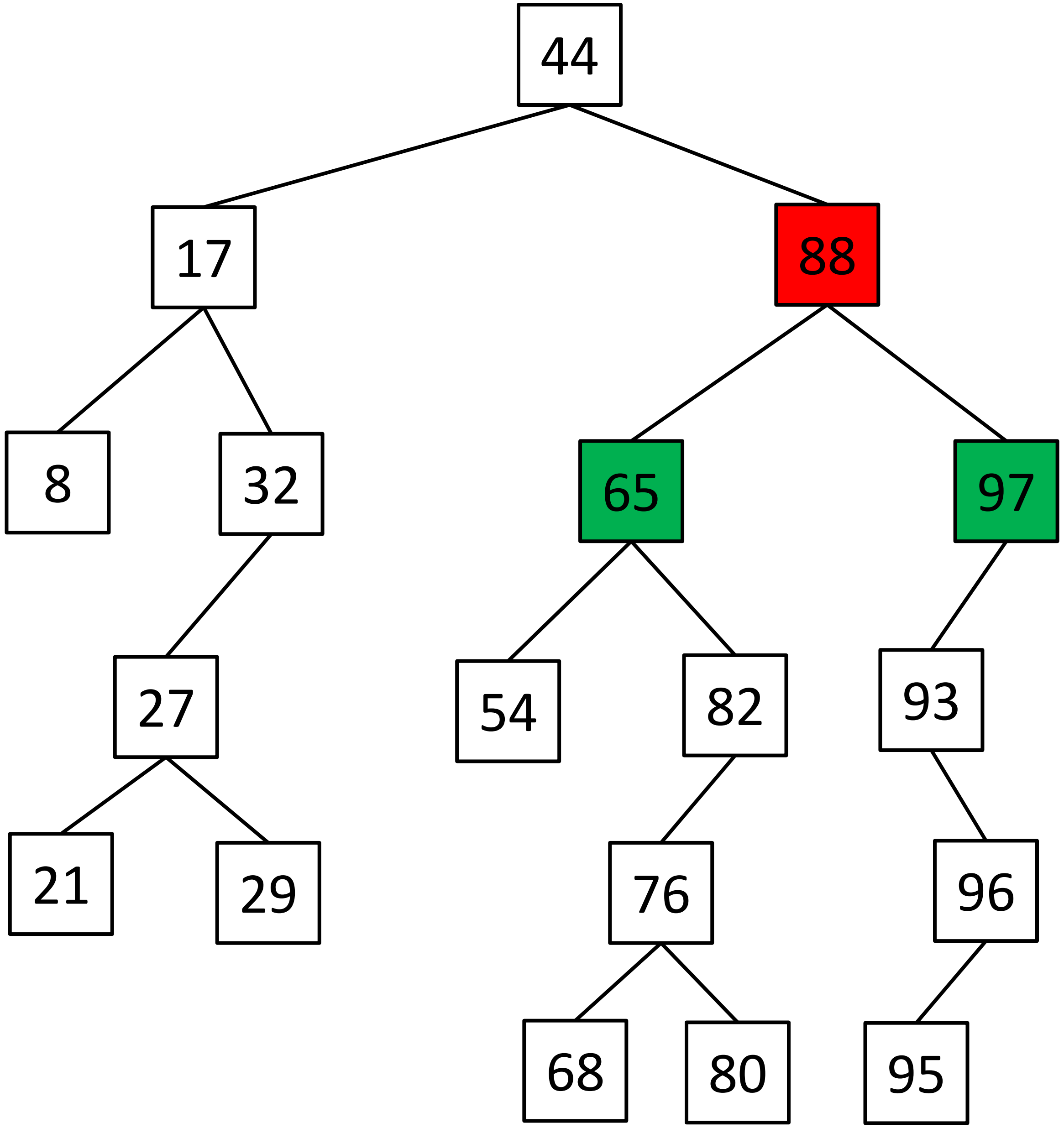


Binary Search Tree- Removal

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

```
remove(88);
```

Which child to use?

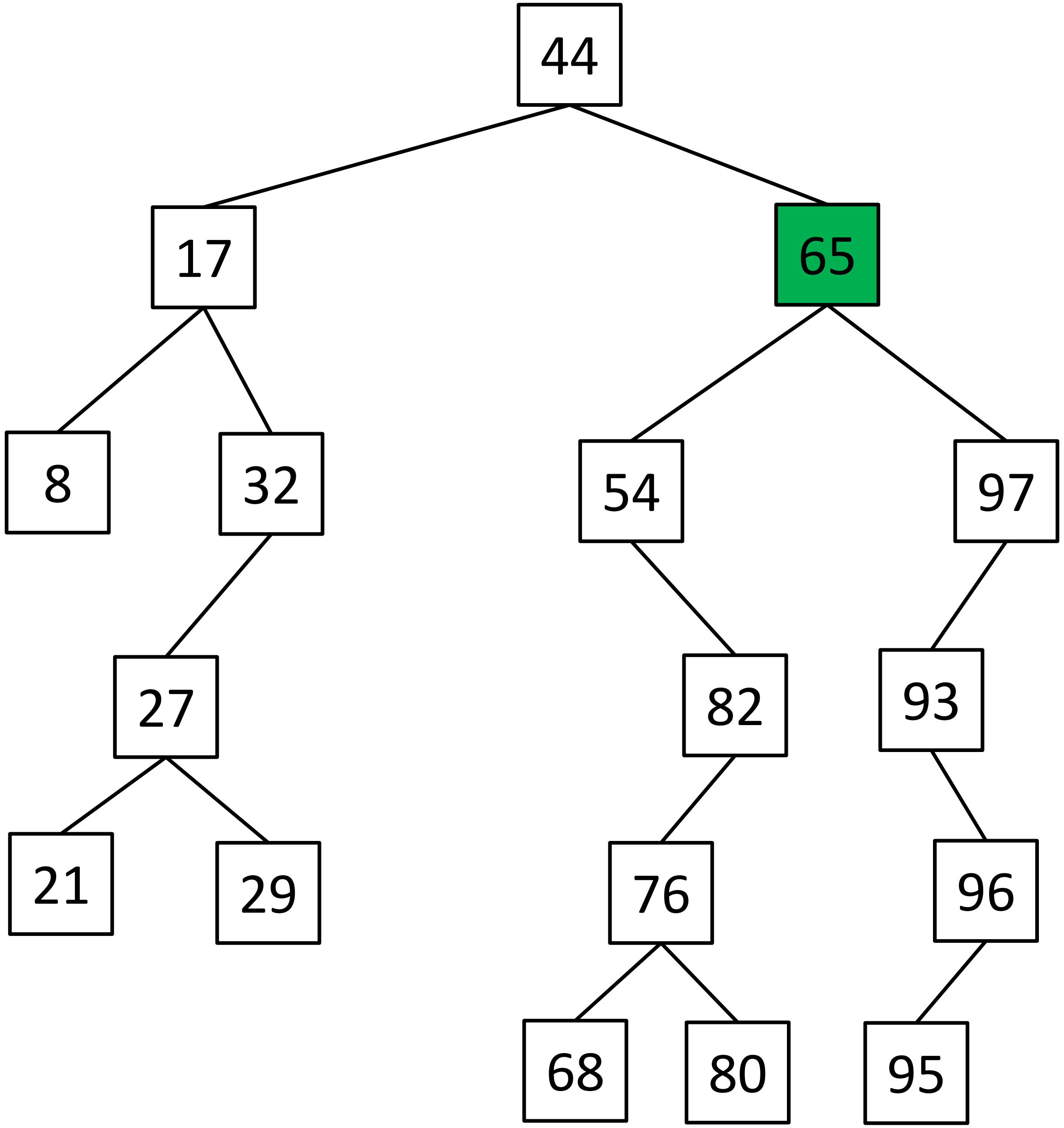


Binary Search Tree- Removal

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

```
remove(88);
```

Which child to use?



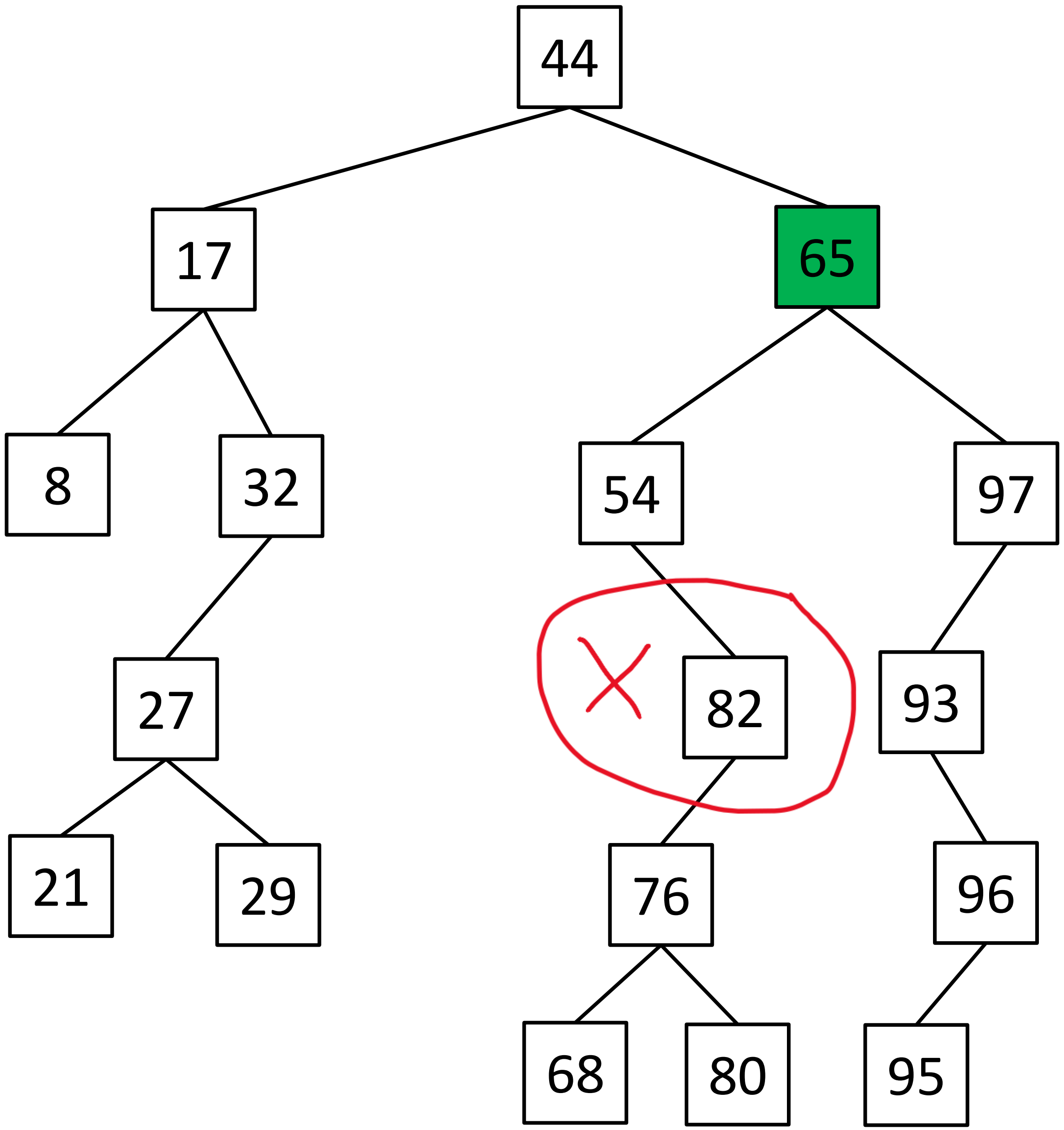
Binary Search Tree- Removal

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

```
remove(88);
```

Which child to use?

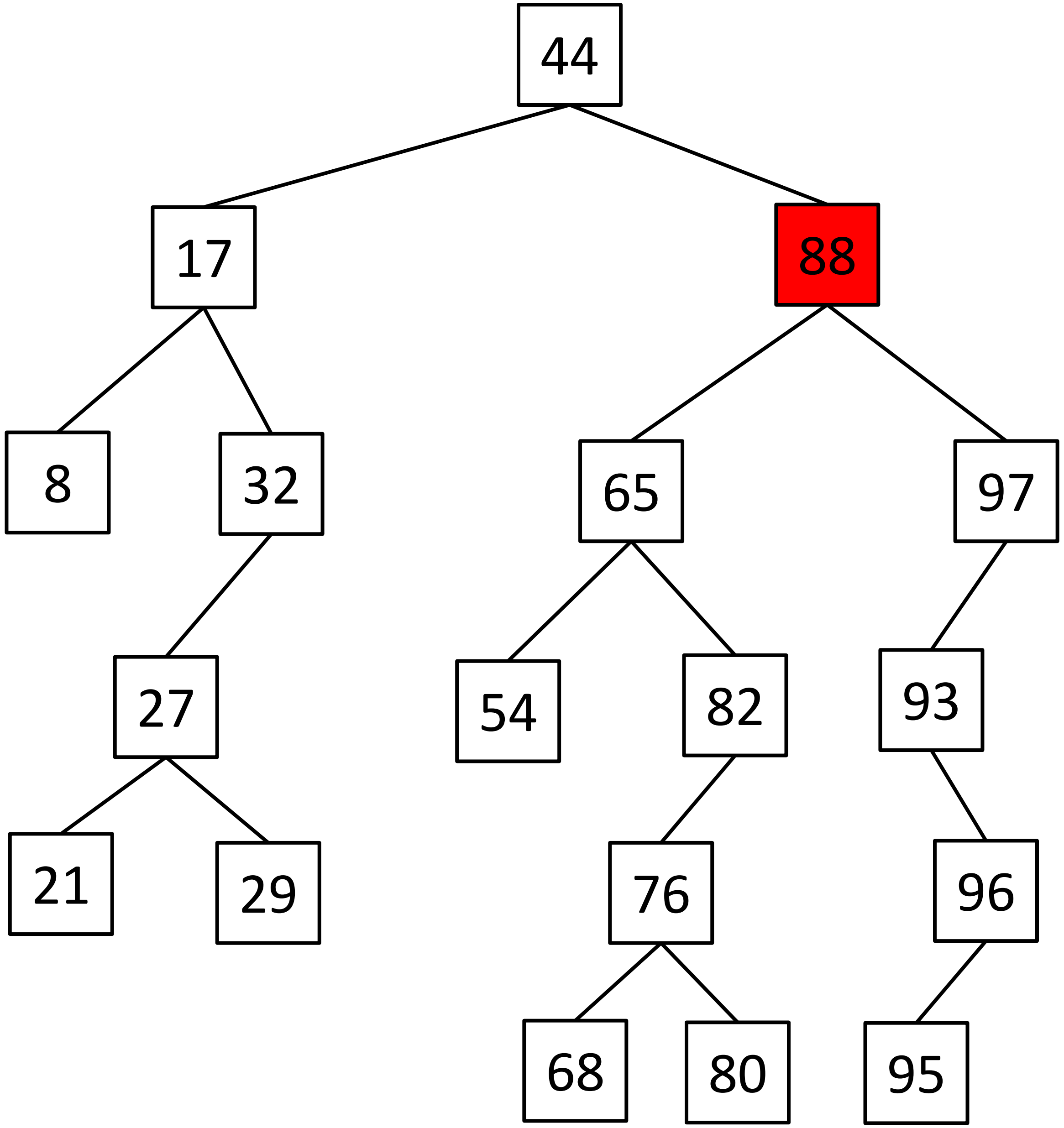
Left child
doesn't work!



Binary Search Tree- Removal

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

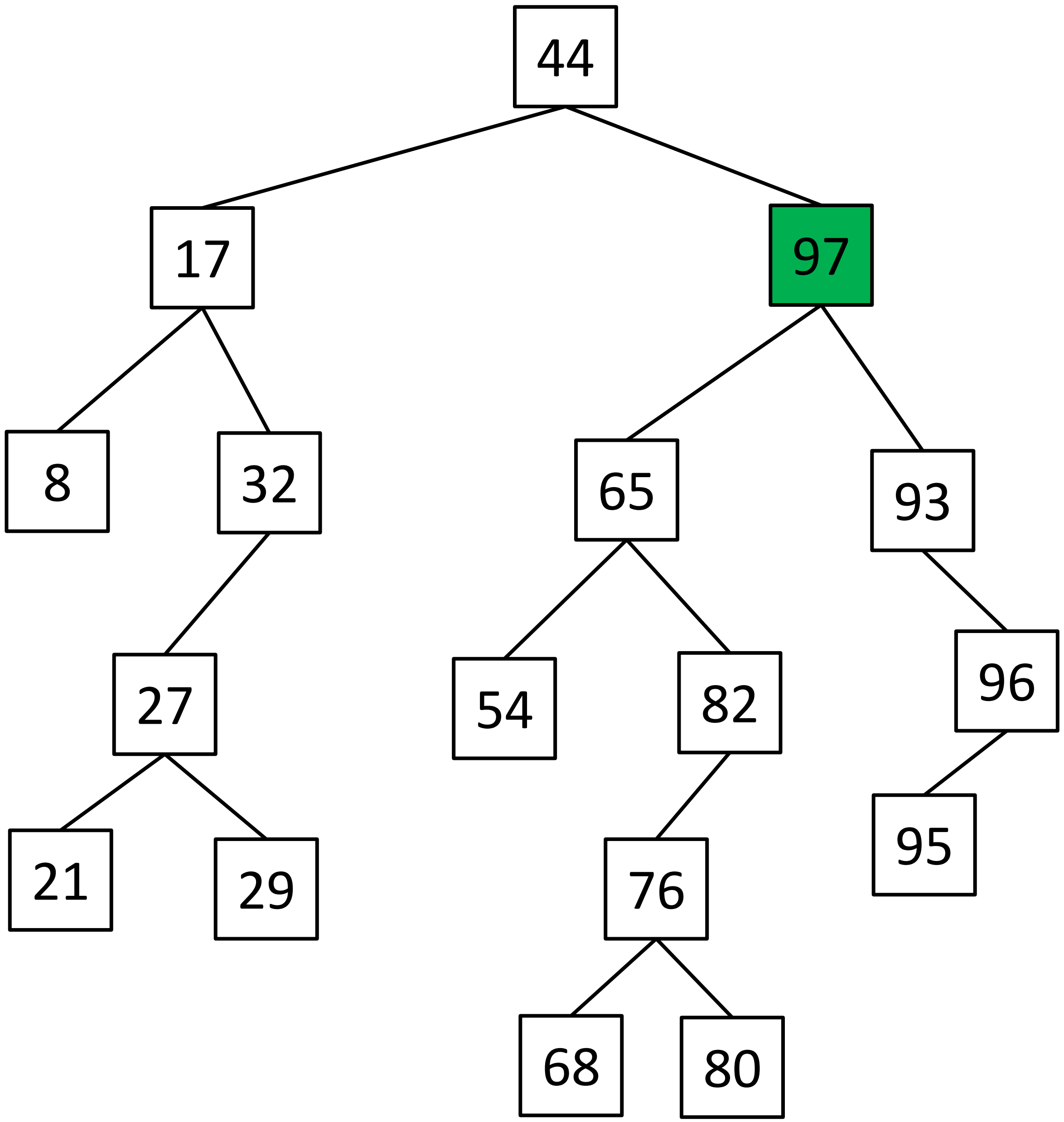
```
remove(88);
```



Binary Search Tree- Removal

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

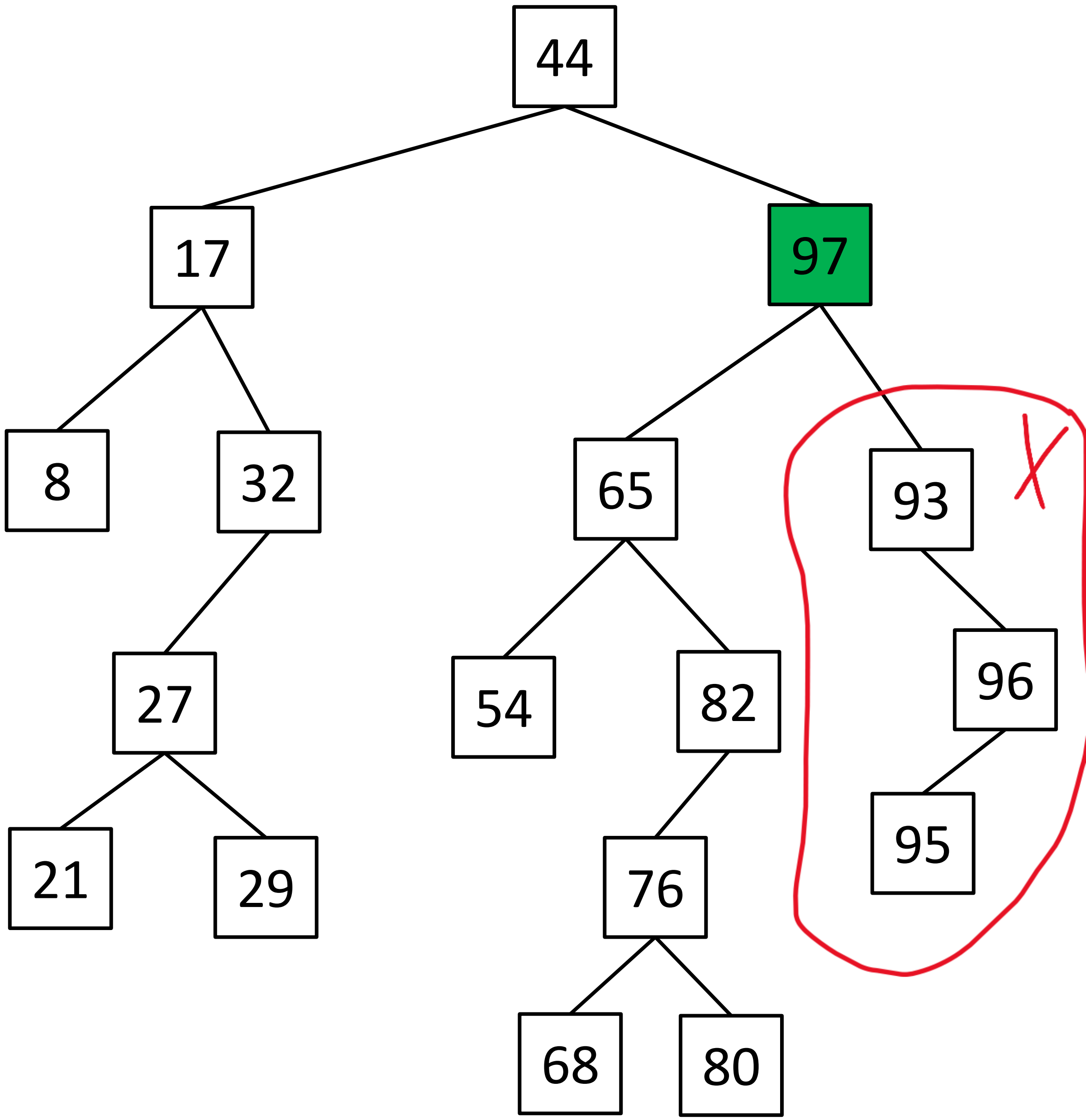
```
remove(88);
```



Binary Search Tree- Removal

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

```
remove(88);
```



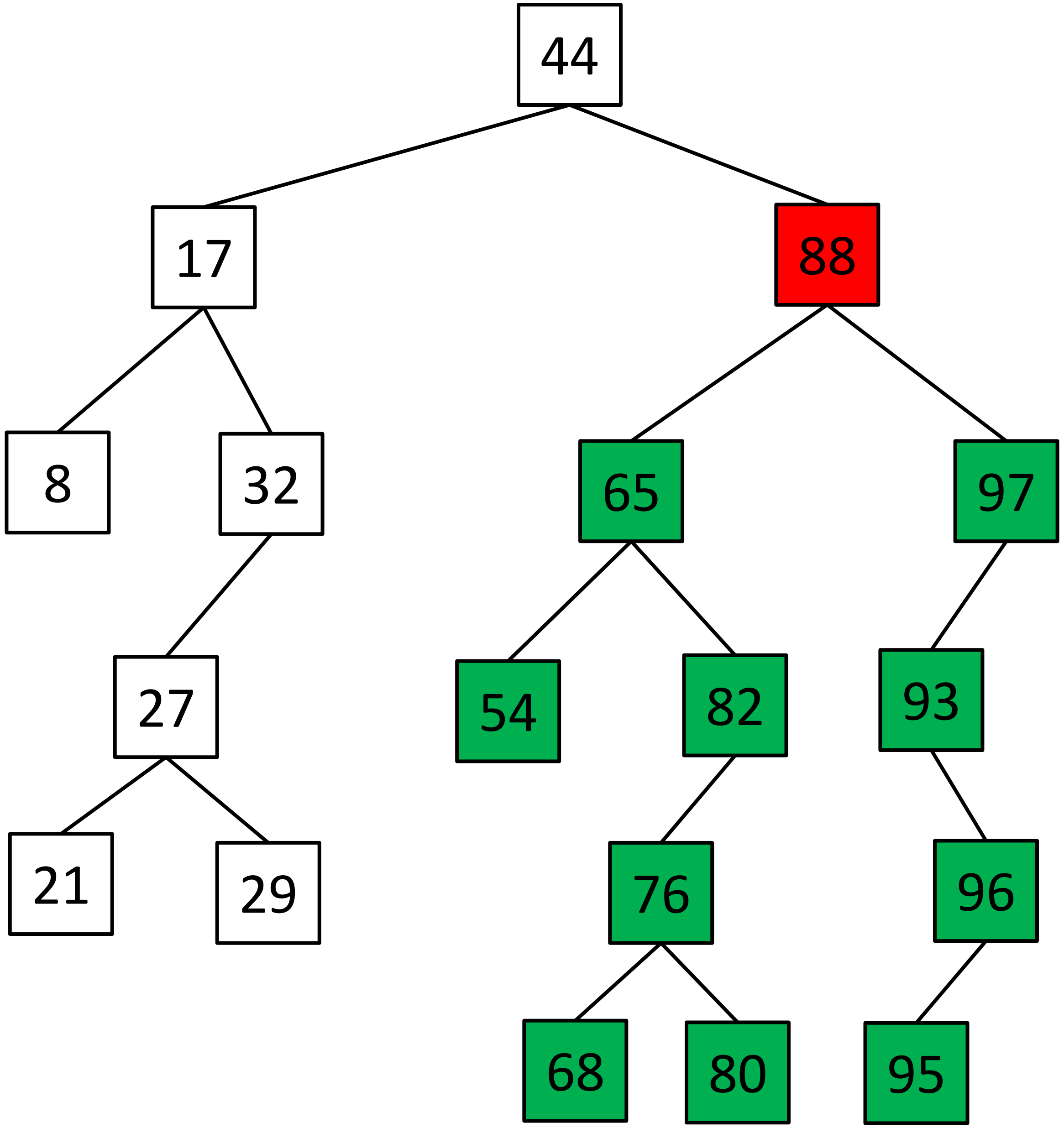
Right child doesn't work!

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?

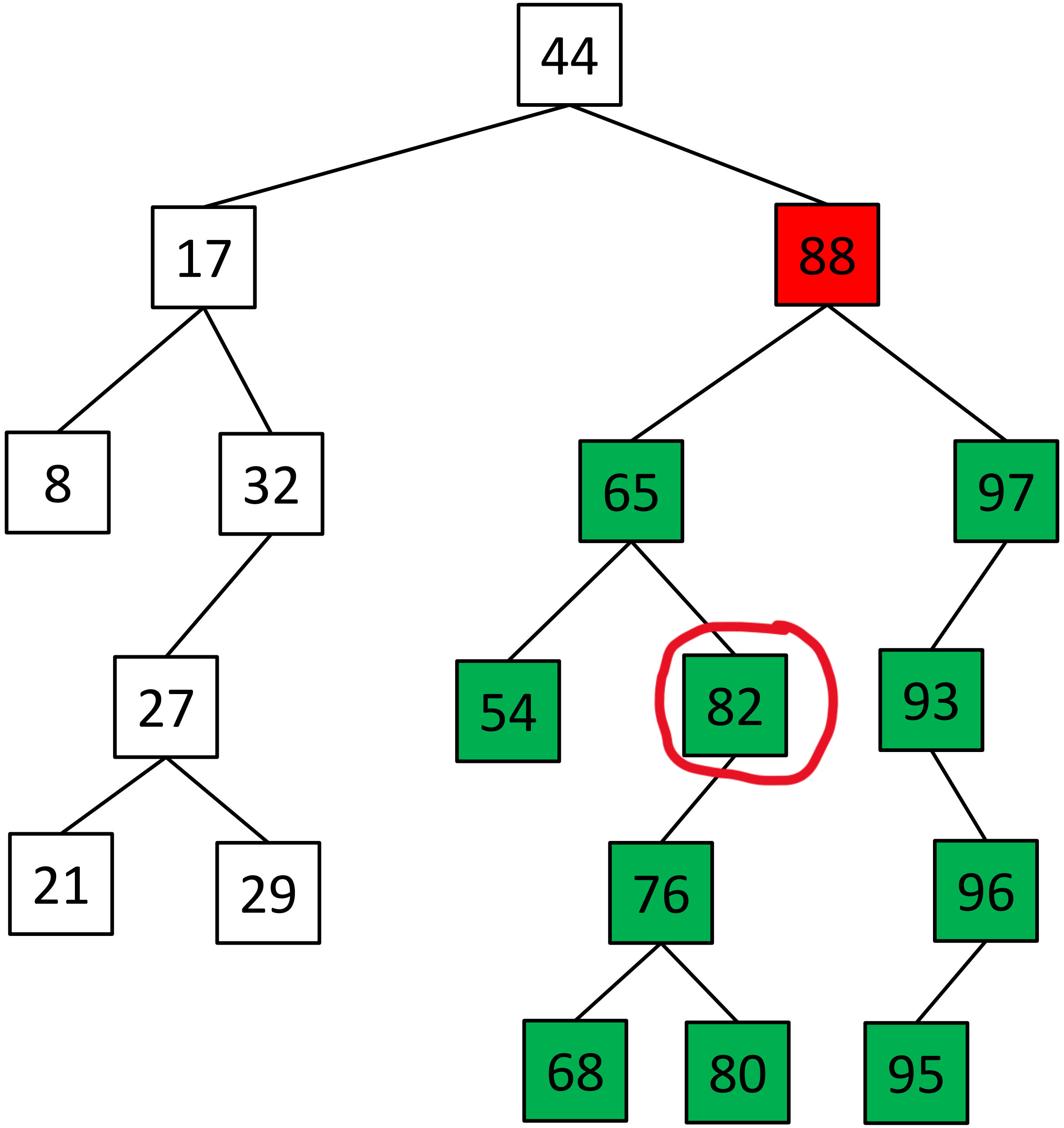


Binary Search Tree- Removal

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?

