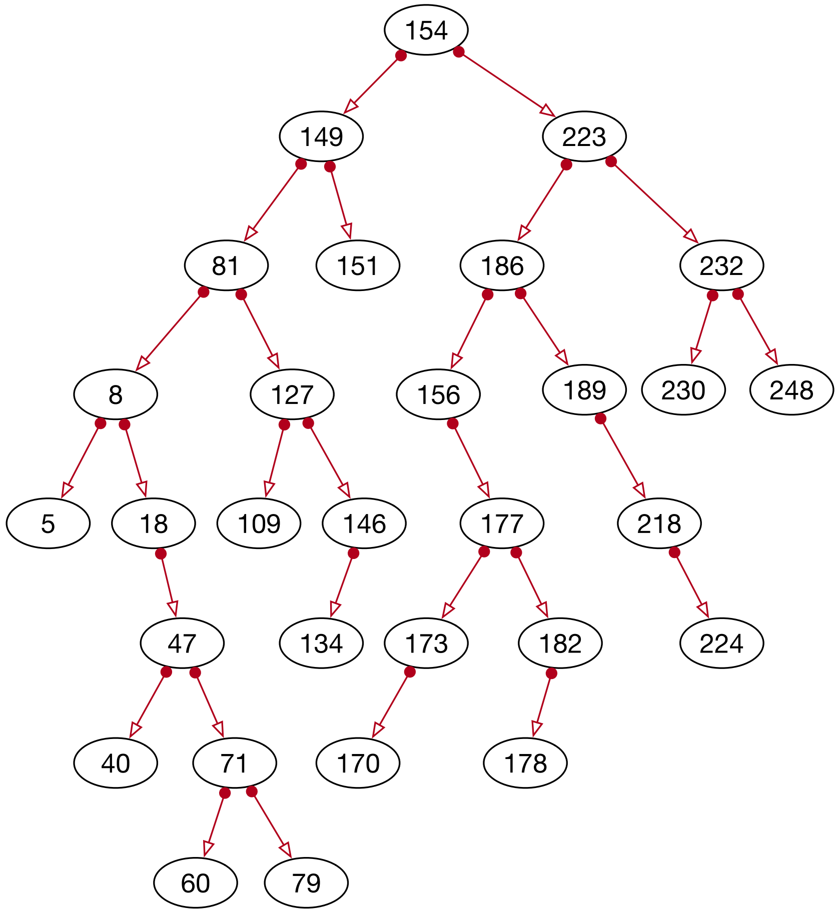
Binary Trees

Thomas Schwarz, SJ

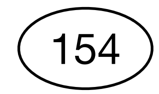
Definition

- Binary trees consists of nodes
  - Each node has:
    - A value (key -- record pair)
    - A left subnode
    - A right subnode
- In computer science, the root of a tree is on top

