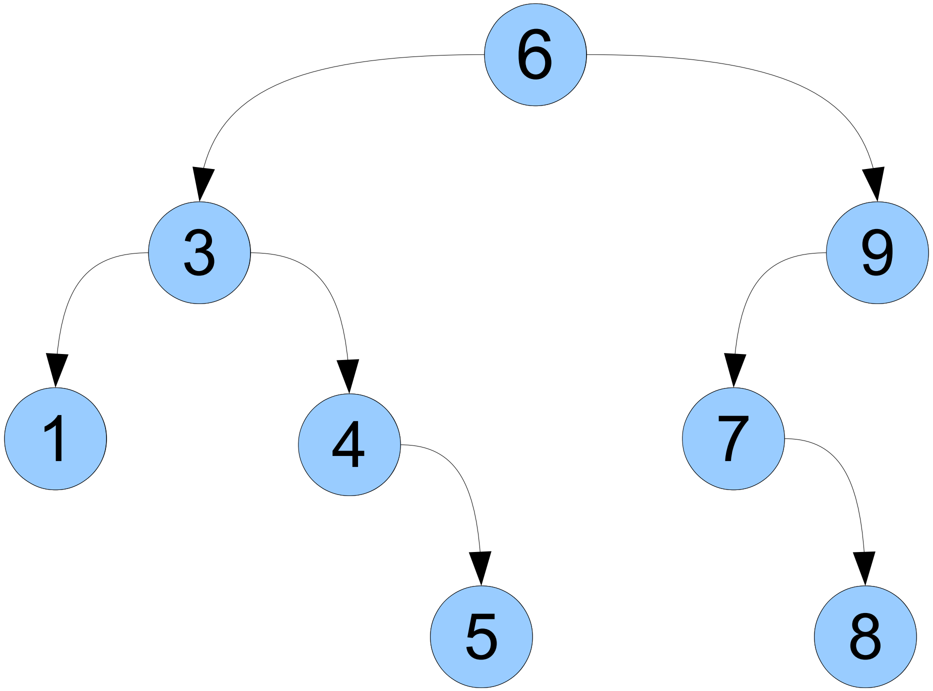
Binary Search Trees

Part Two

Recap from Last Time

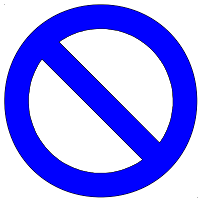
Binary Search Trees

- The data structure we have just seen is called a ***binary search tree*** (or ***BST***).
- The tree consists of a number of ***nodes***, each of which stores a value and has zero, one, or two ***children***.
- All values in a node's left subtree are ***smaller*** than the node's value, and all values in a node's right subtree are ***greater*** than the node's value.

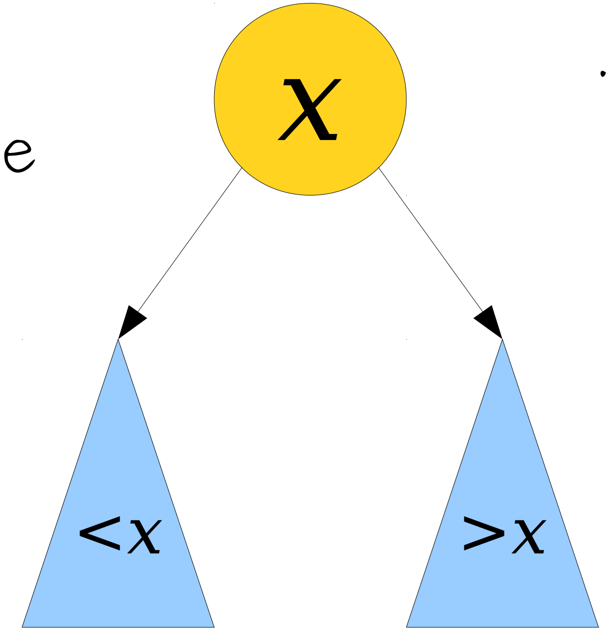


A Binary Search Tree Is Either…

an empty tree,
represented by
nullptr, or…



… a single node,
whose left subtree
is a BST of
smaller values …



… and whose right
subtree is a BST
of larger values.

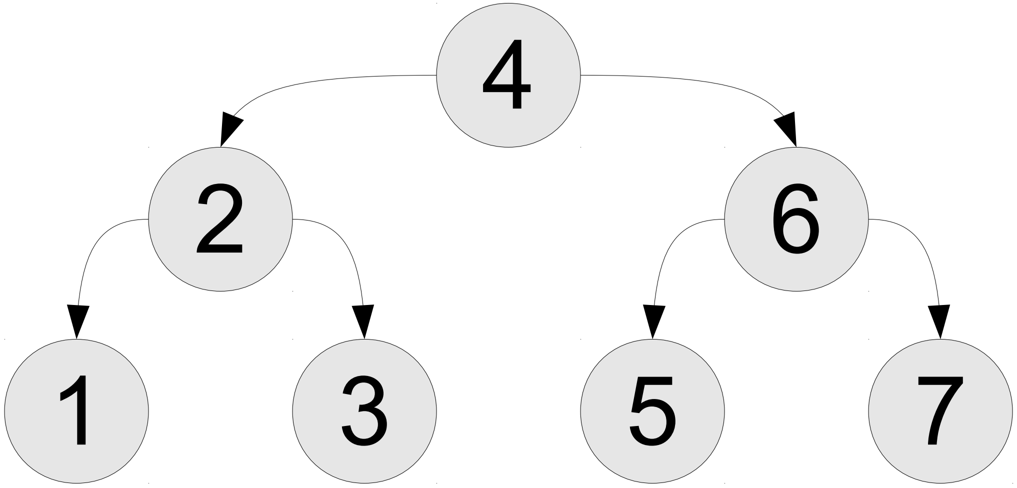
New Stuff!