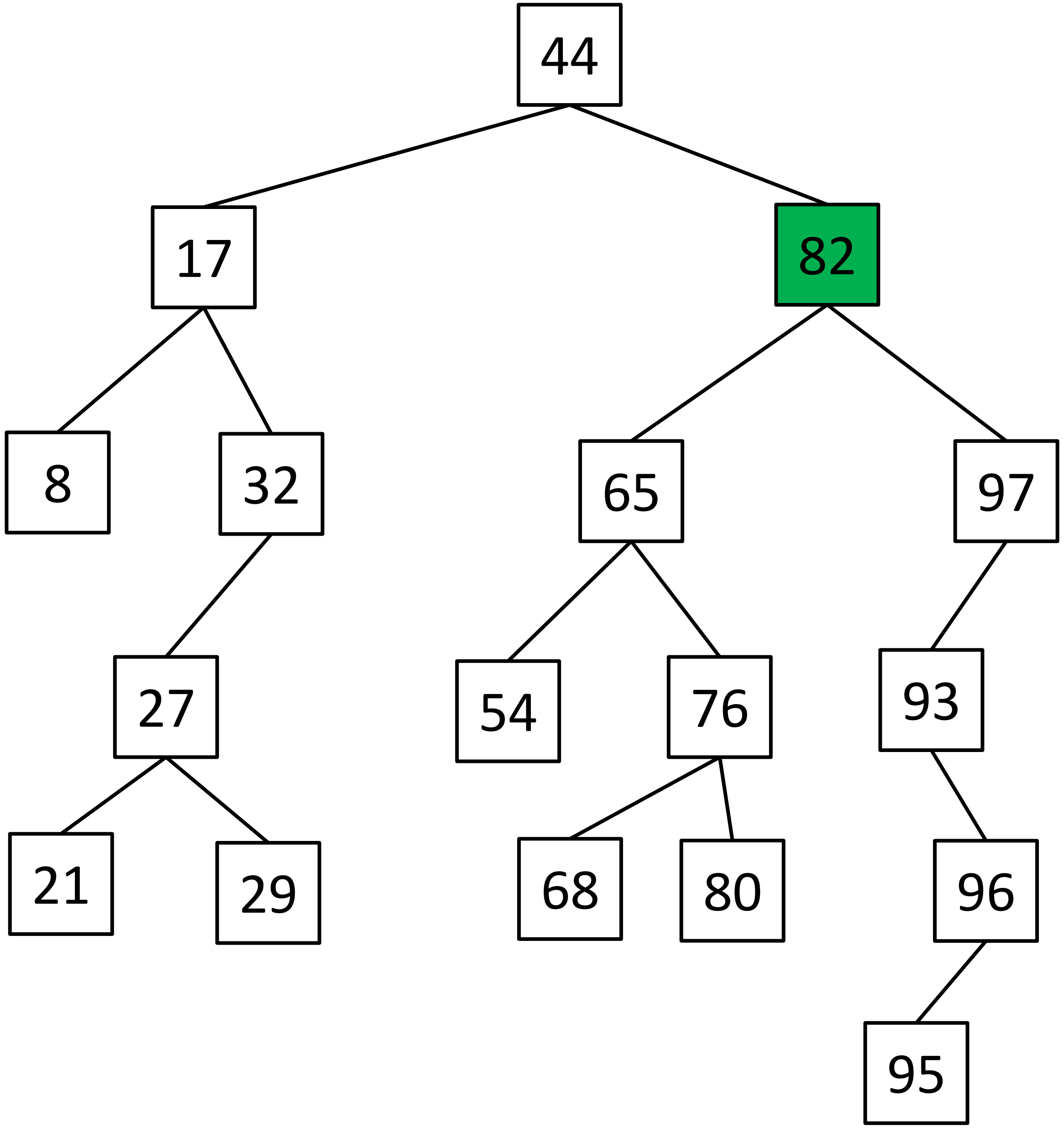


Binary Search Tree- Removal

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?

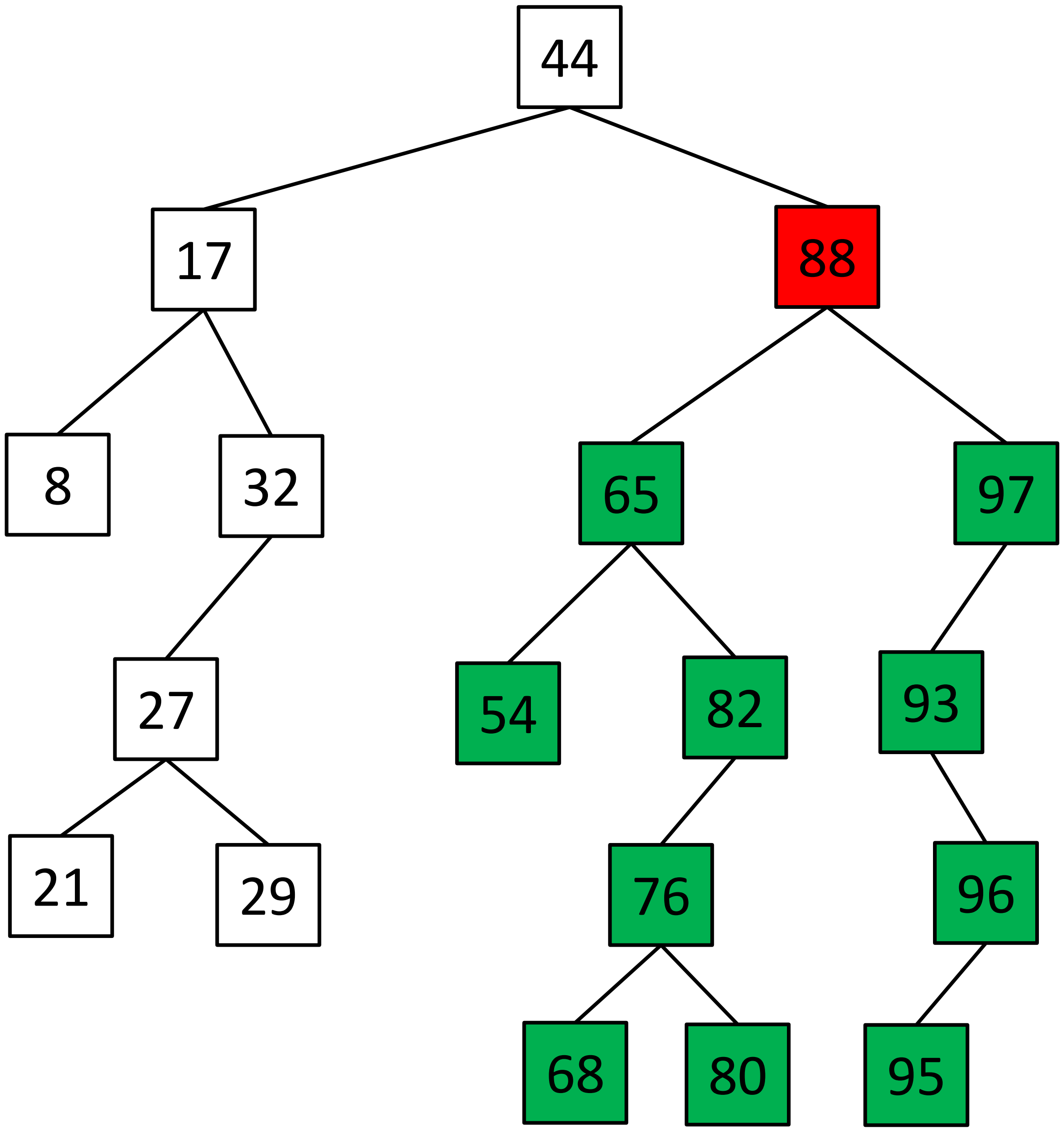


Binary Search Tree- Removal

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?

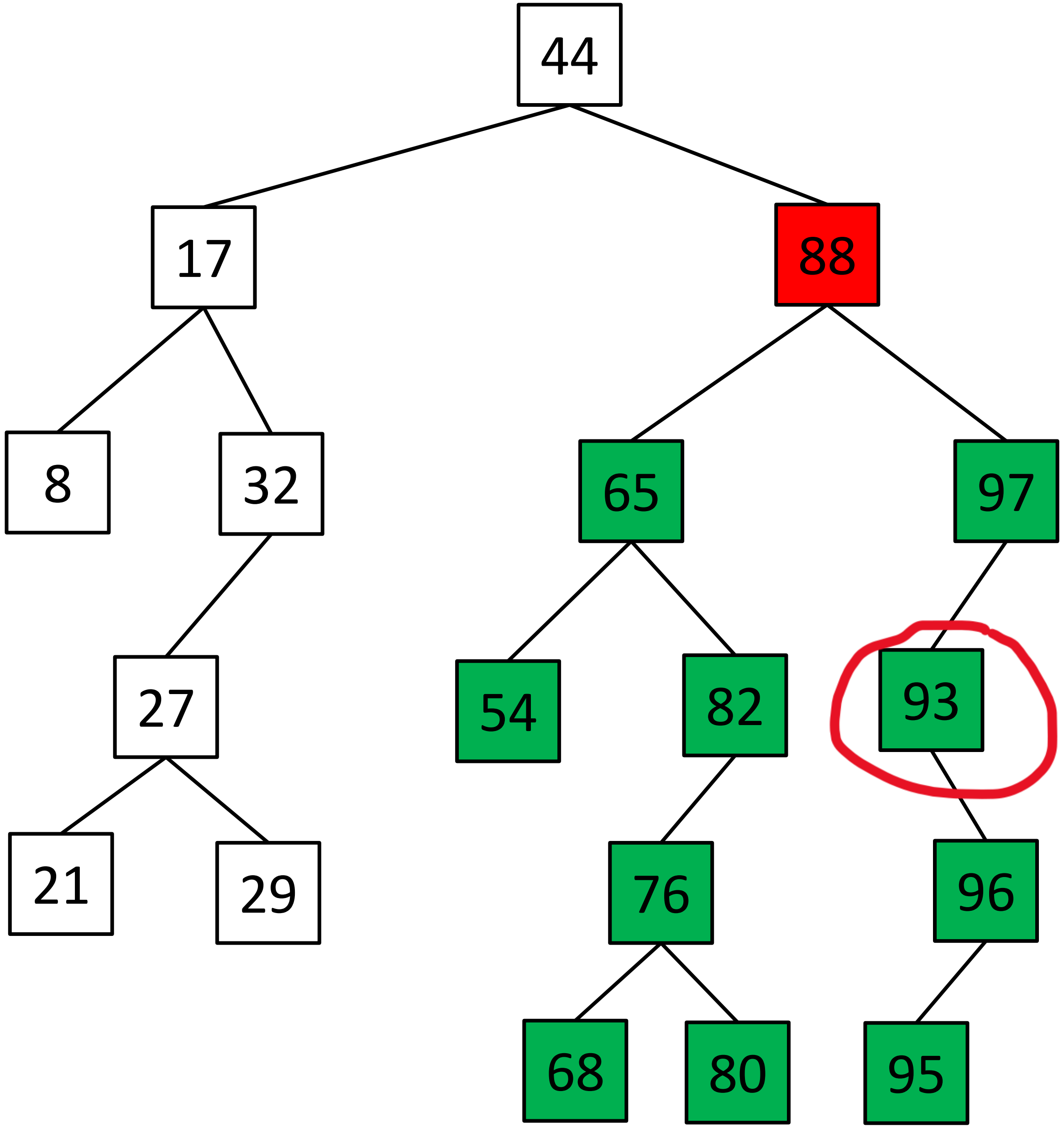


Binary Search Tree- Removal

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?

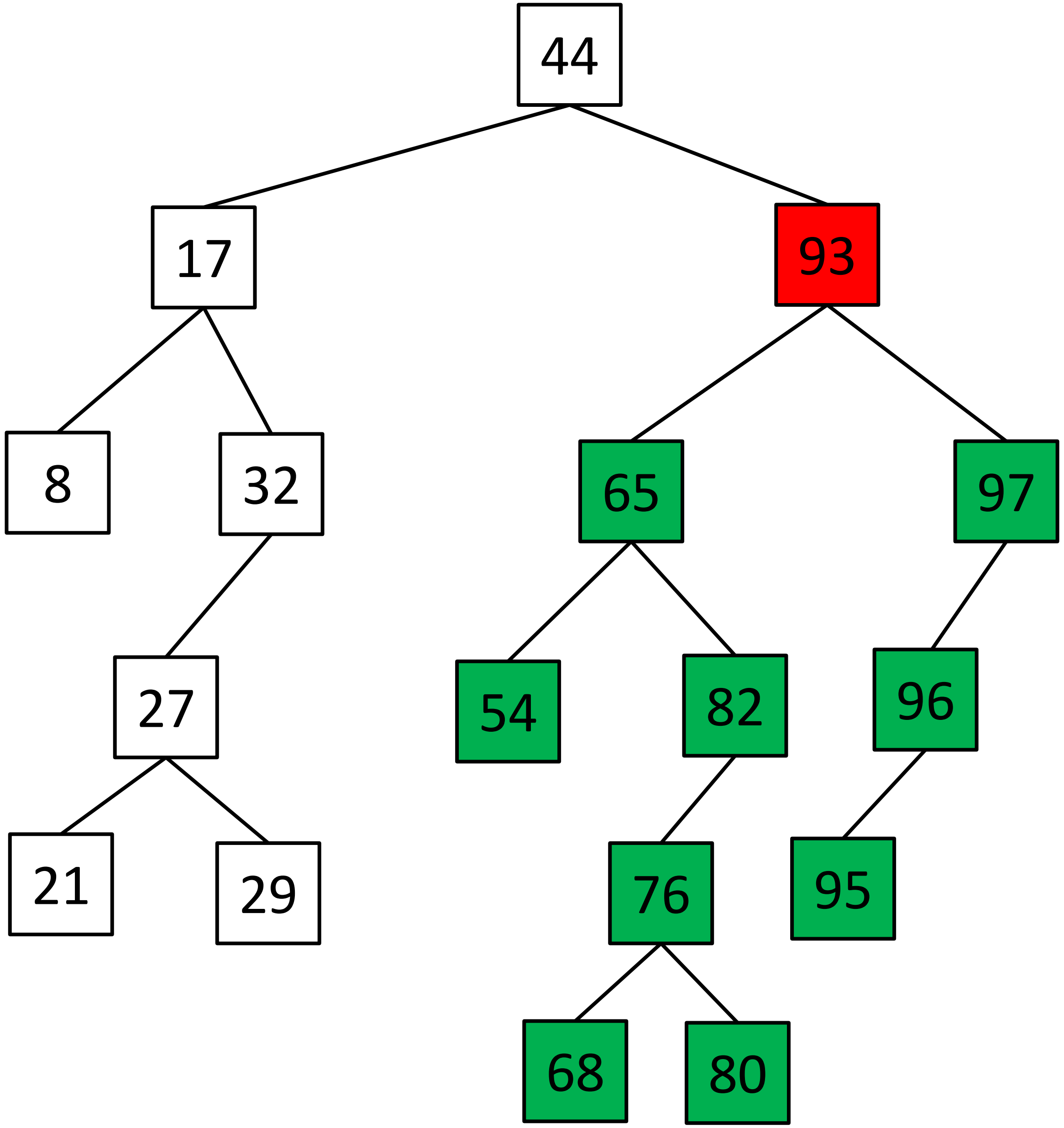


Binary Search Tree- Removal

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

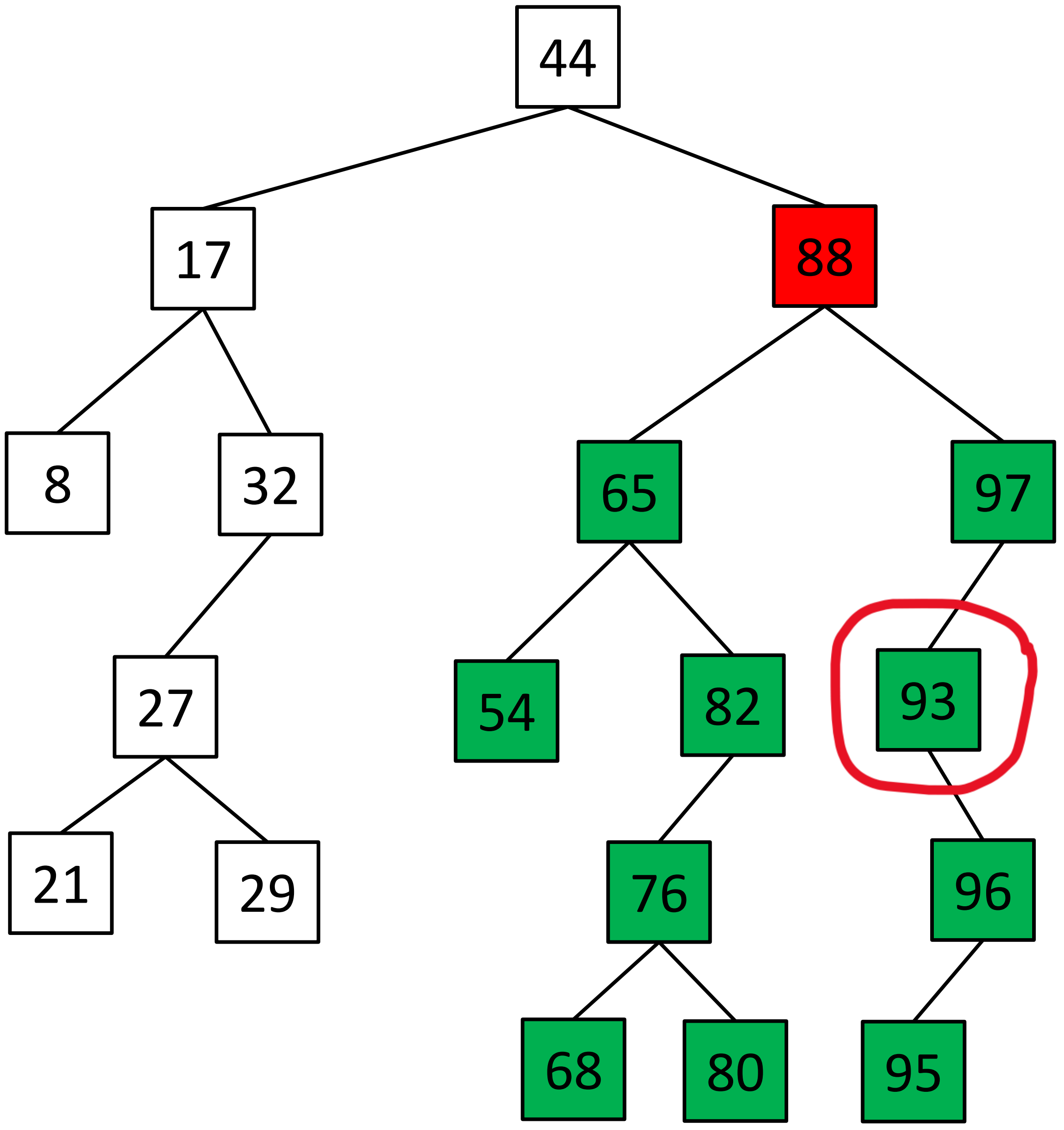
Case 1: Node has no children
Case 2: Node has one child
Case 3: Node has two children

```
remove(88);
```

Which ~~child~~ **descendant** to use?

The lowest value in the right subtree

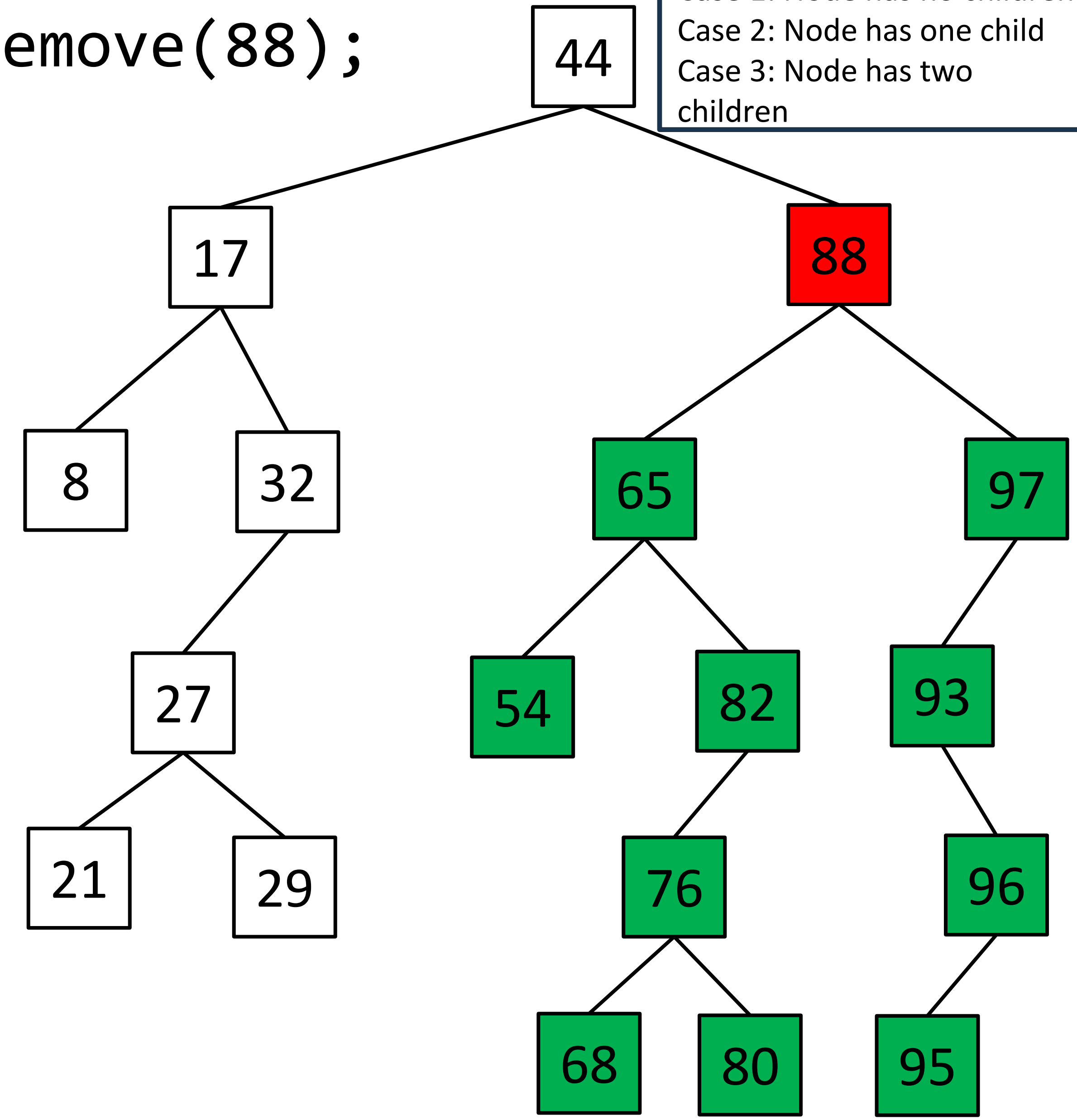
or the highest value in the left subtree



Binary Search Tree- Removal

```
remove(88);
```

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

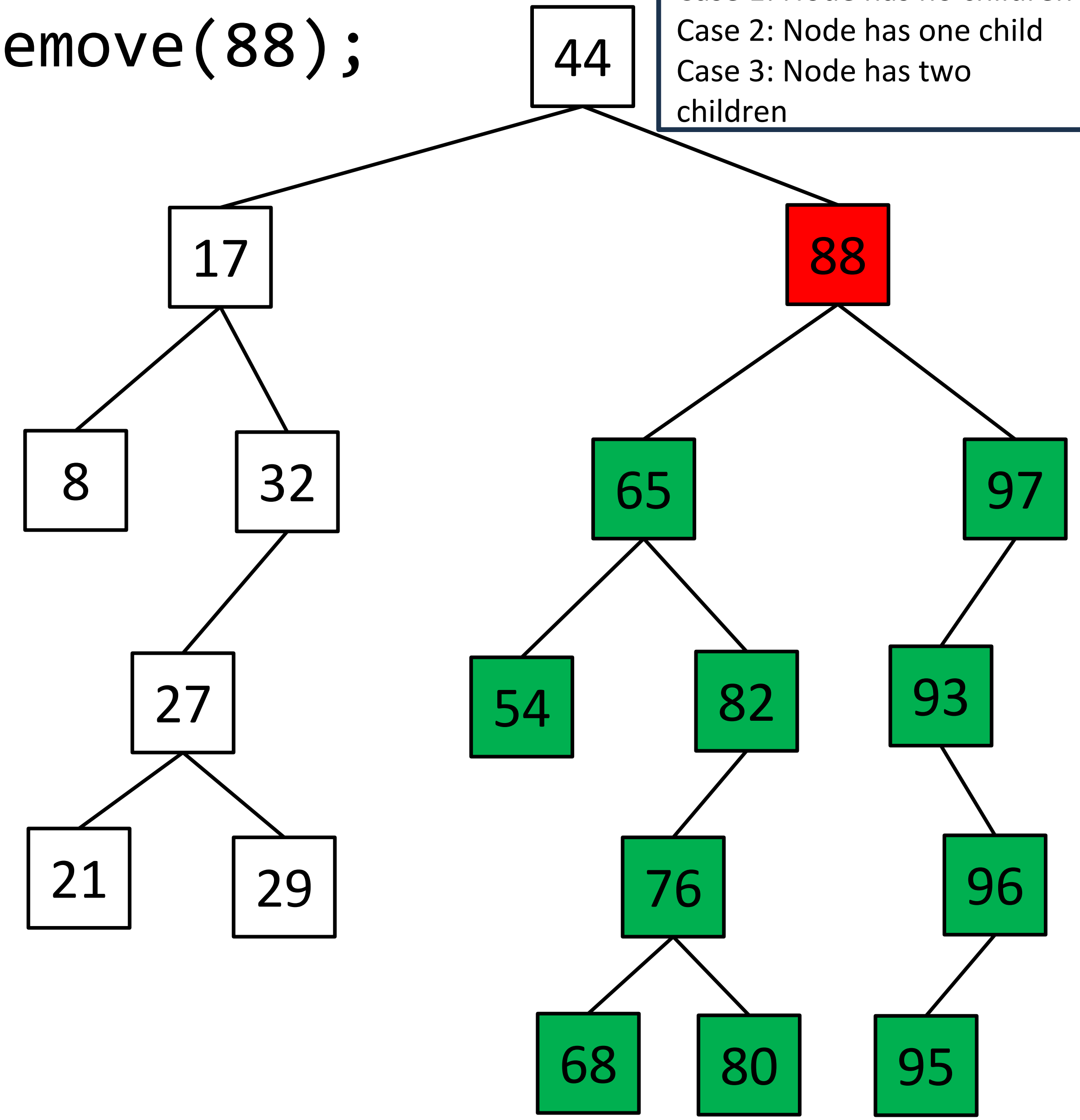


Binary Search Tree- Removal

`remove(88);`

We will solve this *recursively*

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

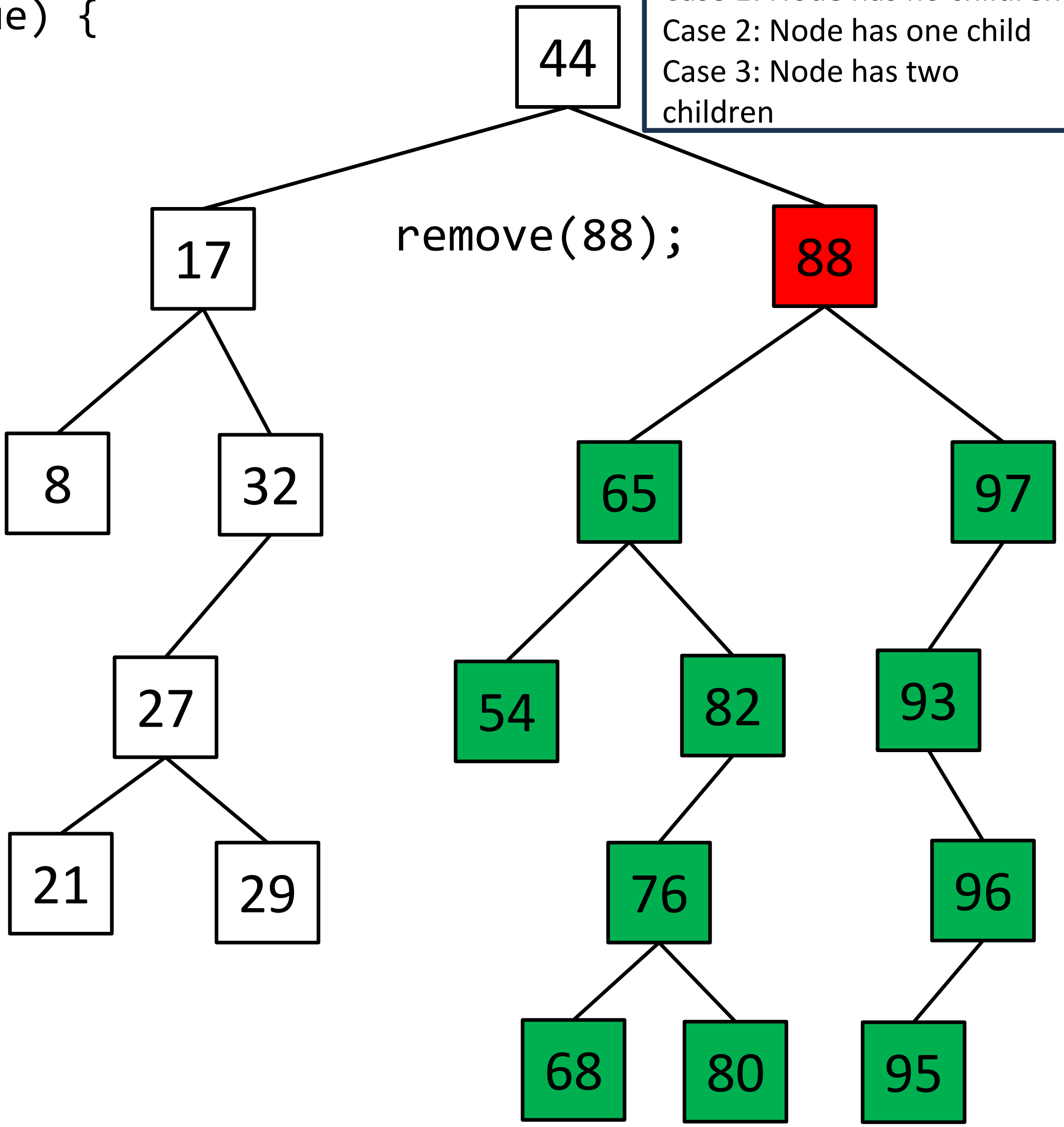


```
public Node deleteNode(Node current, int searchValue) {
```