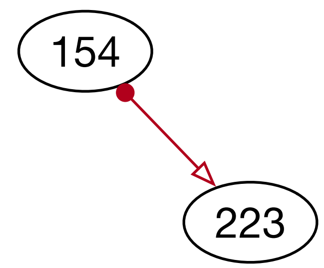


Example

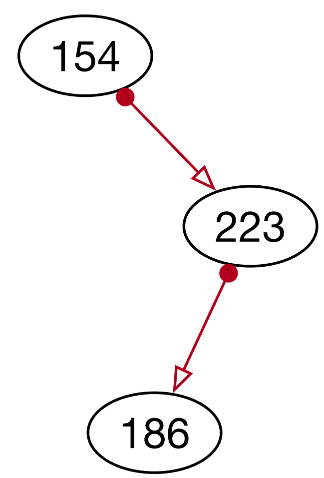
- Insert 154



- Insert 223

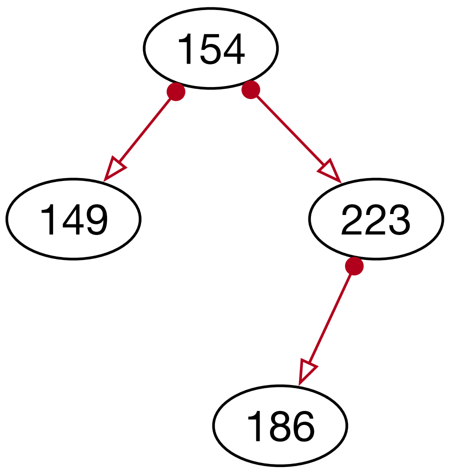


- Insert 186

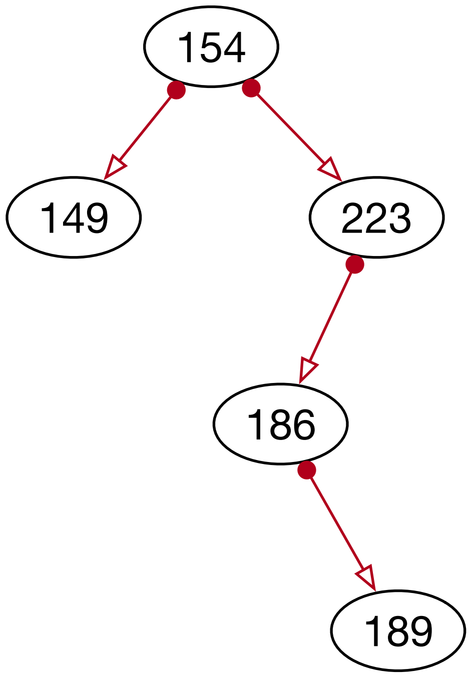


Example

- Insert 149

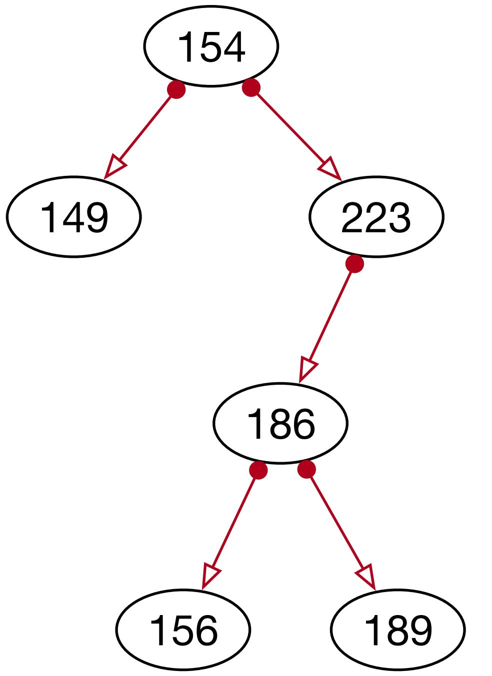


- Insert 189

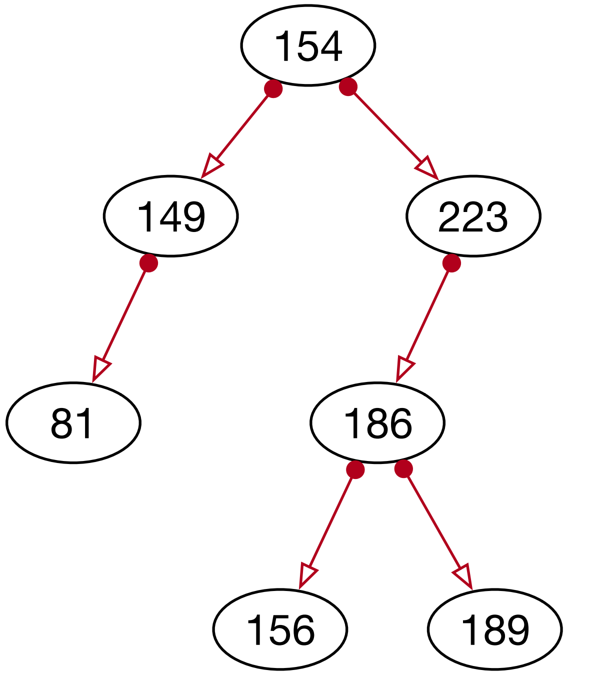


Example

- Insert 156

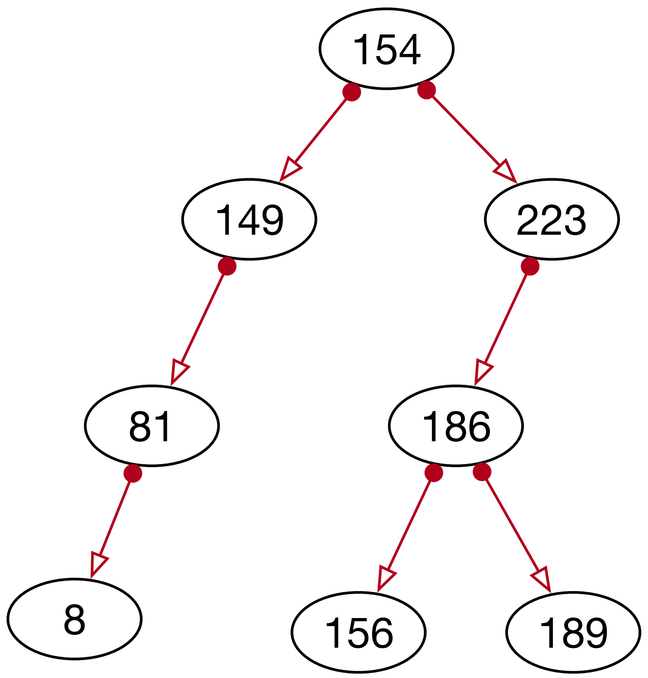


- Insert 81

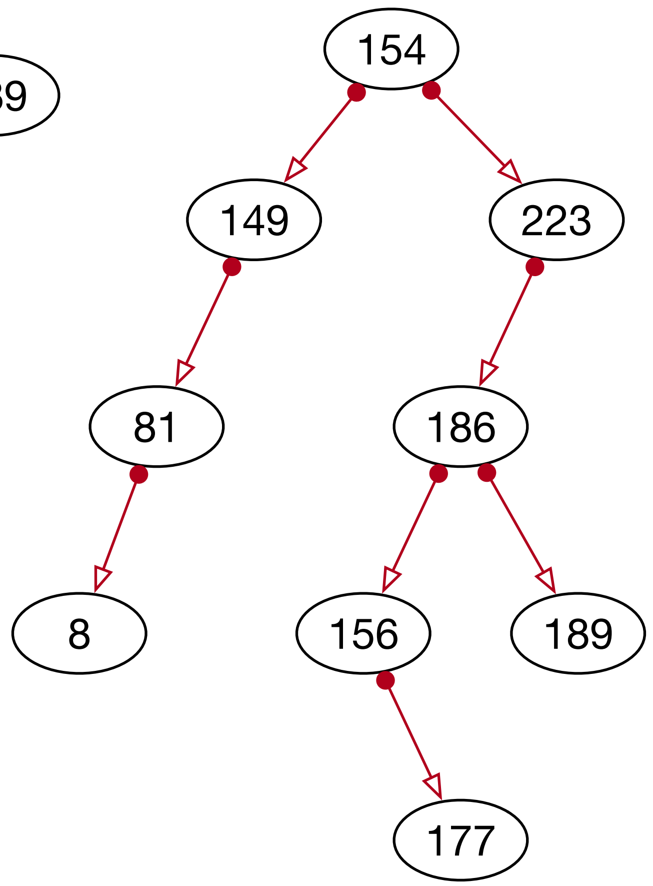


Example

- Insert 8

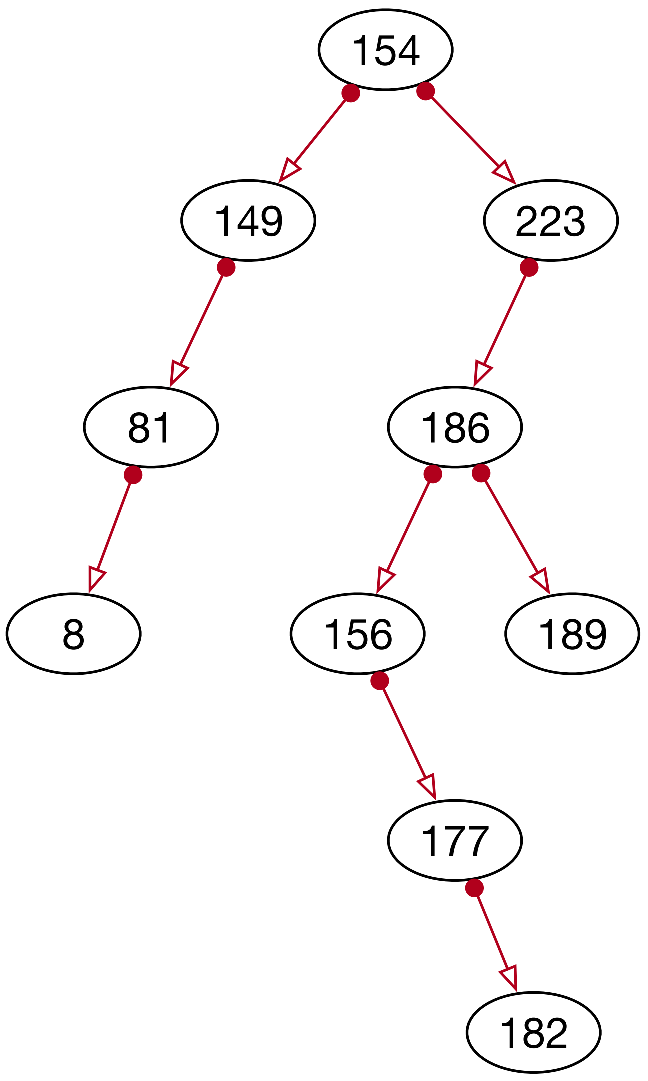


- Insert 177



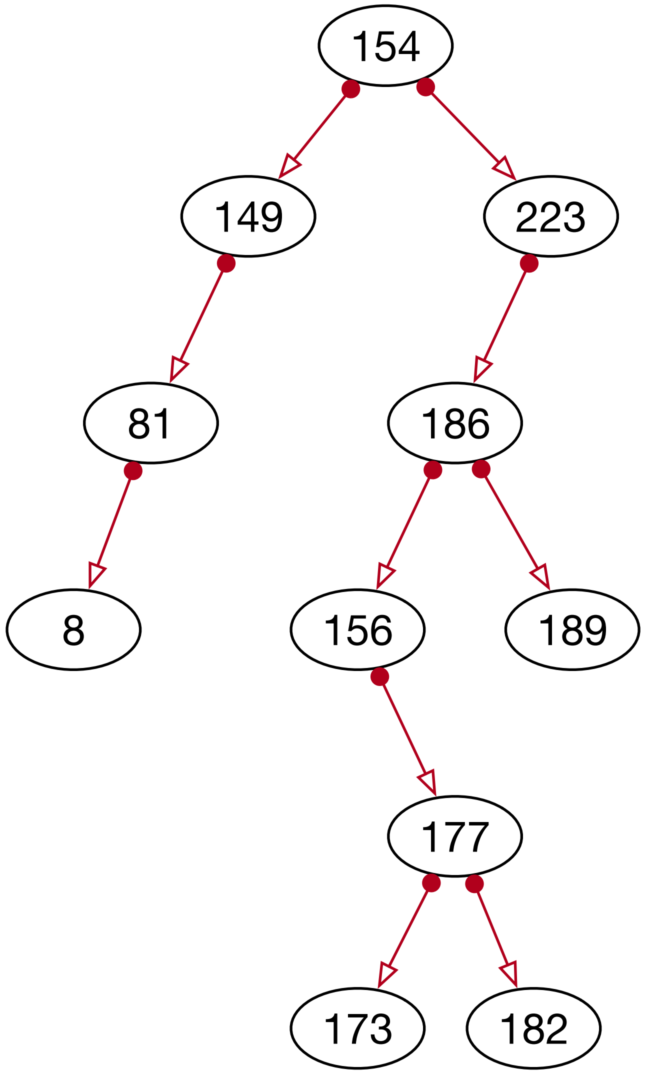
Example

- Insert 182



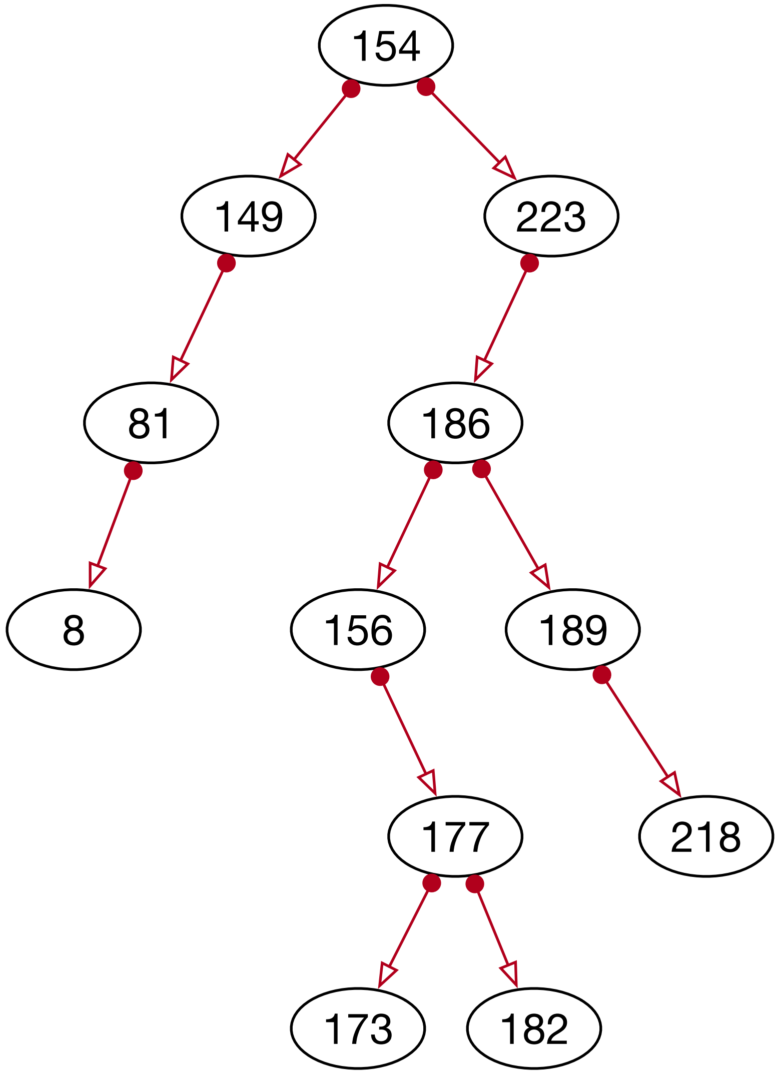
Example

- Insert 173



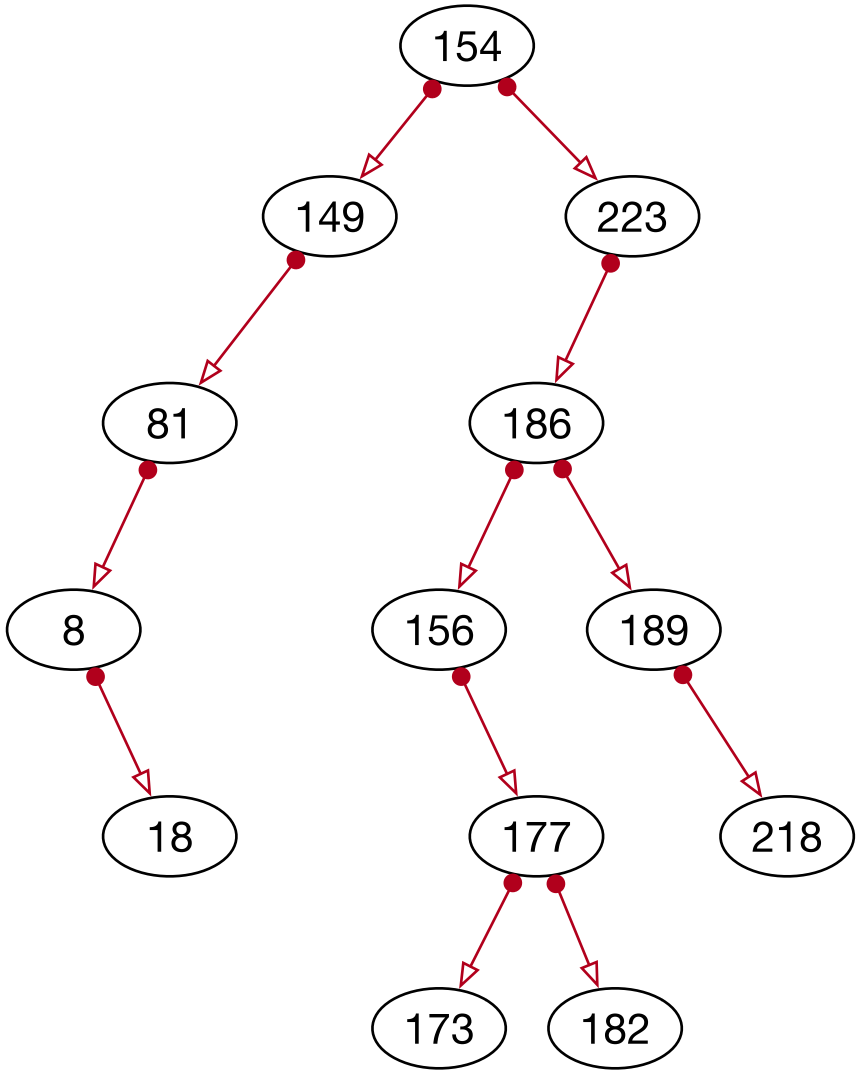
Example

- Insert 218



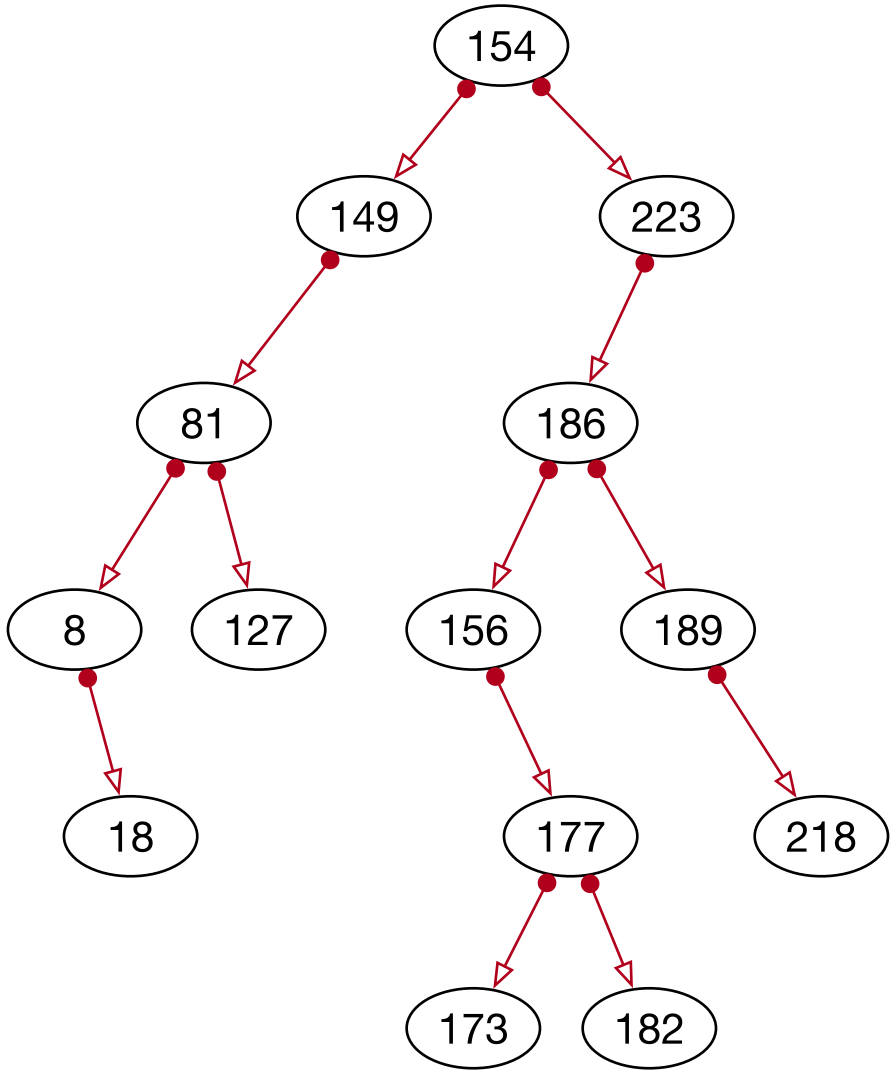
Example

- Insert 18



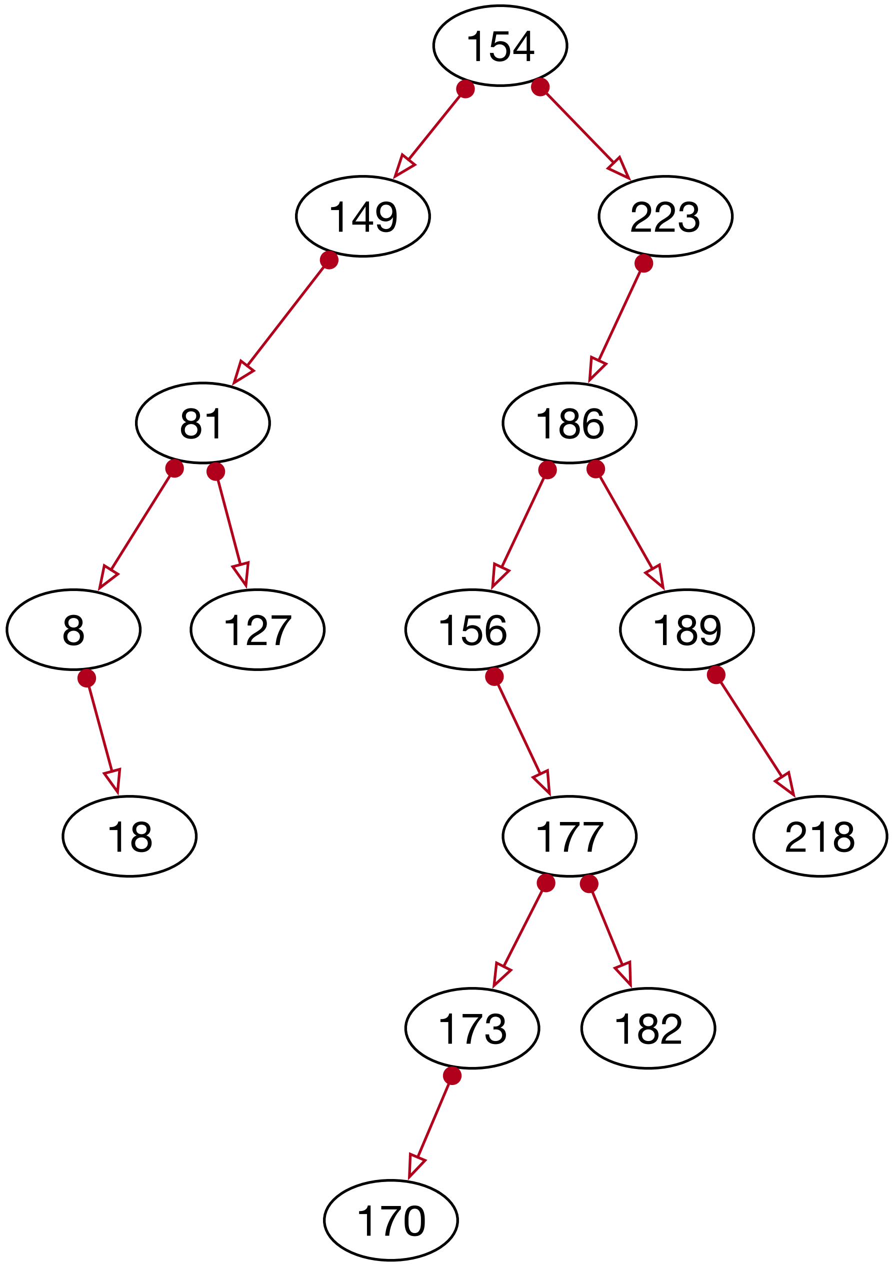
Example

- Insert 127



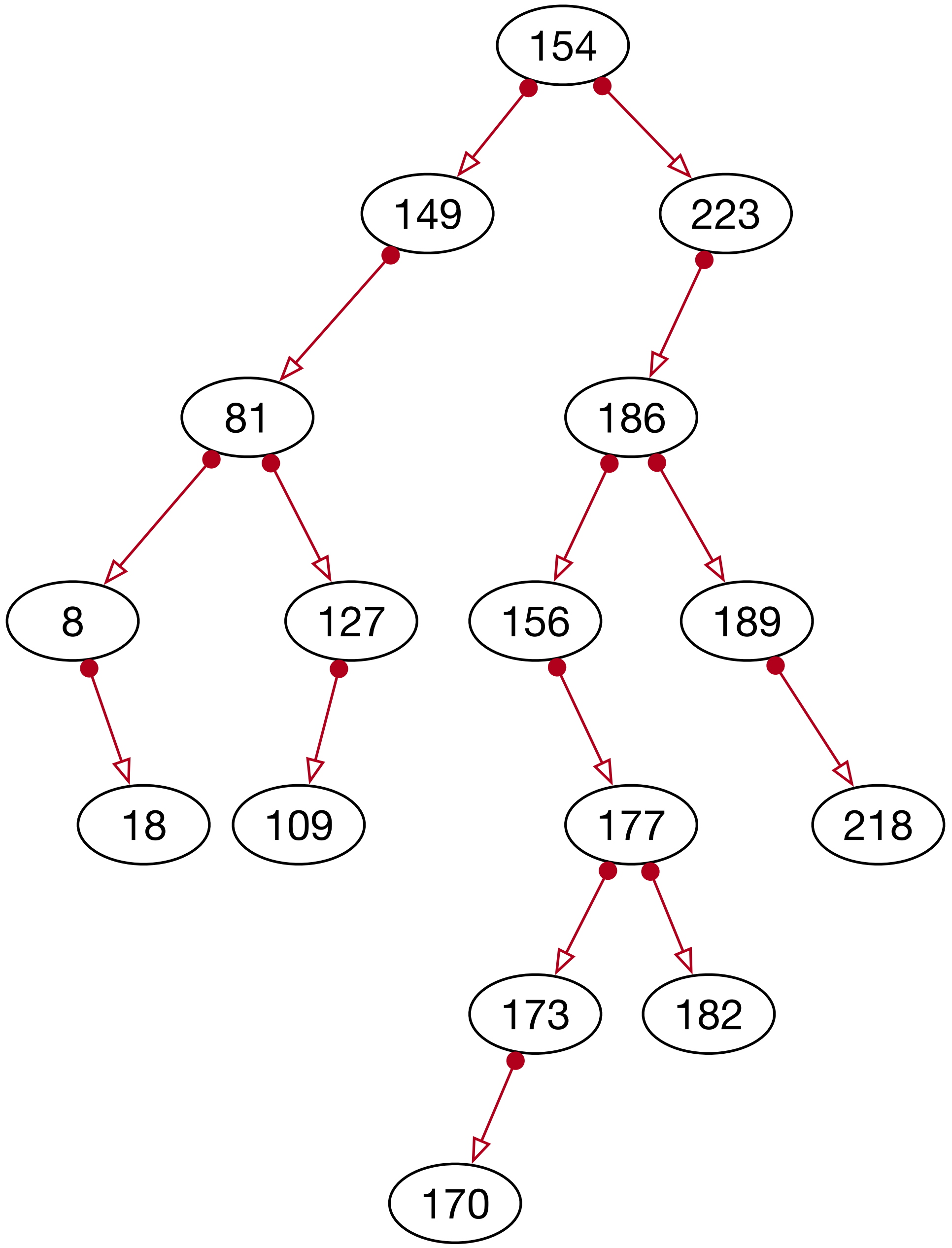
Example

- Insert 170



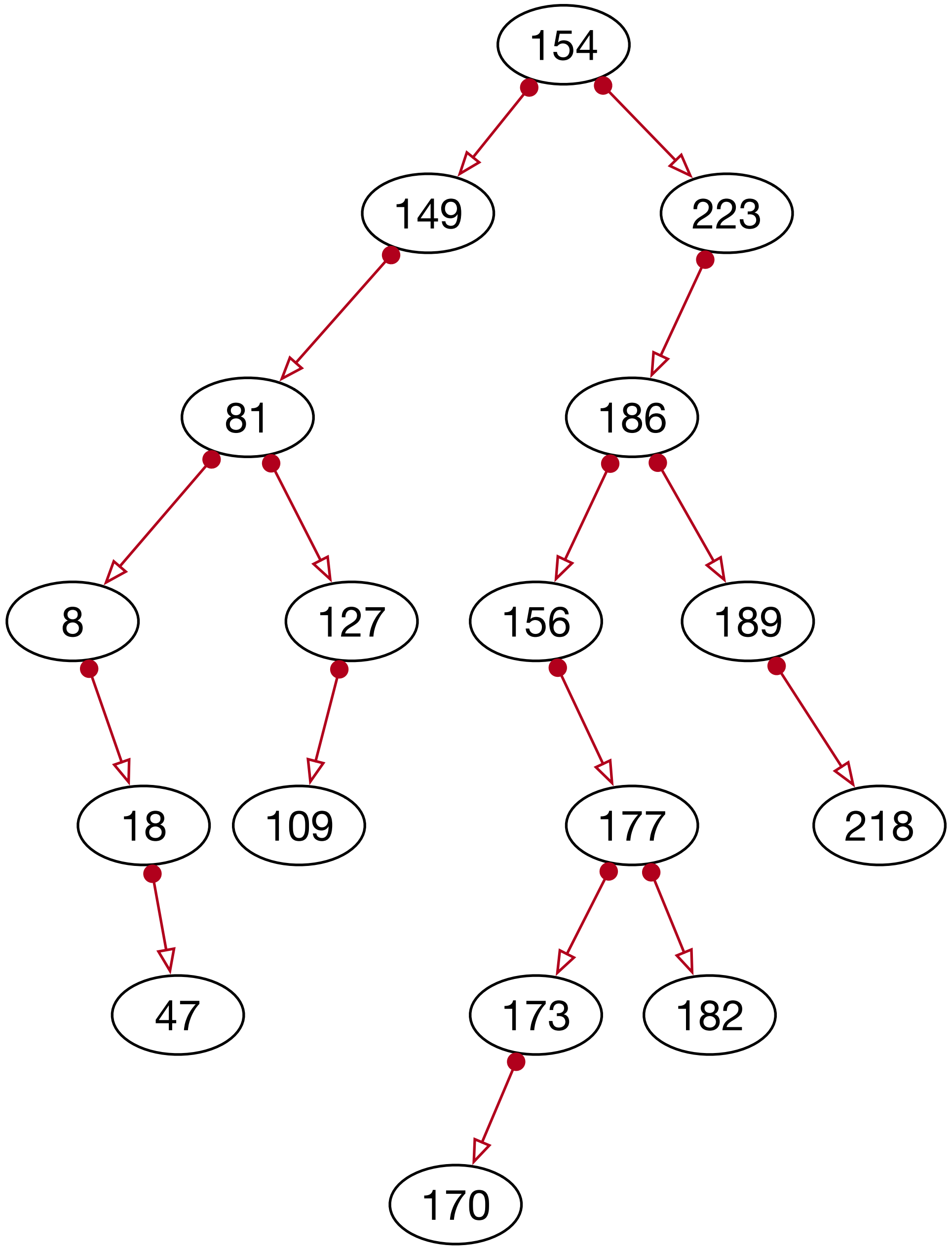
Example

- Insert 109



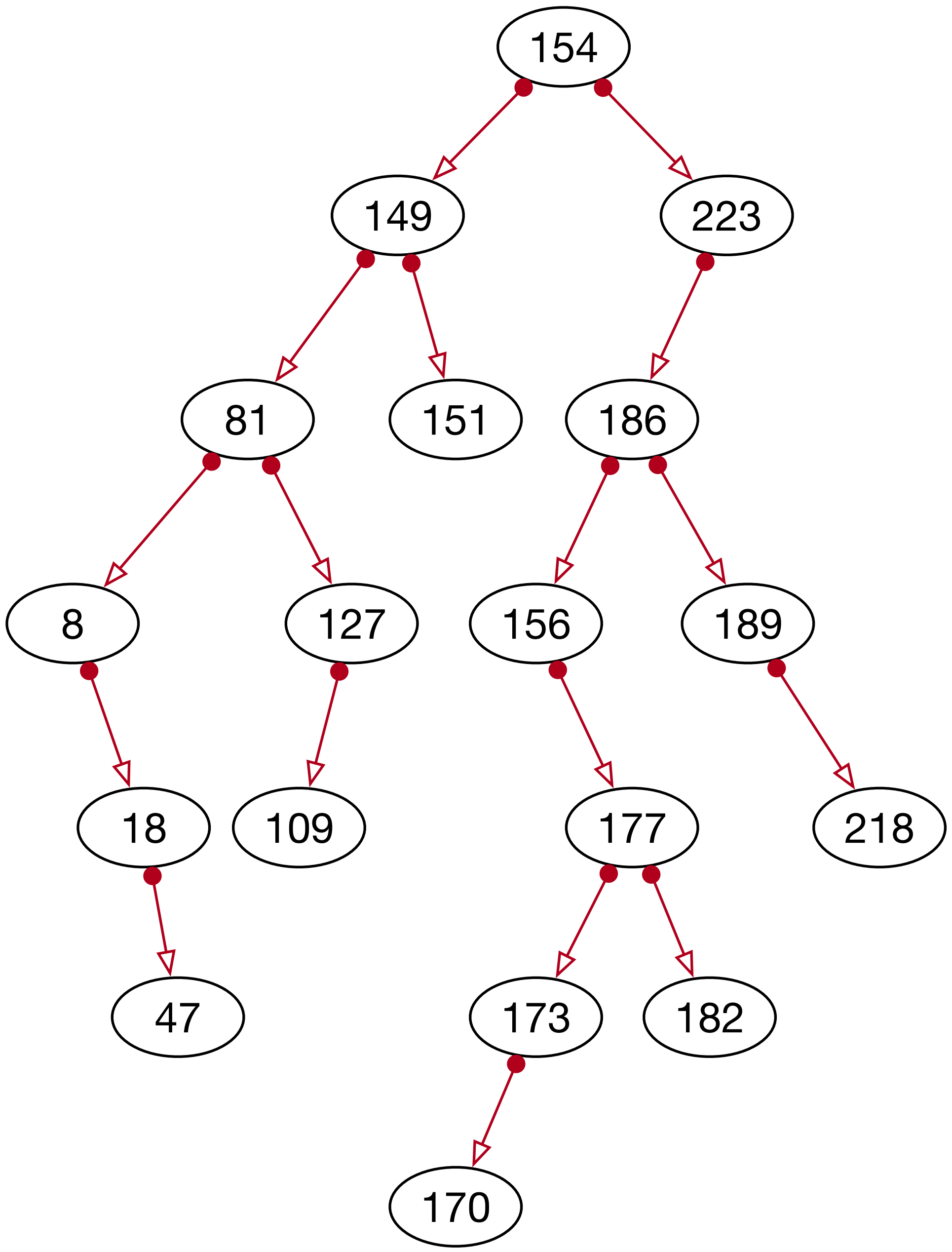
Example

- Insert 47



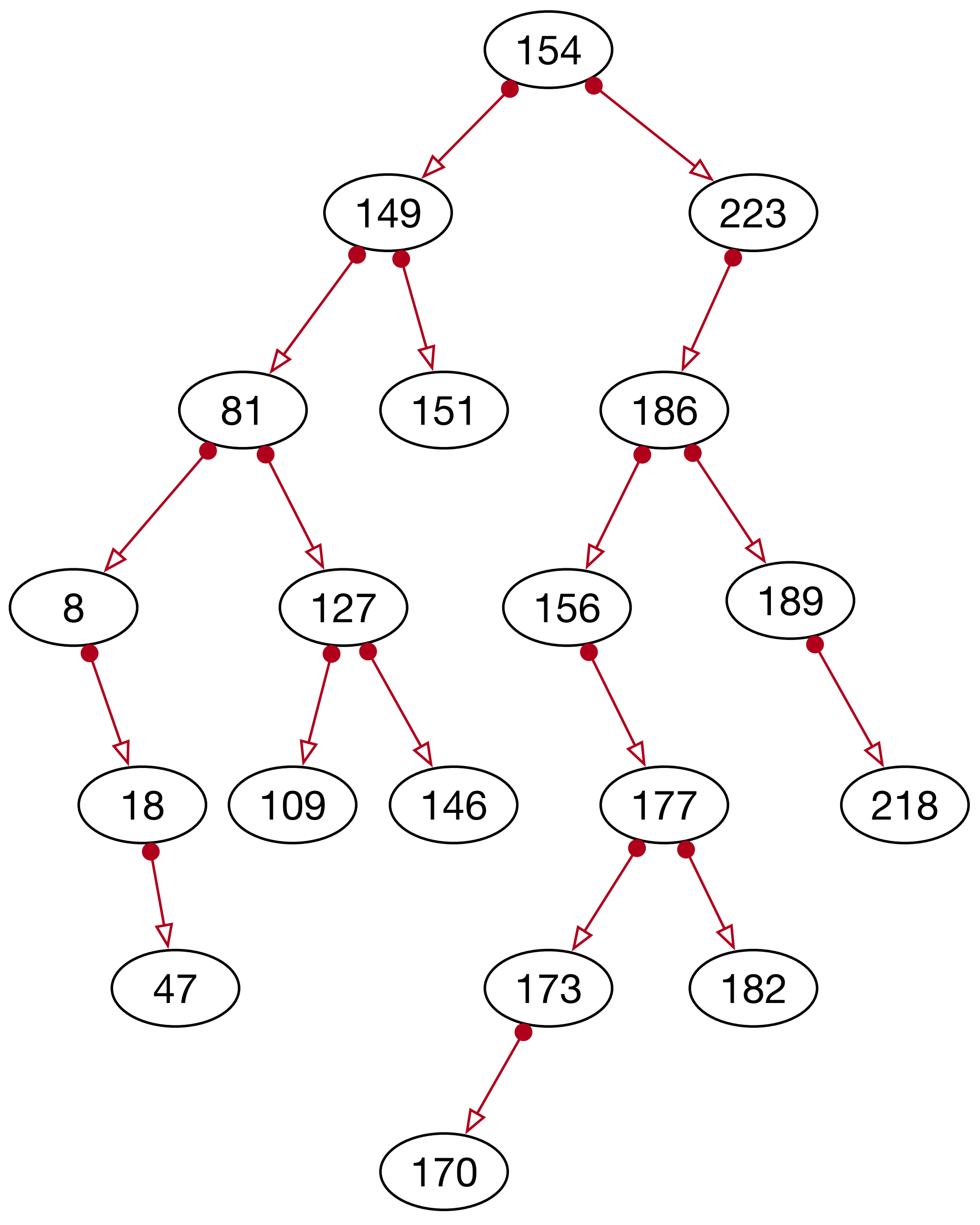
Example

- Insert 151



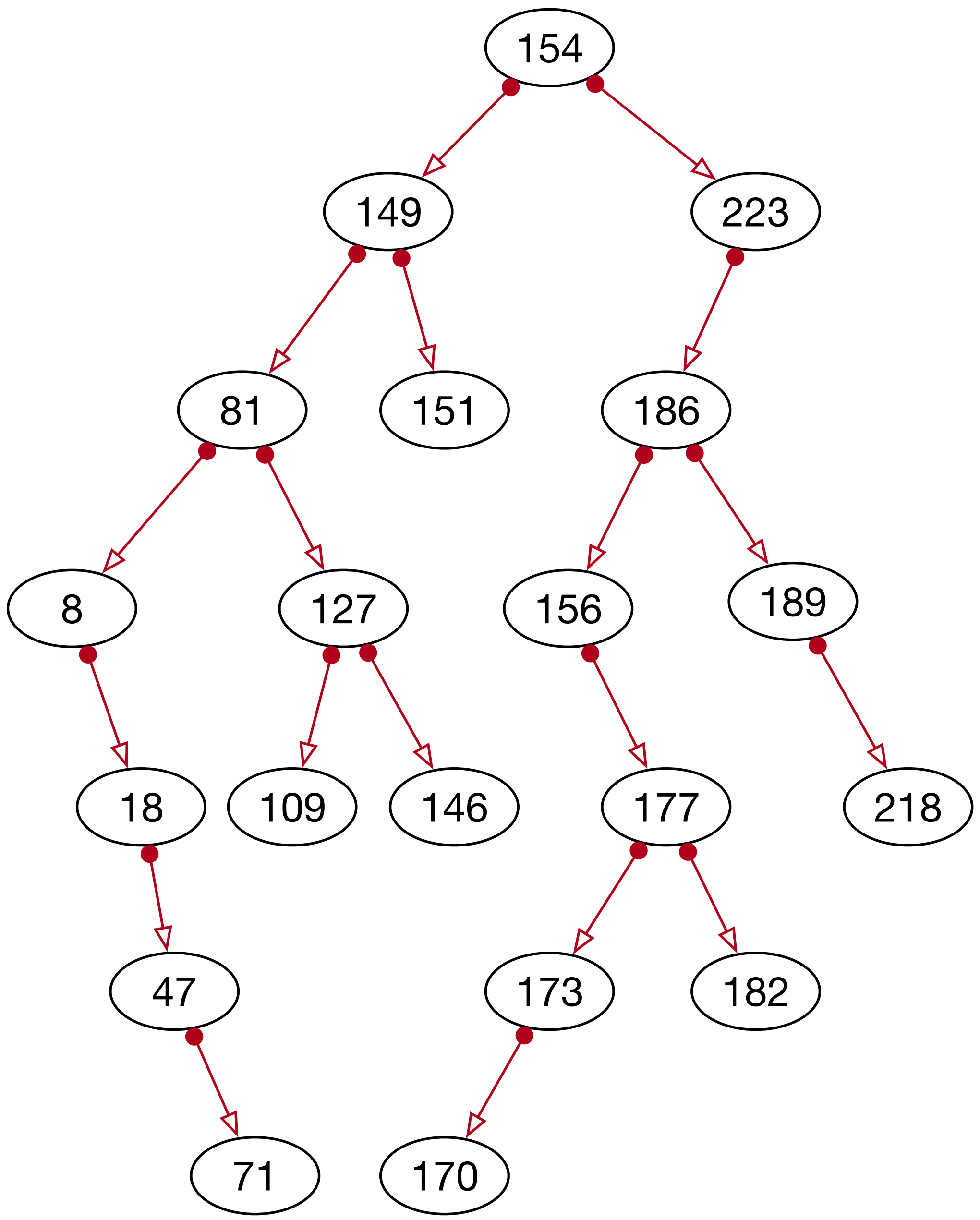
Example

- Insert 146



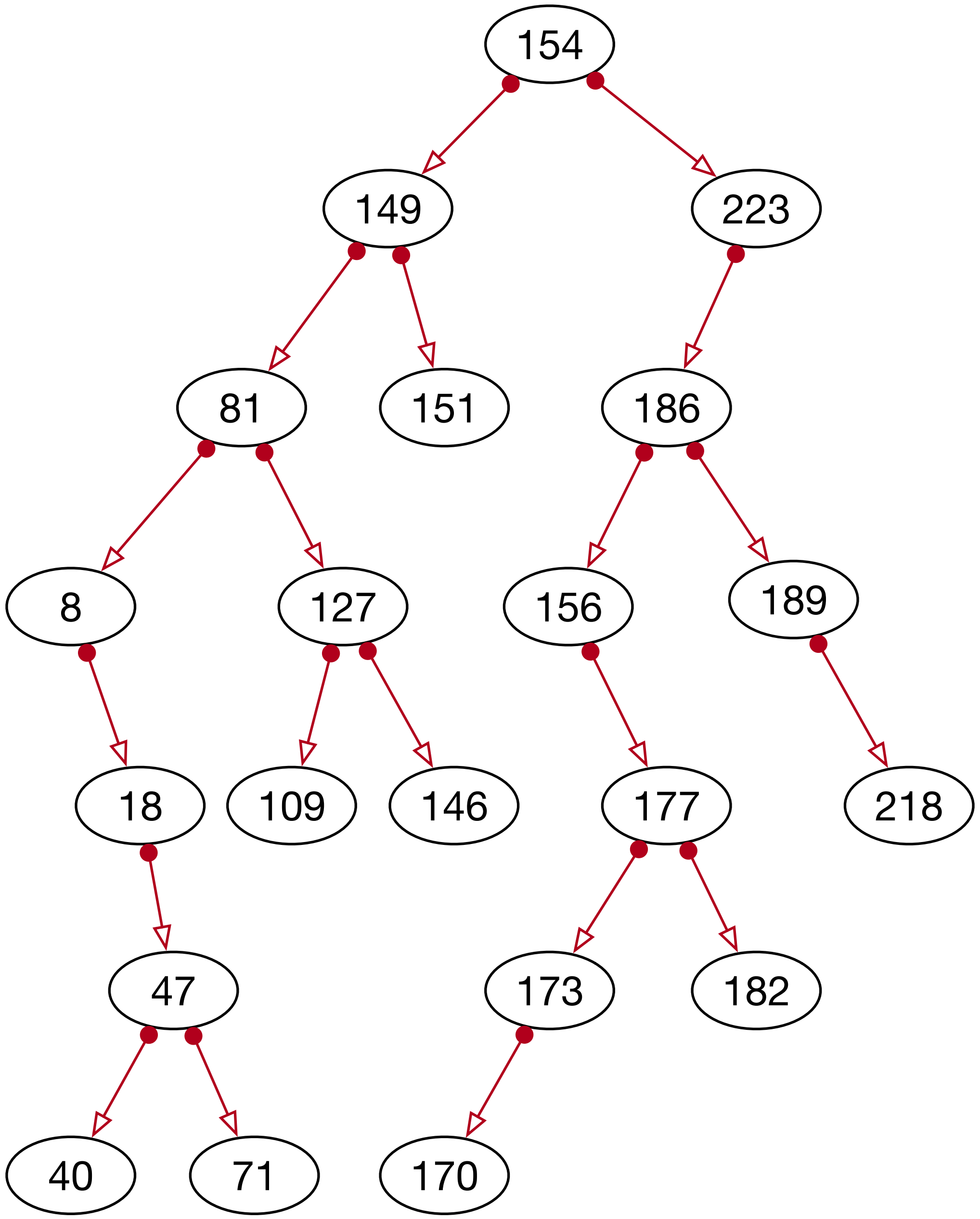
Example

- Insert 71



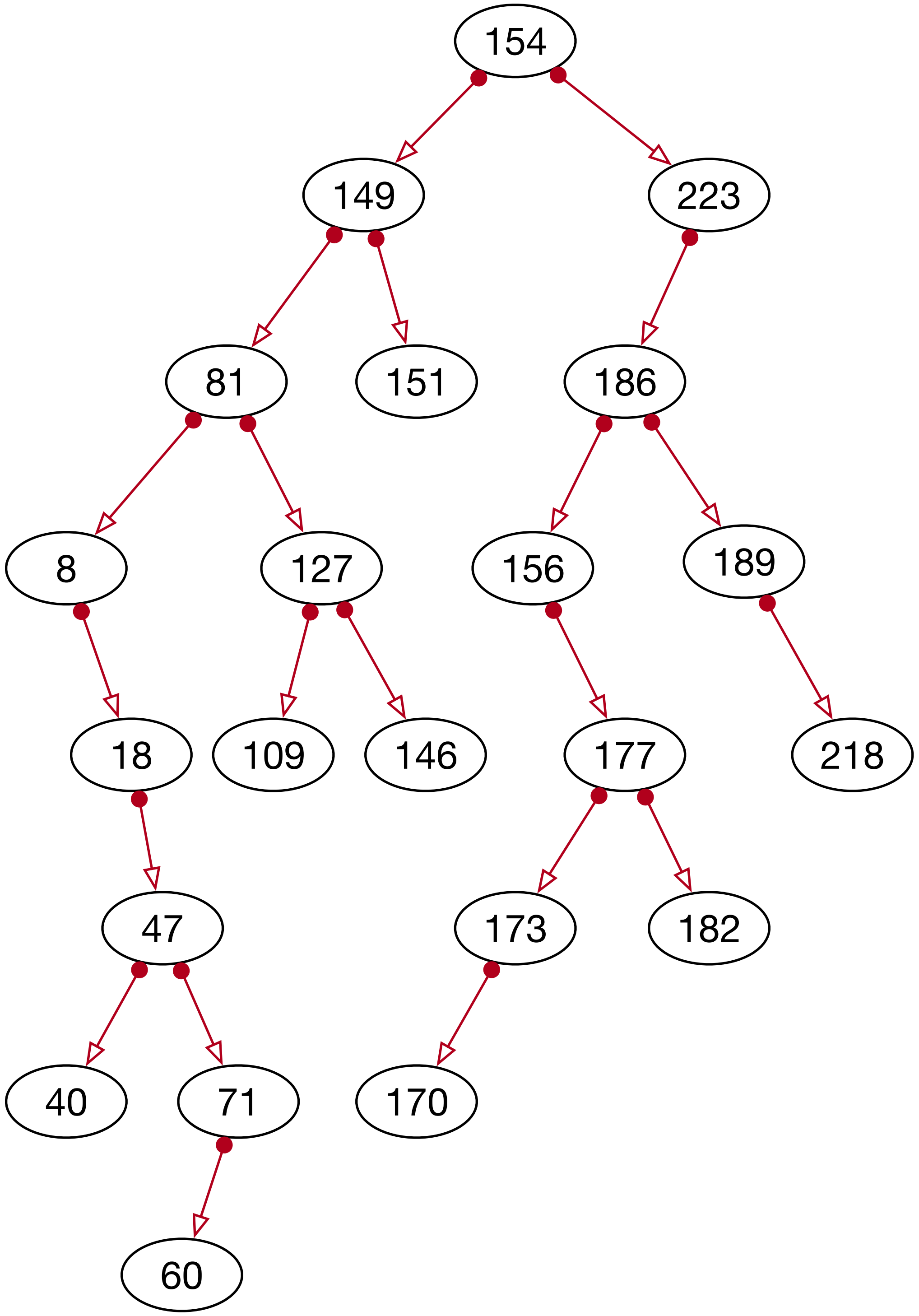
Example

- Insert 40



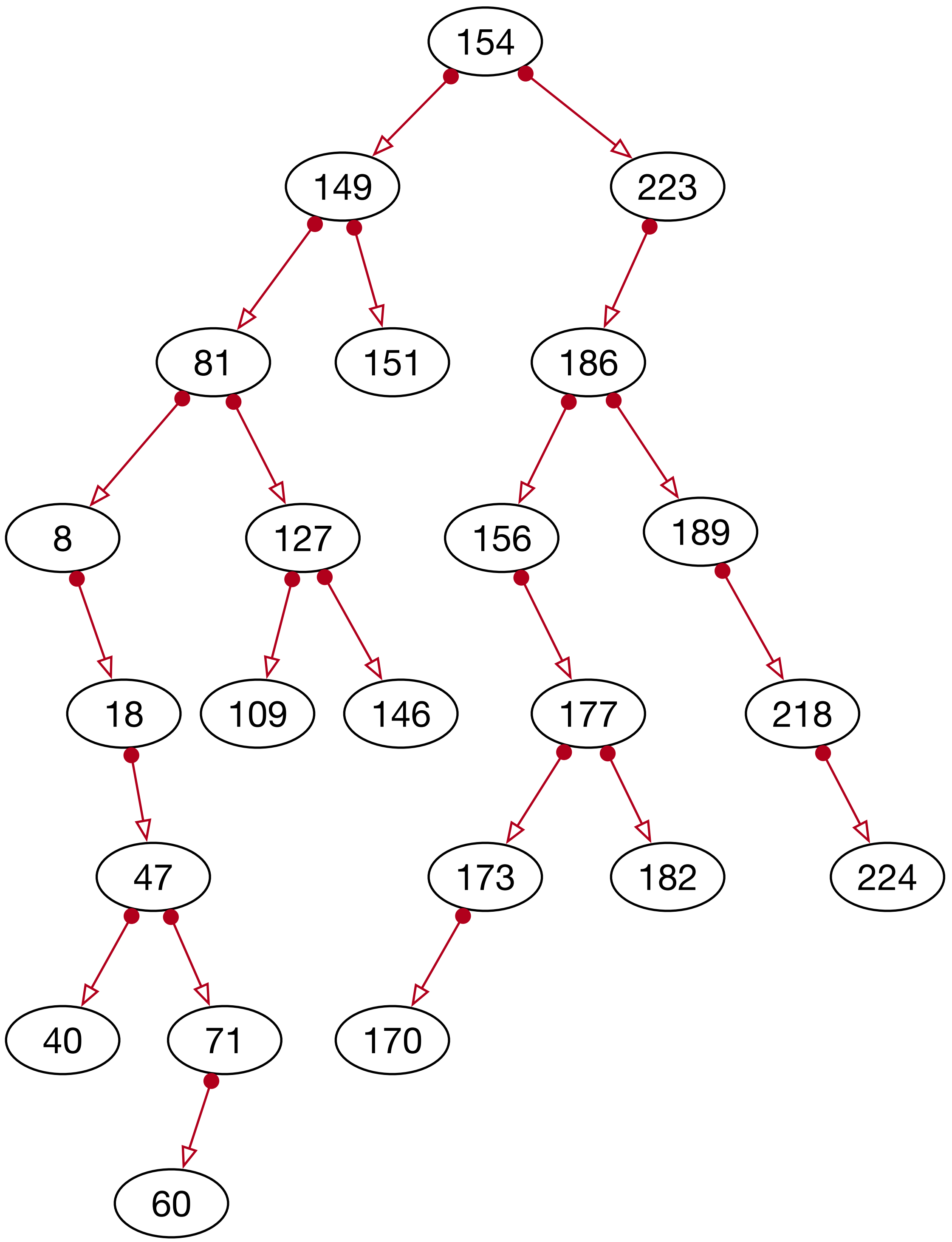
Example

- Insert 60



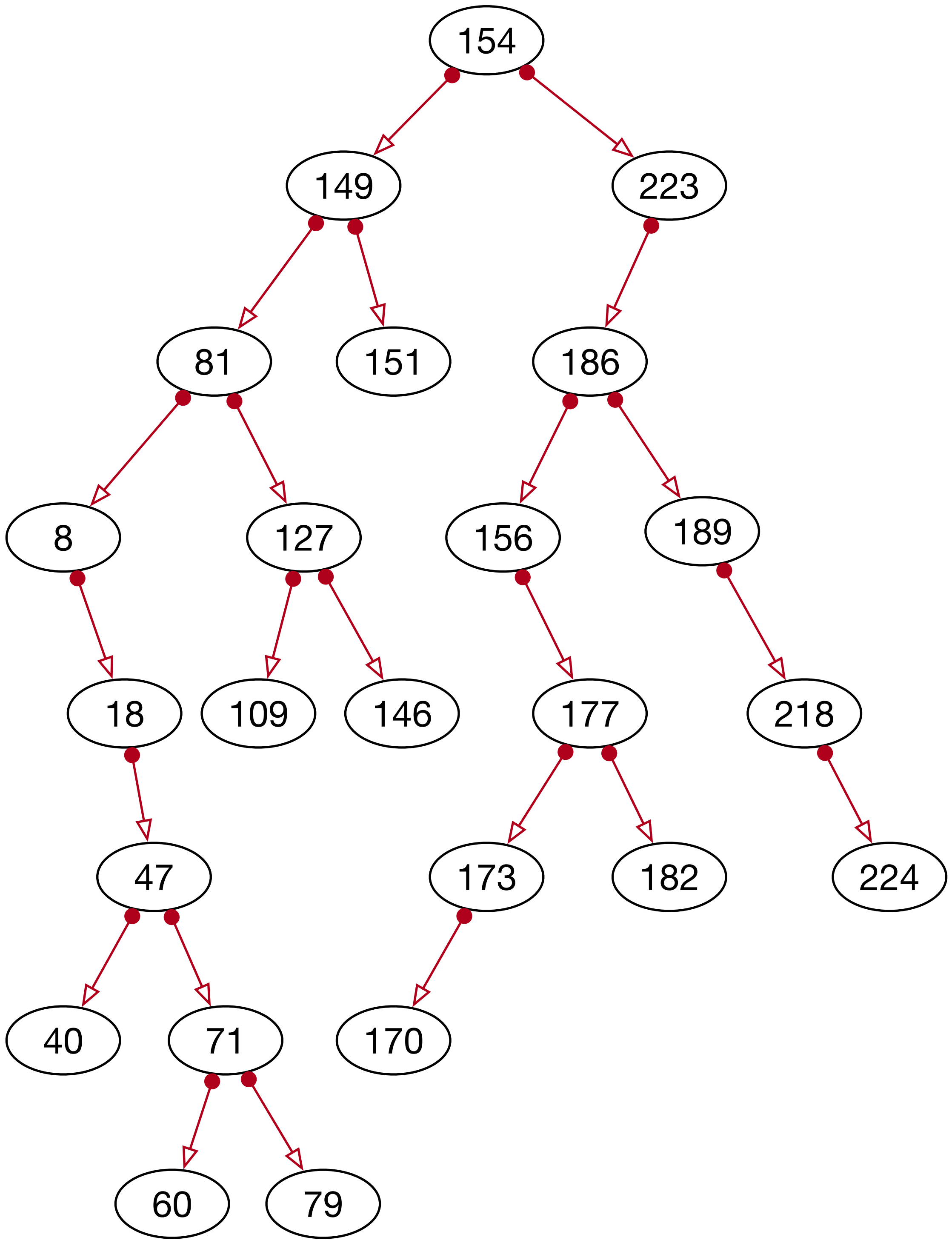
Example

- Insert 224



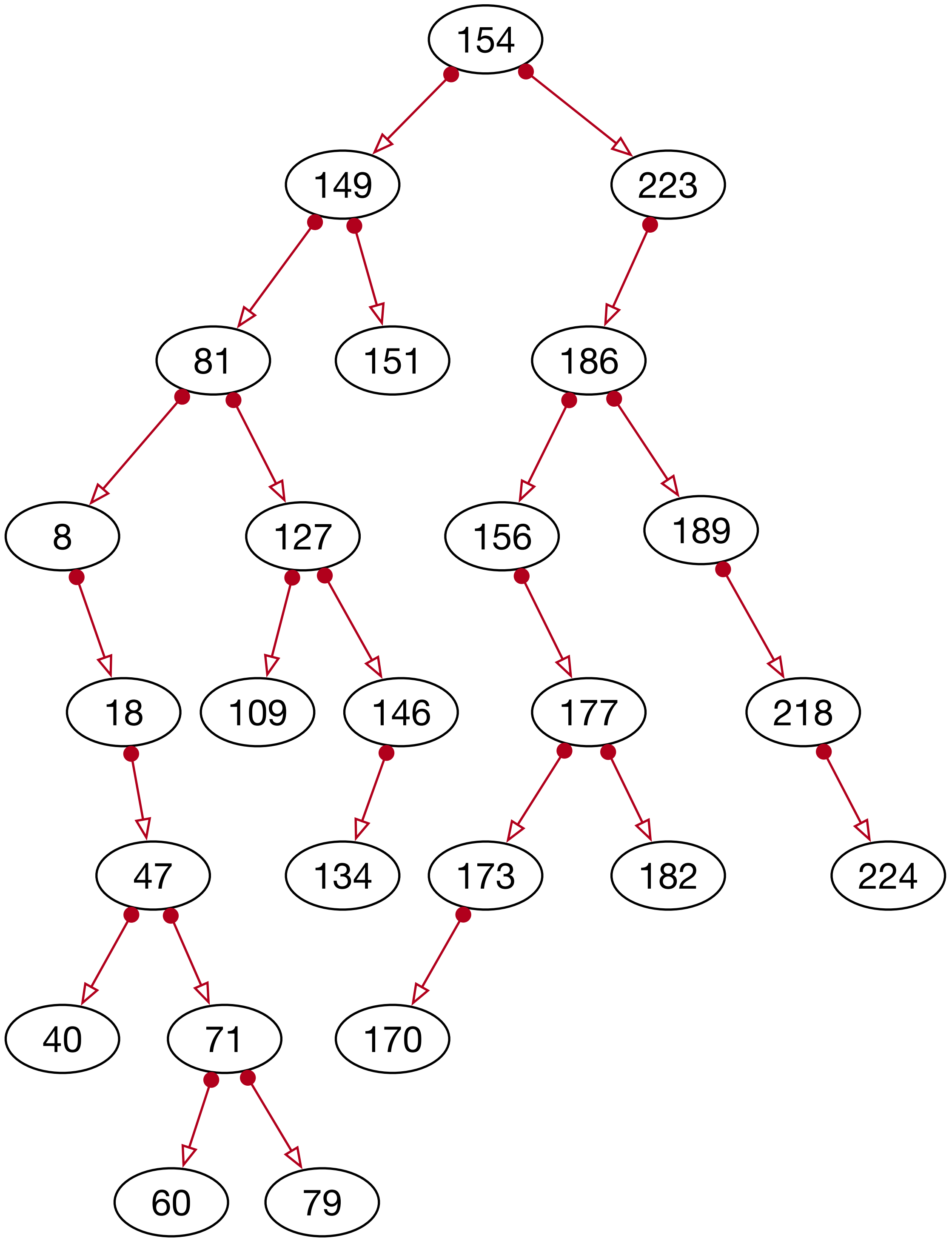
Example

- Insert 79



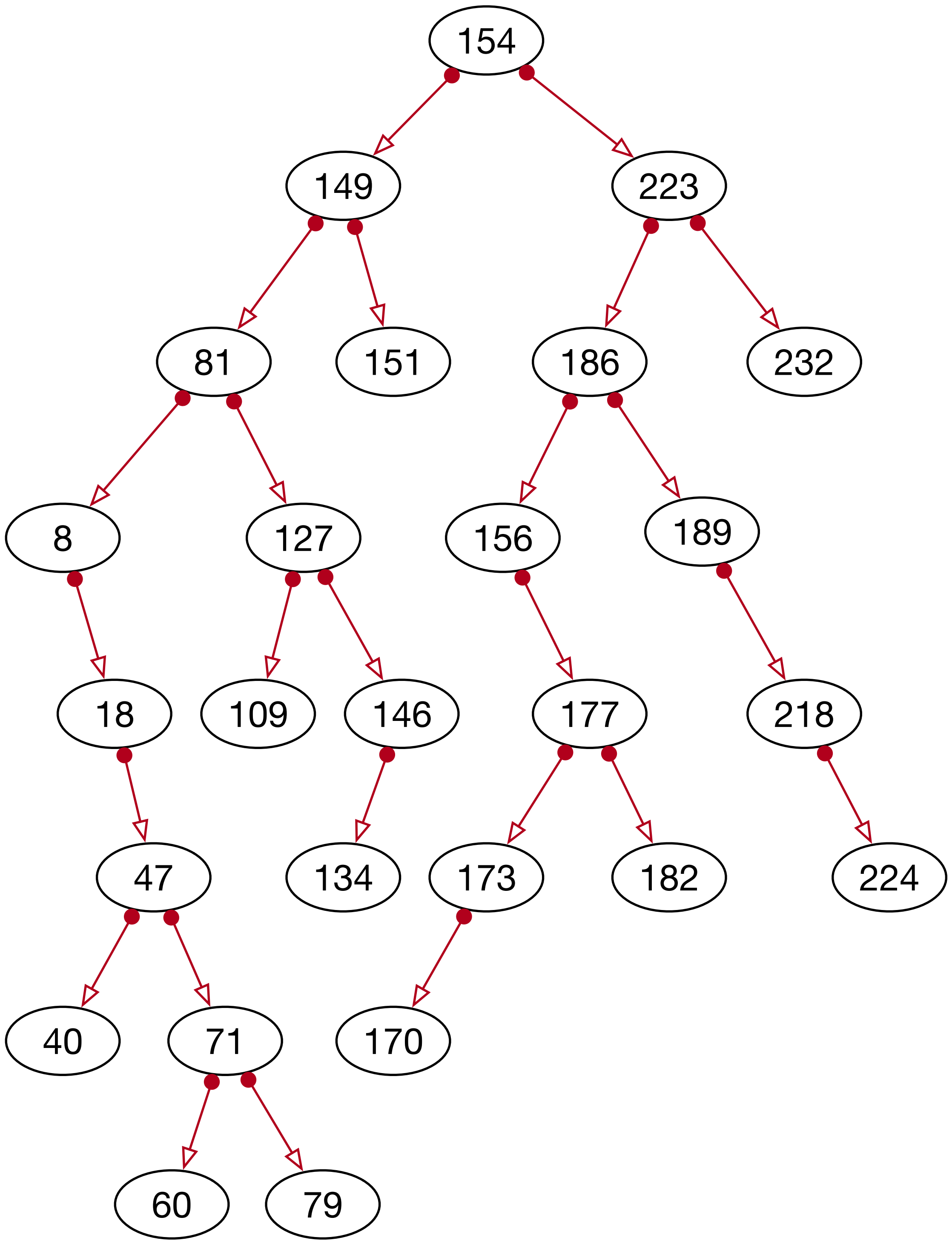
Example

- Insert 134



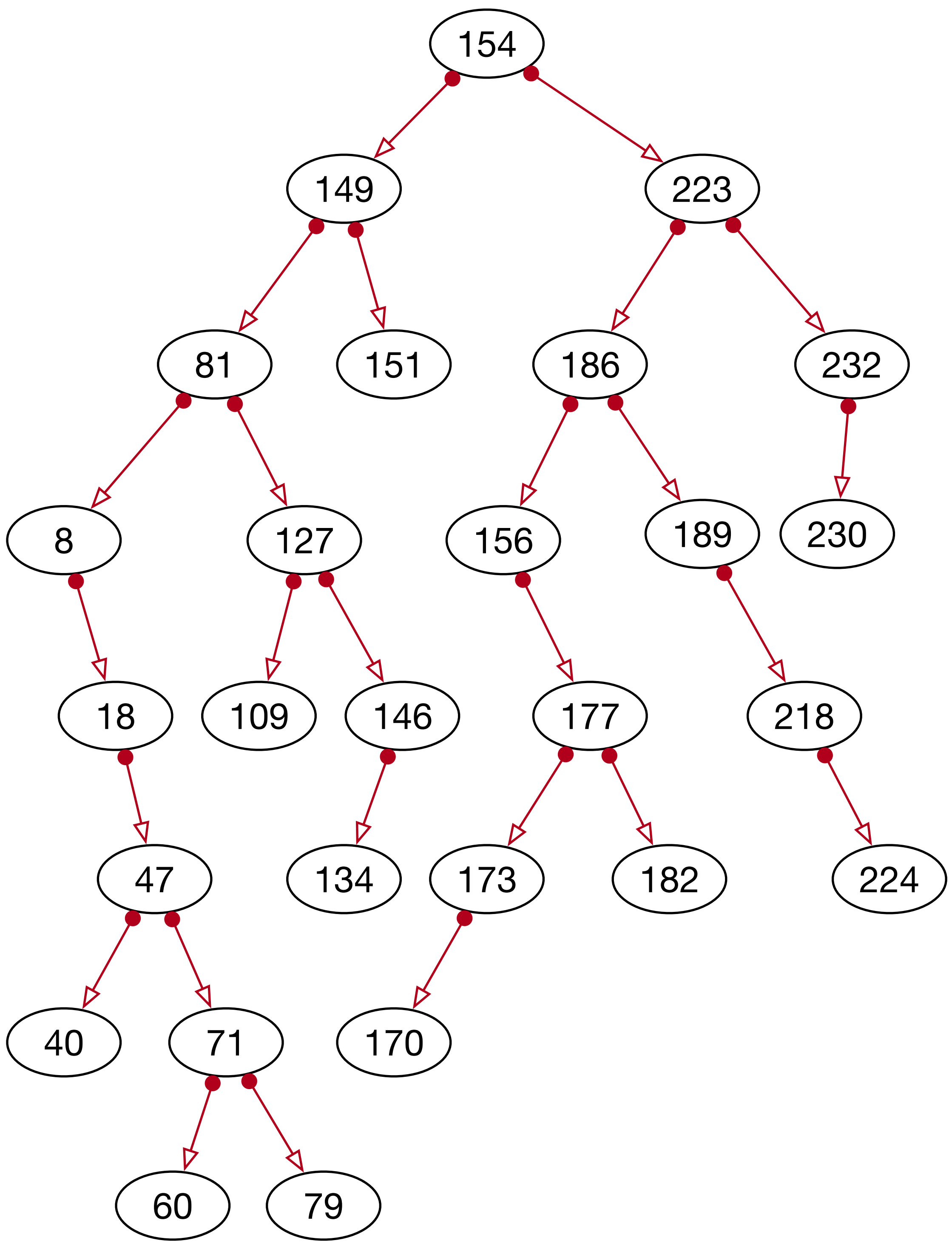
Example

- Insert 232



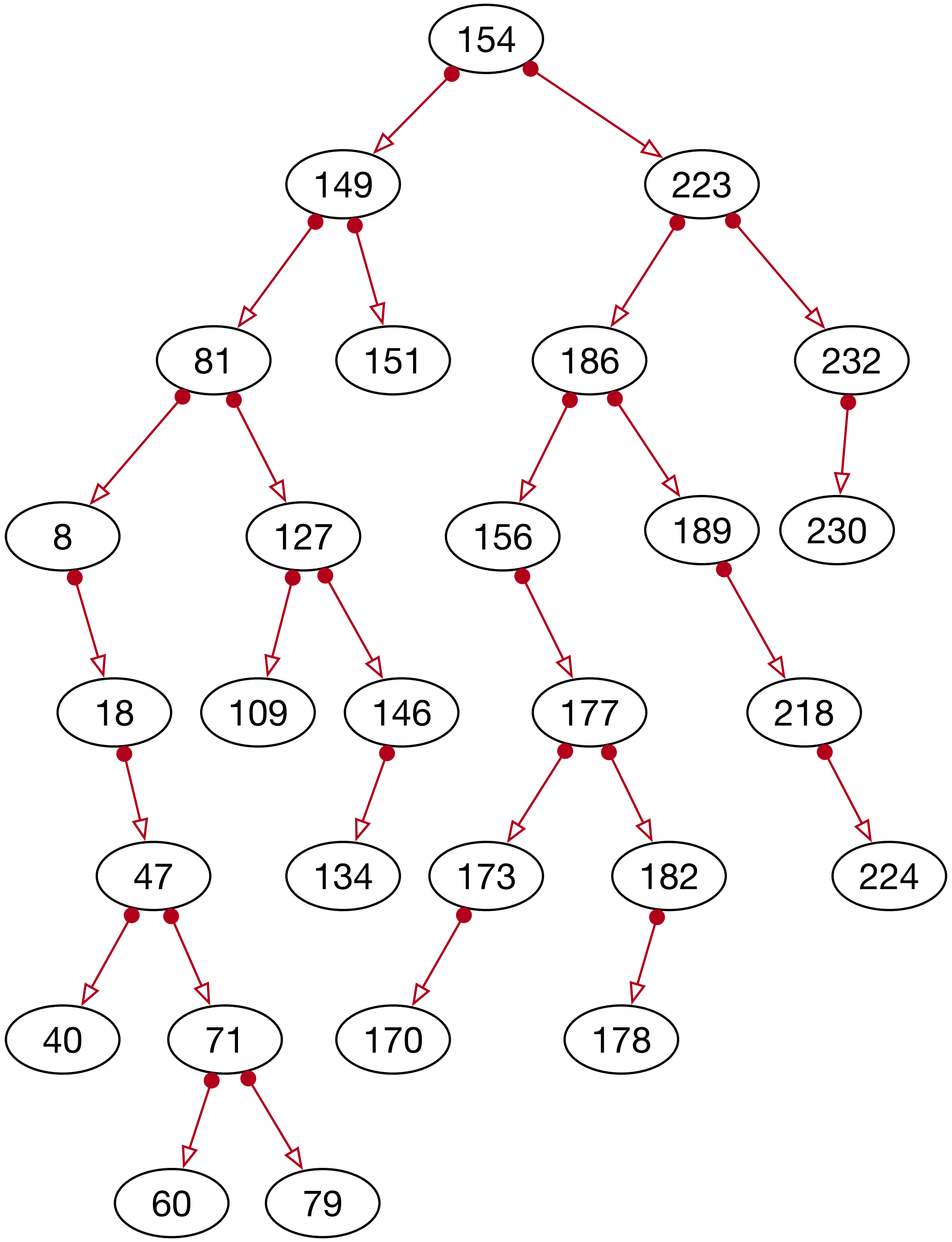
Example

- Insert 230



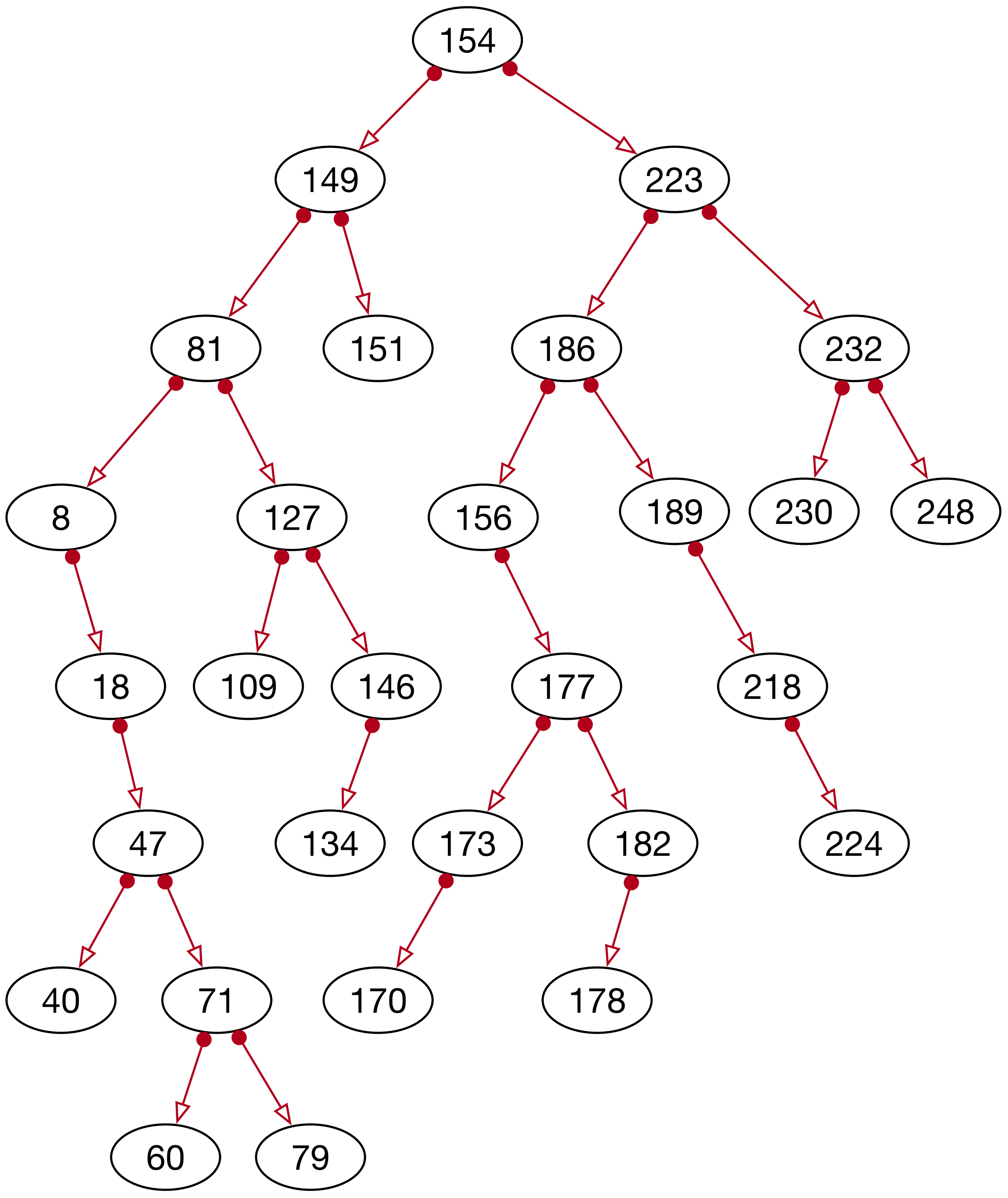
Example

- Insert 178



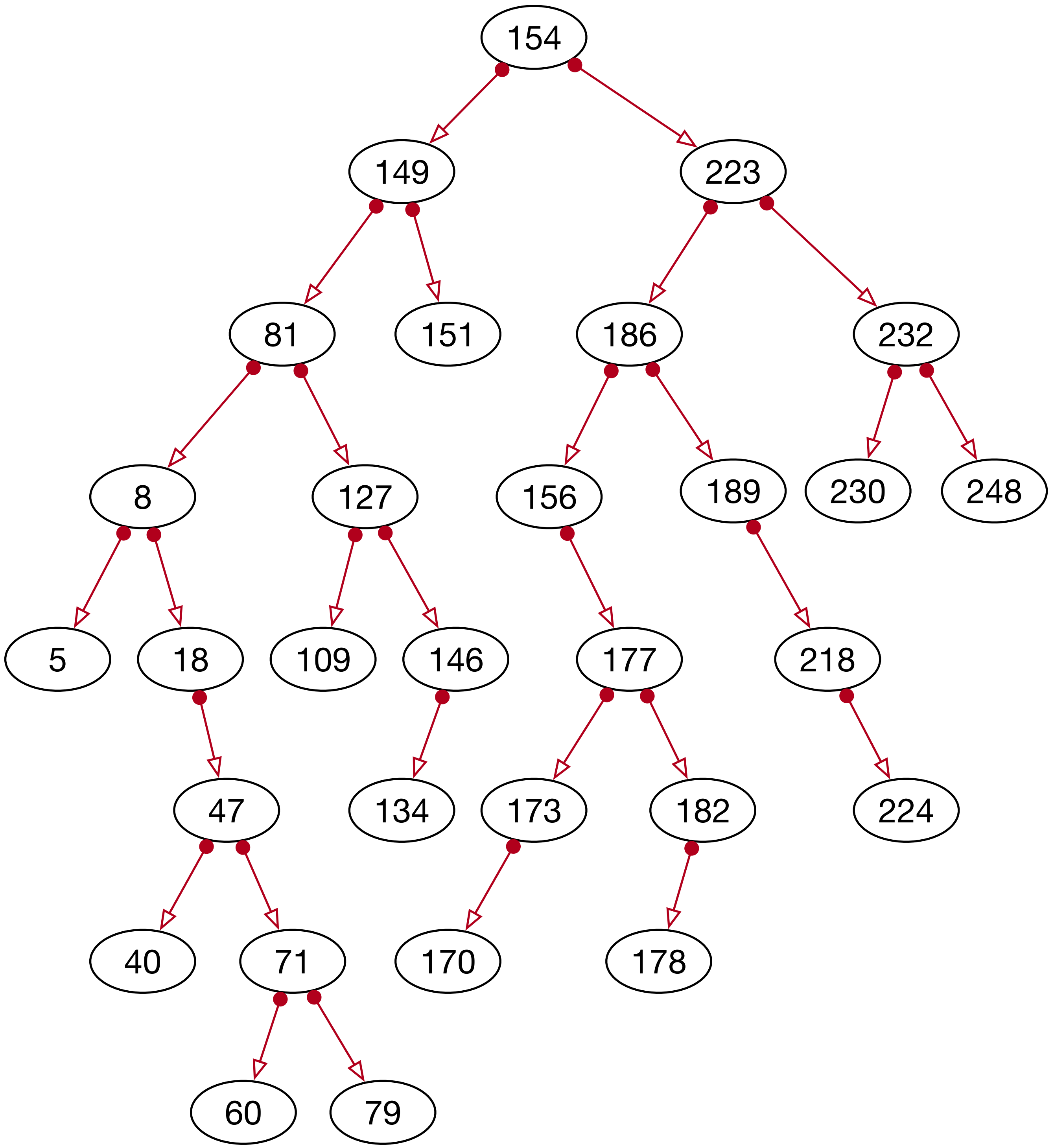
Example

- Insert 248



Example

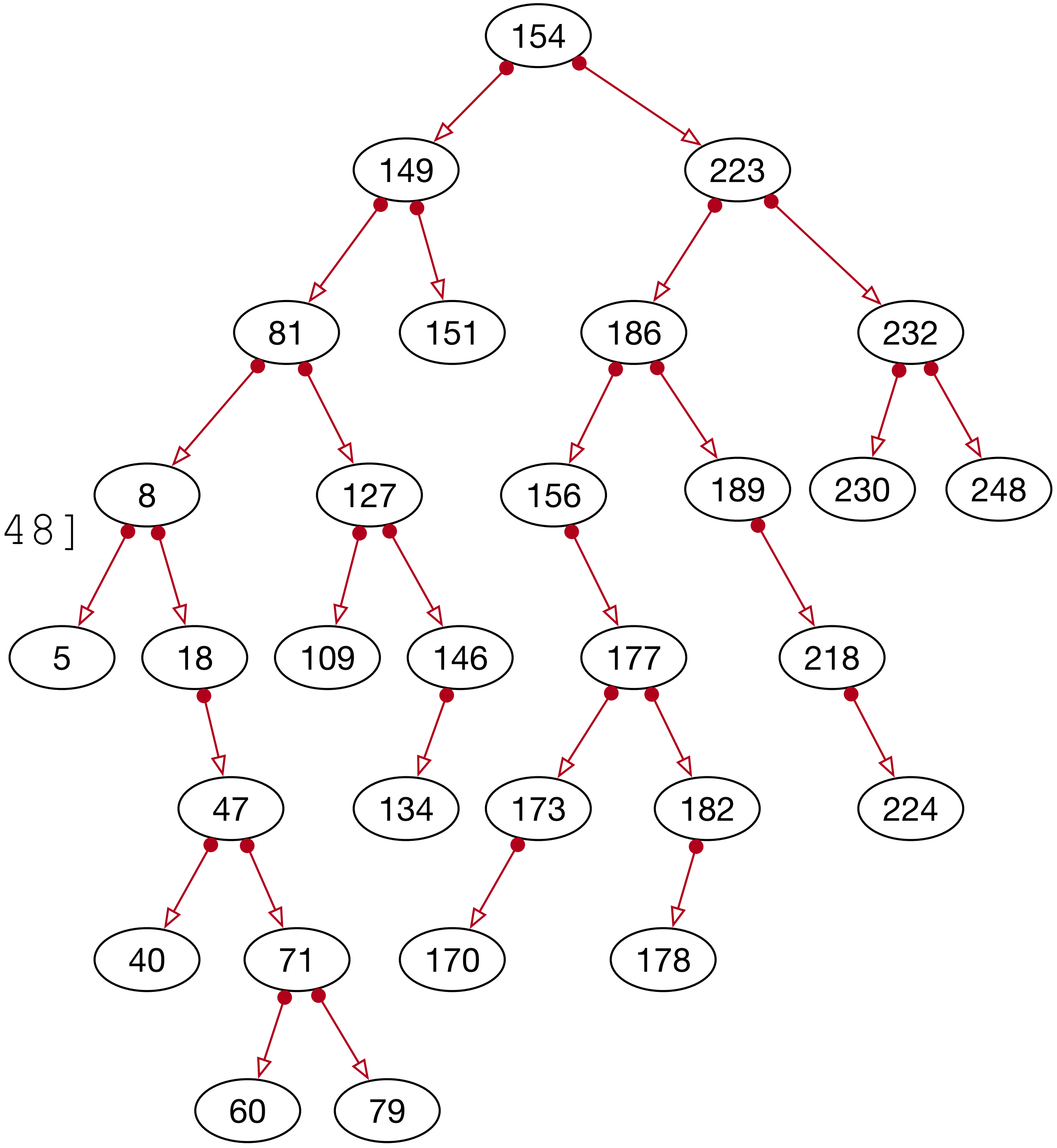
- Insert 5



Example

- Printing out per layer

```
[154]
[149, 223]
[81, 151, 186, 224]