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

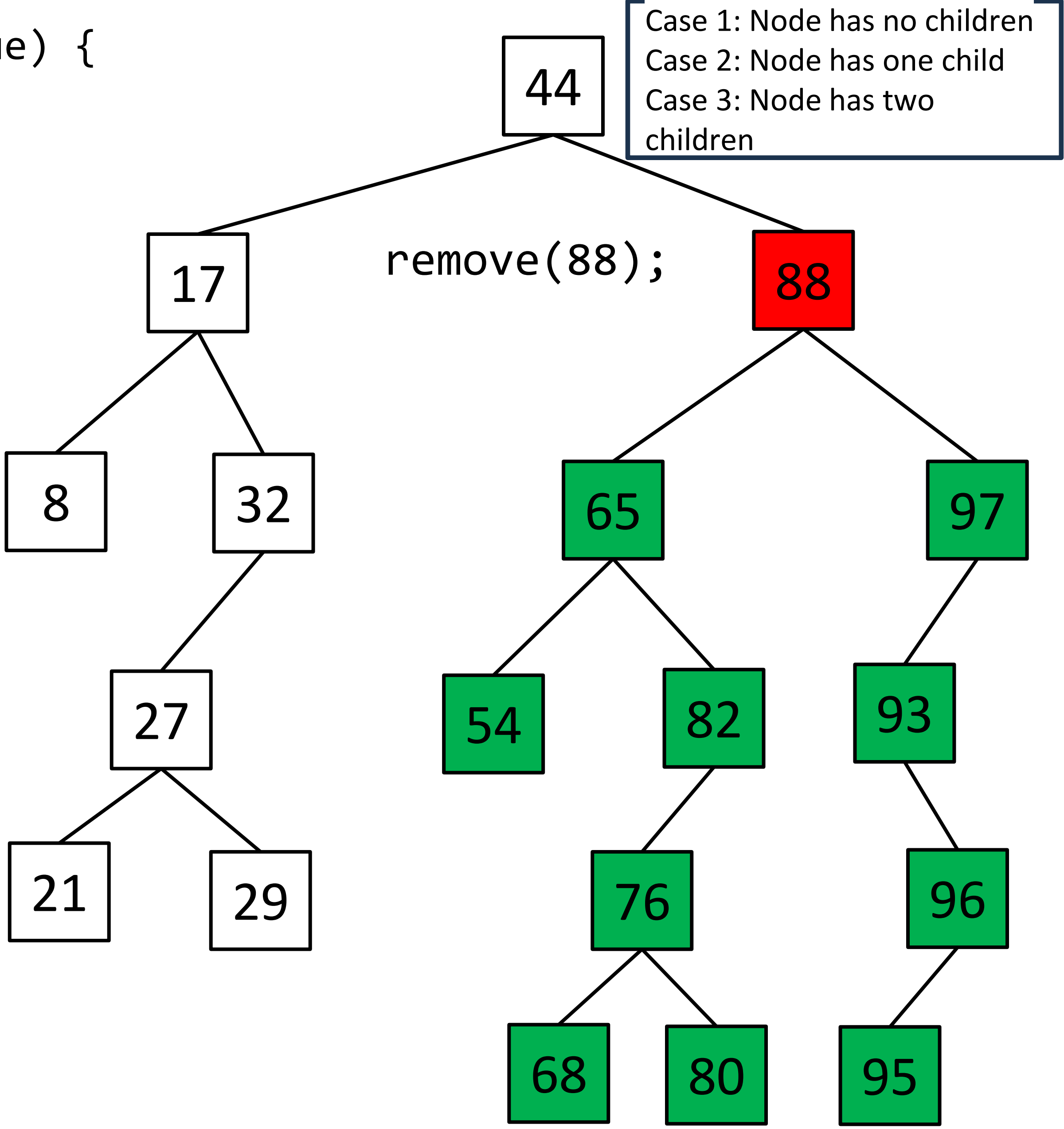
We are going to recursively work our way down the tree, and remove the searchValue, and then return the new *root* of the subtree

(if it got modified)



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

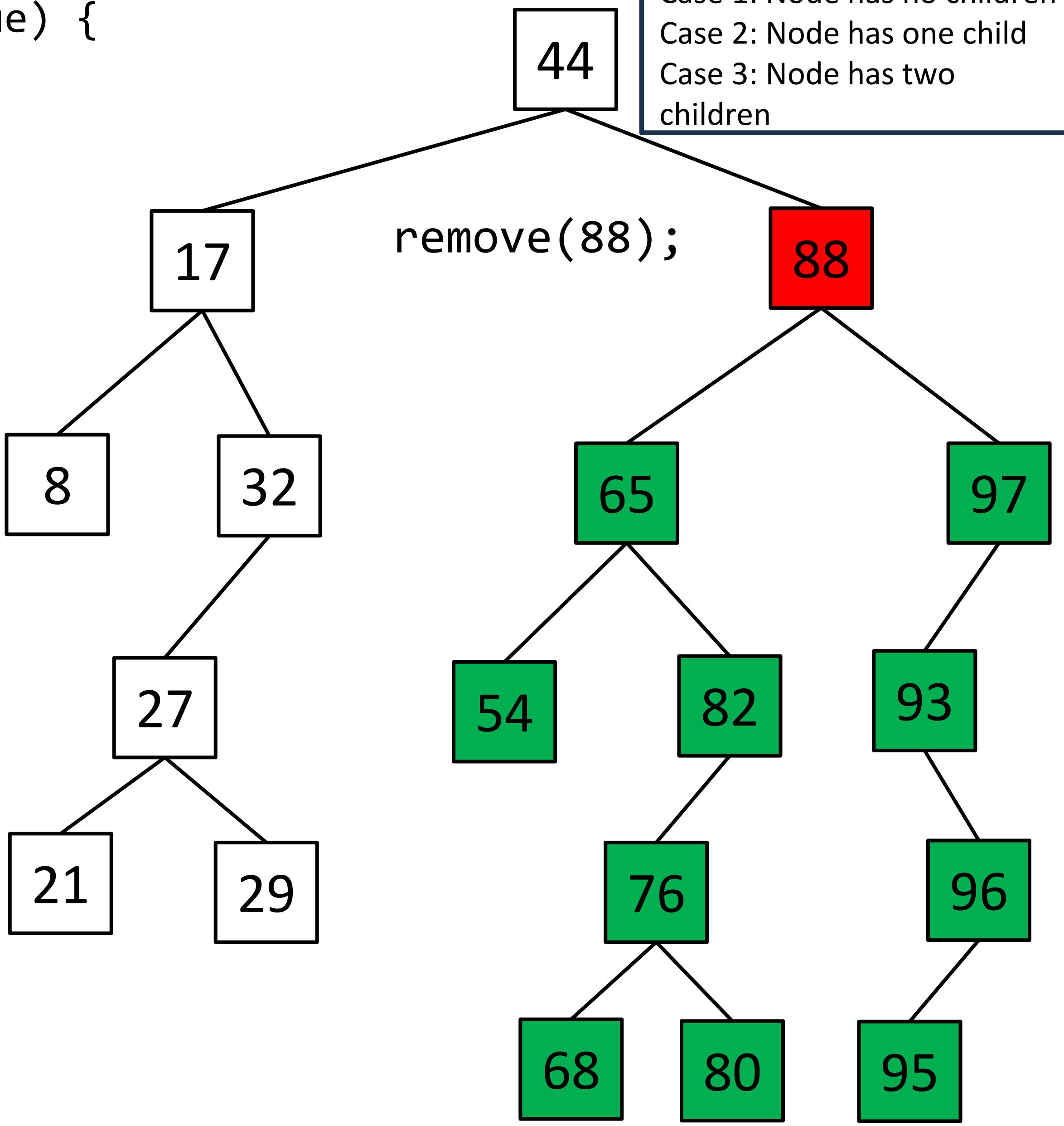
If we ever hit a null value, return



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

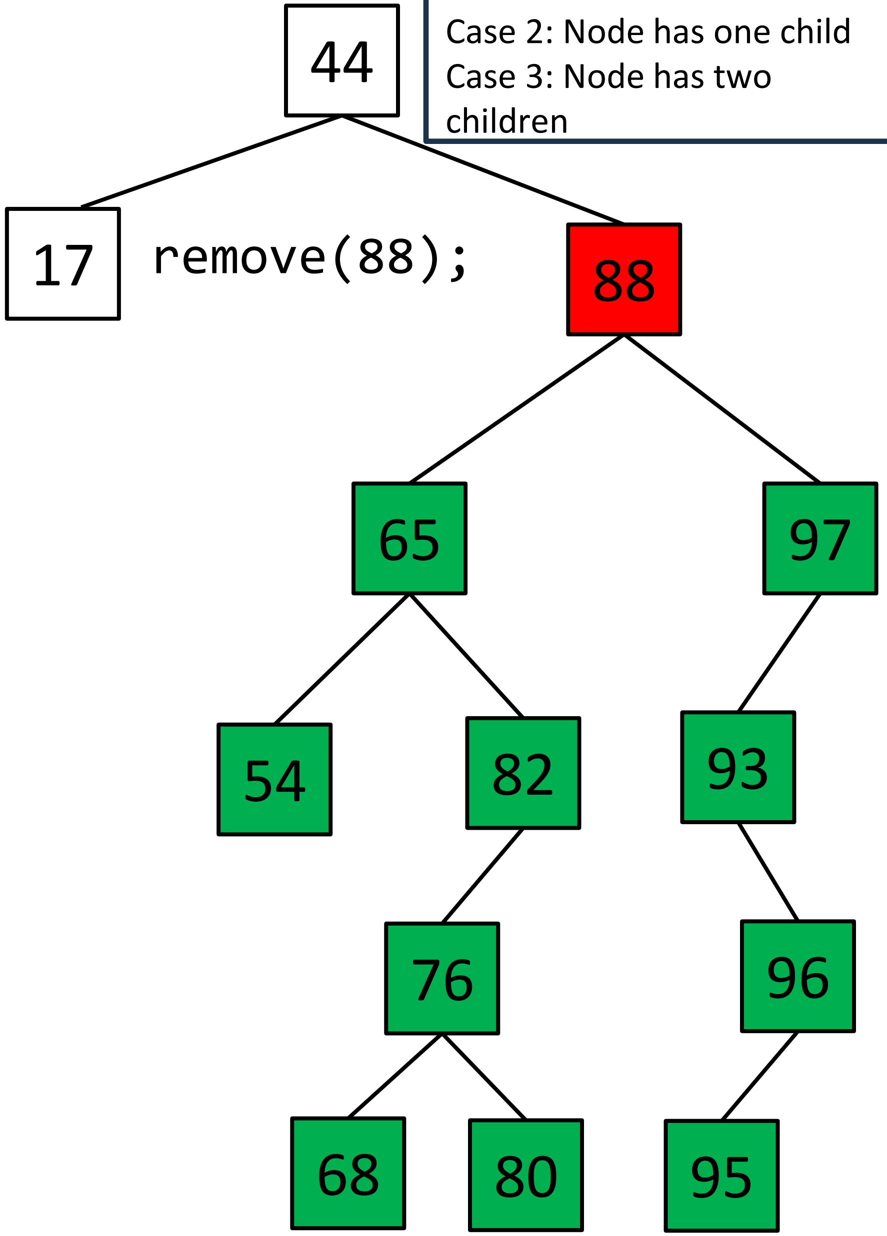
Step 1. Search

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

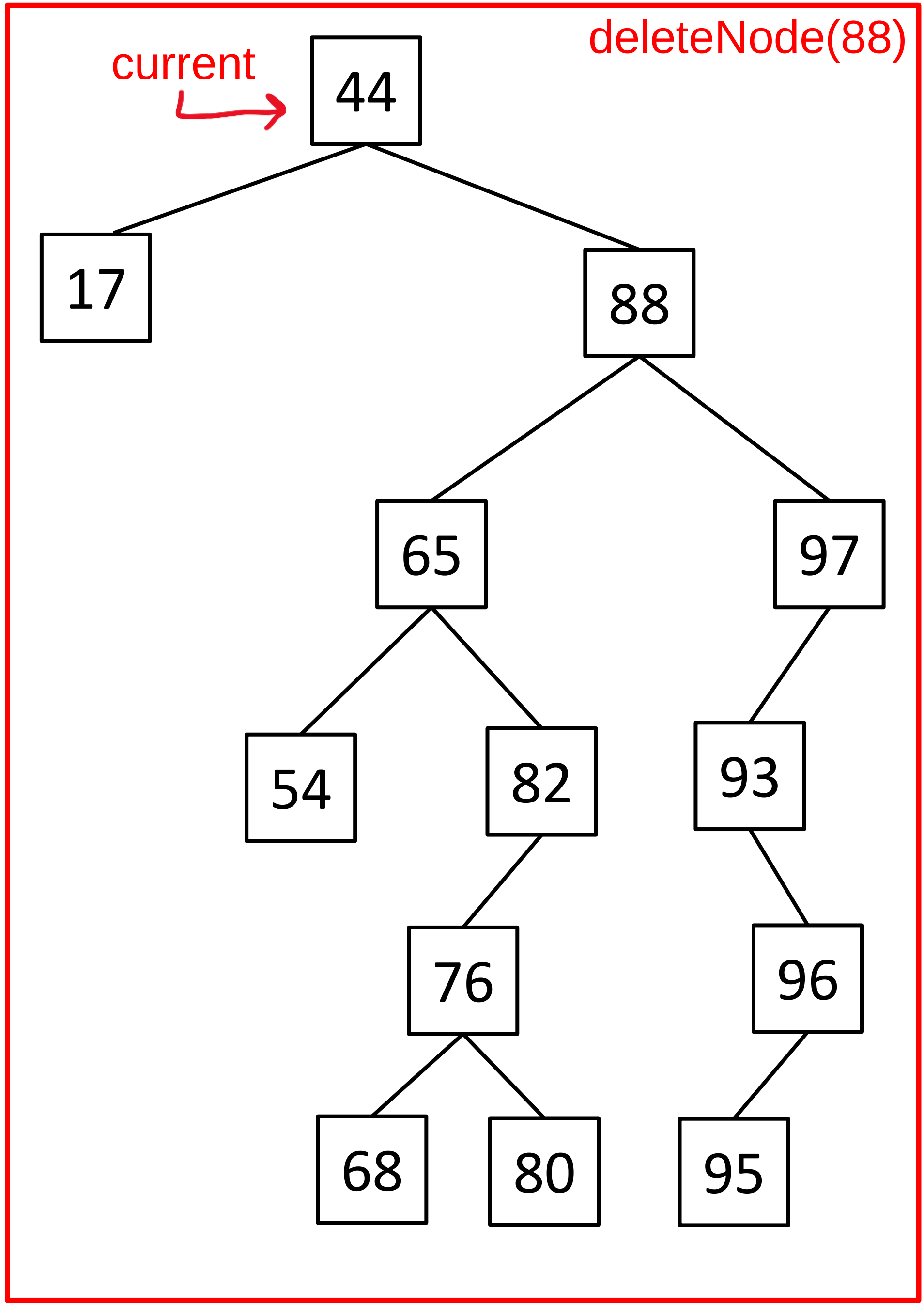


```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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

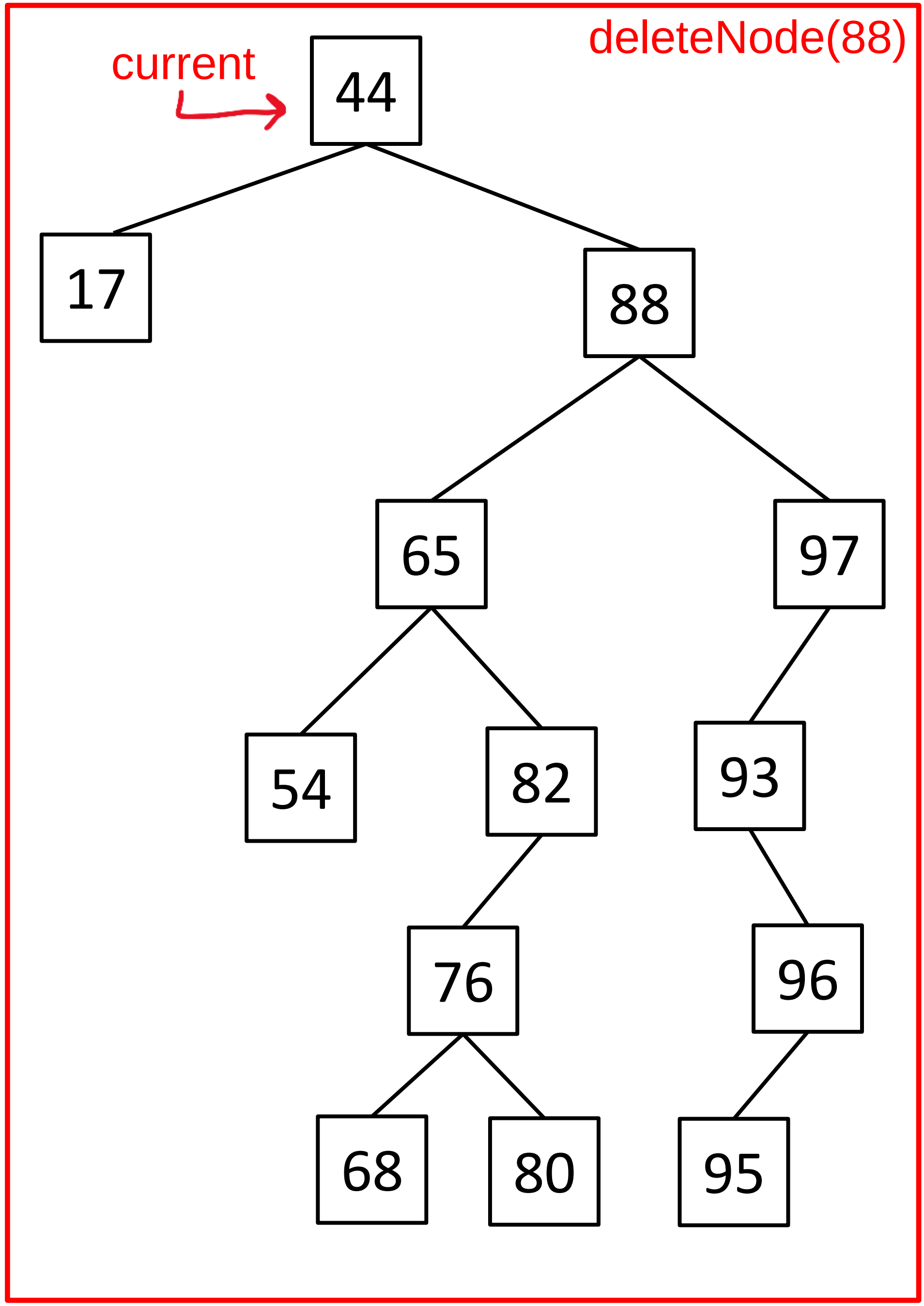
Case 1: Node has no children
Case 2: Node has one child
Case 3: Node has two children



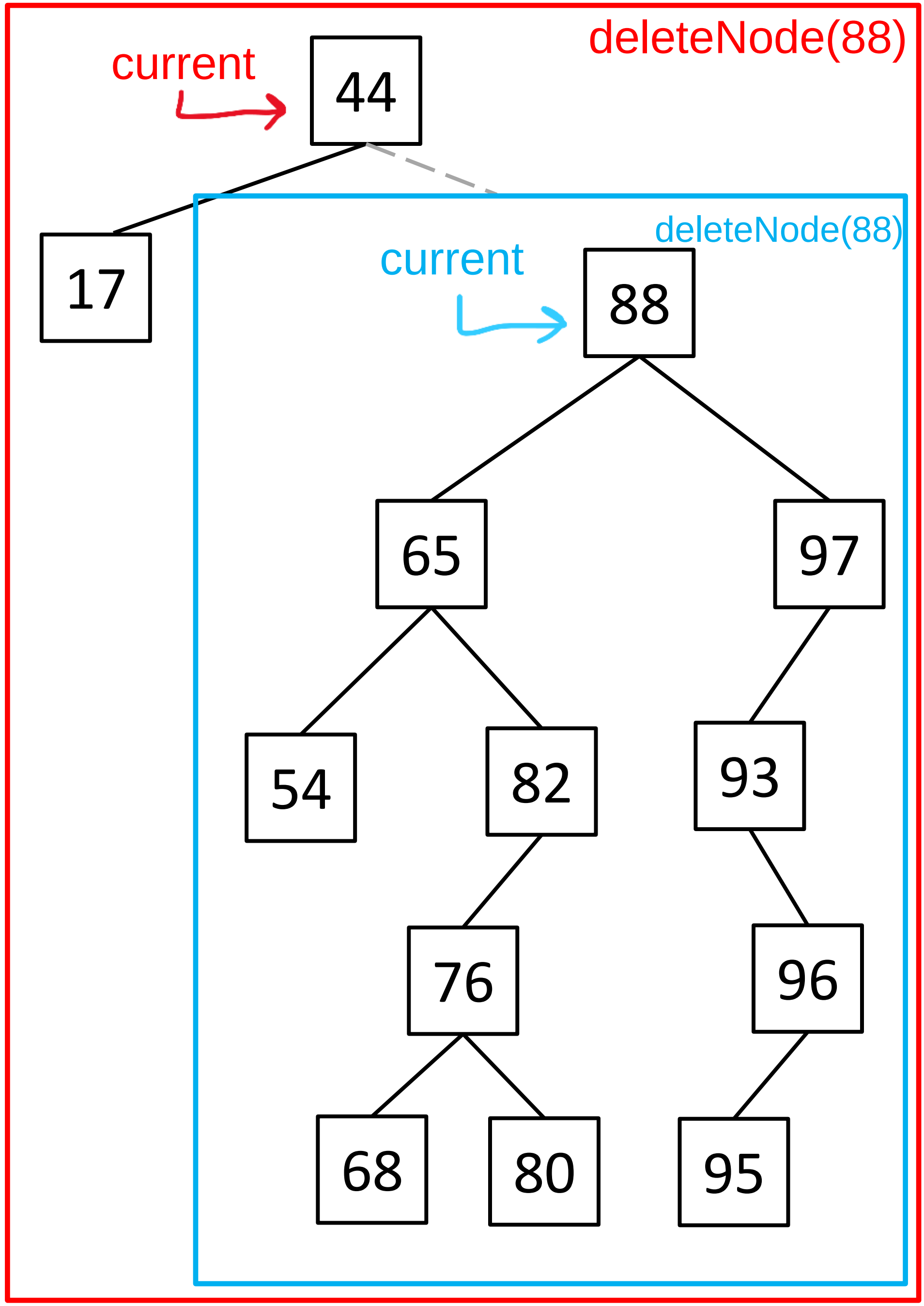
```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



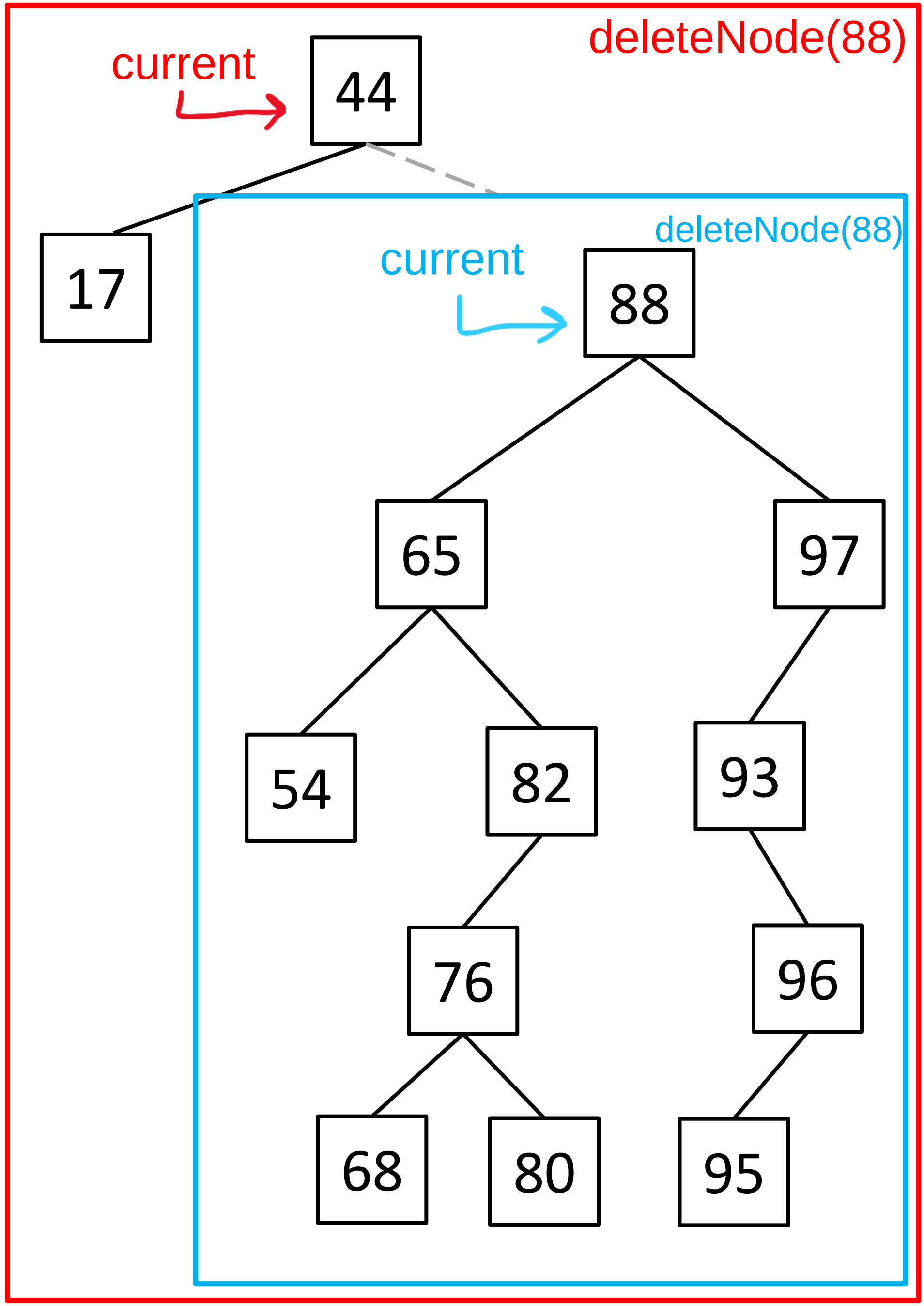
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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