[8, 127, 156, 189, 232]
[5, 18, 109, 146, 177, 218, 230, 248]
[47, 134, 173, 182]
[40, 71, 170, 178]
[60, 79]
```



Example

- This example shows very nicely:
  - Trees under random insertion become too thin
  - Not well balanced

Implementation

- Node class

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
    def __repr__(self):
        return "Node : {}, Value: {}, Left: {}, Right: {}".format(
            hex(id(self)),
            self.value,
            hex(id(self.left)),
            hex(id(self.right)))
```

Implementation

- Tree class:
  - Empty tree has a None root

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

Implementation

- Insert is more difficult:
  - Start at the root
    - If the value to be inserted is smaller than the value at the root, go to the left
    - Otherwise go to the right
  - Continue until you reach a None-child
    - Insert there

Implementation

- Special Case:
  - The tree is empty

```
def insert(self, value):
        new_node = Node(value)
        if not self.root:
            self.root = new_node
        else:
             ...
```

Implementation

```
def insert(self, value):
        new_node = Node(value)
        if not self.root:
            self.root = new_node
        else:
            current = self.root
            while True:
                if value < current.value:
                    if current.left:
                        current = current.left
                    else:
                        current.left = new_node
                        return
                else:
                    if current.right:
                        current = current.right
                    else:
                        current.right = new_node
                        return
```

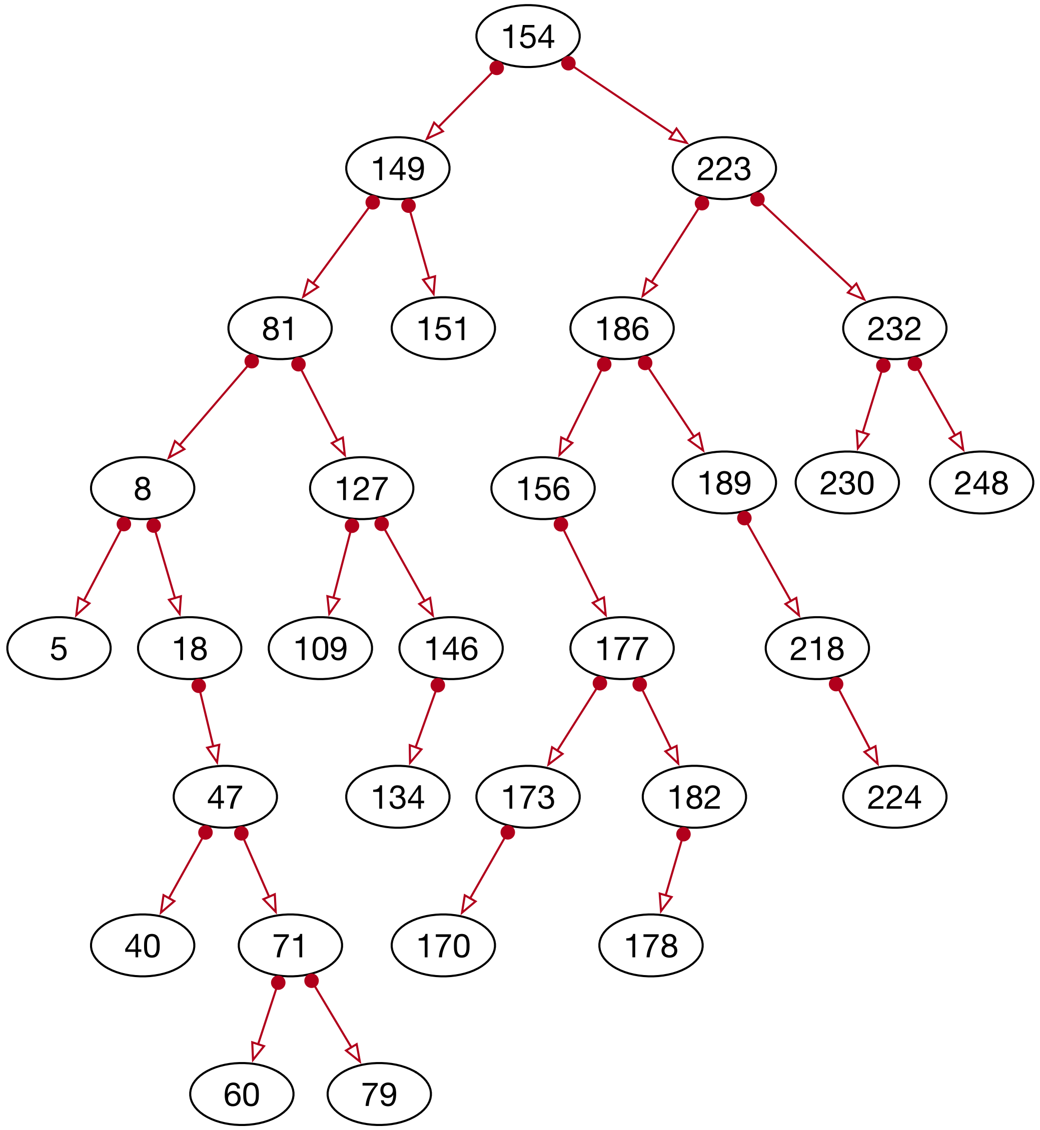
Implementation

- Note:
  - What do we have to change to prevent the same key to be inserted twice?

Showing Contents

- Show by level
  - Give all the records inserted at a certain level

```
[154]
[149, 223]
[81, 151, 186, 224]
[8, 127, 156, 189, 232]
[5, 18, 109, 146, 177, 218, 230, 248]
[47, 134, 173, 182]
[40, 71, 170, 178]
[60, 79]
```

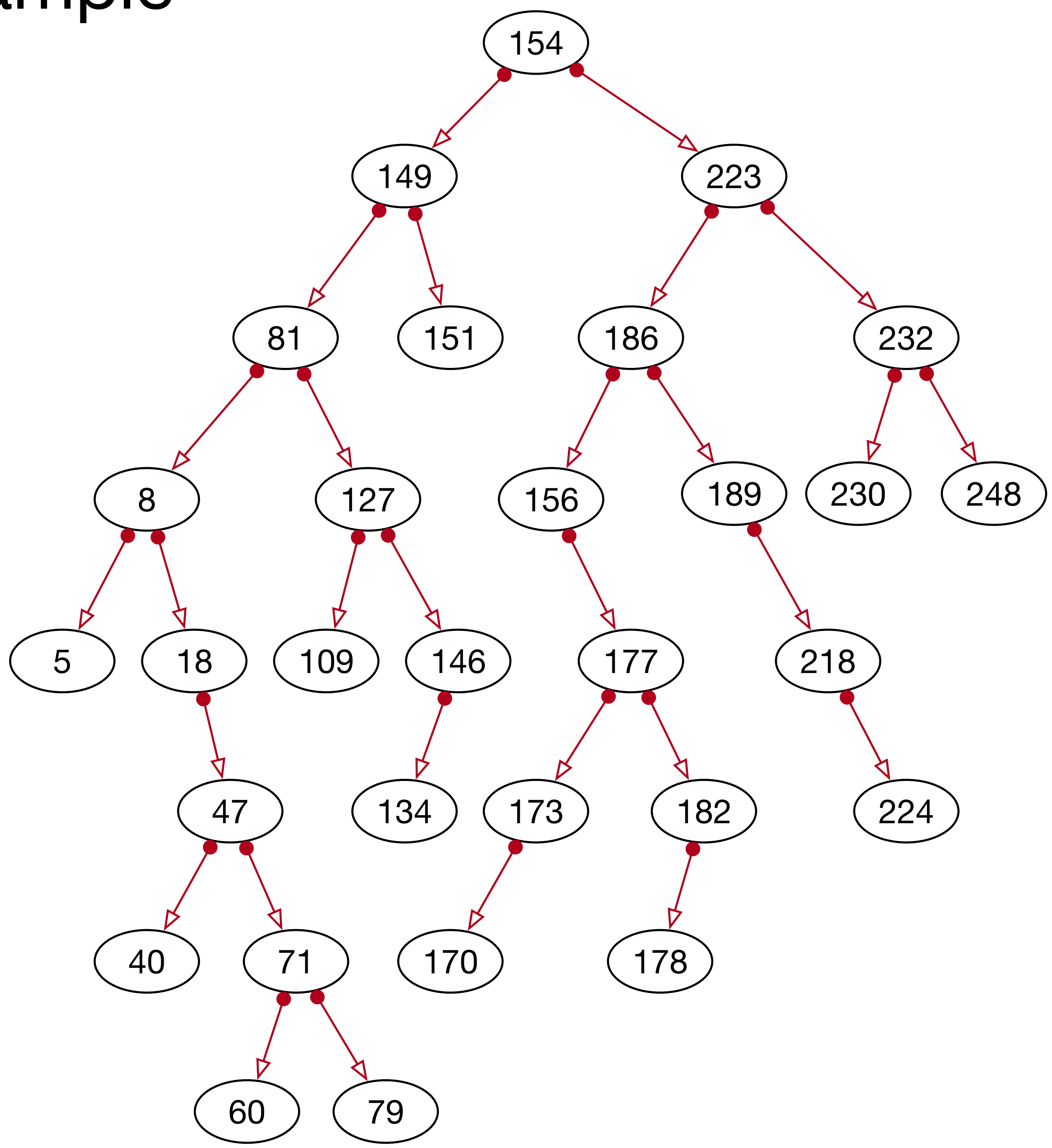


Showing Contents

- Idea:
  - Put all nodes at a certain level in a queue
  - Process the queue by:
    - Printing all values
    - Adding children to the next queue

Showing Contents

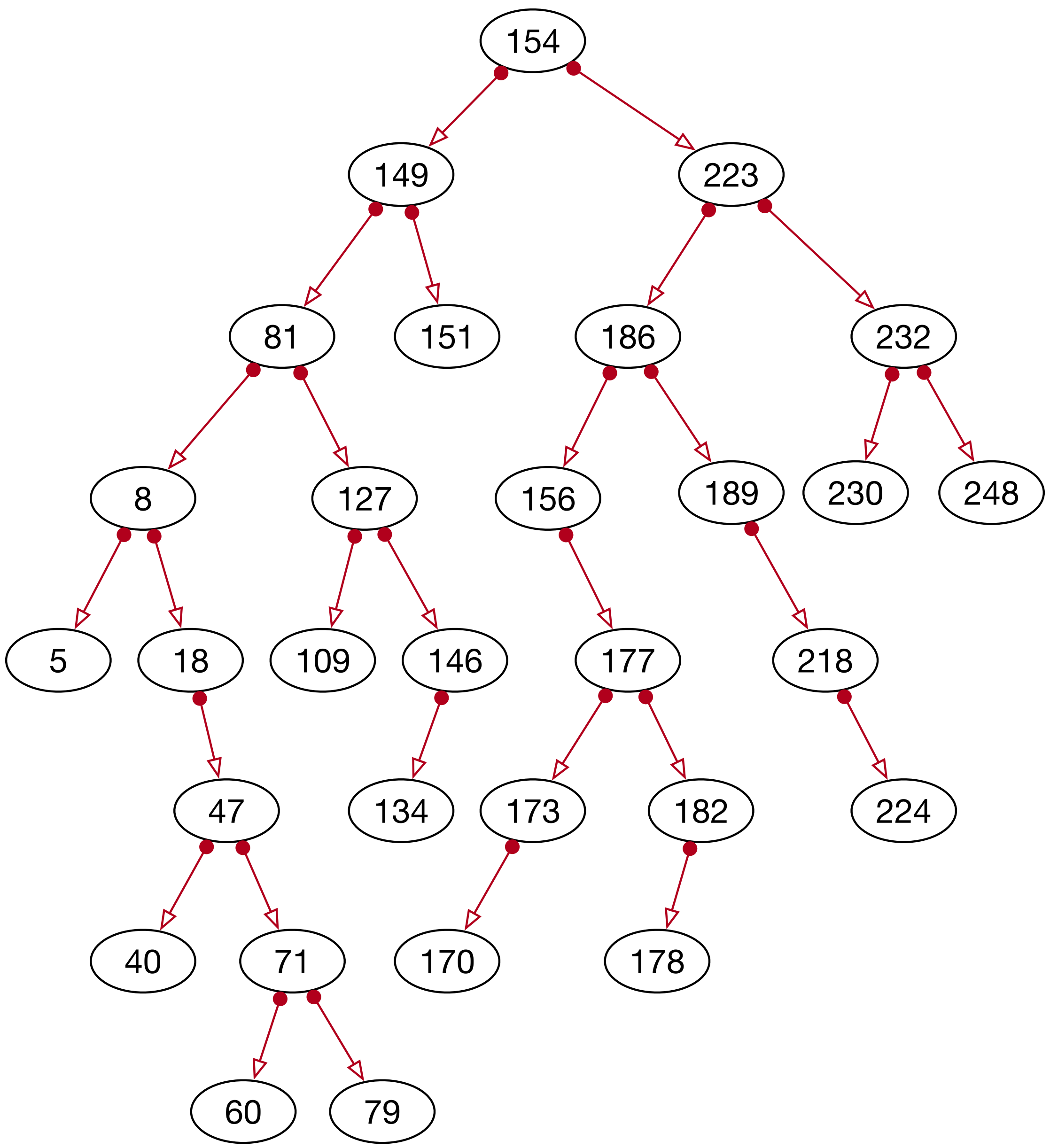
- Example



current: 154
next: 149, 223

Showing Contents

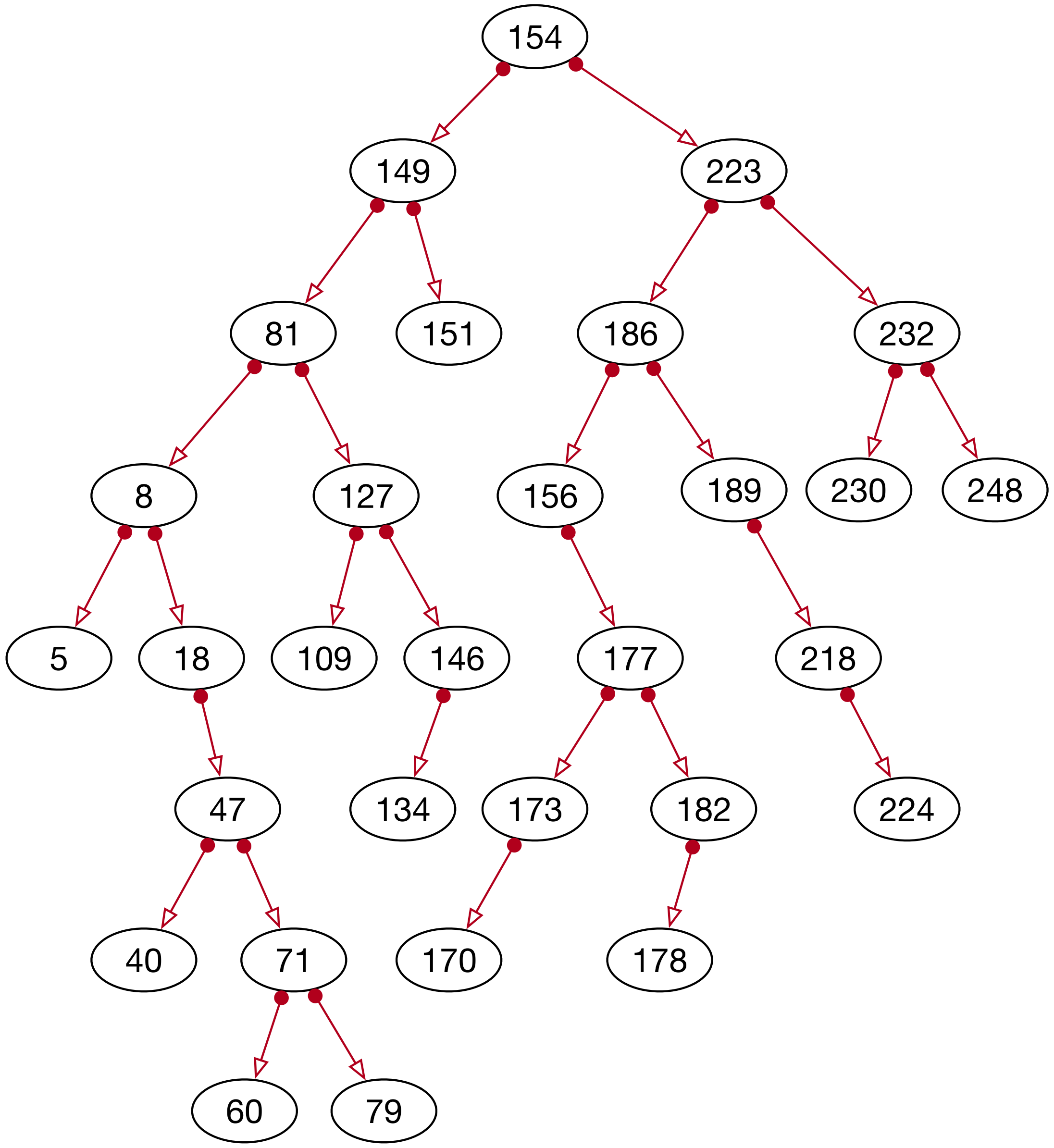
- Example (cont.)



current: 149, 223
next: 81, 151, 186, 232

Showing Contents

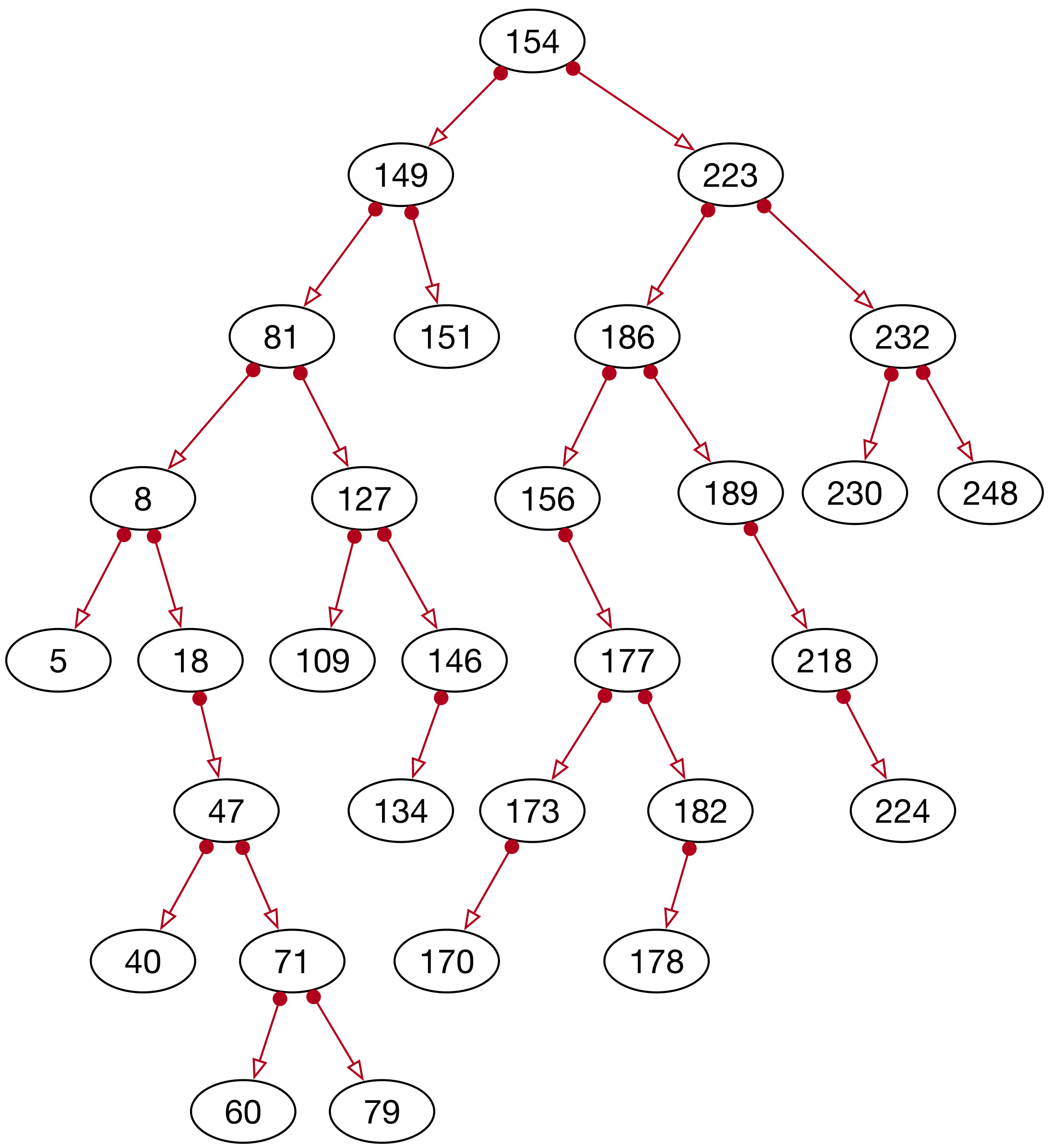
- Example (cont.)



current: 81, 151, 186, 232
next: 8, 127, 156, 189, 230, 248

Showing Contents

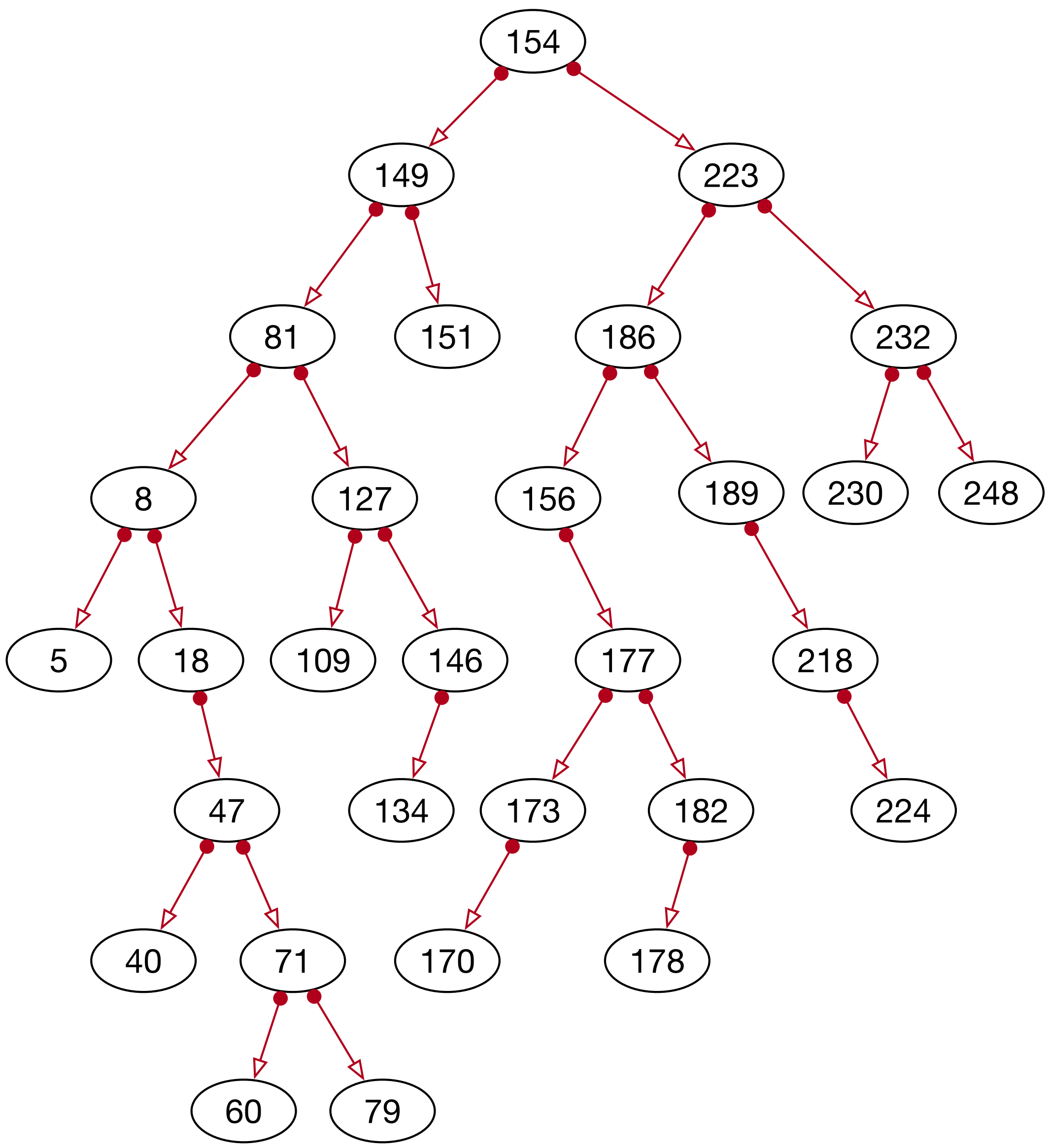
- Example (cont.)



current: 8, 127, 156, 189, 230, 248
next: 5, 18, 109, 146, 177, 218

Showing Contents

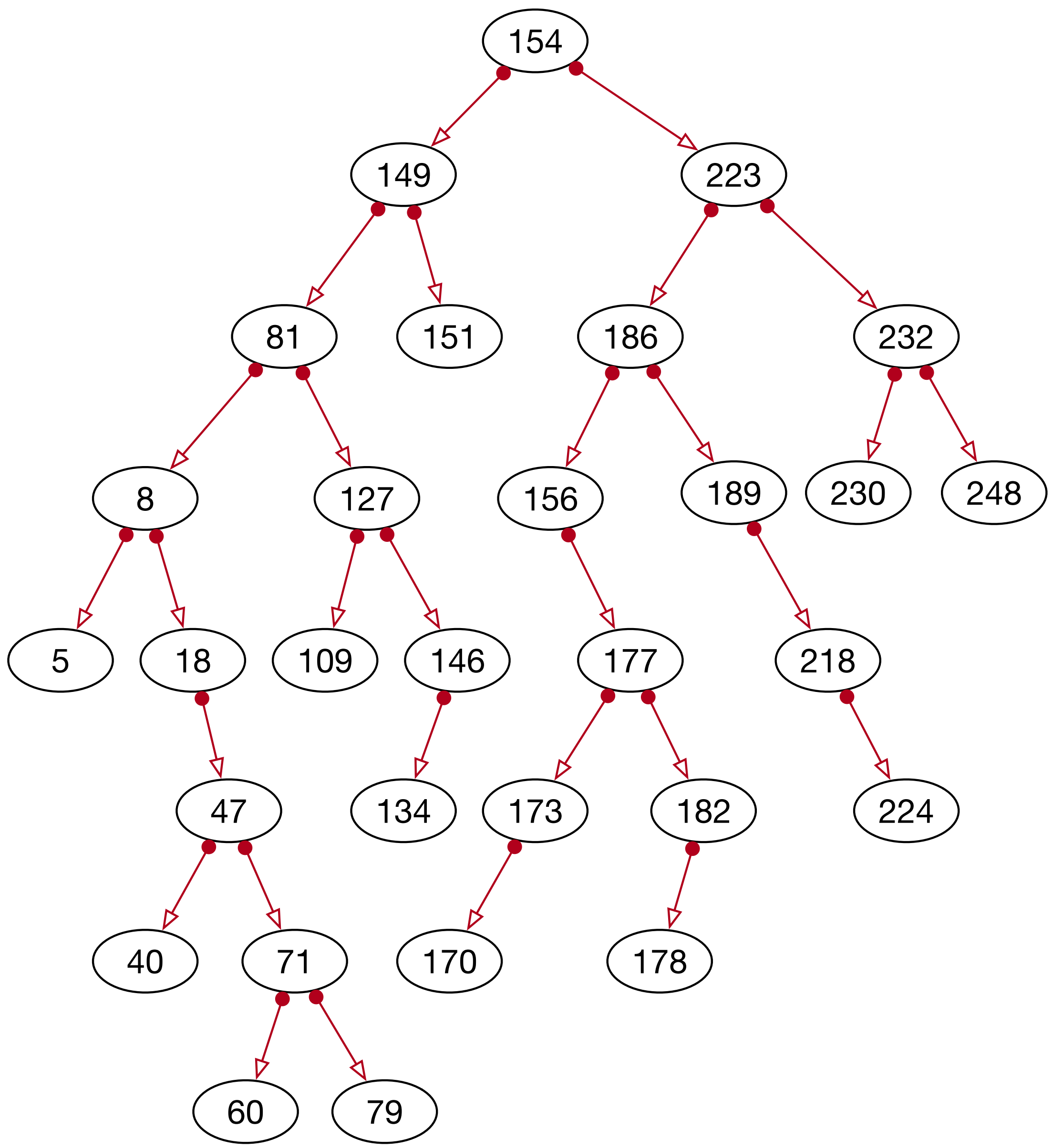
- Example (cont.)



current: 5, 18, 109, 146, 177, 218
next: 47, 134, 173, 182, 224

Showing Contents

- Example (cont.)



current: 47, 134, 173, 182, 224
next: 40, 71, 170, 178

Showing Contents

- We can use different implementations for queues
  - Such as linked lists
    - But it is easier to use Python lists

Showing Contents