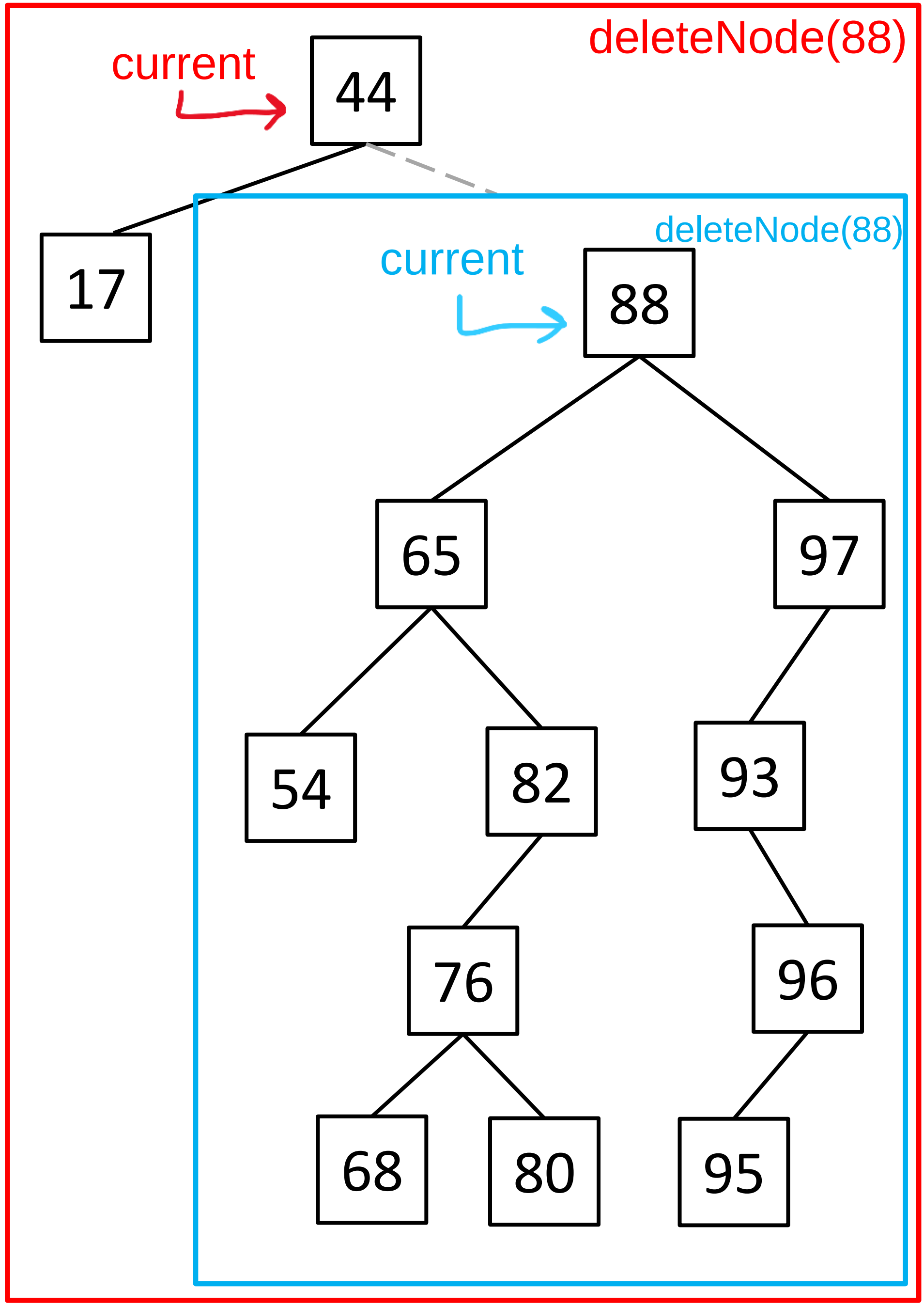


```
public Node deleteNode(Node current, int searchValue) {
if (current == null) {
  return current;
}
if (current.getValue() > searchValue) {
  Current.setLeft( deleteNode(current.getLeft(), searchValue));
}
else if (current.getValue() < searchValue) {
  current.setRight( deleteNode(current.getRight(),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

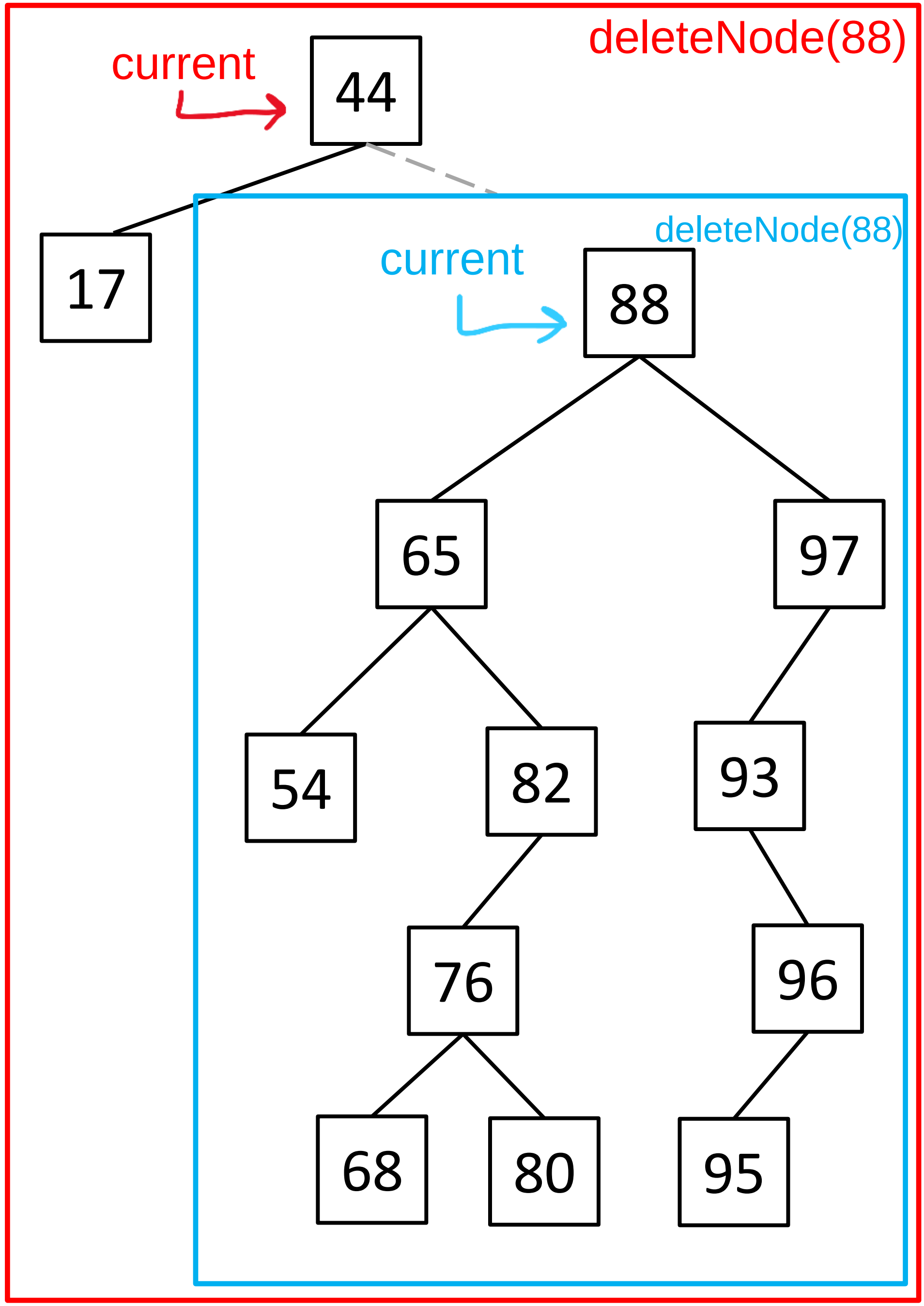


```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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

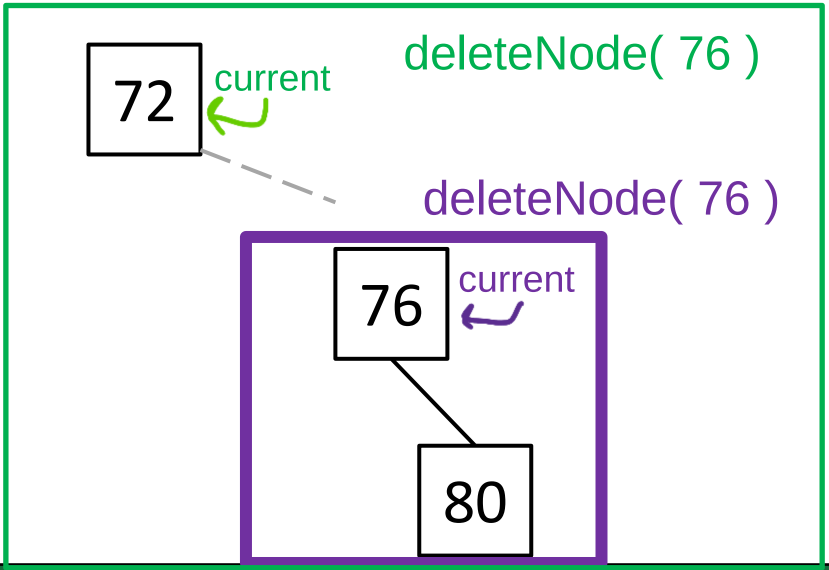
Case 1: Node has no children
Case 2: Node has one child
Case 3: Node has two children

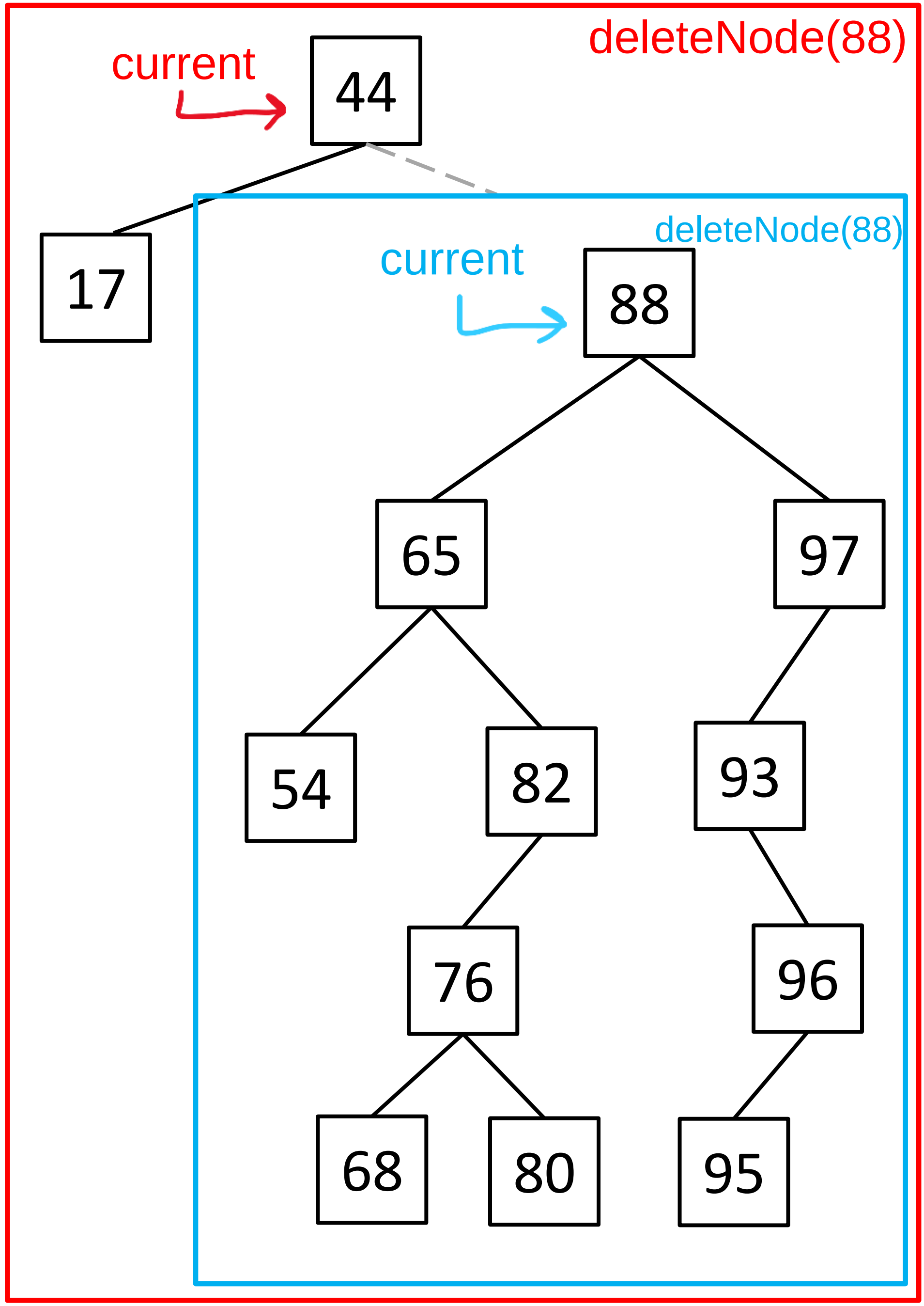


```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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

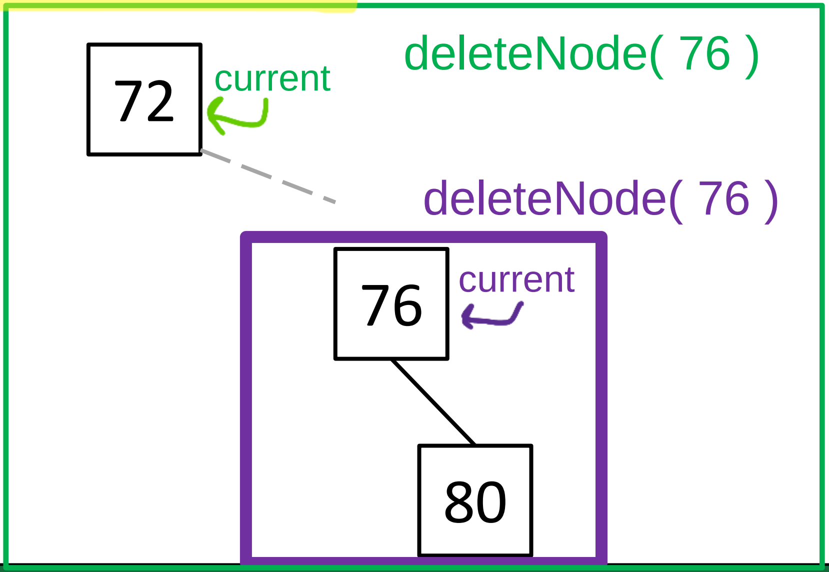


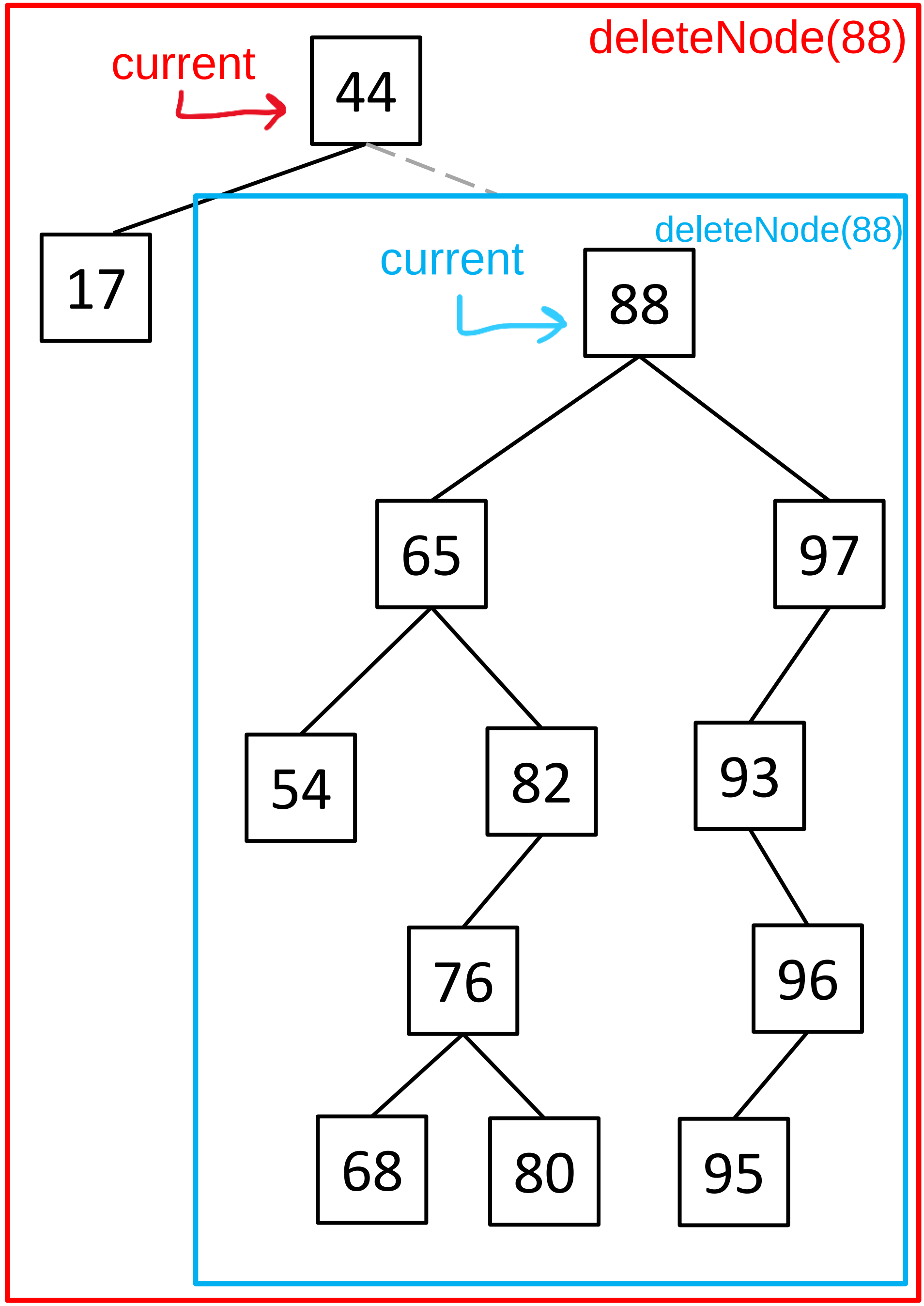
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



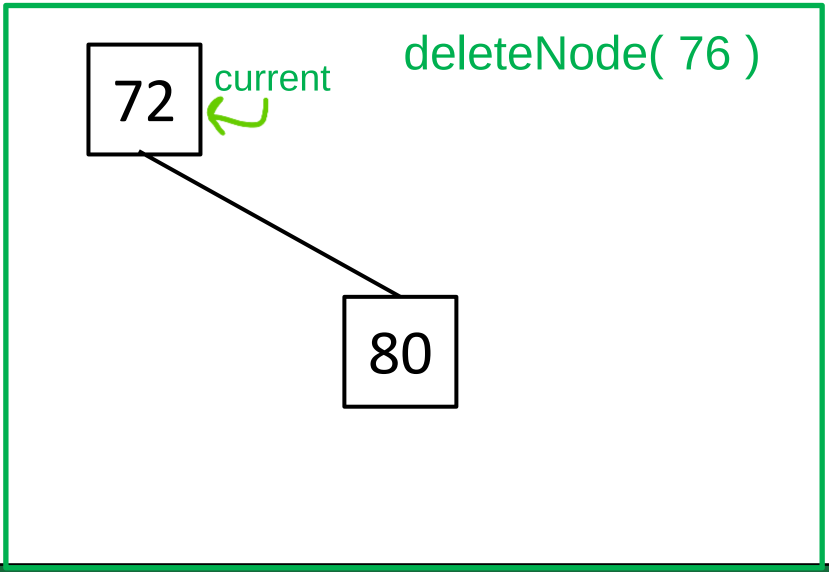


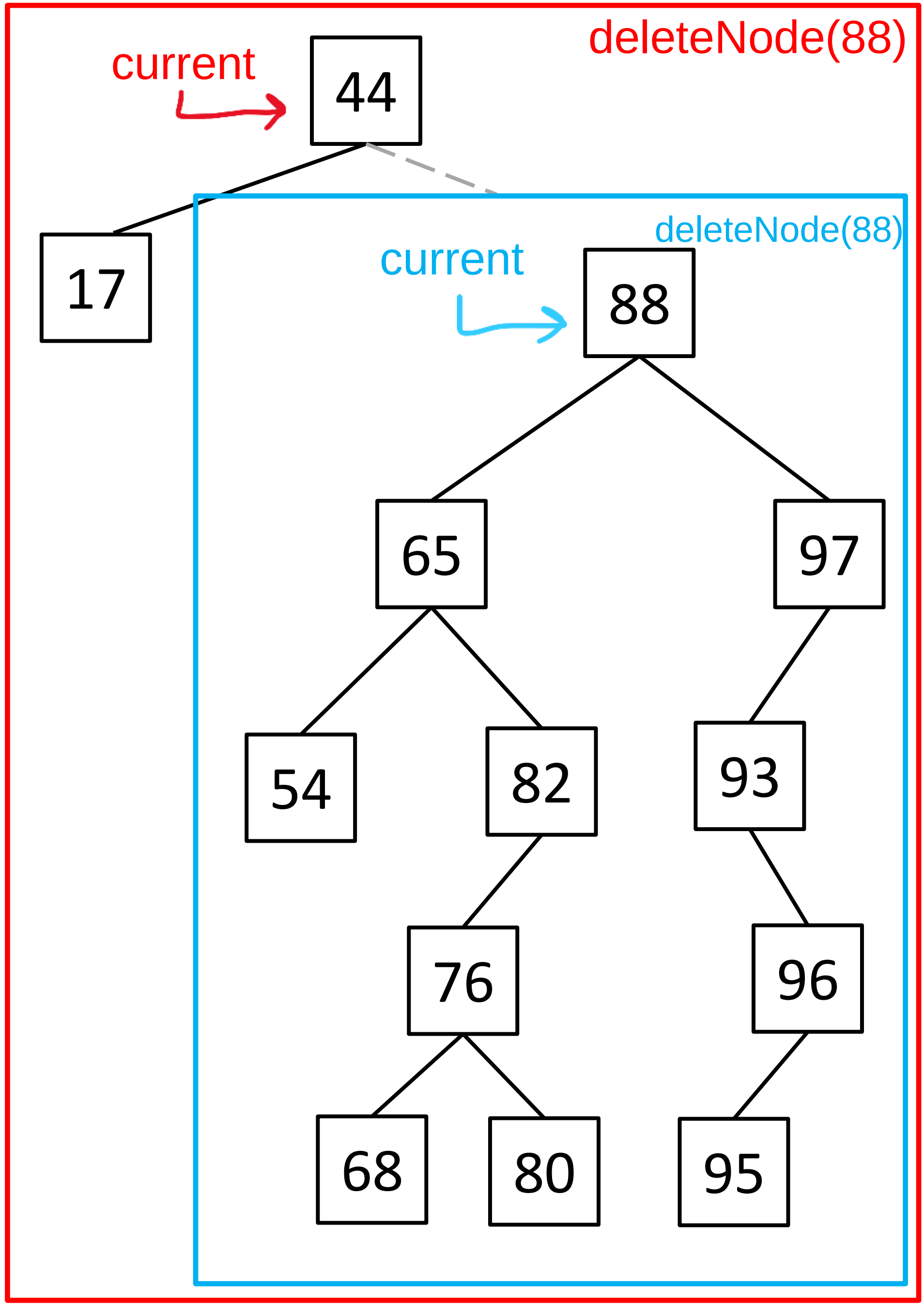
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



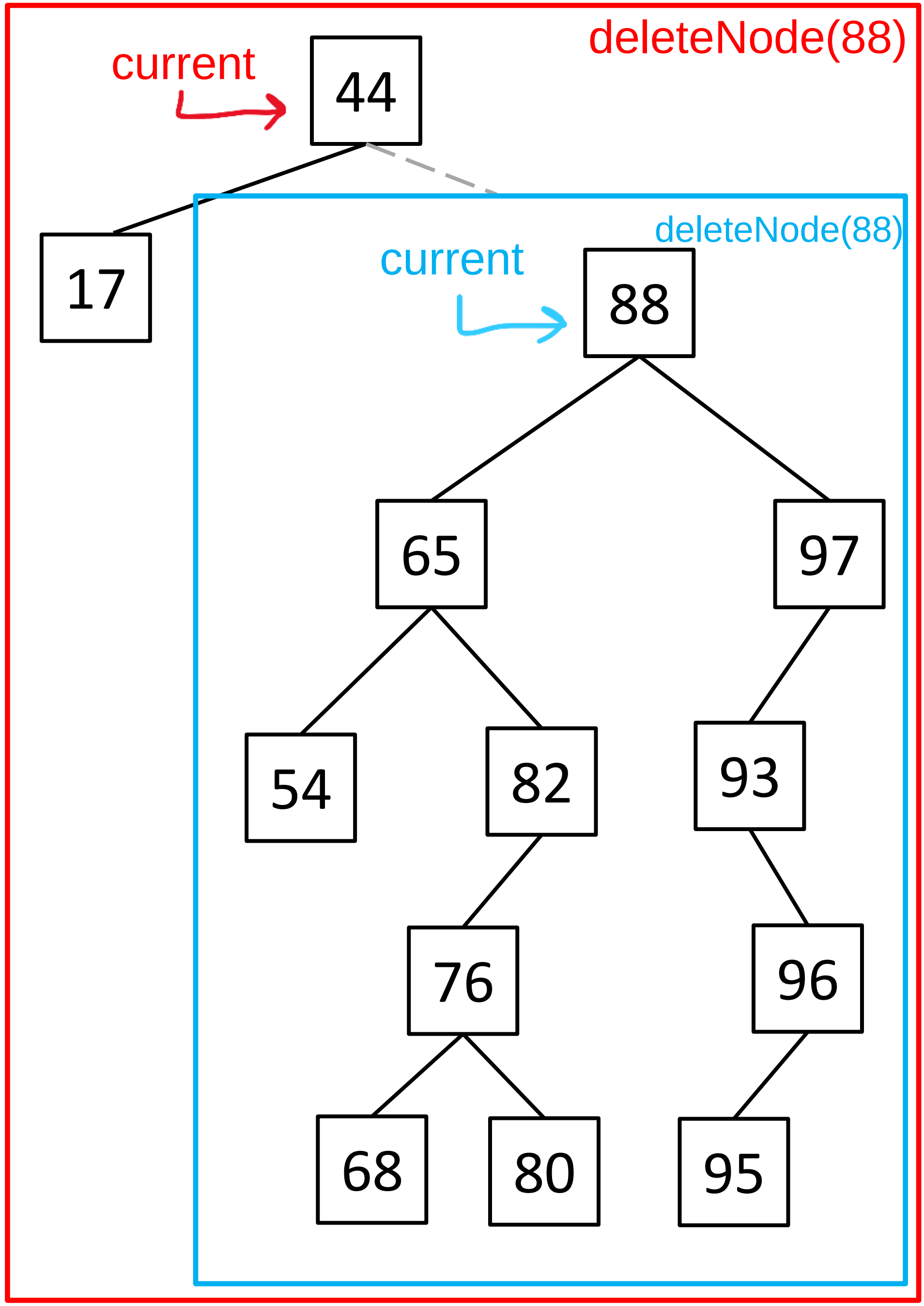


```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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