```
def print_layer(self):
       current_generation = [self.root]
       while True:
           print([node.value for node in
                          current_generation])
           next_generation = [ ]
           for node in current_generation:
               if node and node.left:
                   next_generation.append(node.left)
               if node and node.right:
                   next_generation.append(node.right)
           current_generation = next_generation
           if not current_generation:
               return
```

Showing Contents

- Other tree traversals are more important
  - In-order traversal:
    - Traverse the left sub-tree
    - Get the root
    - Traverse the right sub-tree

Showing Contents

- These use recursion:
  - A function that calls itself
    - For us: have to define a left sub-tree and right sub-tree

```
def left_tree(self):
        result = Binary_Tree()
        result.root = self.root.left
        return result
```

```
def right_tree(self):
        result = Binary_Tree()
        result.root = self.root.right
        return result
```

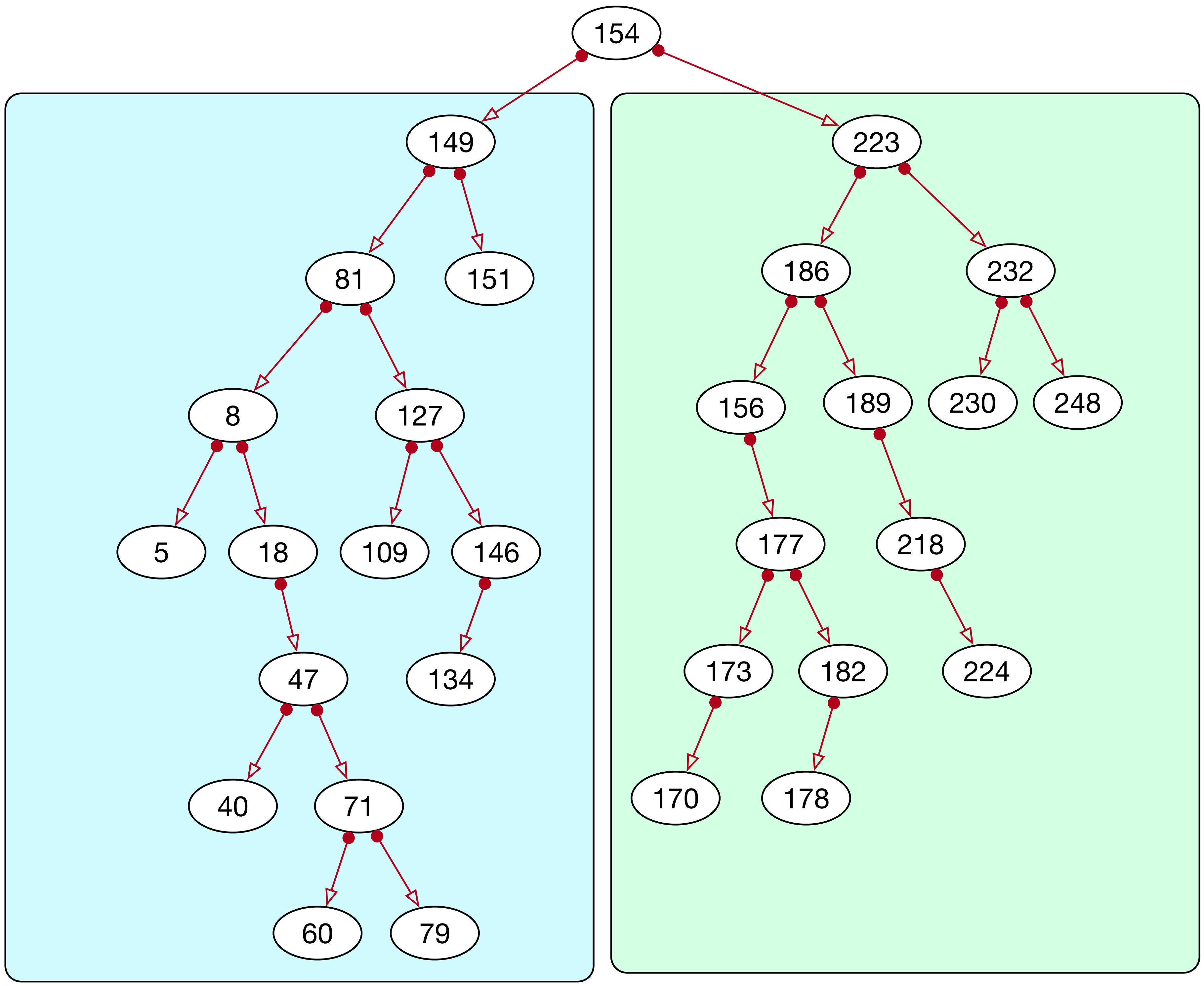
Showing Contents

- Then to implement in-order

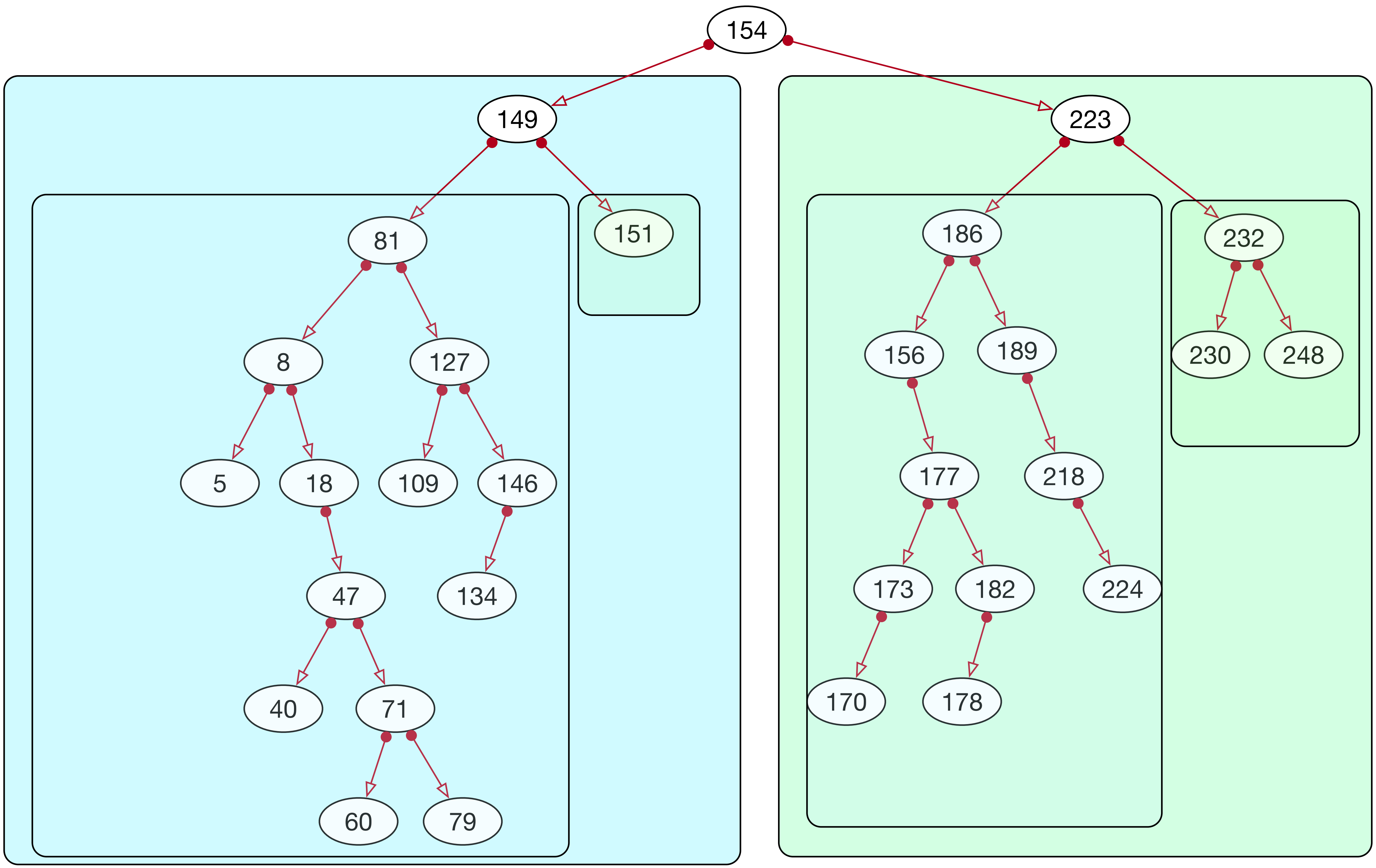
```
def in_order(self):
        if not self.root:
            return []
        return self.left_tree().in_order() +
               [self.root.value] +
               self.right_tree().in_order()
```

- Notice how we are calling the function on the smaller tree

Showing Contents



Showing Contents



Showing Contents

- In-order traversal:
  - Records are printed in order

```
b = Binary_Tree()
lista = [154,223,186,149,189,156,81,8,177,182,173,
         218,18,127,170,109,47,151,146,71,40,60,
         224,79,134,232,230,178,248,5, 150, 130]
for x in lista:
    b.insert(x)
b.print_layer()
print(b.in_order())
```

Showing Contents

- Result

```
[5, 8, 18, 40, 47, 60, 71, 79, 81, 109, 127, 130,
134, 146, 149, 150, 151, 154, 156, 170, 173, 177,
178, 182, 186, 189, 218, 223, 224, 230, 232, 248]
```

- This is incidentally our first ordering algorithm
    - Though not necessarily a very good one

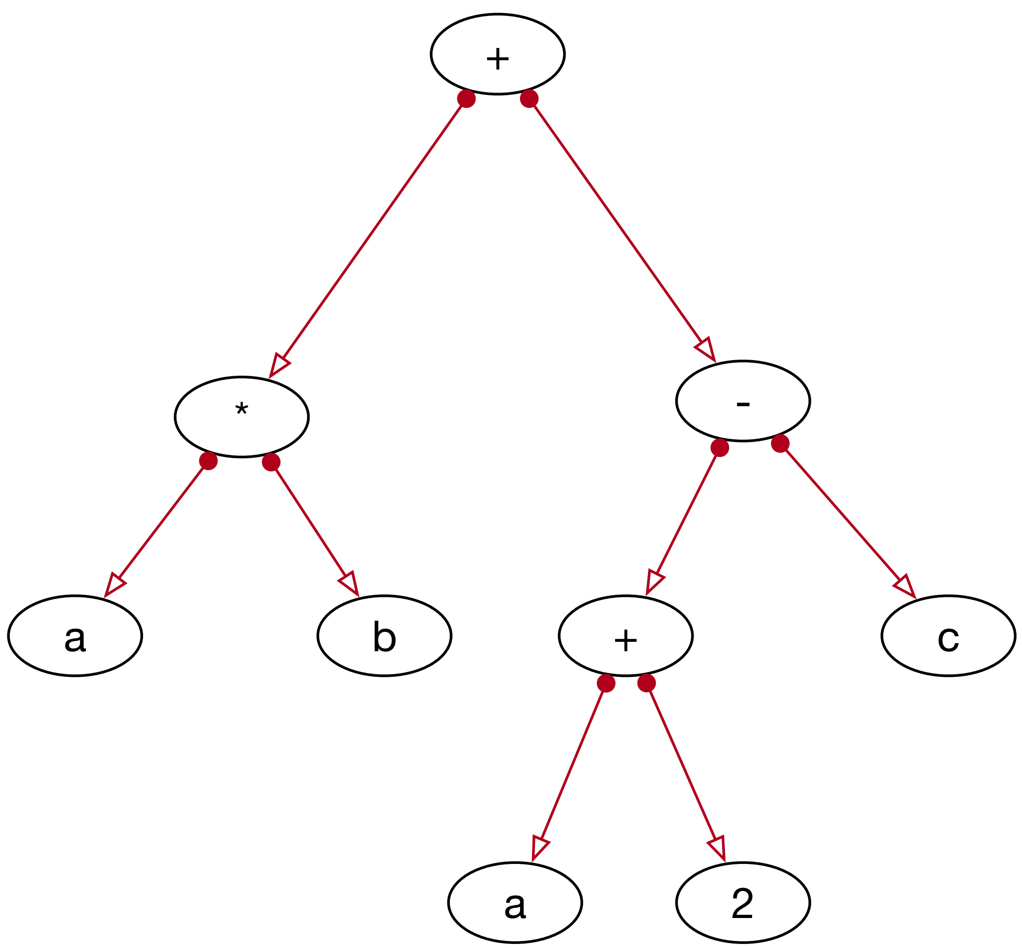
Showing Contents

- Pre-order:
    - First root
    - Then left-tree
    - Then right-tree

```
def pre_order(self):
        if not self.root:
            return []
        return [self.root.value] +
                self.left_tree().pre_order() +
                self.right_tree().pre_order()
```

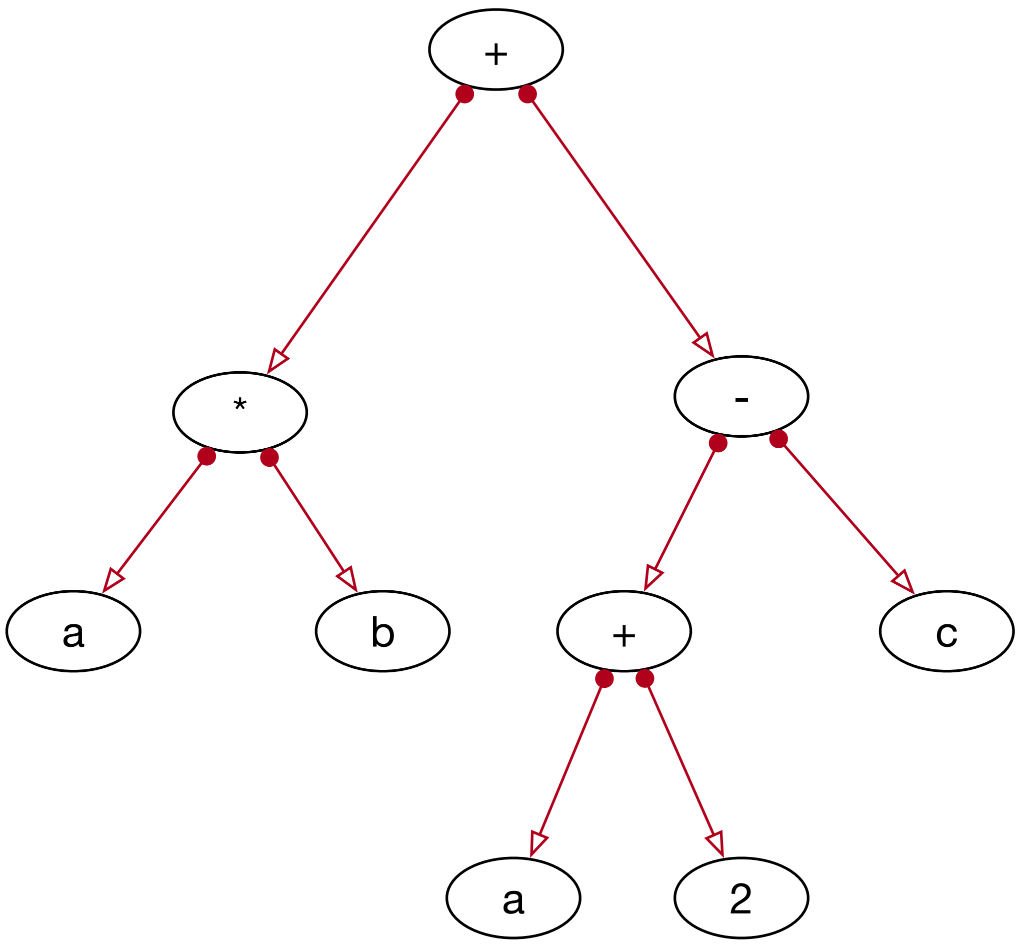
Showing Contents

- Application:
  - Expression trees
  - $a \cdot b + (a + 2) - c$



Showing Contents

- Pre-order
  - Polish notation:
    - $+ * ab - + a2c$



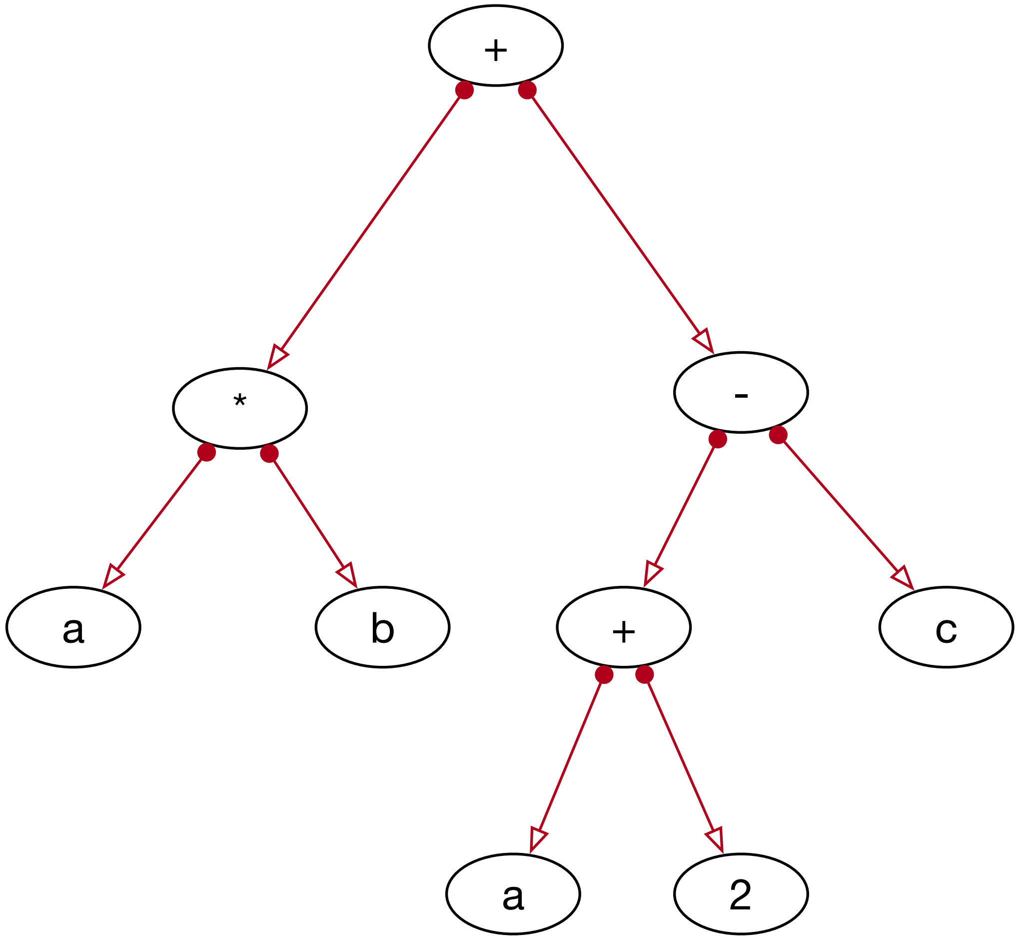
Showing Contents

- Post-order
  - First left tree
  - Then right tree
  - Then root