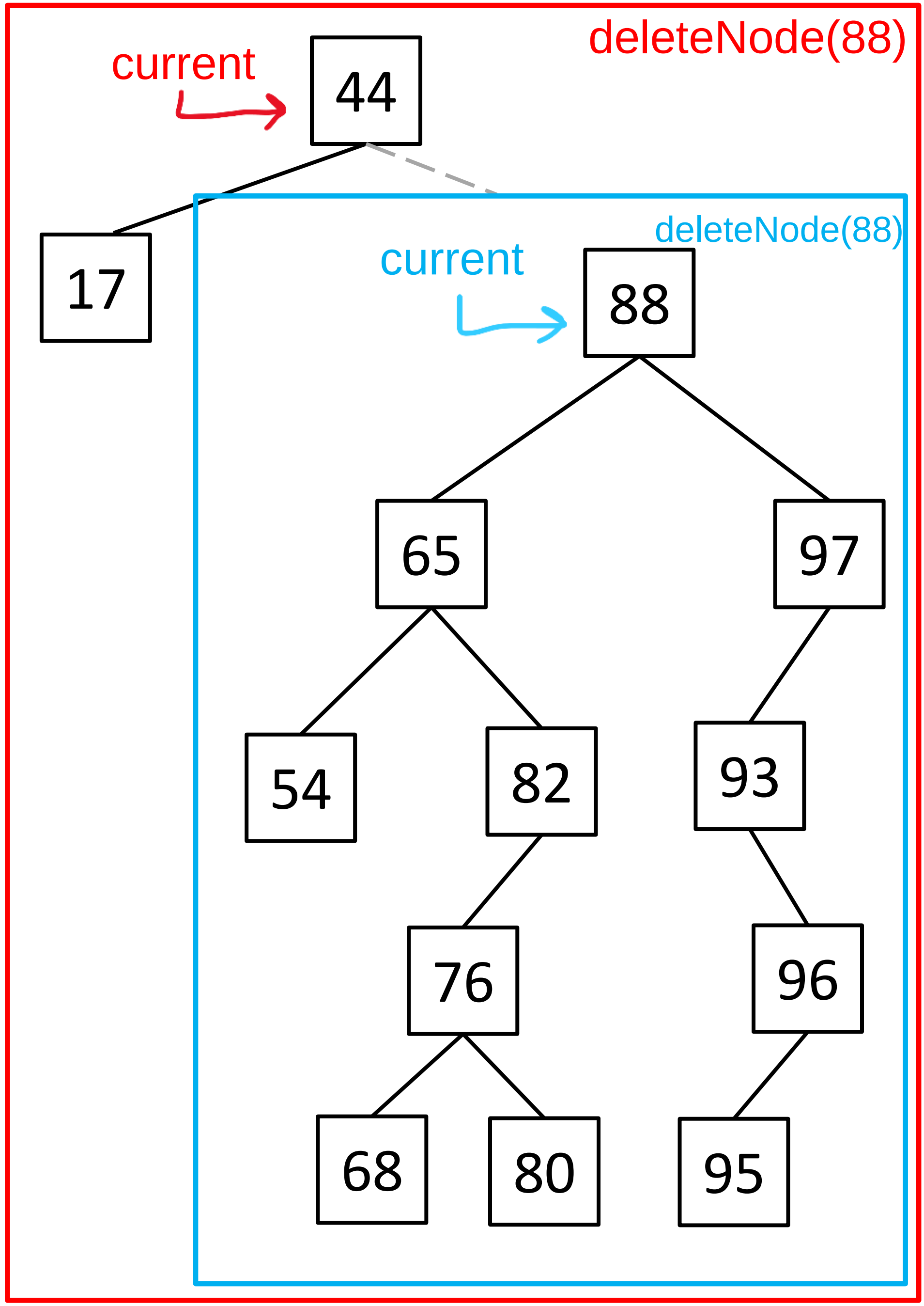




```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



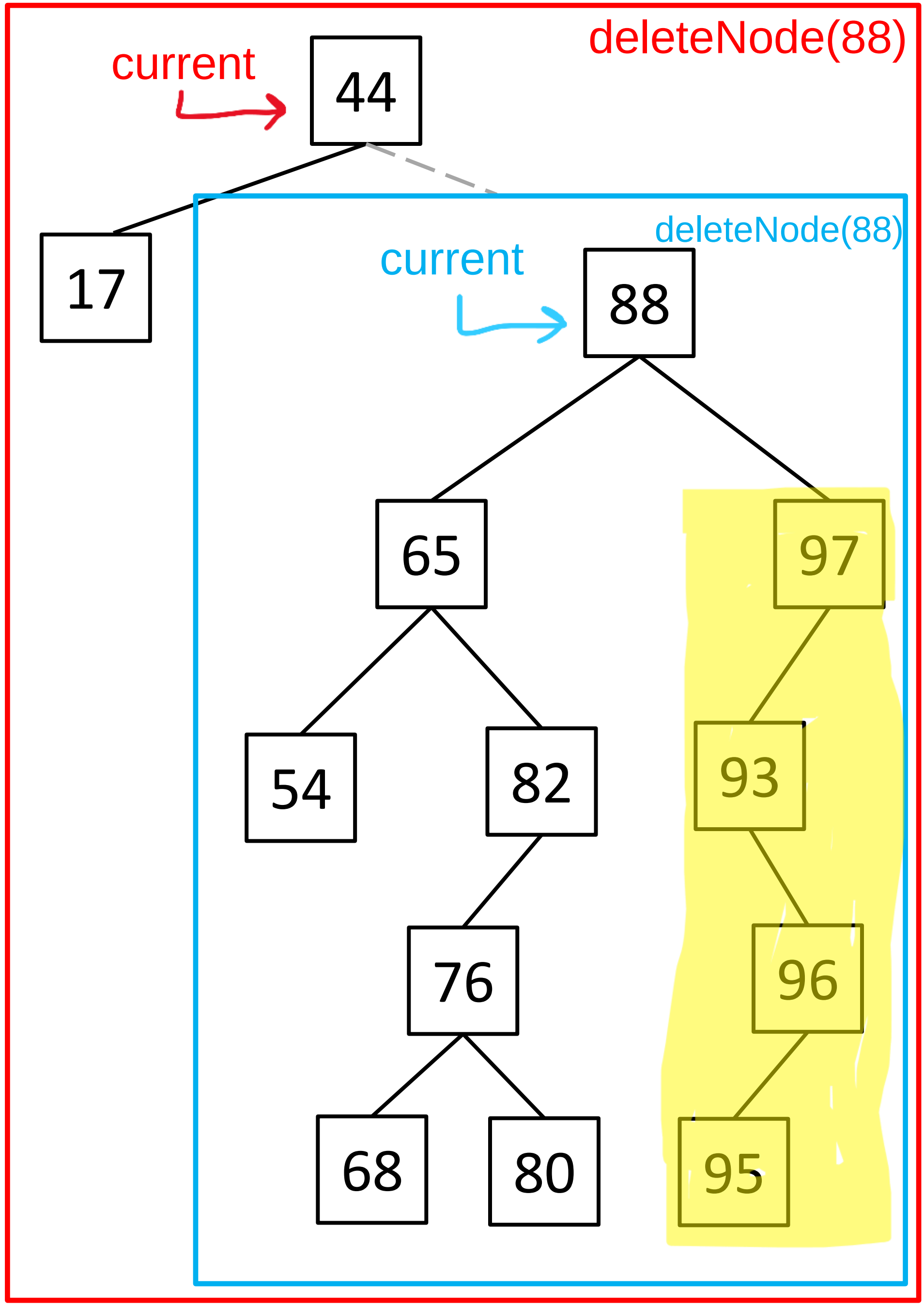
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



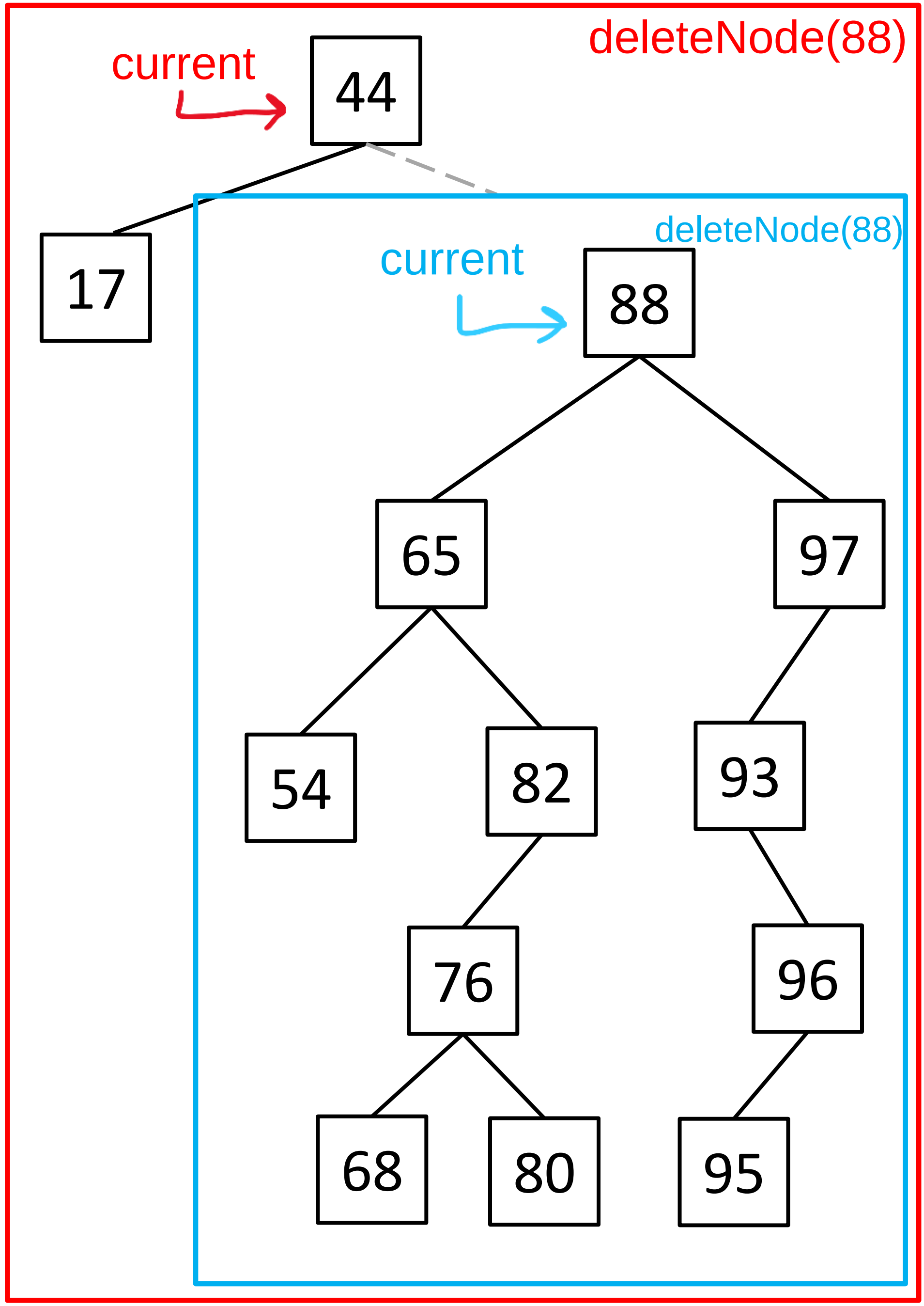
In our current recursive call (blue square), the node we are currently at has two children

```java
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);
```

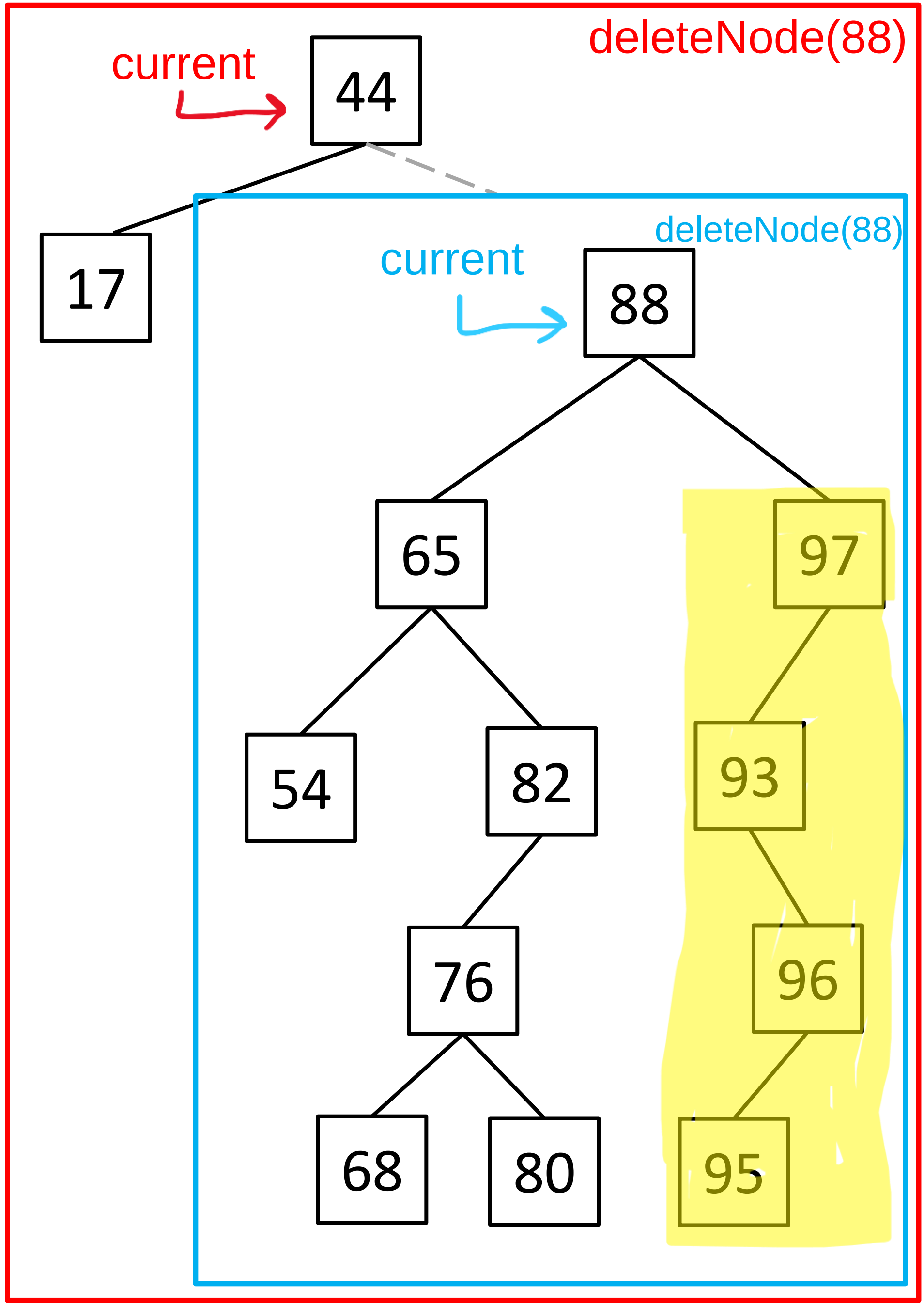
Helper method to find replacement (smallest value in right subtree)



```java
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);
```

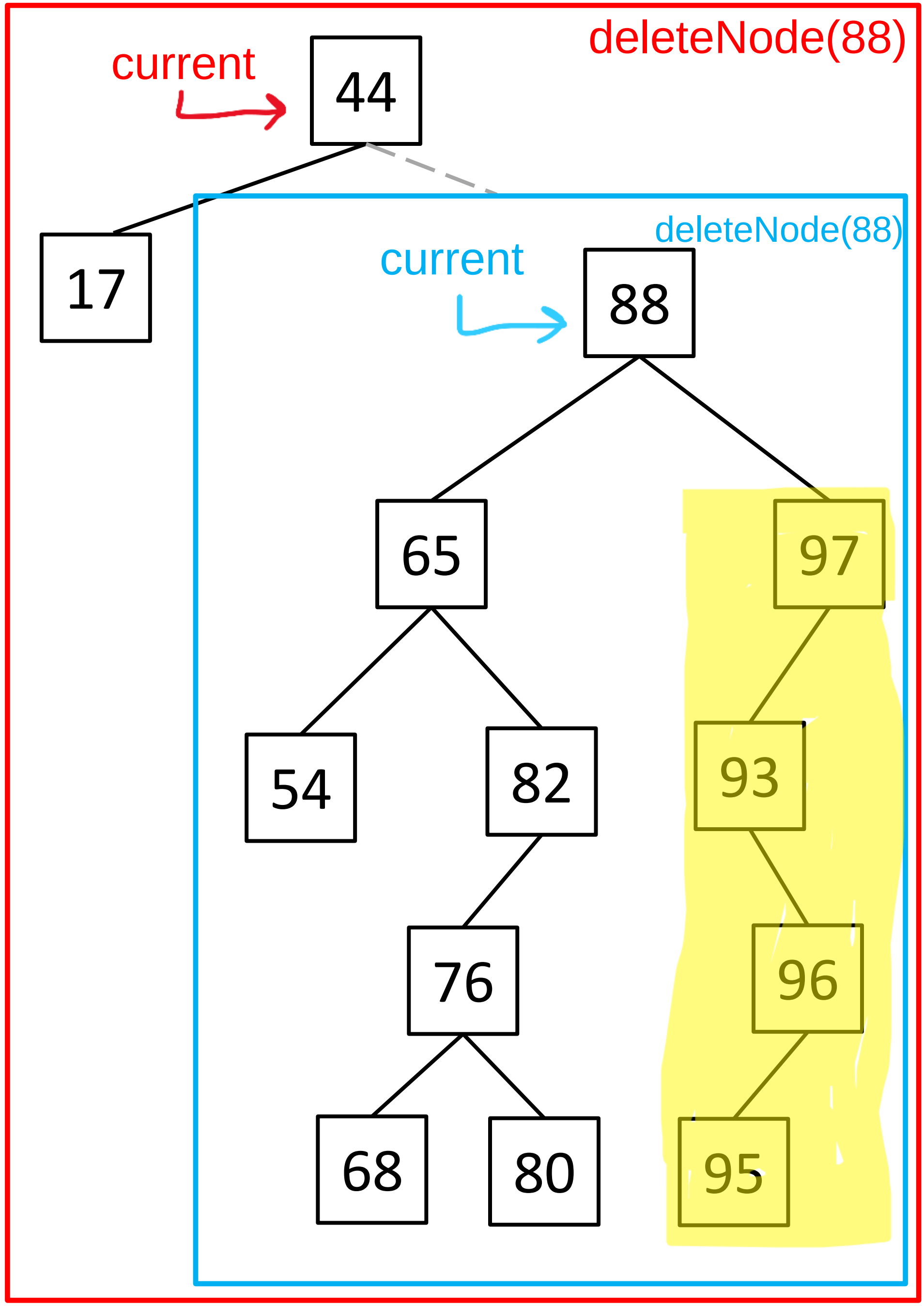


```java
public Node findReplacement(Node current) {
  current = current.getRight();
  while(current != null && current.getLeft() != null) {
    current = current.getLeft();
  }
  return current;
}
```



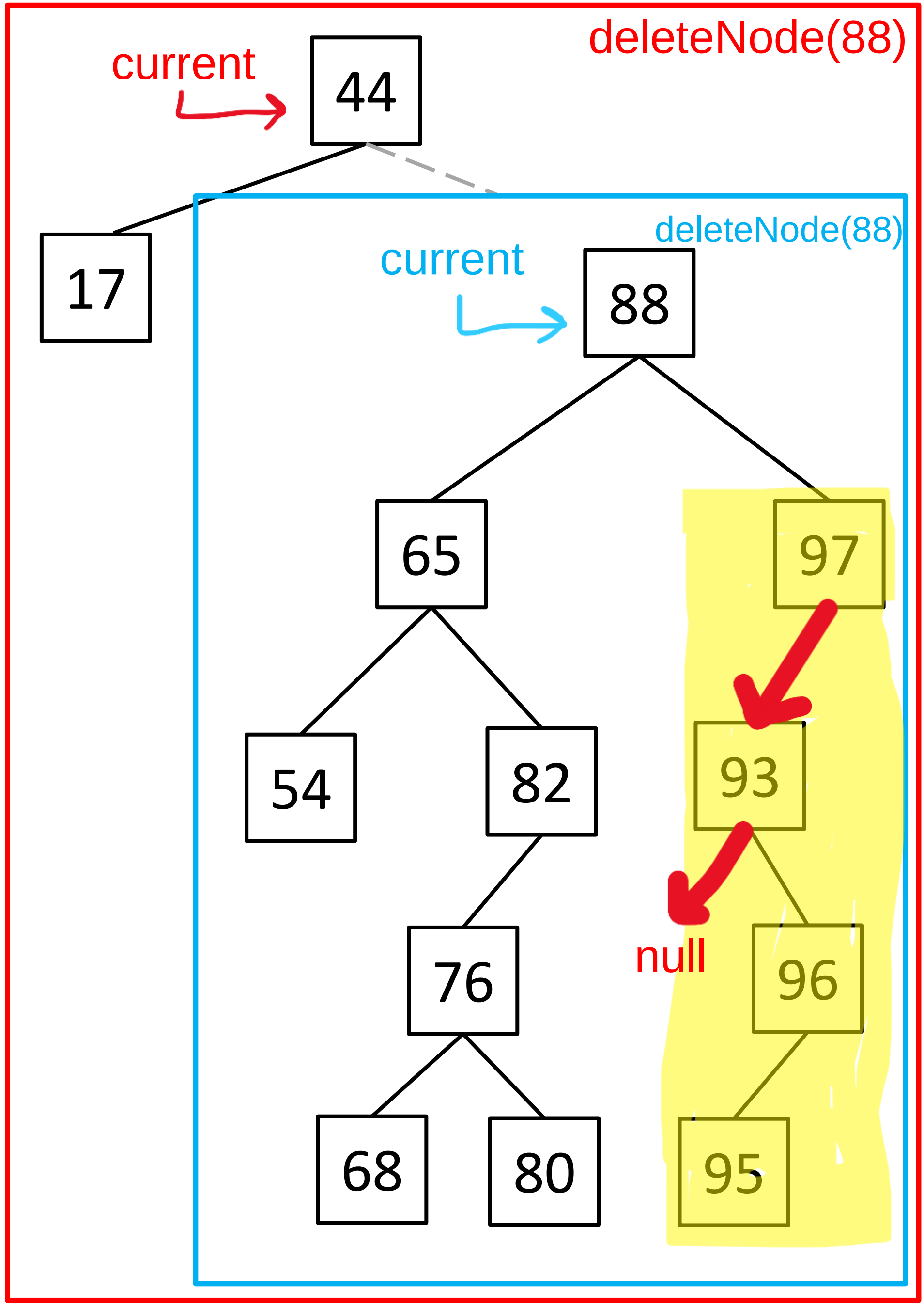
```java
public Node findReplacement(Node current) {
  current = current.getRight();
  while(current != null && current.getLeft() != null) {
    current = current.getLeft();
  }
  return current;
}
```

1. Go into right subtree



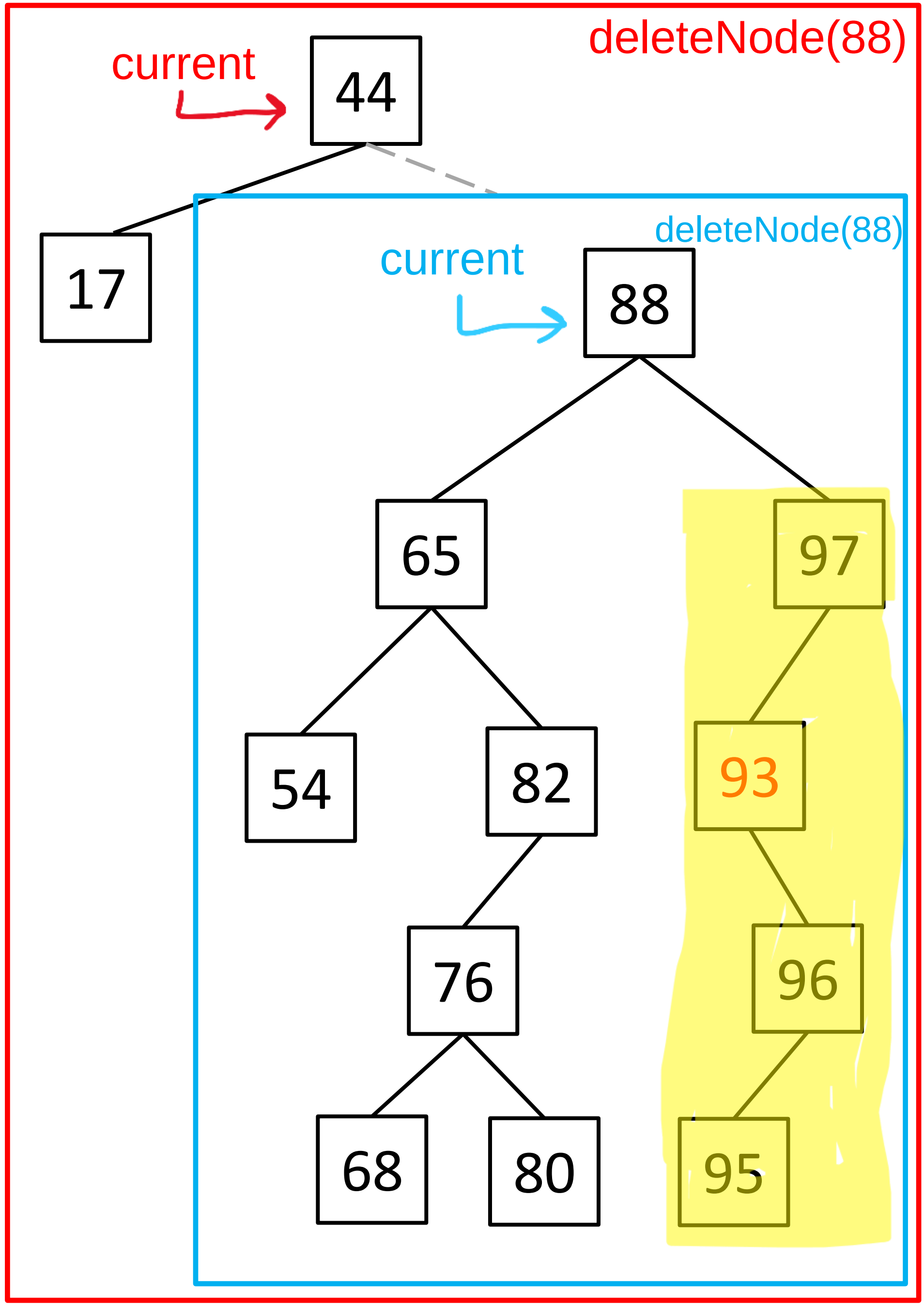
```
public Node findReplacement(Node current) {
  current = current.getRight();
  while(current != null && current.getLeft() != null) {
    current = current.getLeft();
  }
  return current;
}
```

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

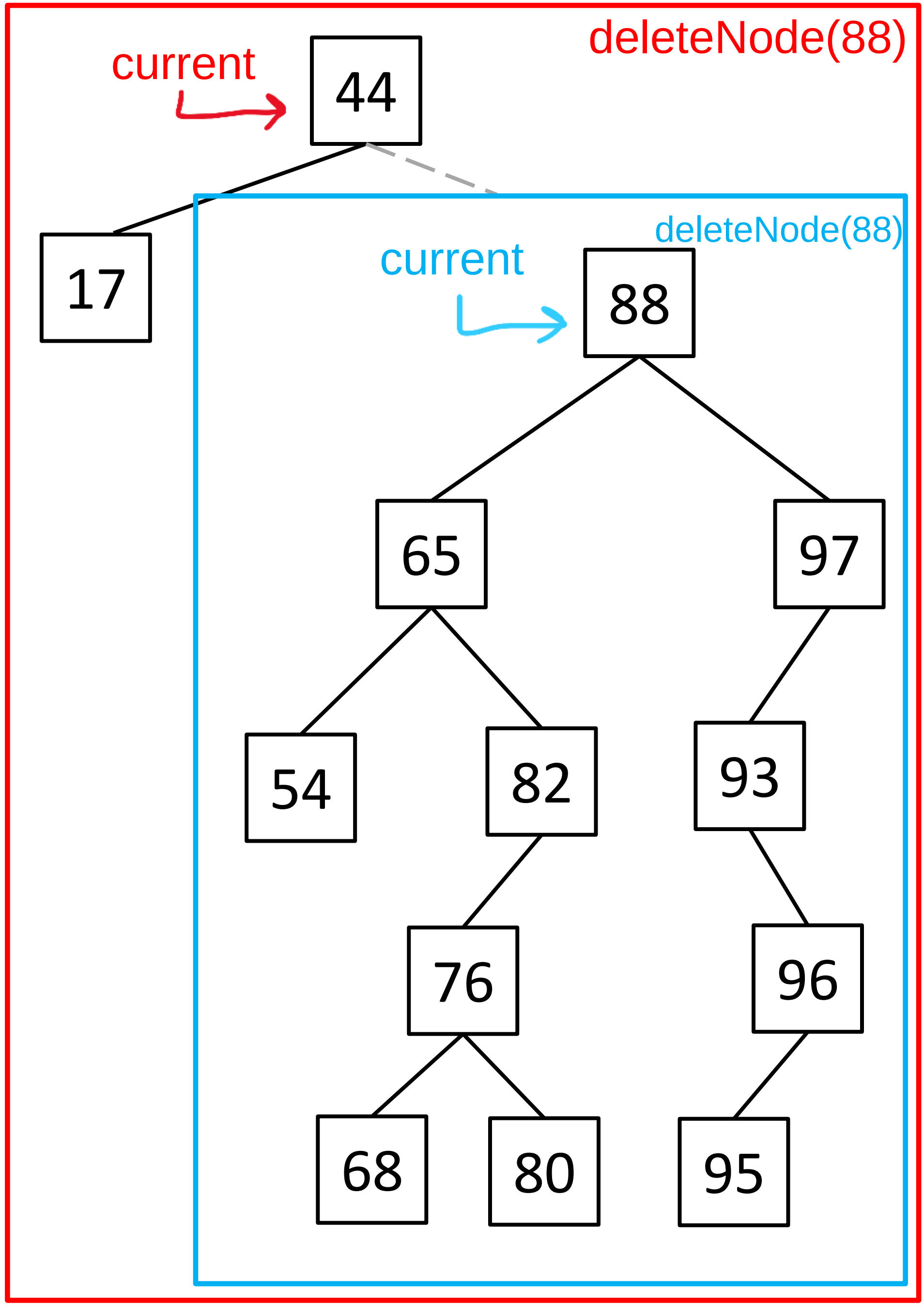


```java
public Node findReplacement(Node current) {
  current = current.getRight();
  while(current != null && current.getLeft() != null) {
    current = current.getLeft();
  }
  return current;
}
```

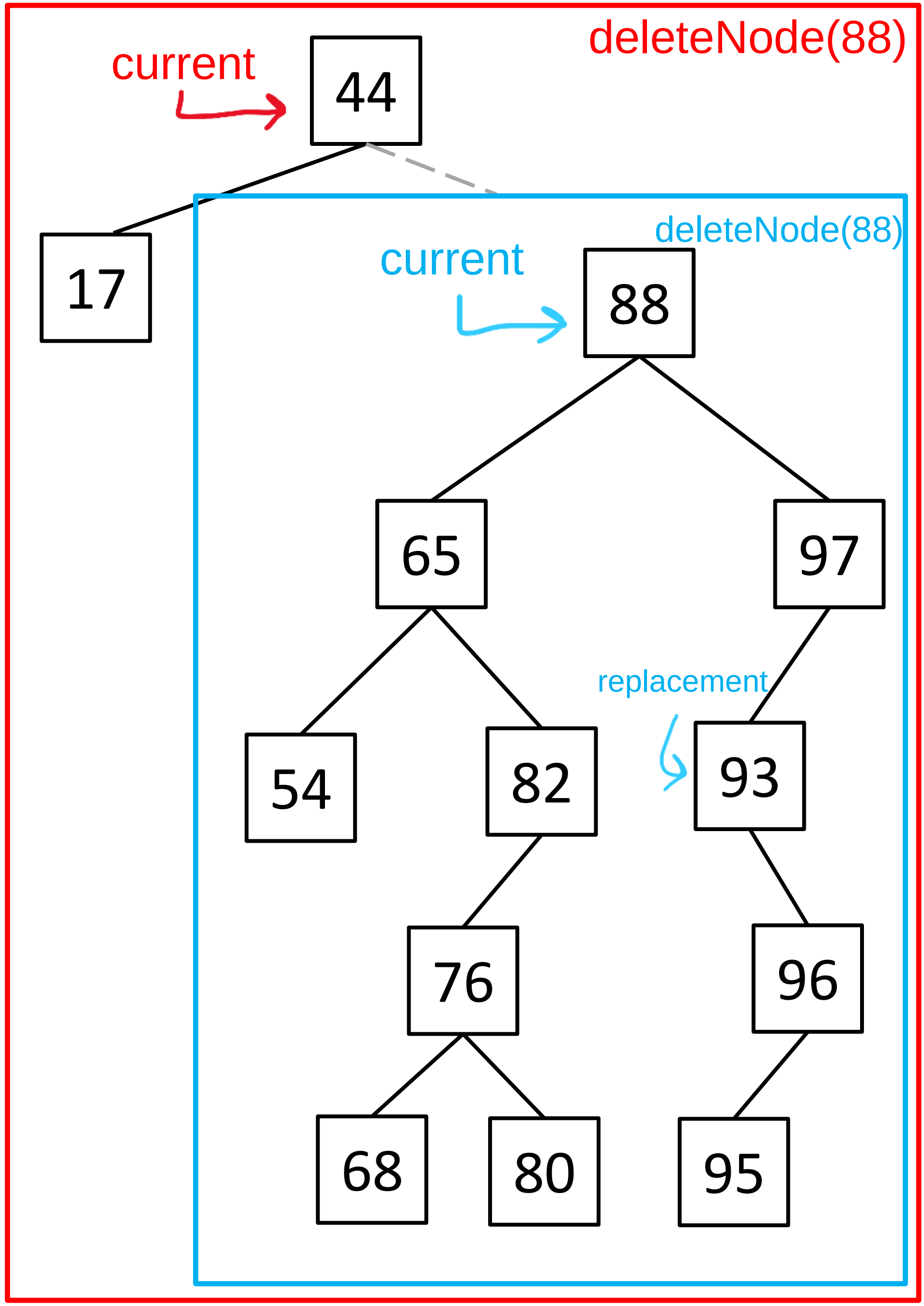
1. Go into right subtree
2. Keep going left in the subtree
3. When we cant go left anymore, return node we are at



```java
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);
```

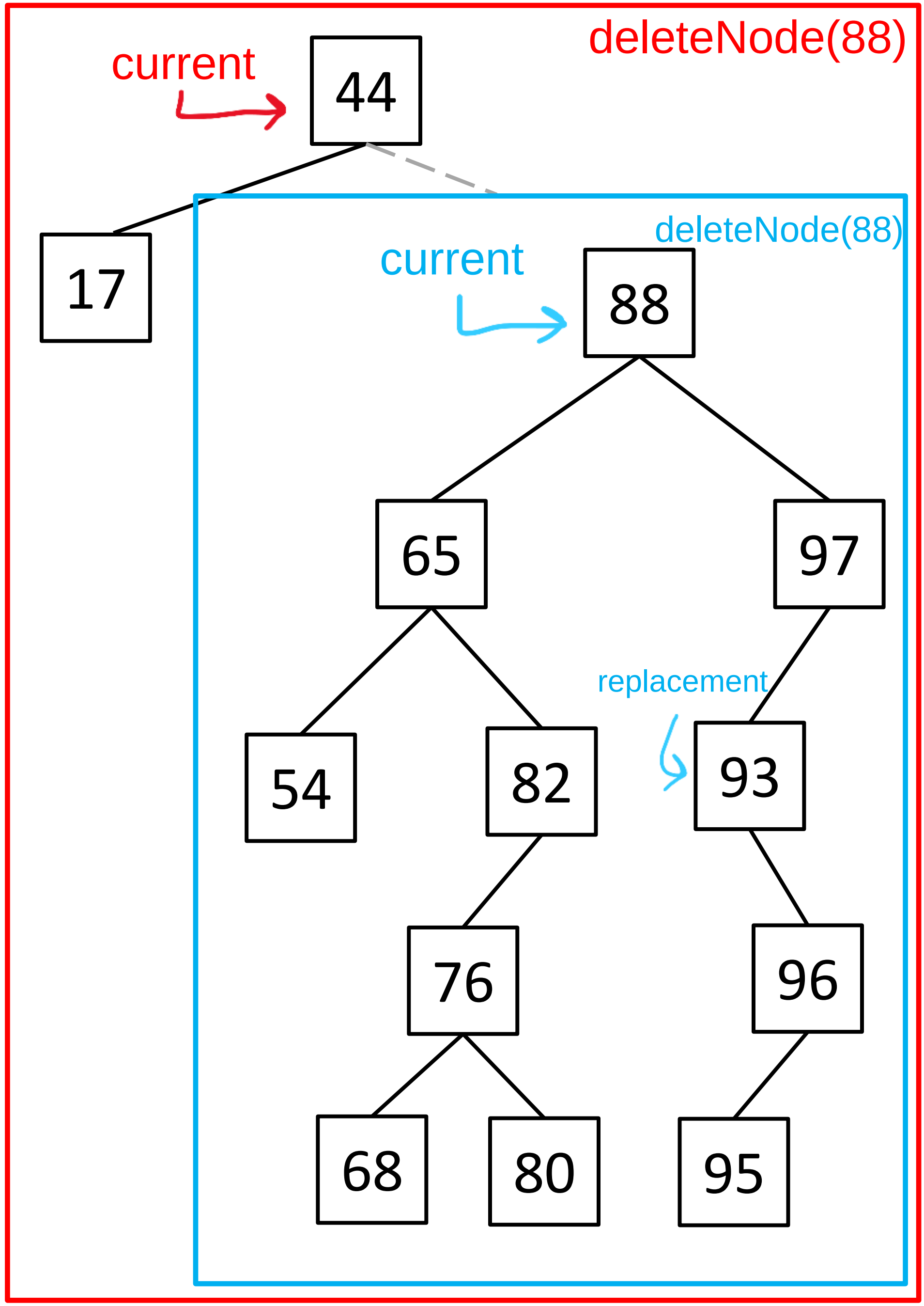


```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



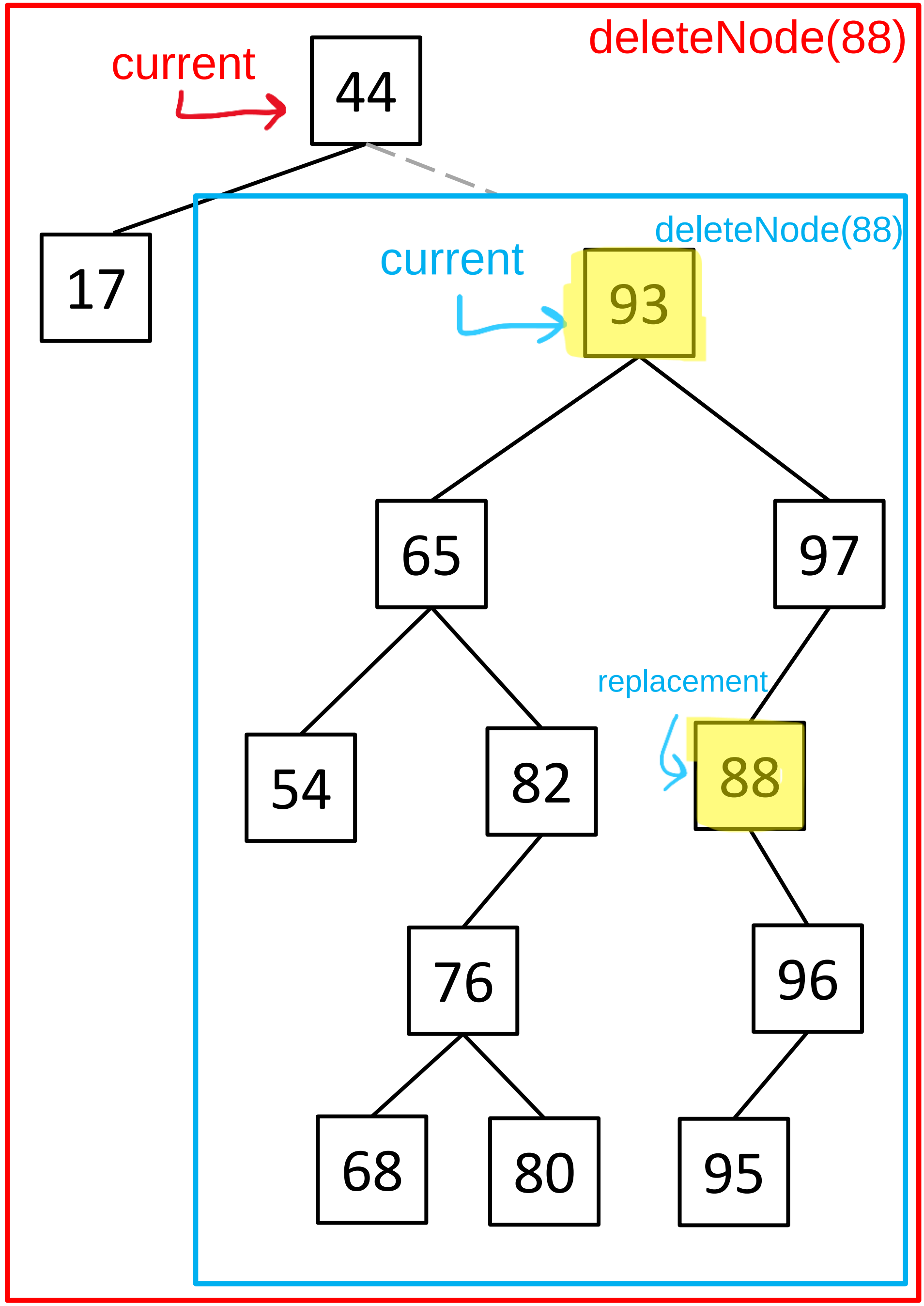
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



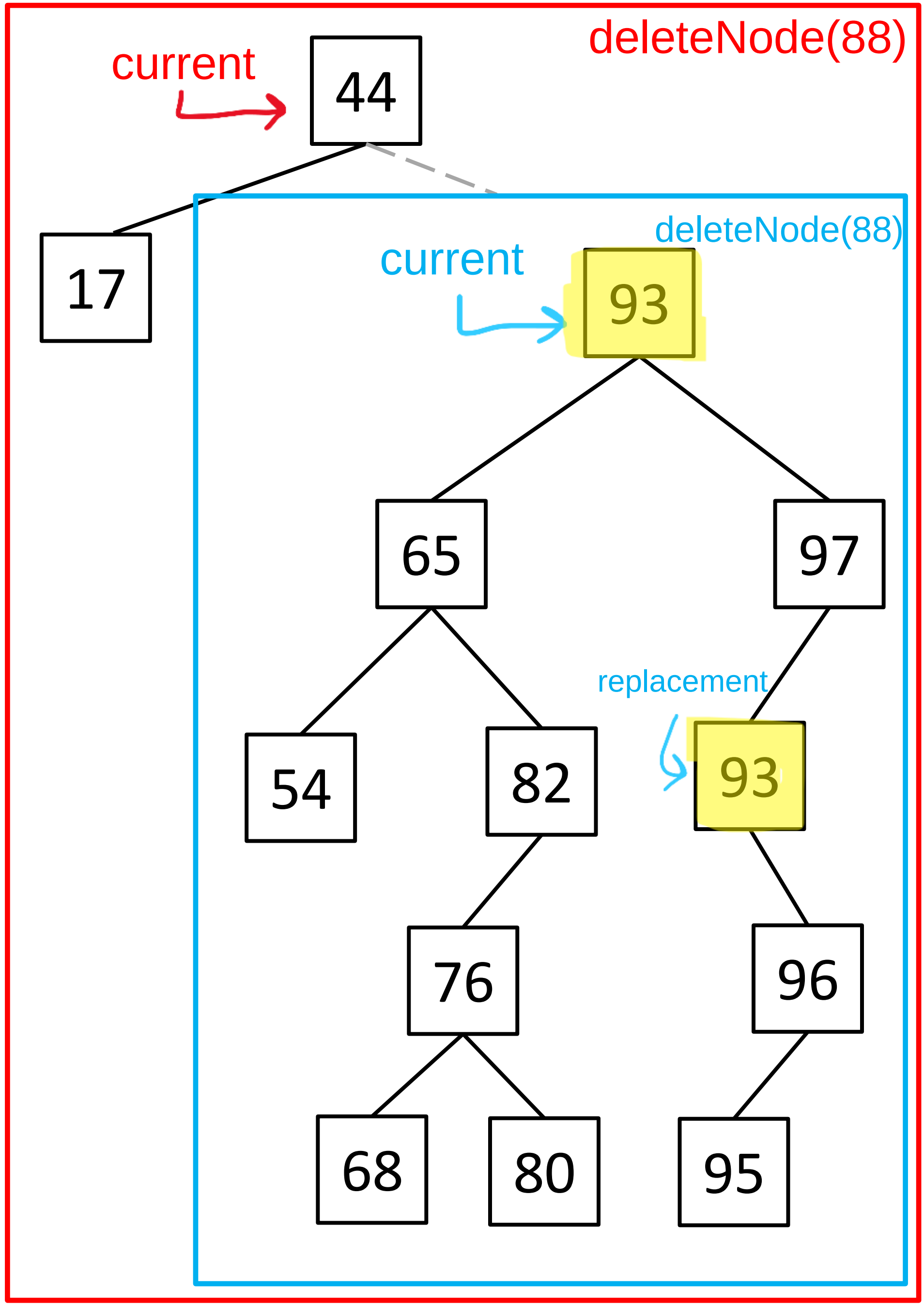
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



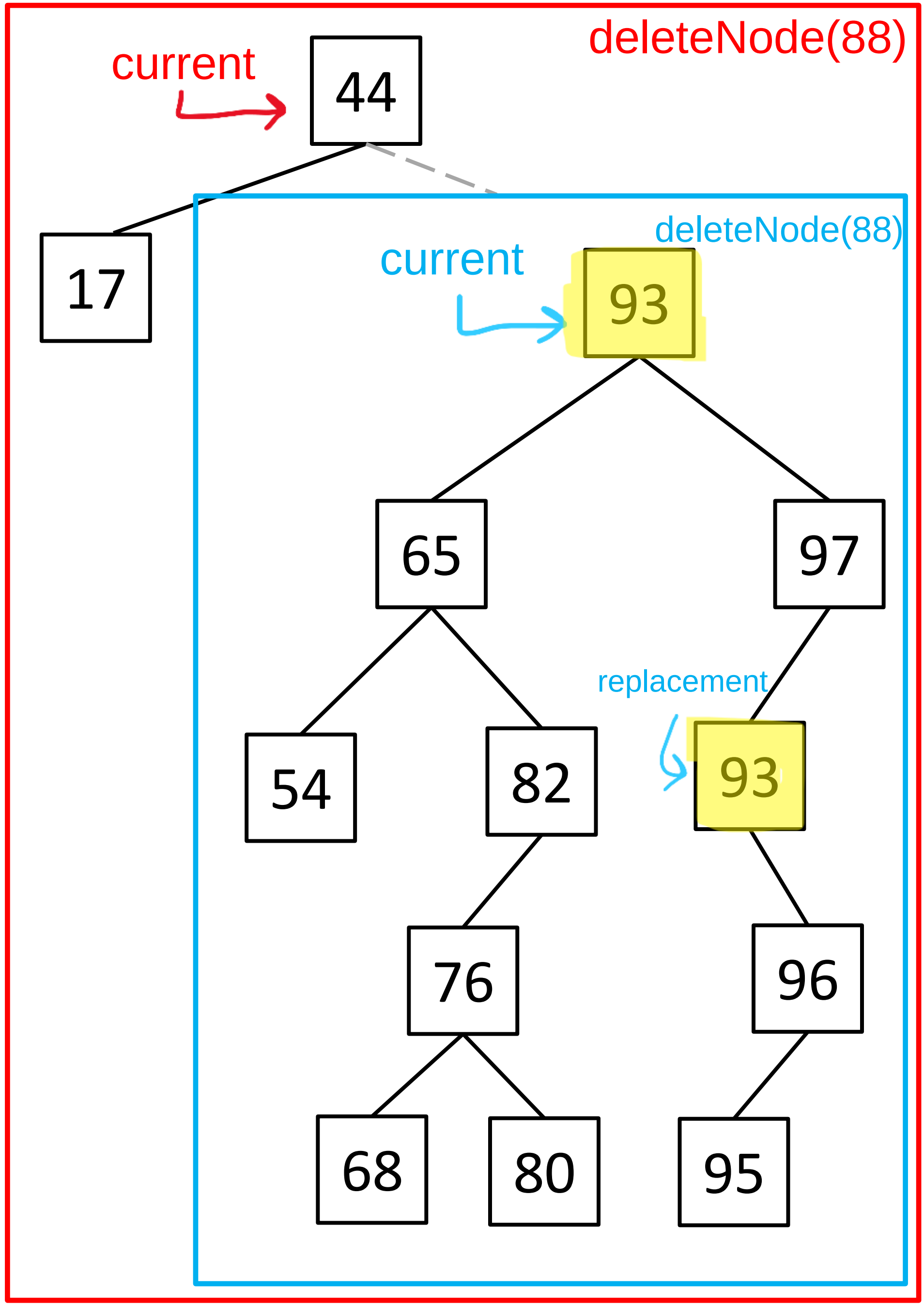
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



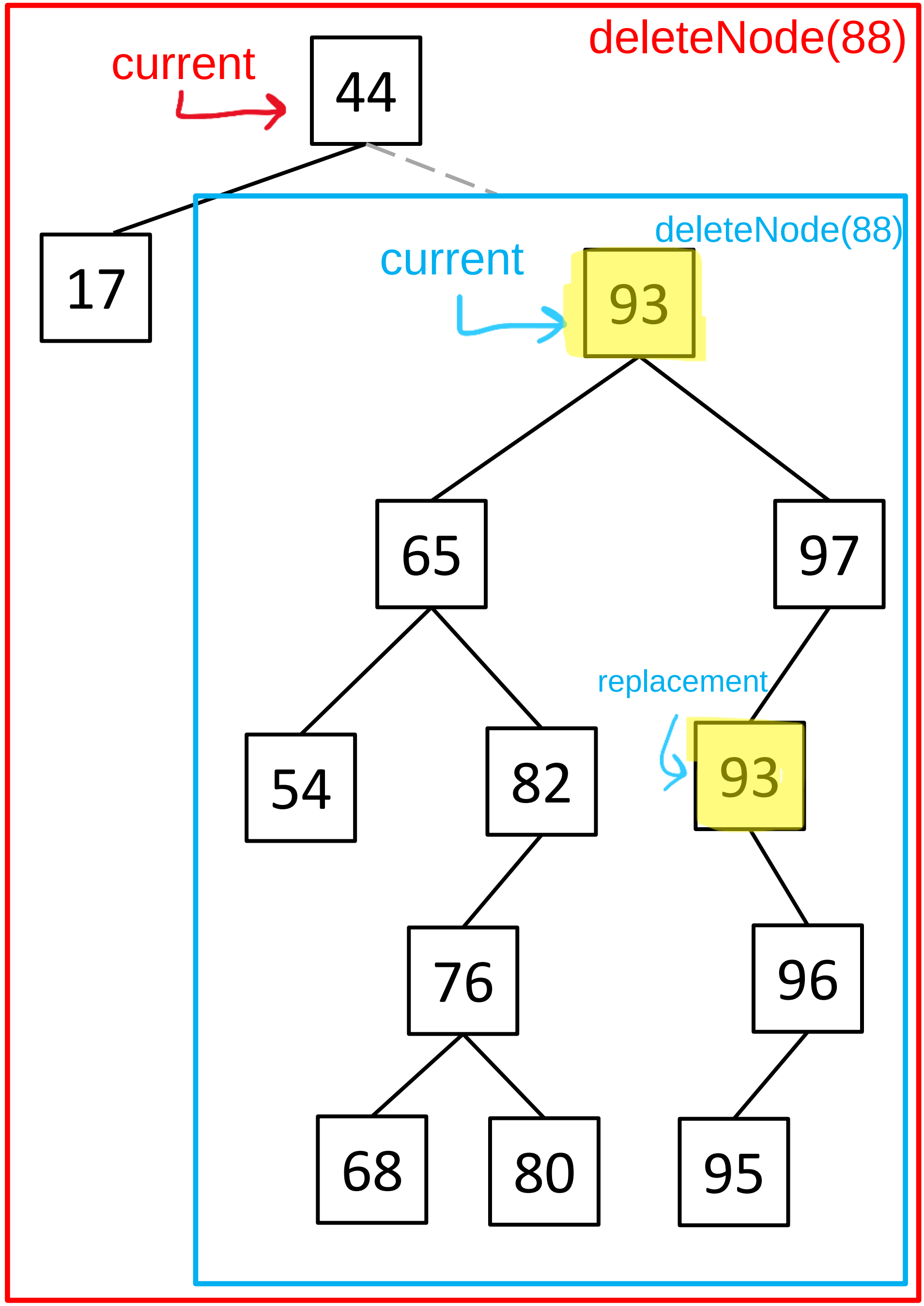
```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



```java
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());
```

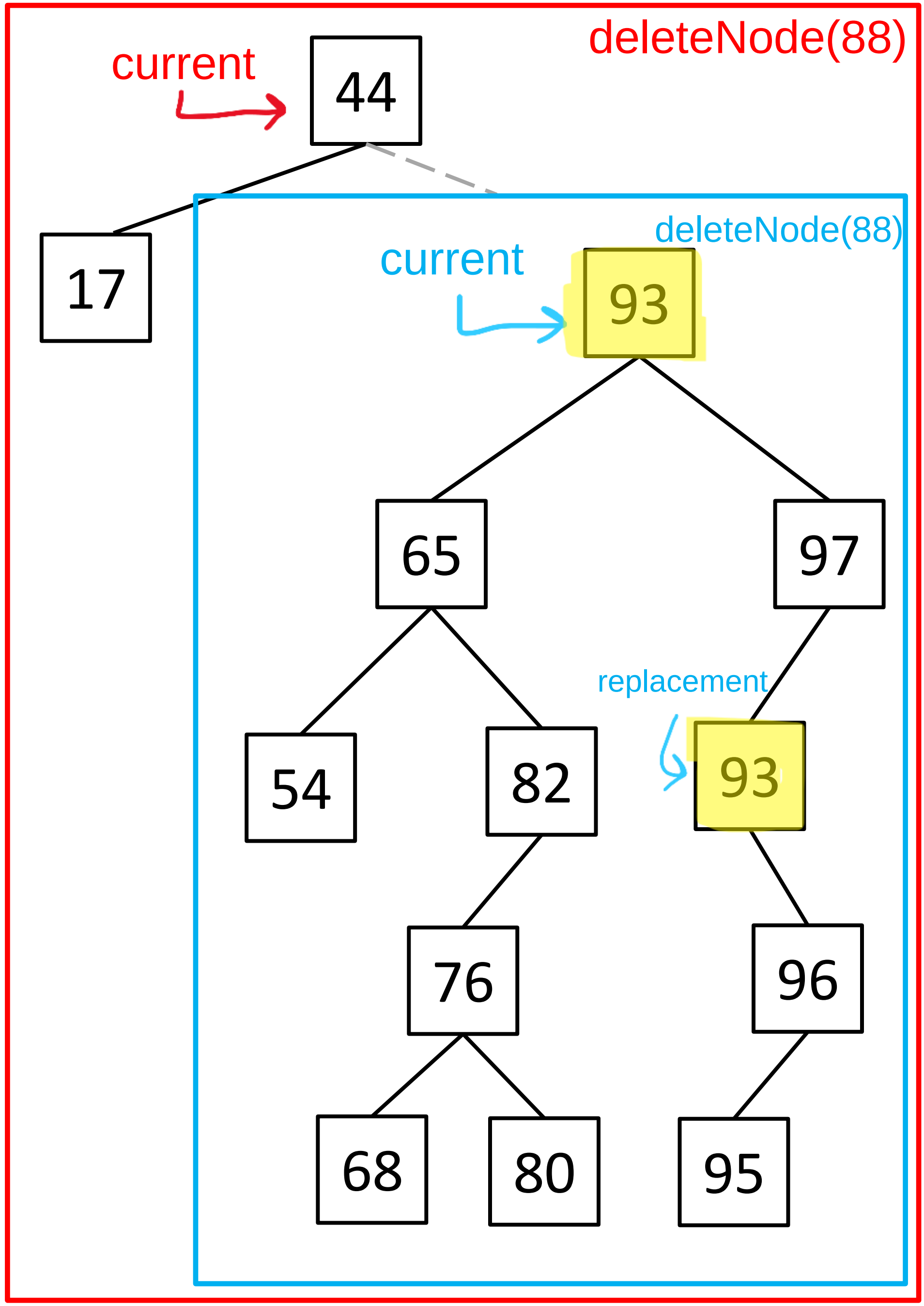
*How to remove a ~~duplicate~~ **node** in a BST?*



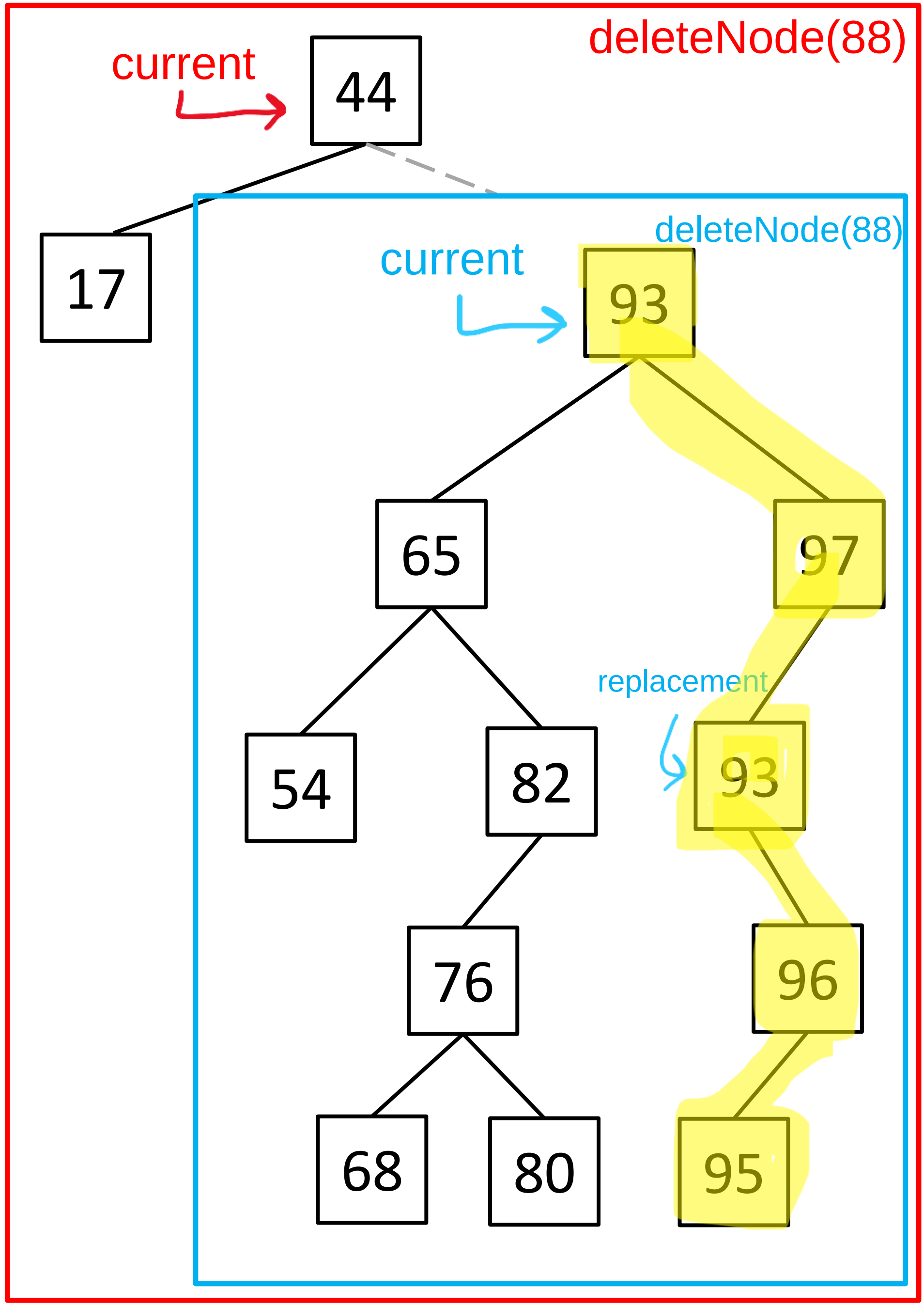
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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

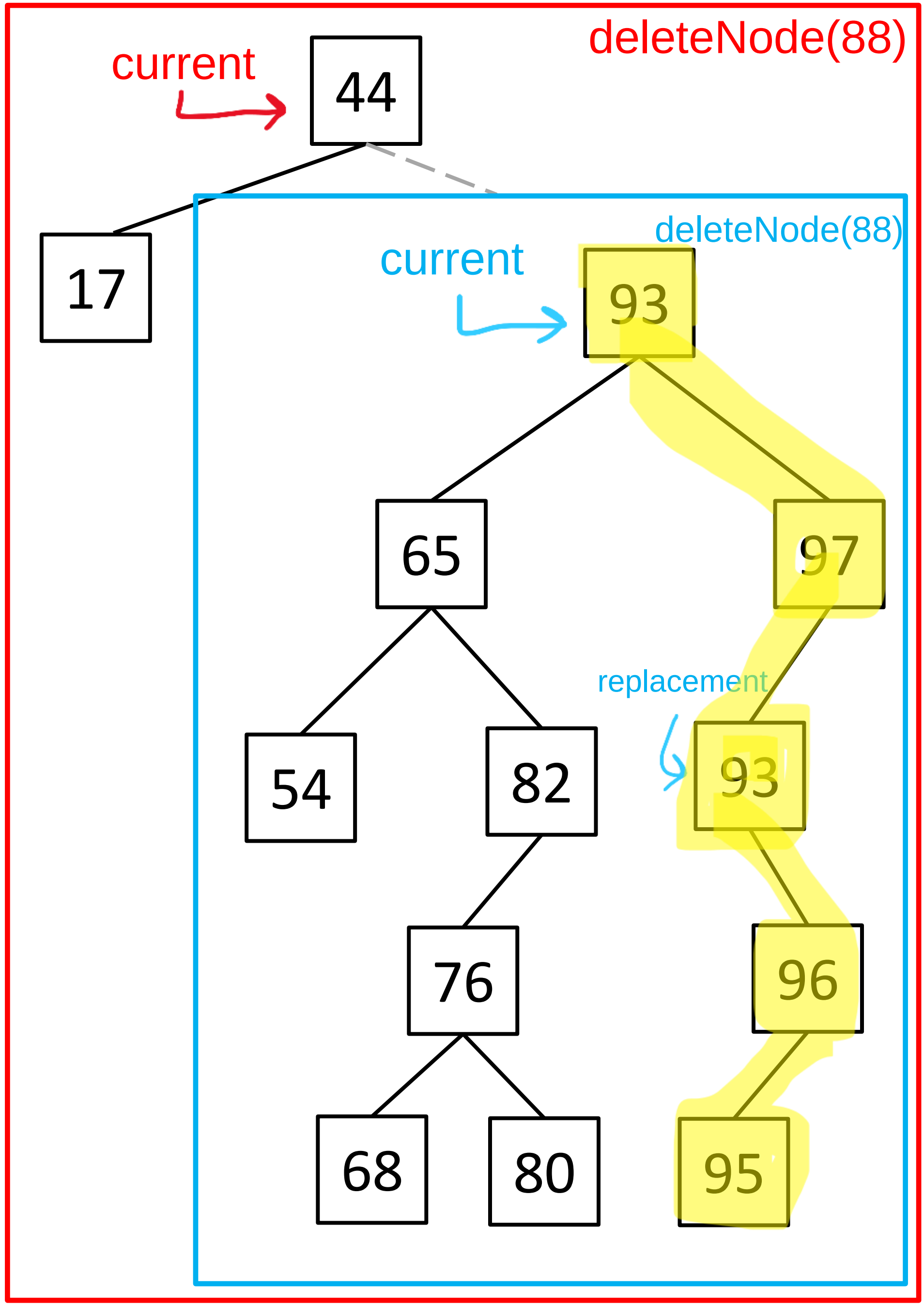


```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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