```
def post_order(self):
    if not self.root:
        return []
    return   self.left_tree().post_order()
           + self.right_tree().post_order()
           + [self.root.value]
```

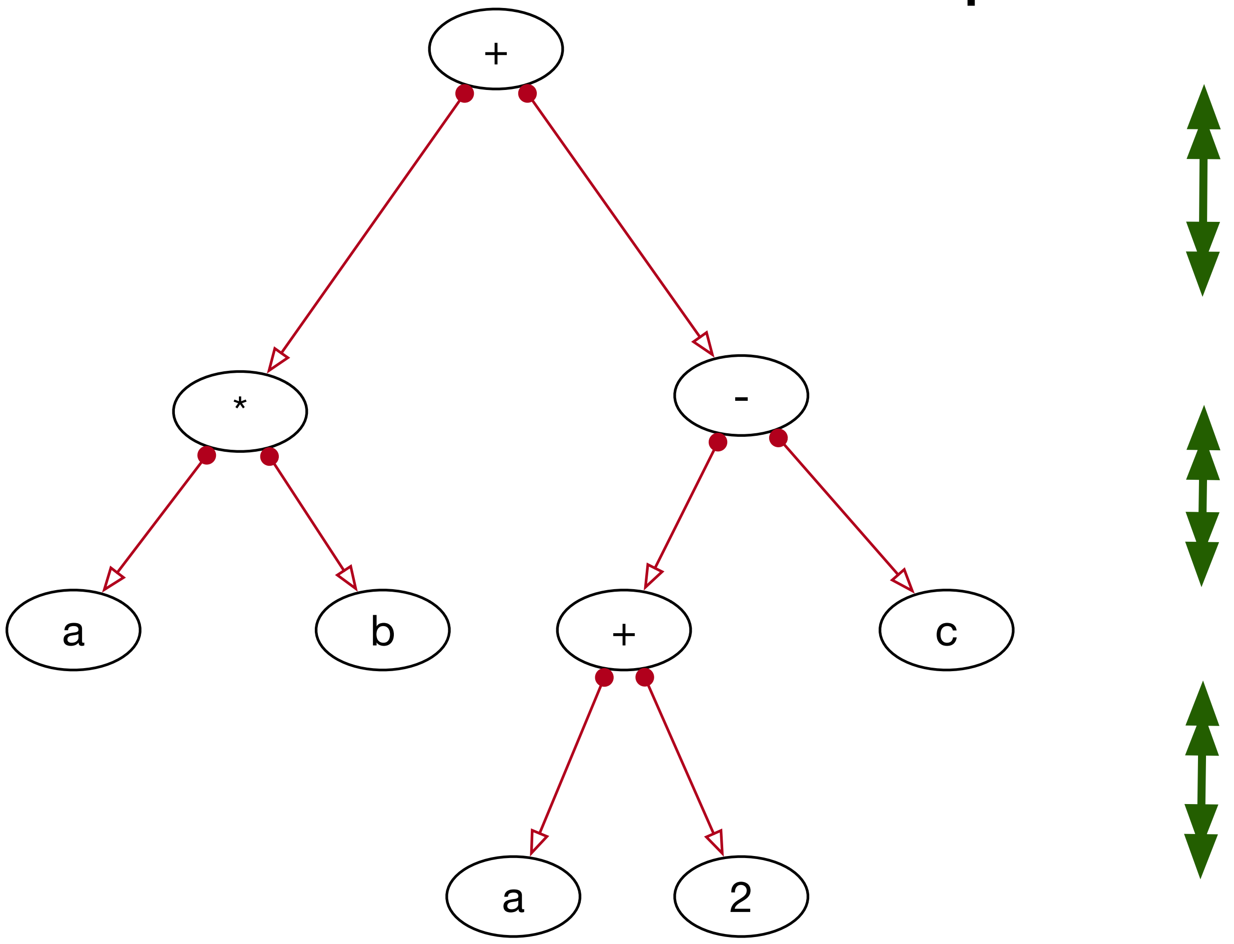
Showing Contents

- Post-order
  - Reverse Polish notation:
    - $ab * a2 + c-$



Geometry

- Depth of a tree:
  - Number of links from root to the furthest node
    - Depth is three in this example



Geometry

- Use recursion

```
def depth(self):
    if not self.root:
        return -1
    else:
        return 1 + max(self.left_tree().depth(),
                       self.right_tree().depth())
```

Geometry

- Number of elements in a tree
  - Let's use Python length
    - Again, use recursion on left and right subtree

```
def __len__(self):
    if not self.root:
        return 0
    else:
        return 1+len(self.left_tree())
                +len(self.right_tree())
```

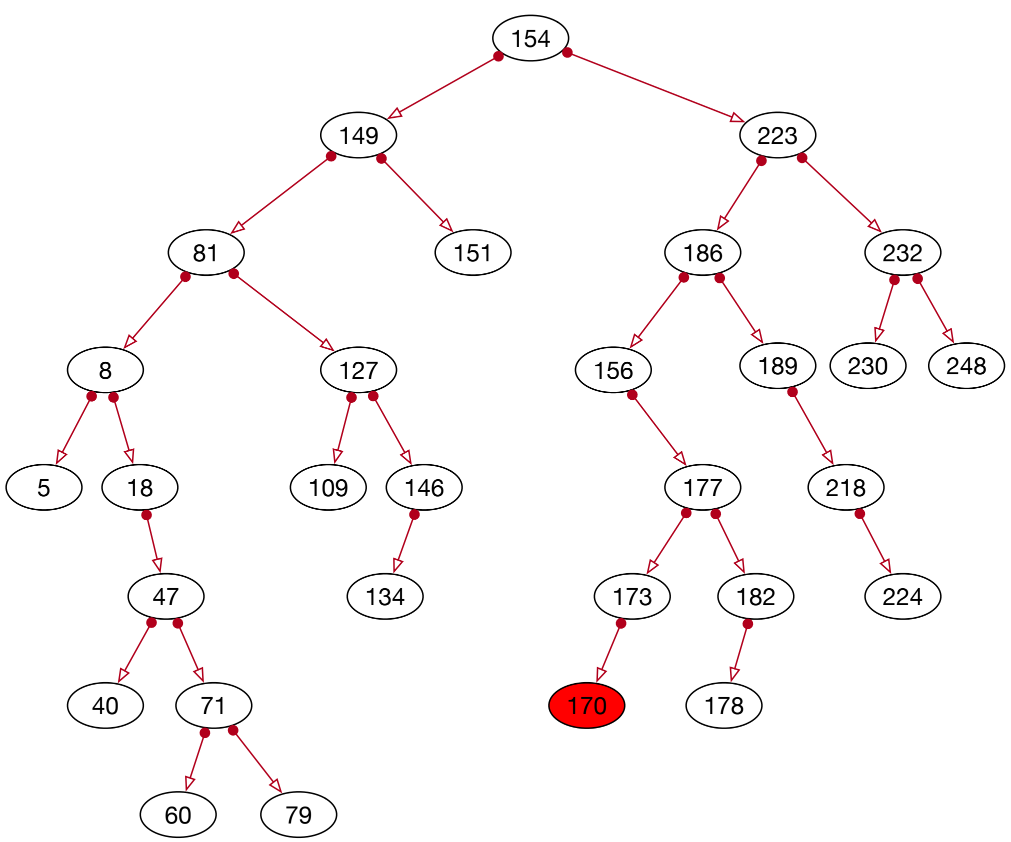
Finding Data

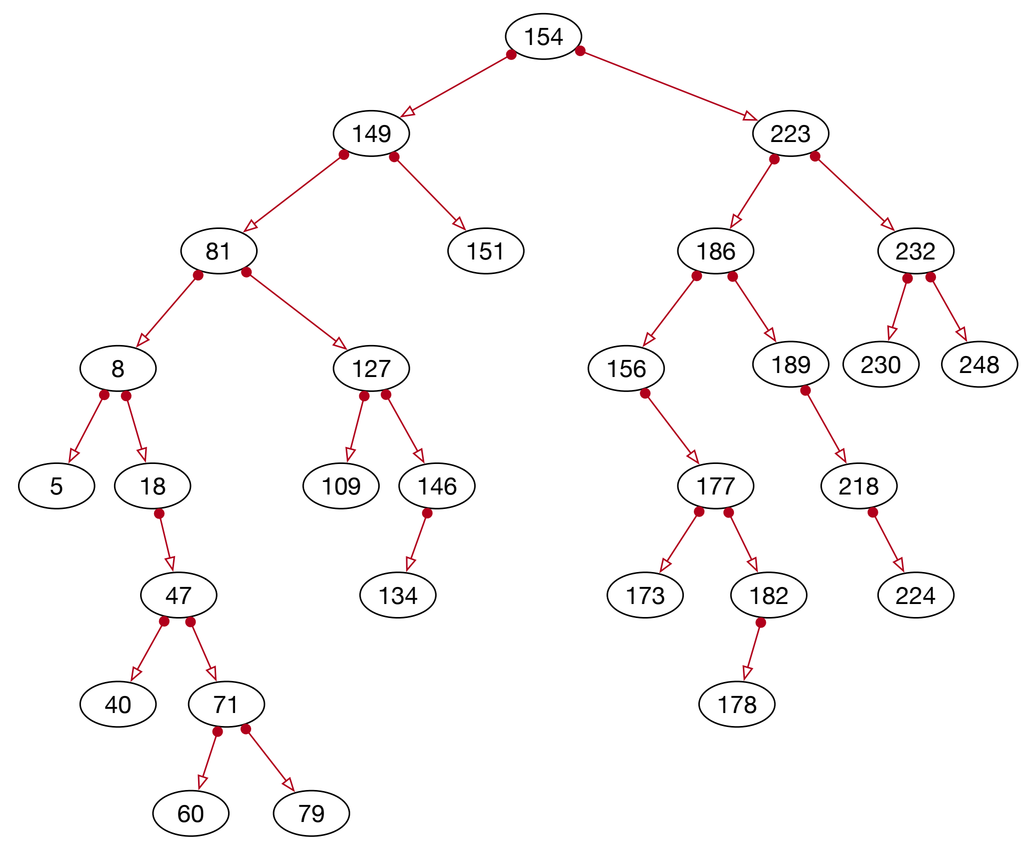
- To find record with certain value:
  - Start at root
    - Go to the left or the right, depending on whether the value is

```
def contains(self, value):
    if not self.root:
        return False
    current = self.root
    while True:
        if not current:
            return False
        if value == current.value:
            return True
        if value < current.value:
            current = current.left
        else:
            current = current.right
```

Deleting Data

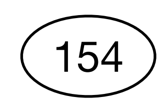
- If the node is a leaf:
  - We just delete the leaf





Deleting Data

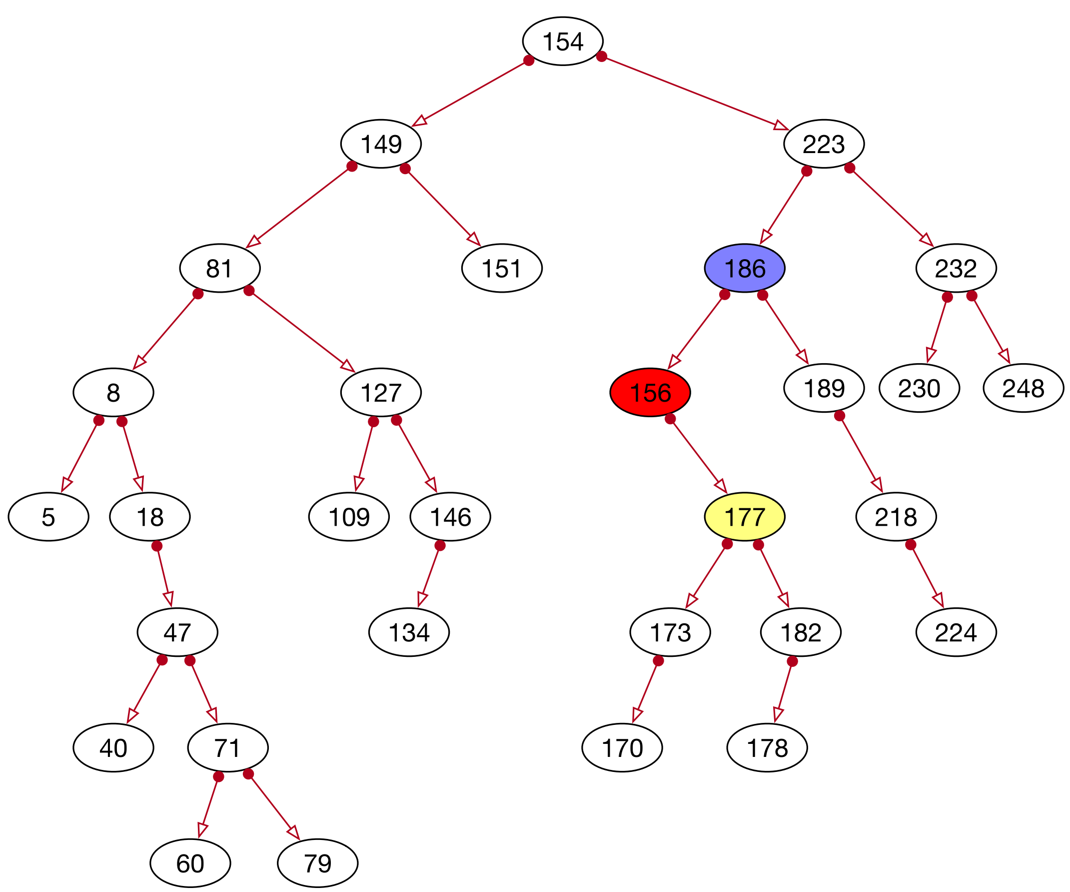
- Special case:
  - The node to be deleted is the root

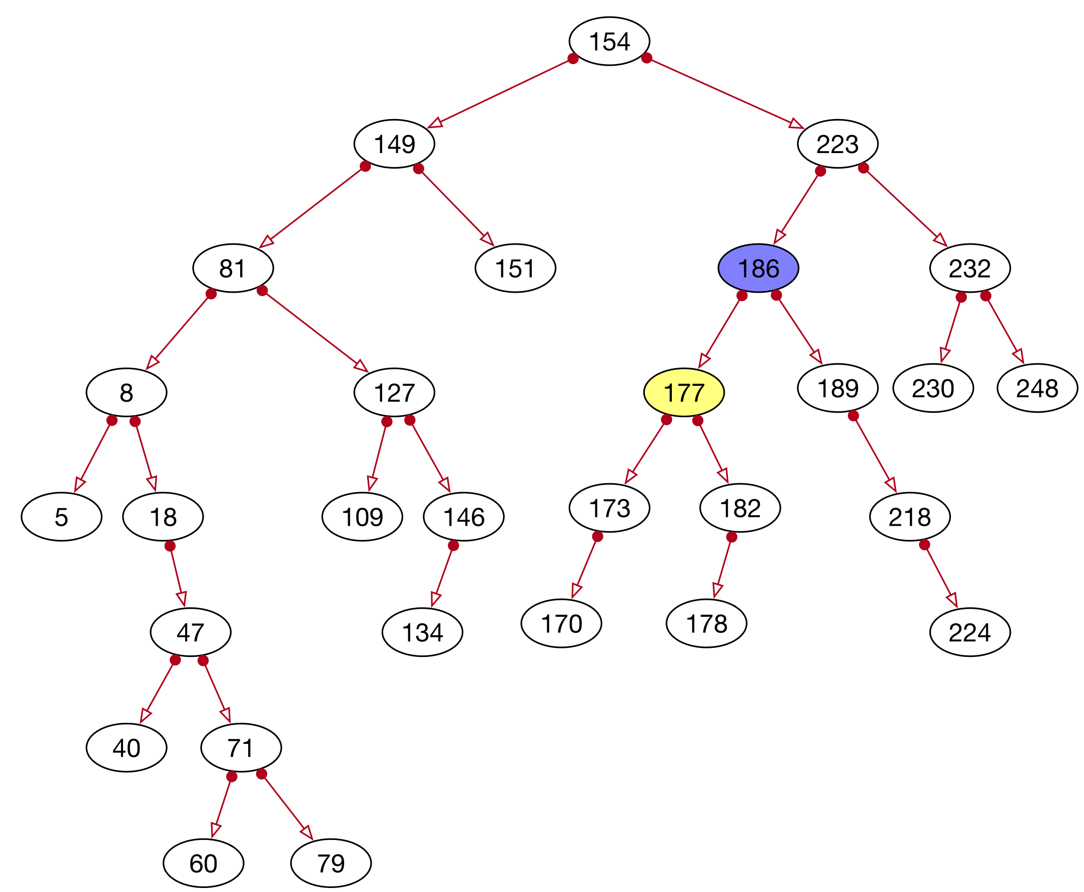


  - In which case we need to set the root to None

Deleting Data

- If the node to be deleted has only one child:
  - Replace the sub-tree rooted in the node with the child tree

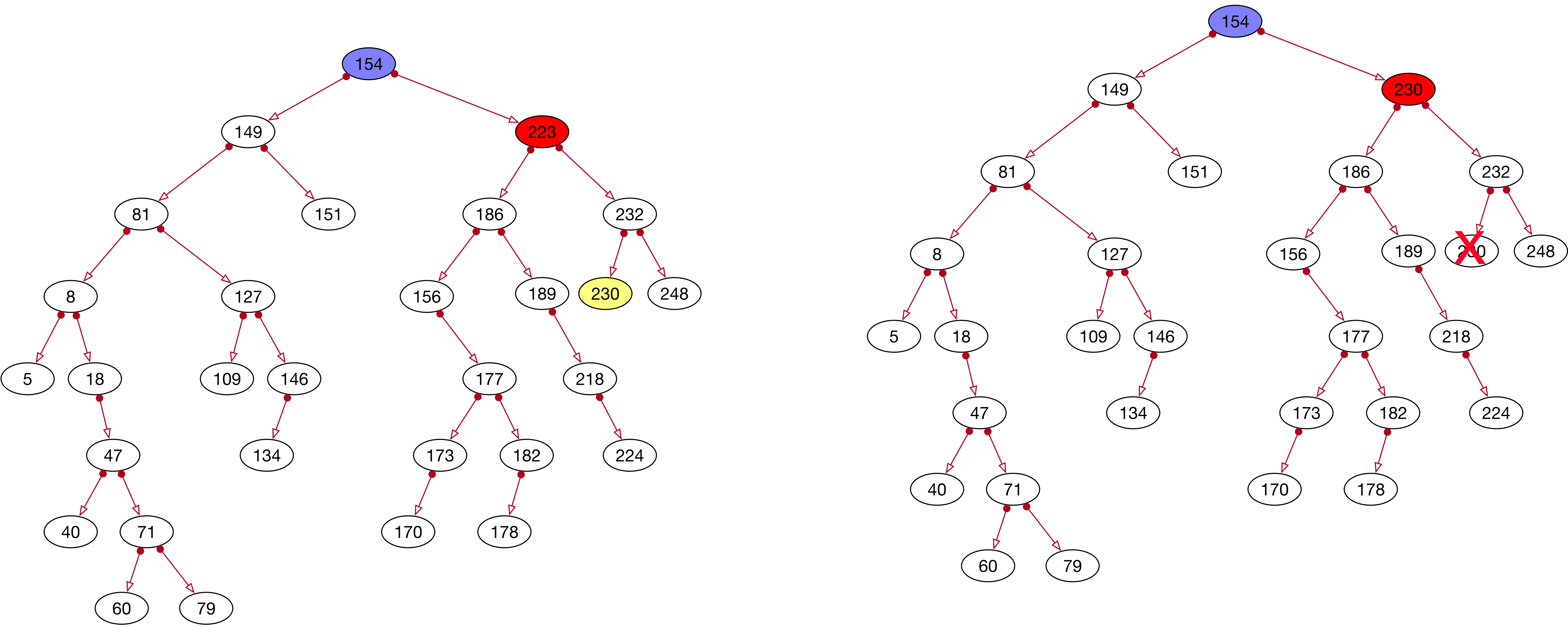




Deleting Data

- If the node to be deleted has two children:
  - Find the successor
    - Leftmost node in the right subtree
  - Save value of the successor
  - Delete the successor
    - For this we need to find its successor
  - Then change the value of the node to be deleted

Deleting Data



Implementation

- Trick:
  - Maintain a queue of all nodes visited
    - So that we can go up
      - Only problem is if we delete at the very top