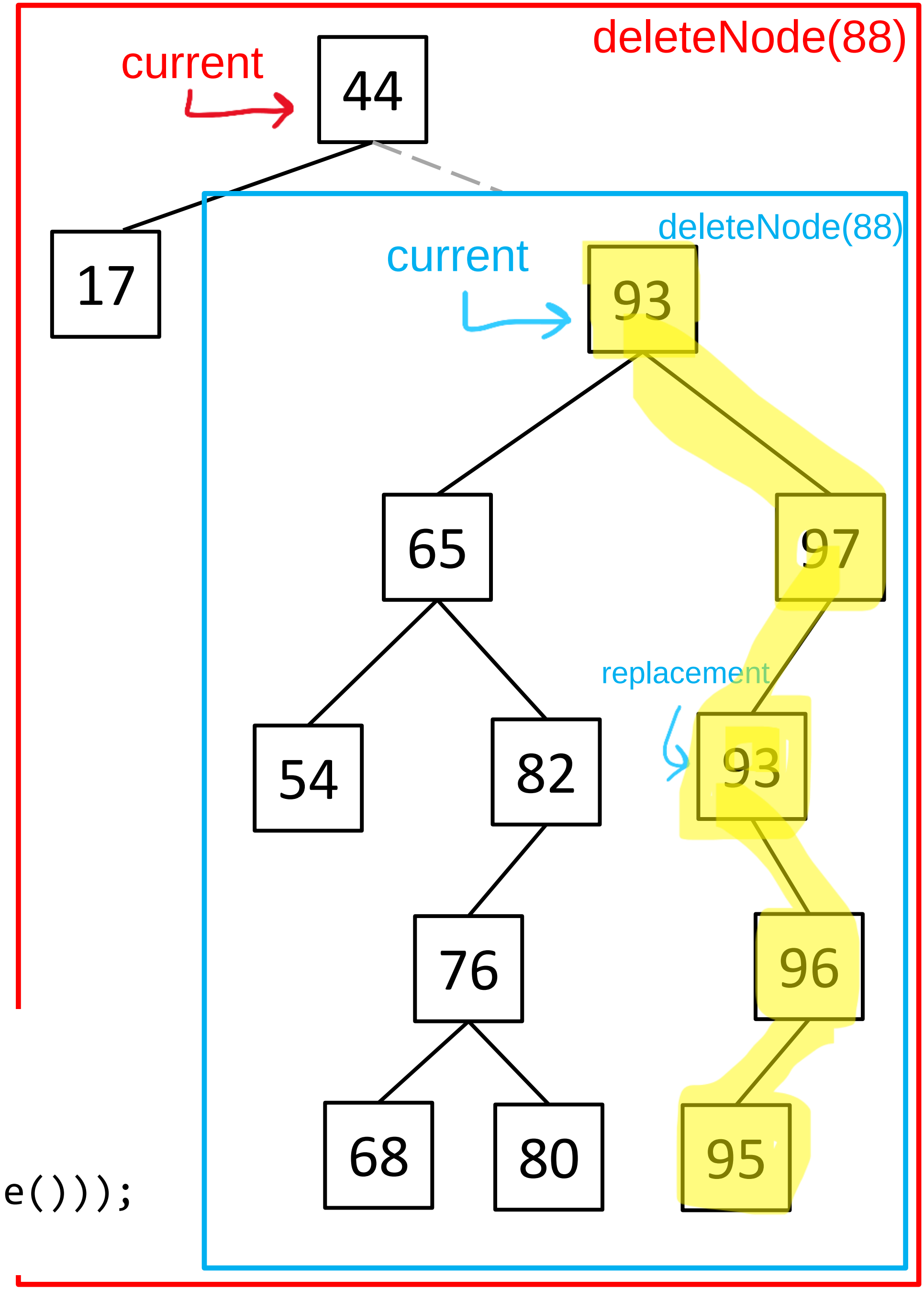


When the replacement was done, the only thing that was affected was the **right subtree**

```java
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( ??? );
```

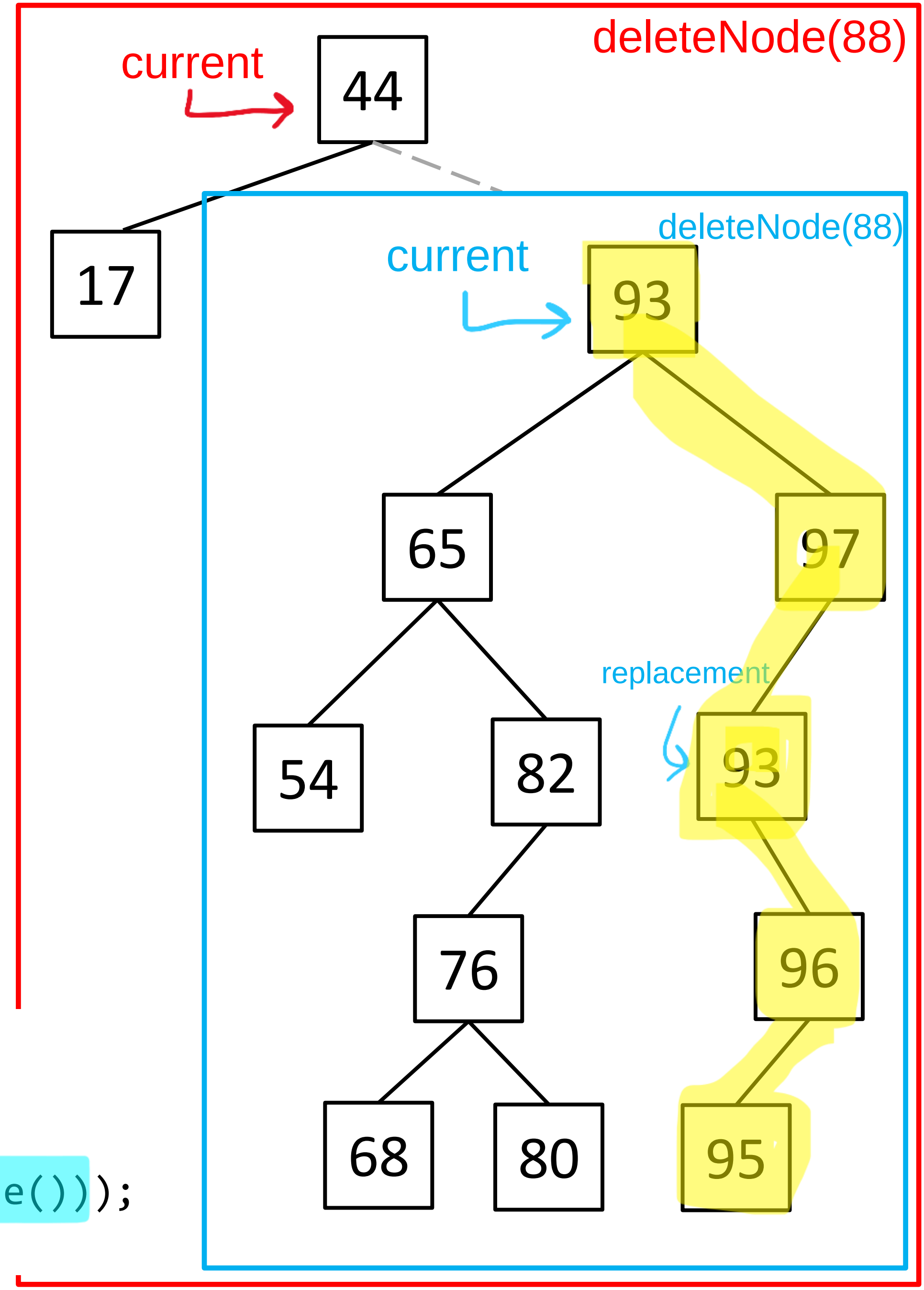


```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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

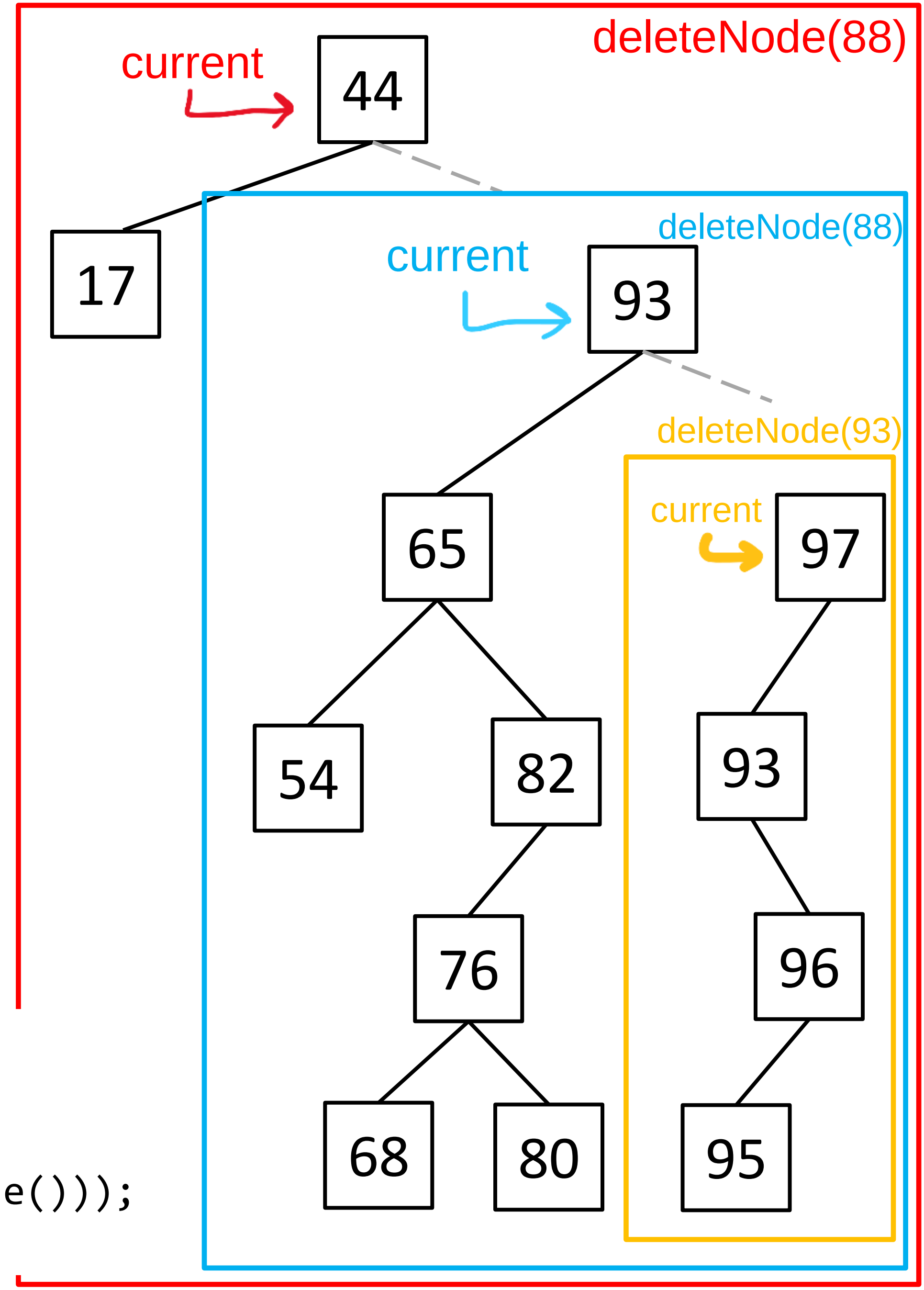


```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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

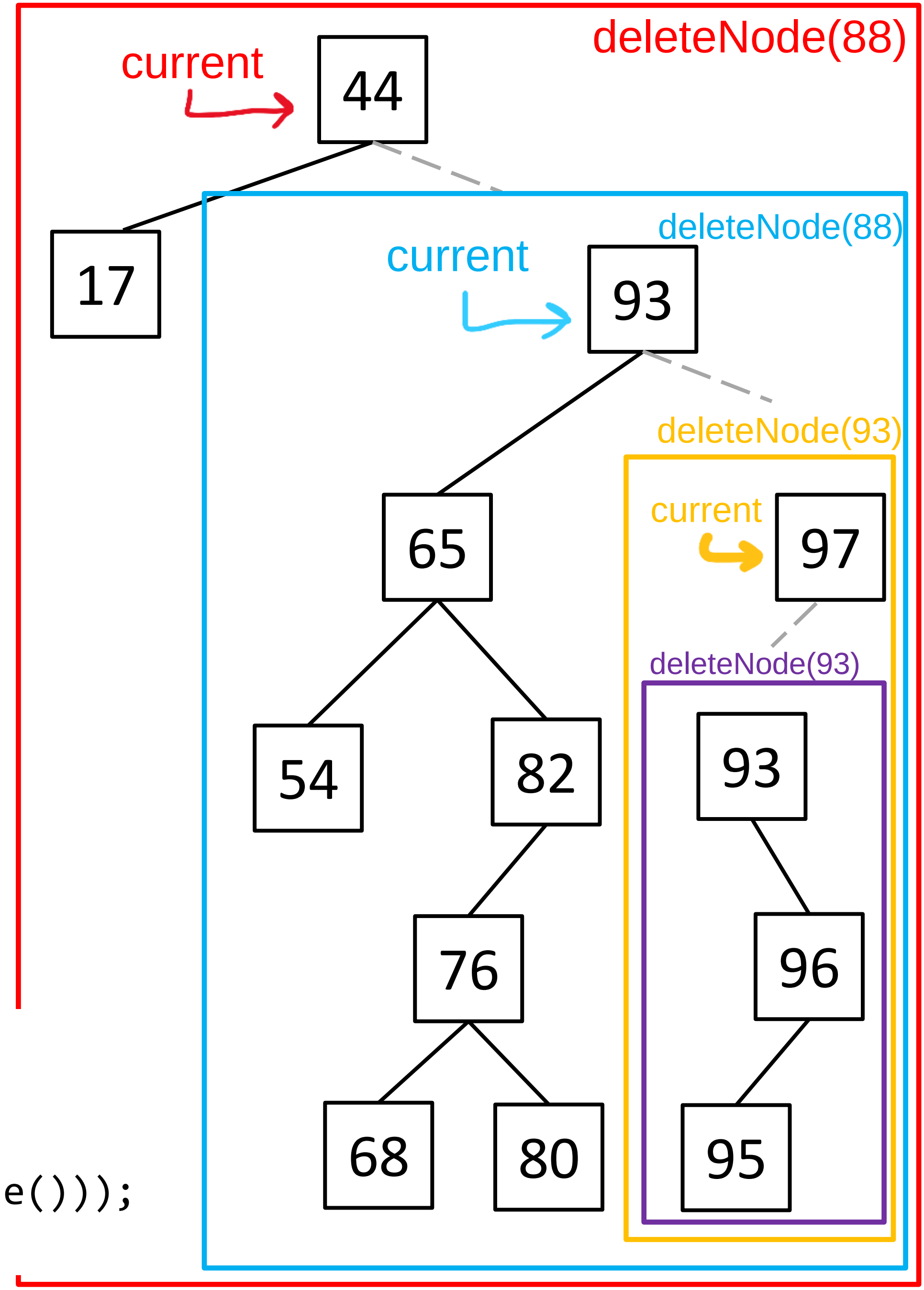
Recursively go through the right subtree and remove the duplicate value



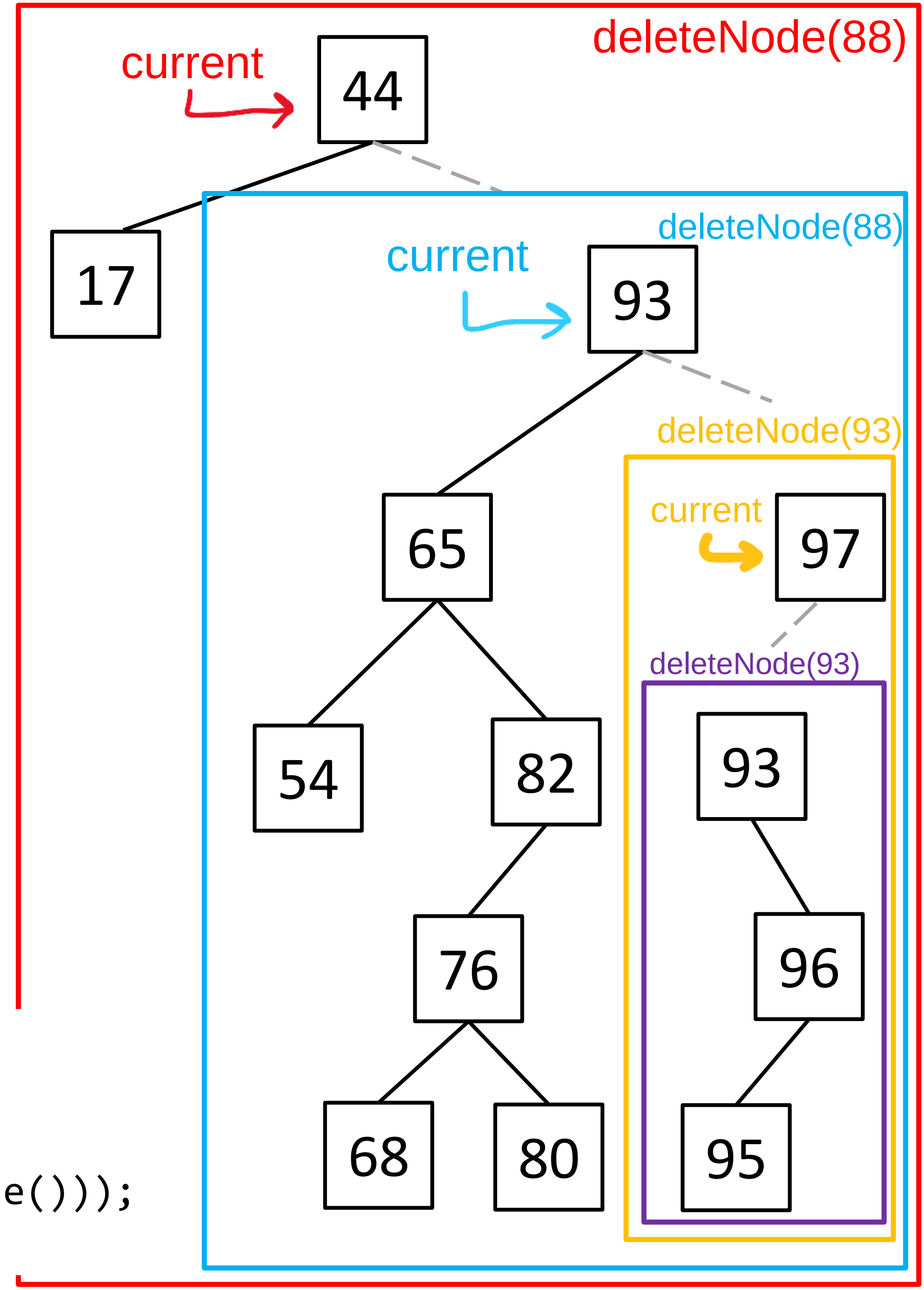
```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



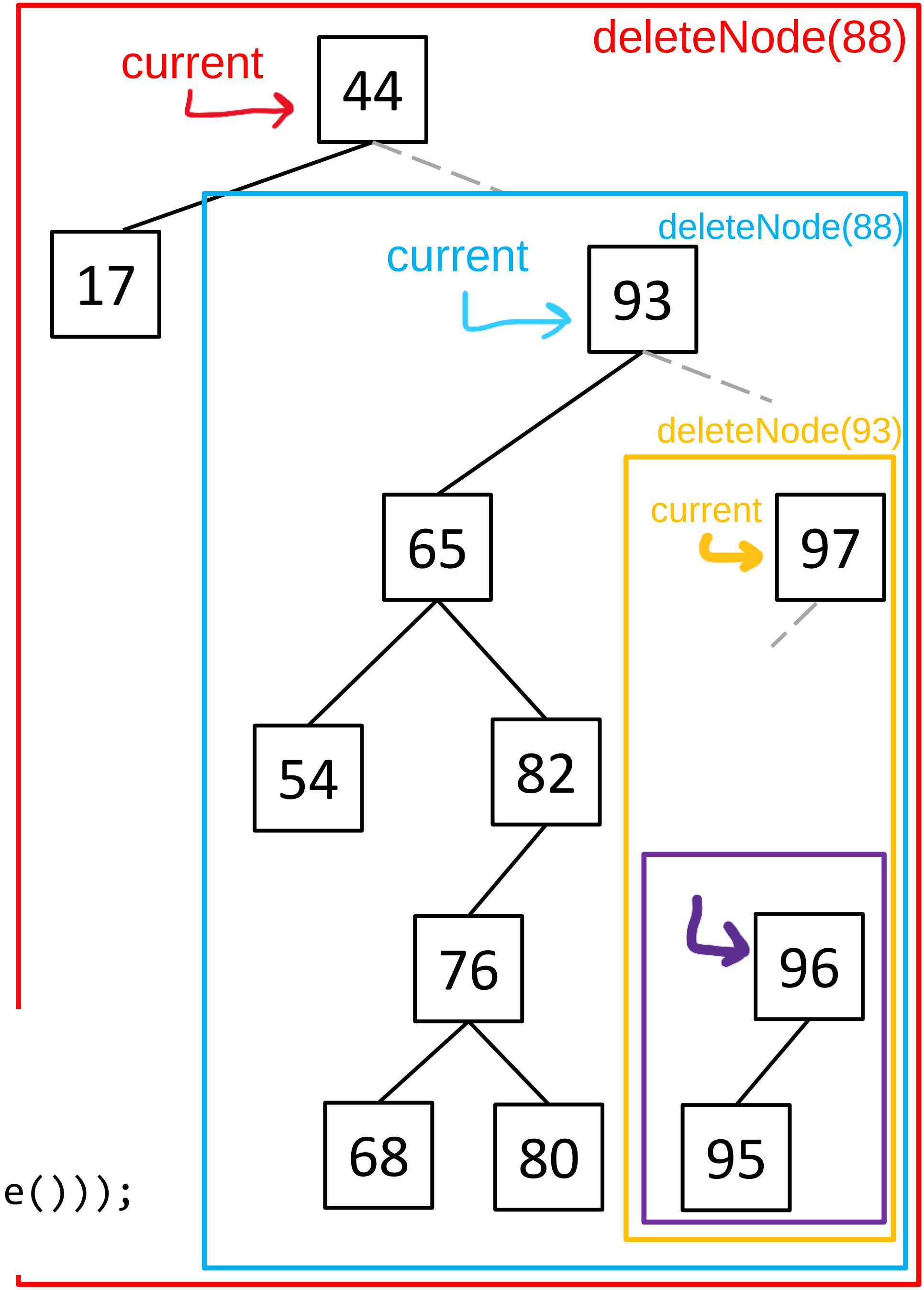
```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



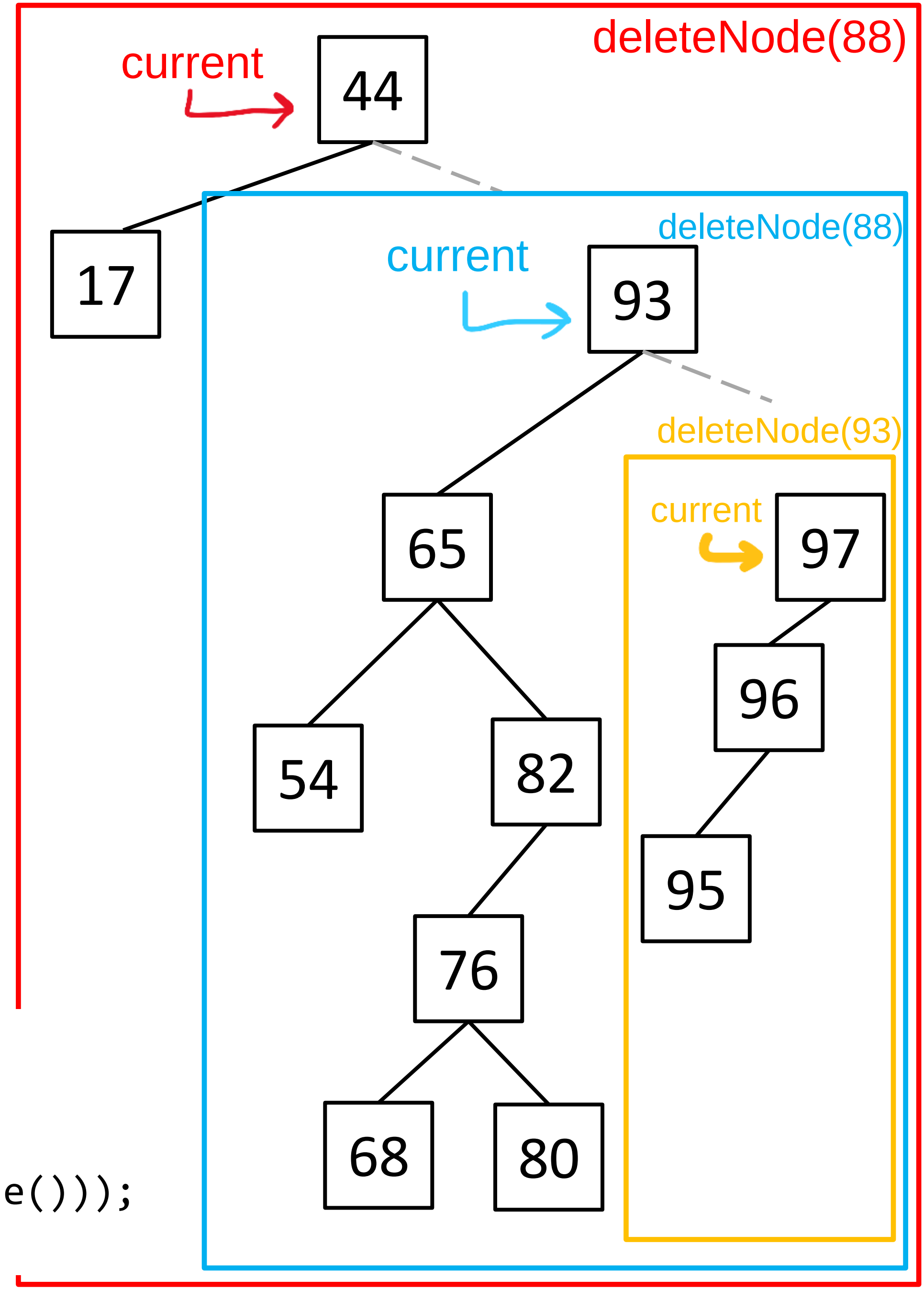
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



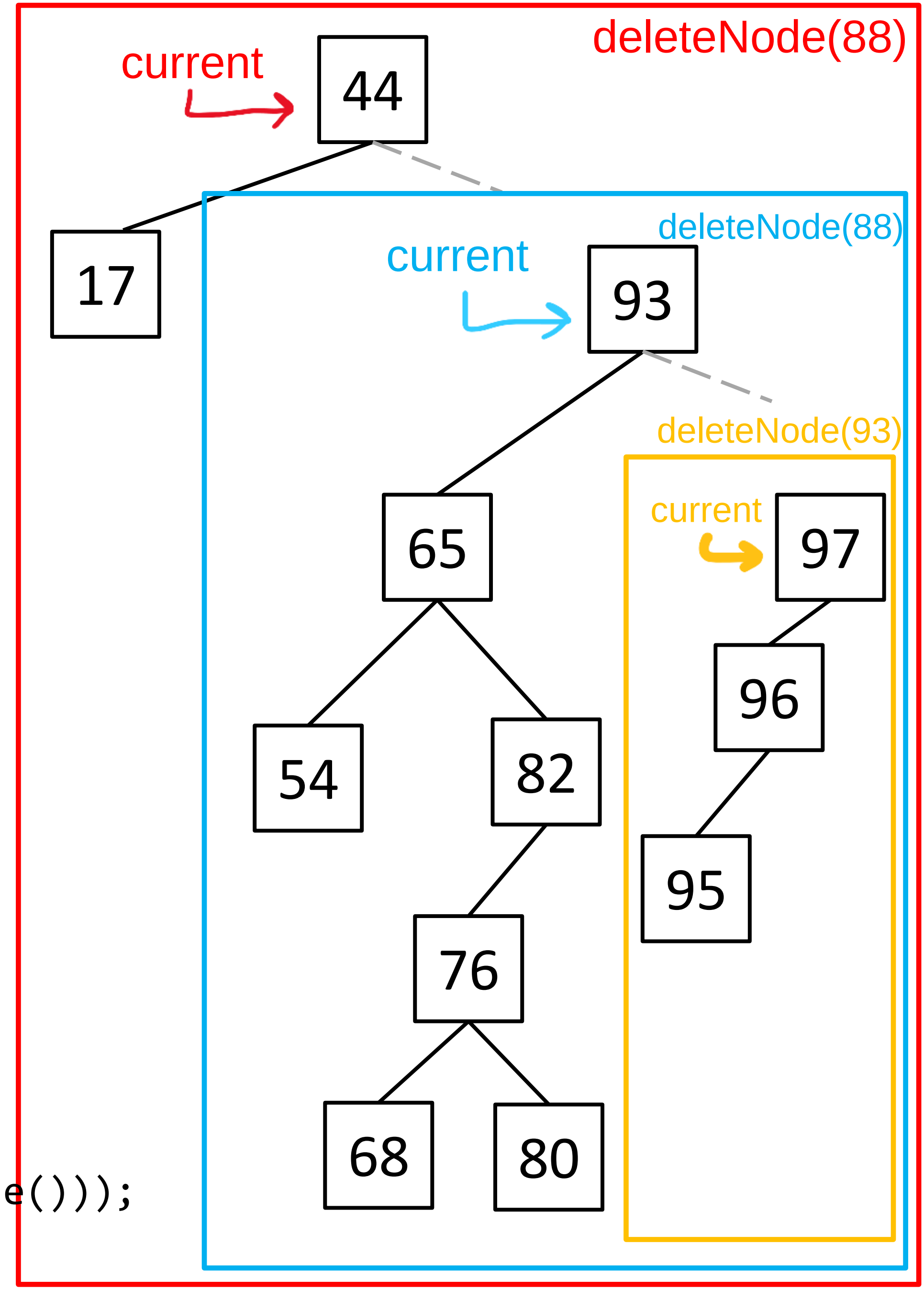
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



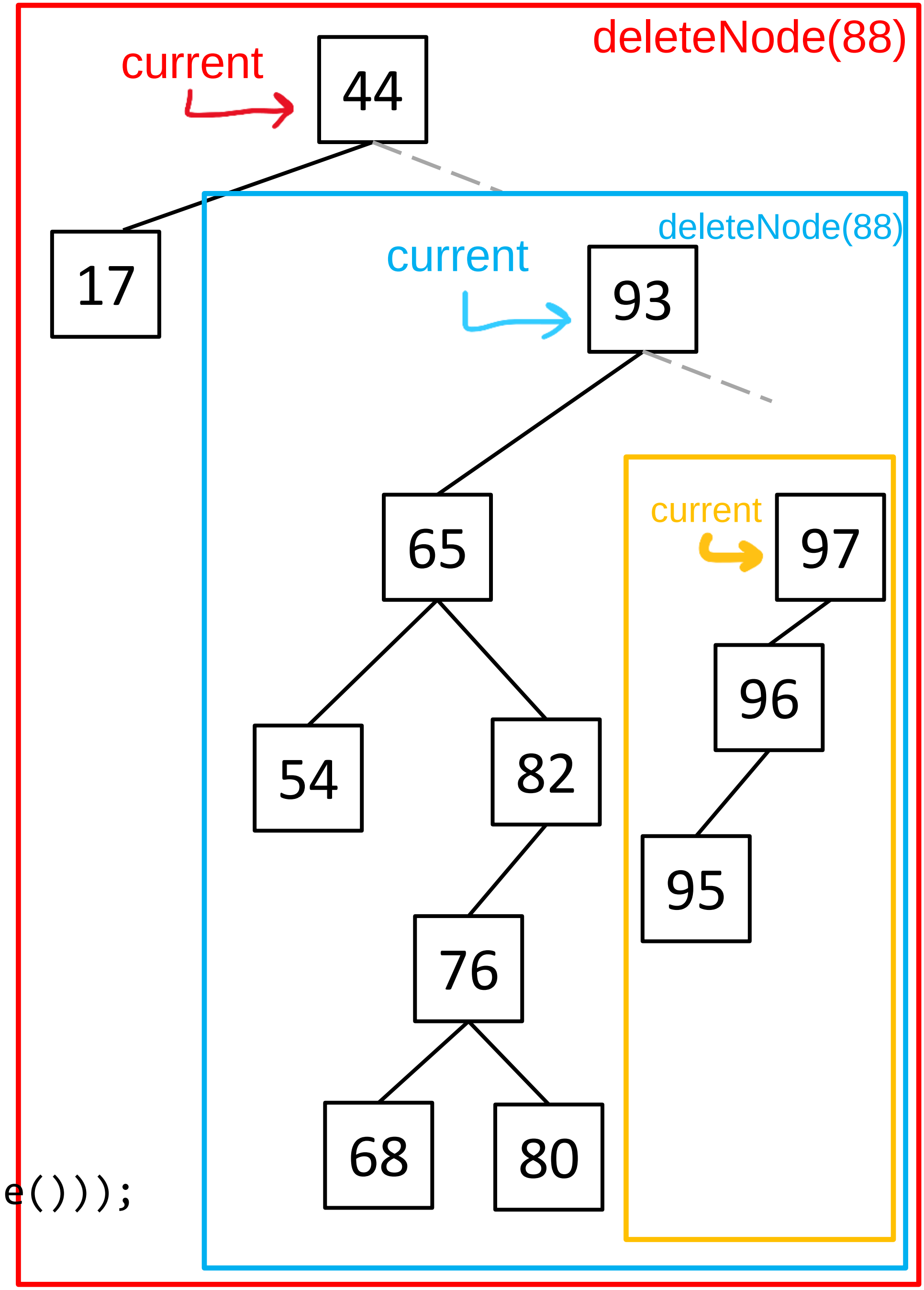
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



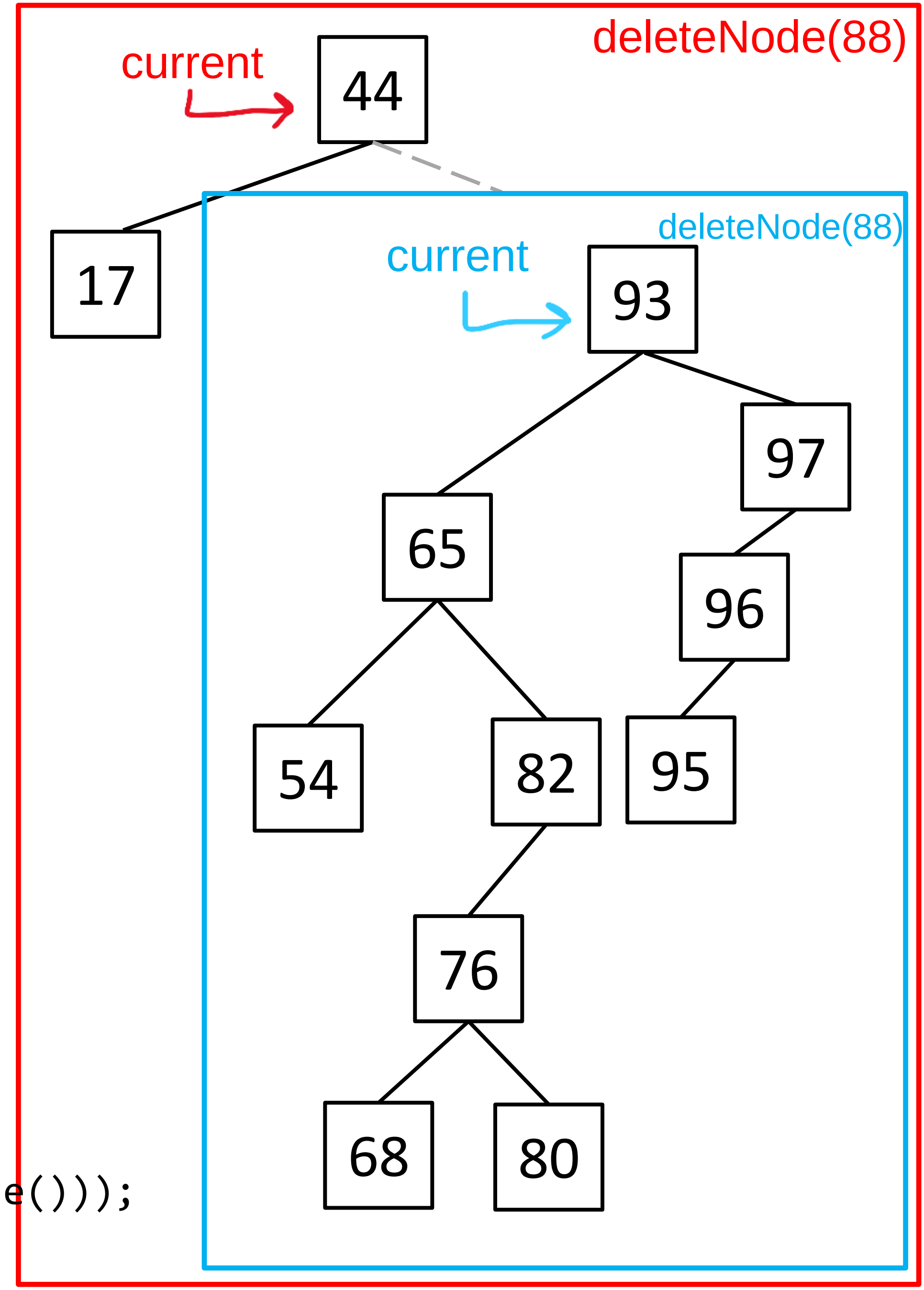
```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



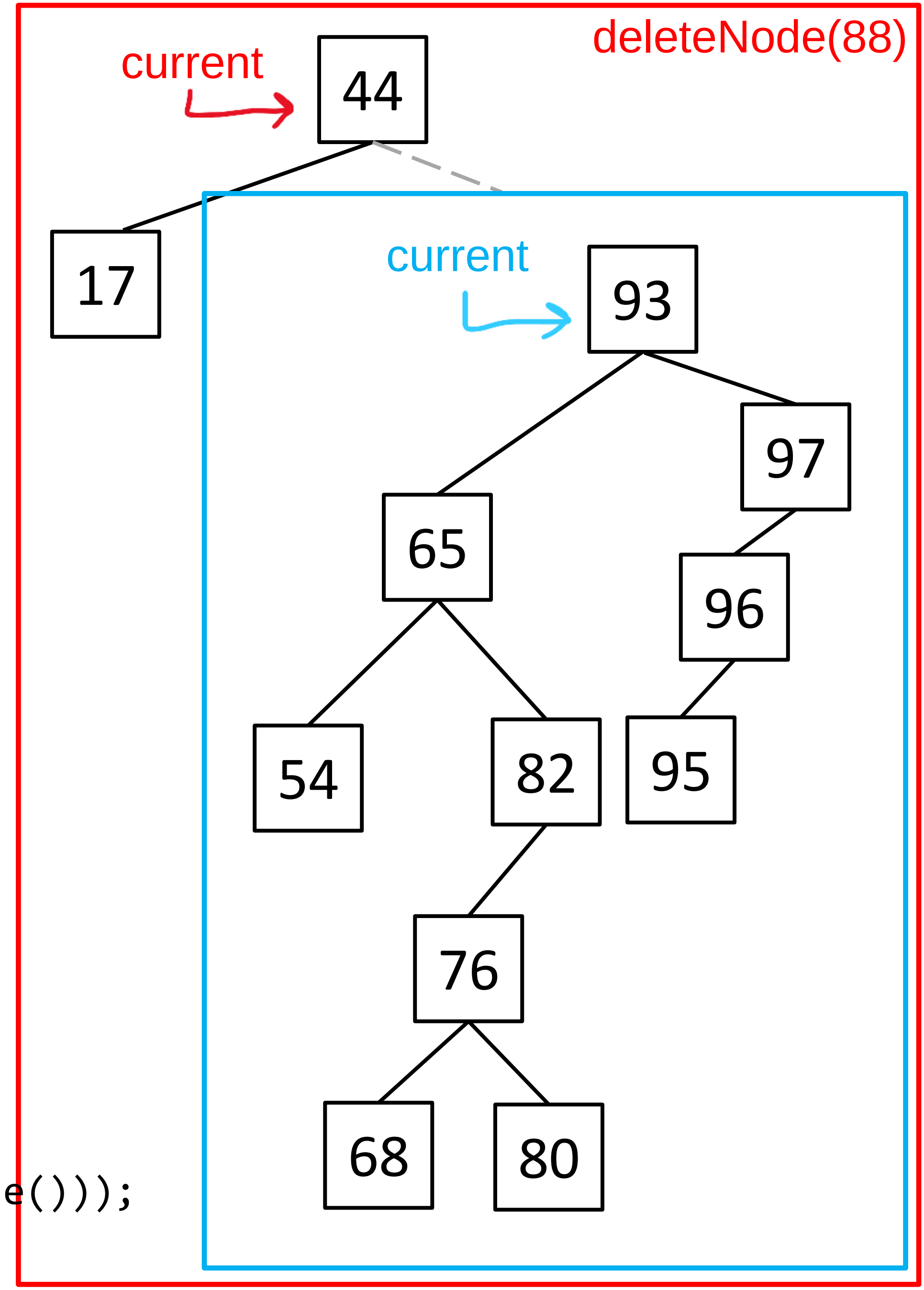
```java
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



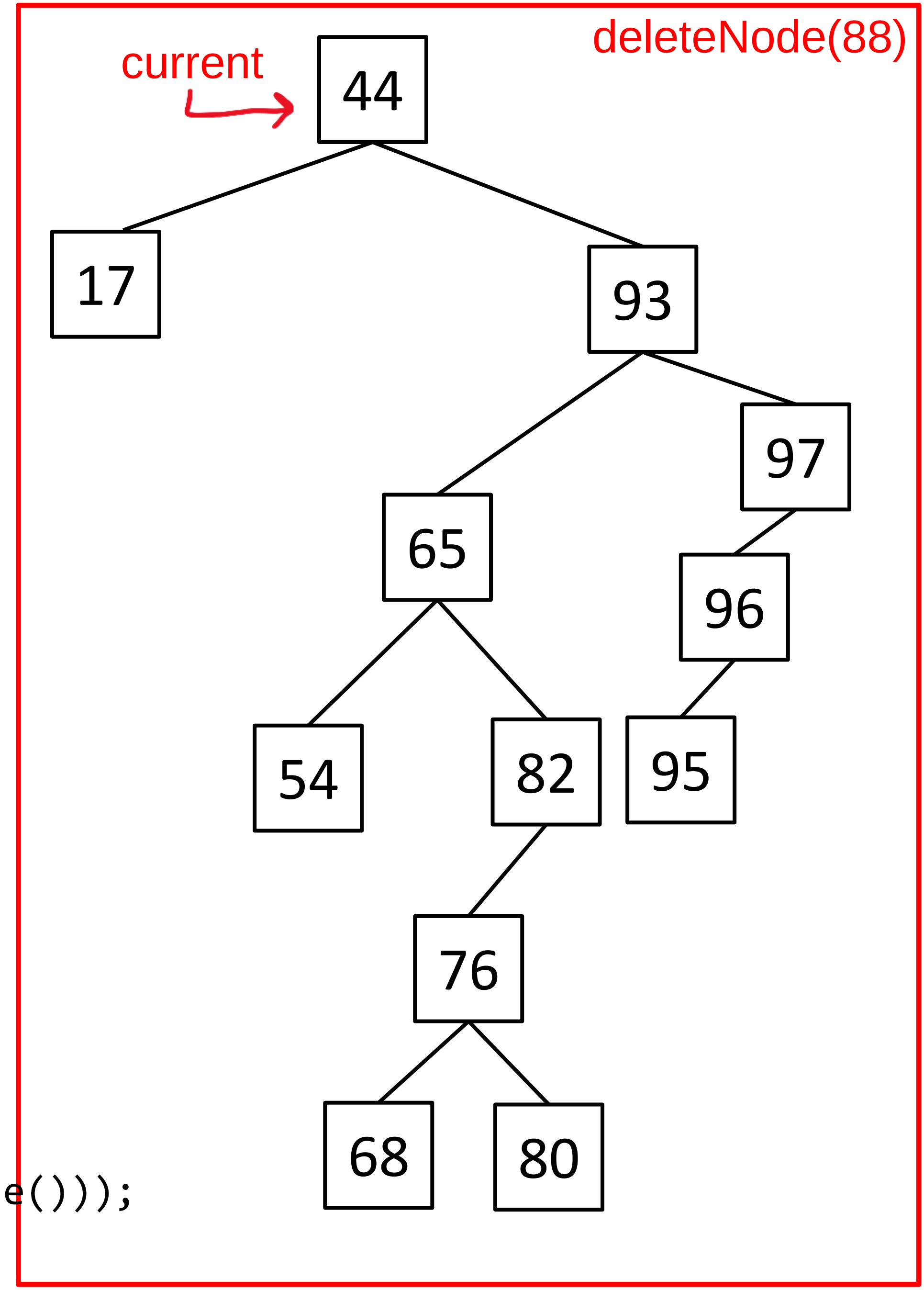
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



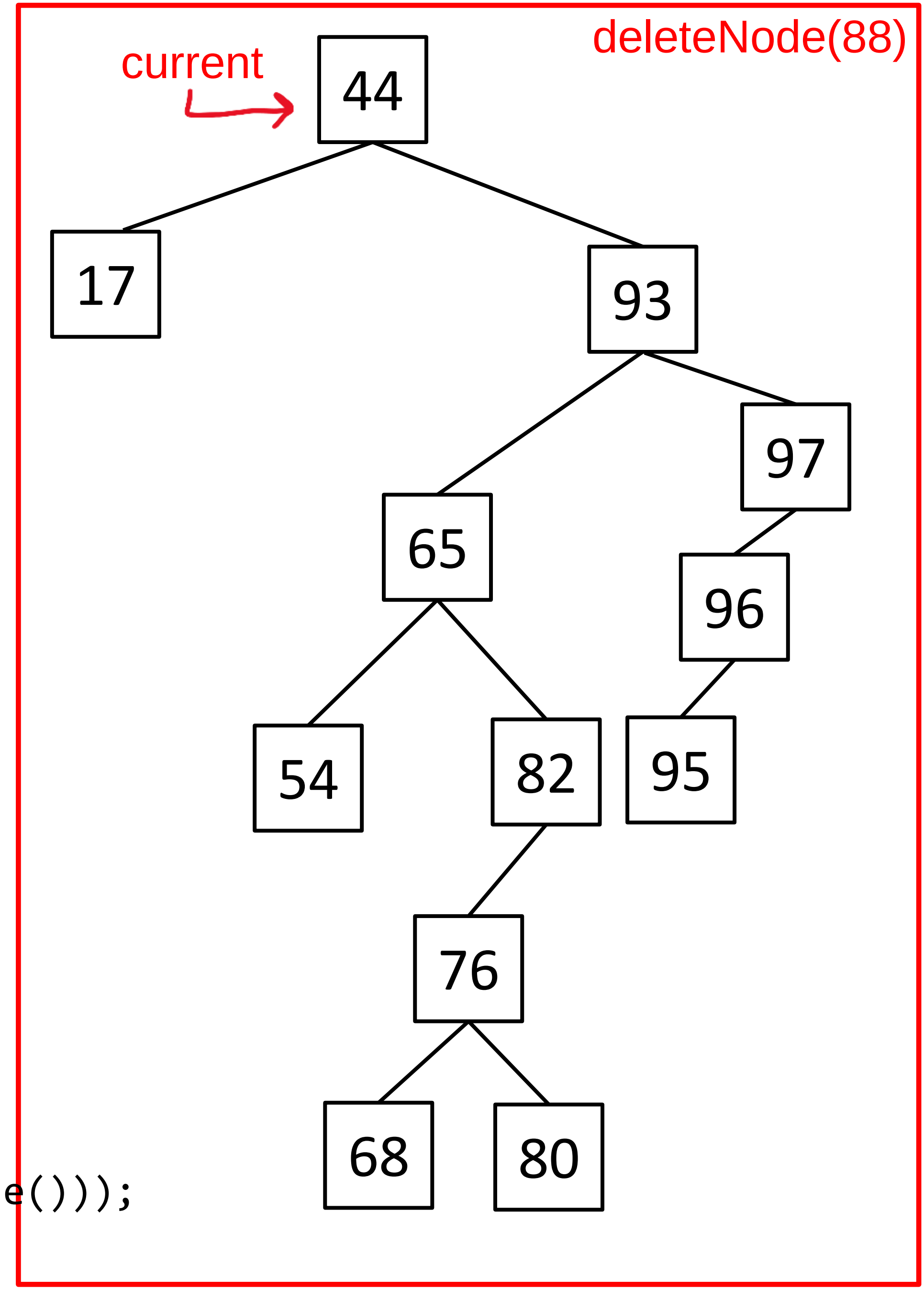
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



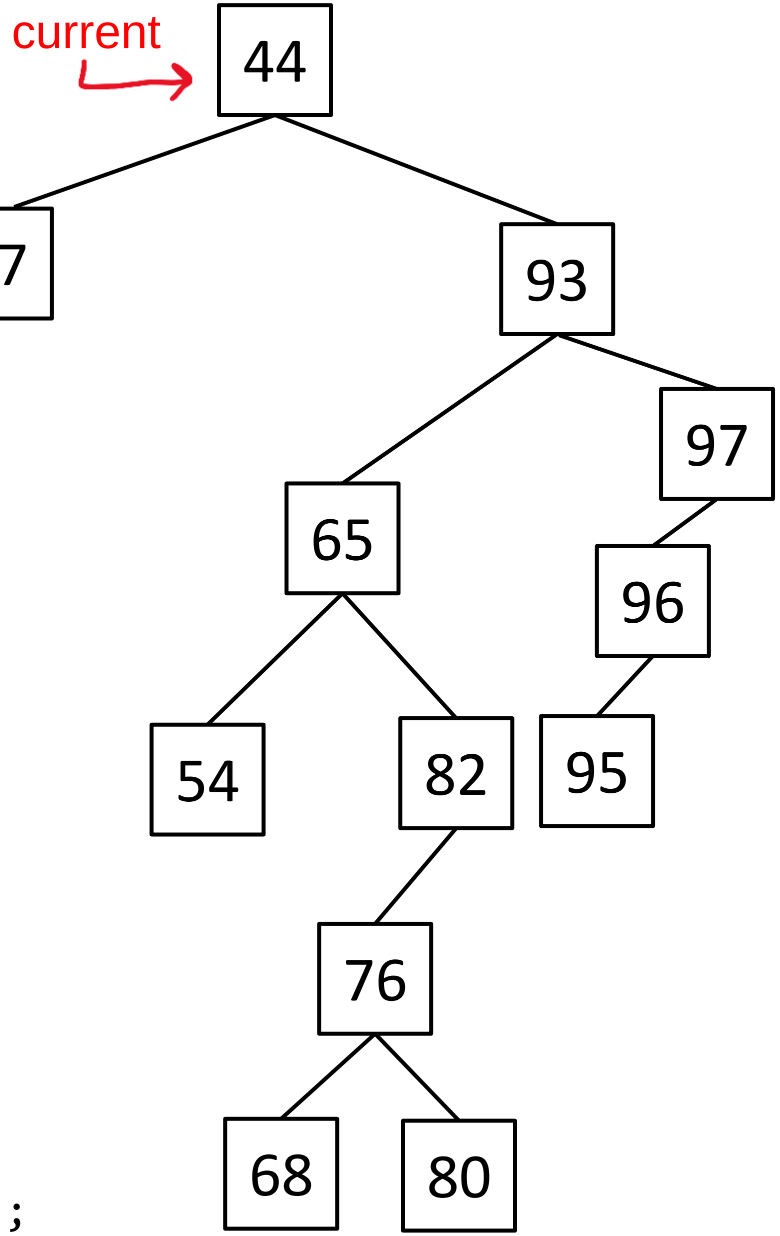
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



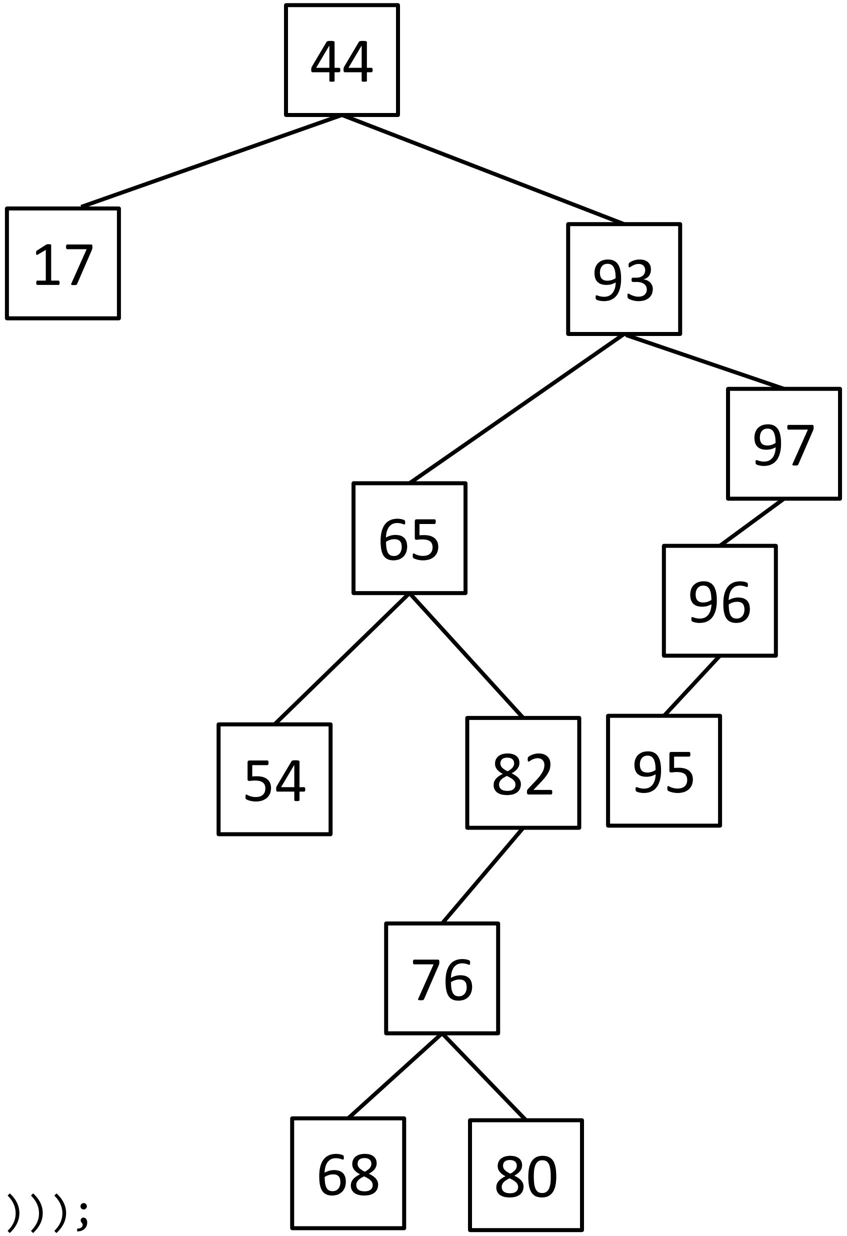
```
public Node deleteNode(Node current, int searchValue) {
 if (current == null) {
   return current;
 }
 if (current.getValue() > searchValue) {
   Current.setLeft( deleteNode(current.getLeft(), searchValue));
 }
 else if (current.getValue() < searchValue) {
   current.setRight( deleteNode(current.getRight(),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



```java
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;
```



```java
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;
```

1. **Find value we are searching for (recursively)**

```java
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**

```java
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(),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**

4. **Copy replacement value into removed node**

```java
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**

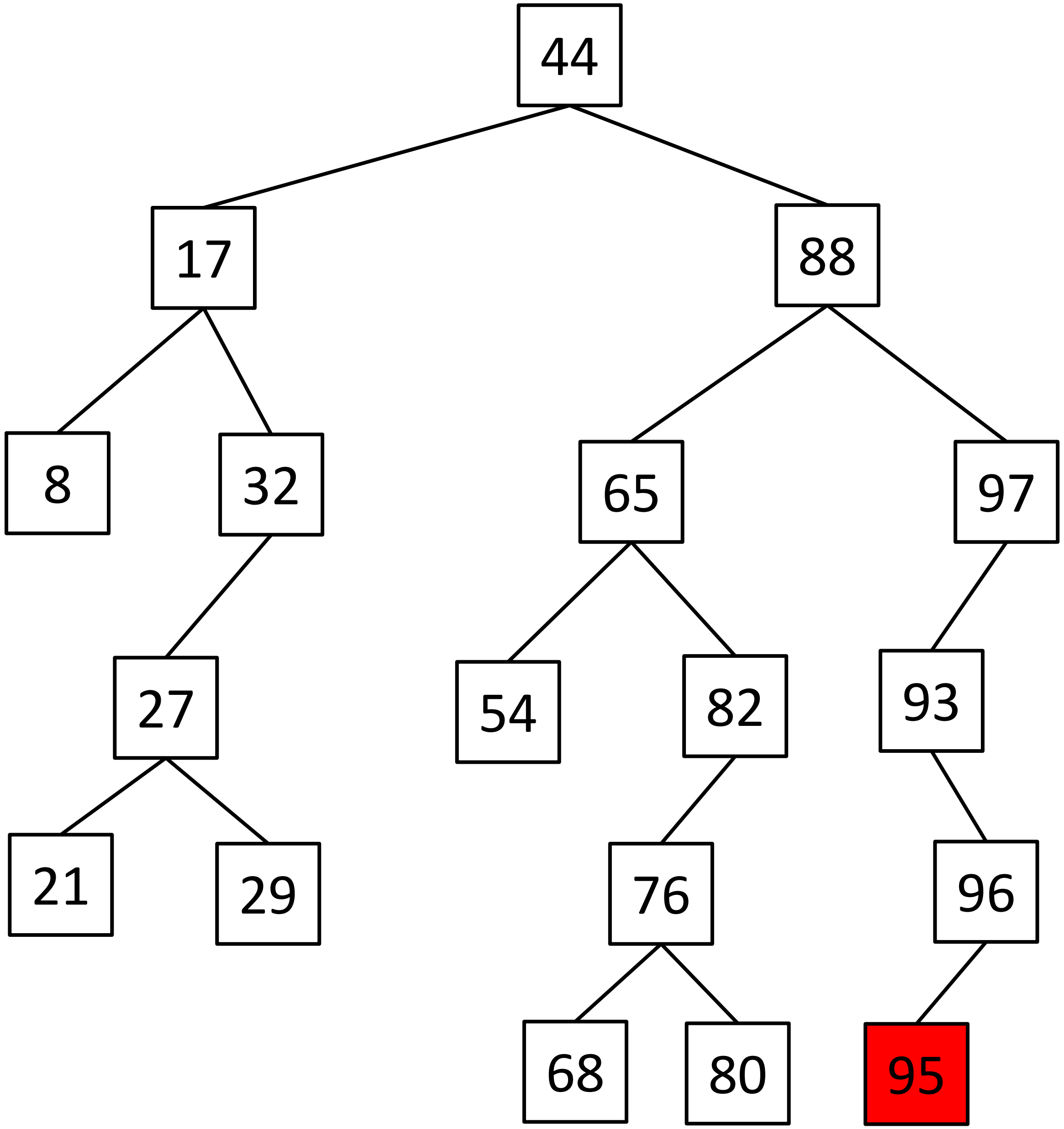
3. **(In case of 2 children) Find replacement/successor of removed node**

4. **Copy replacement value into removed node**

5. **Remove duplicate**

```java
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;
```

Running time?



To remove a node, we may end up traversing the entire tree

If this is a *balanced* tree, to traverse the height of tree is O(logn)

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

Sorted Array

Operation	Running Time
Insertion	O(n)
Removal	O(n)
Searching	O(logn)
Printing	O(n)

LinkedList

Operation	Running Time
Insertion	O(1)
Removal (by element)	O(n)
Searching	O(n)
Printing	O(n)

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal (by element)	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

Inserting/removal in a BST is faster than inserting into a sorted Array

Sorted Array

Operation	Running Time
Insertion	O(n) *(shifting elements)*
Removal (by element)	O(n) *(shifting elements)*
Searching	O(logn) *(binary search)*
Printing	O(n)

LinkedList

Operation	Running Time
Insertion	O(1)
Removal (by element)	O(n)
Searching	O(n)
Printing	O(n)

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal (by element)	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

While LinkedLists provide faster insertion times, navigating a Binary Search Tree is faster than a Linked List

(We don't really have a way to start at the "middle" node of a linked and do binary search)

Sorted Array

Operation	Running Time
Insertion	O(n)
Removal (by element)	O(n)
Searching	O(logn)
Printing	O(n)

LinkedList

Operation	Running Time
Insertion	O(1)
Removal (by element)	O(n) *(linear search)*
Searching	O(n) *(linear search)*
Printing	O(n)

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal (by element)	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

Which is the best tool for the job?

Depends on what you need!

`Sorted Array`

Operation	Running Time
Insertion	O(n)
Removal (by element)	O(n)
Searching	O(logn)
Printing	O(n)

`LinkedList`

Operation	Running Time
Insertion	O(1)
Removal (by element)	O(n)
Searching	O(n)
Printing	O(n)


VIBE CHECK

Lab 3