Getting Rid of Trees

http://www.tigersheds.com/garden-resources/image.axd?picture=2010%2F6%2Fdeforestation1.jpg

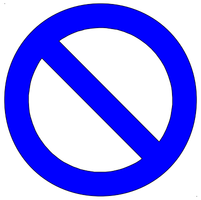
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.

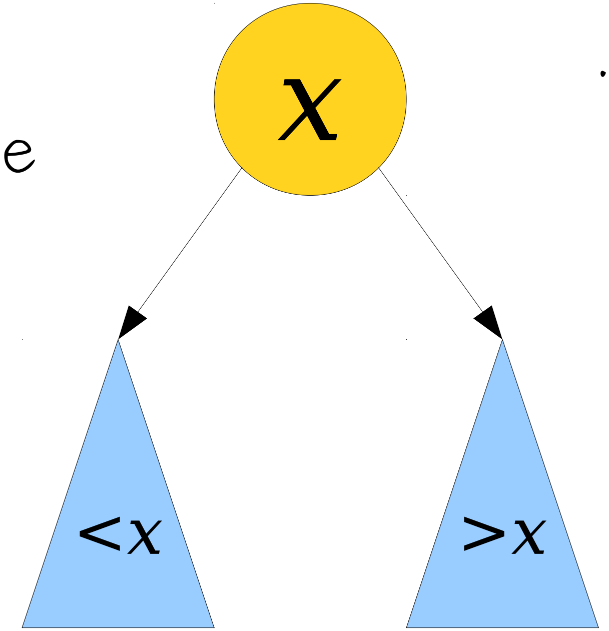


A Binary Search Tree Is Either…

an empty tree,
represented by
nullptr, or…



… a single node,
whose left subtree
is a BST of
smaller values …



… and whose right
subtree is a BST
of larger values.

Postorder Traversals

- The particular recursive pattern we just saw is called a ***postorder traversal*** of a binary tree.
- Specifically:
  - Recursively visit all the nodes in the left subtree.
  - Recursively visit all the nodes in the right subtree.
  - Visit the node itself.

Tree Efficiency

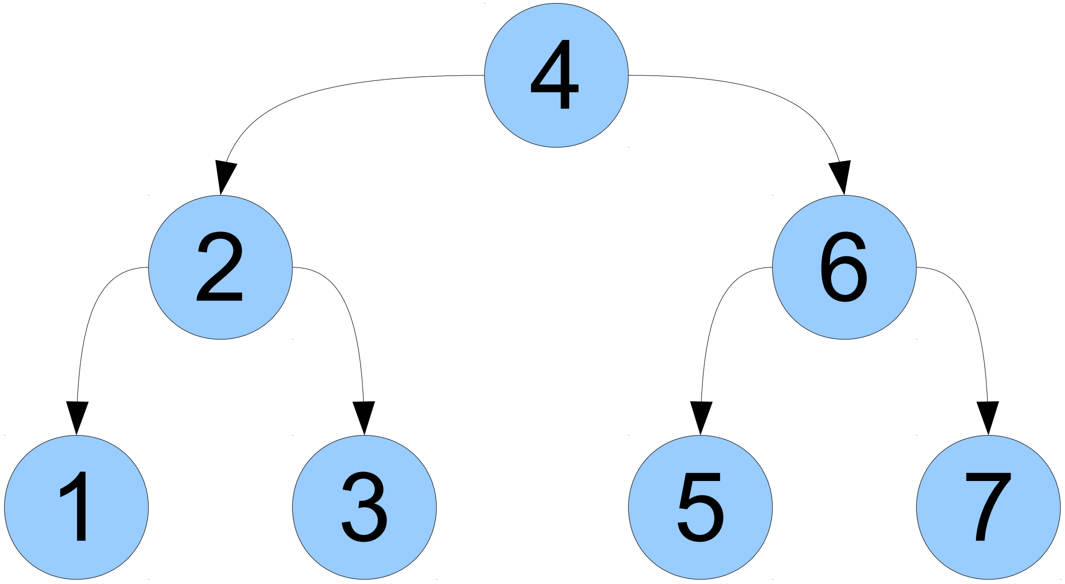
How fast are BST lookups?

How fast are BST insertions?

Insertion Order Matters

- You can have multiple BSTs holding the same elements
- Here’s the BST we get by inserting these elements in this order:

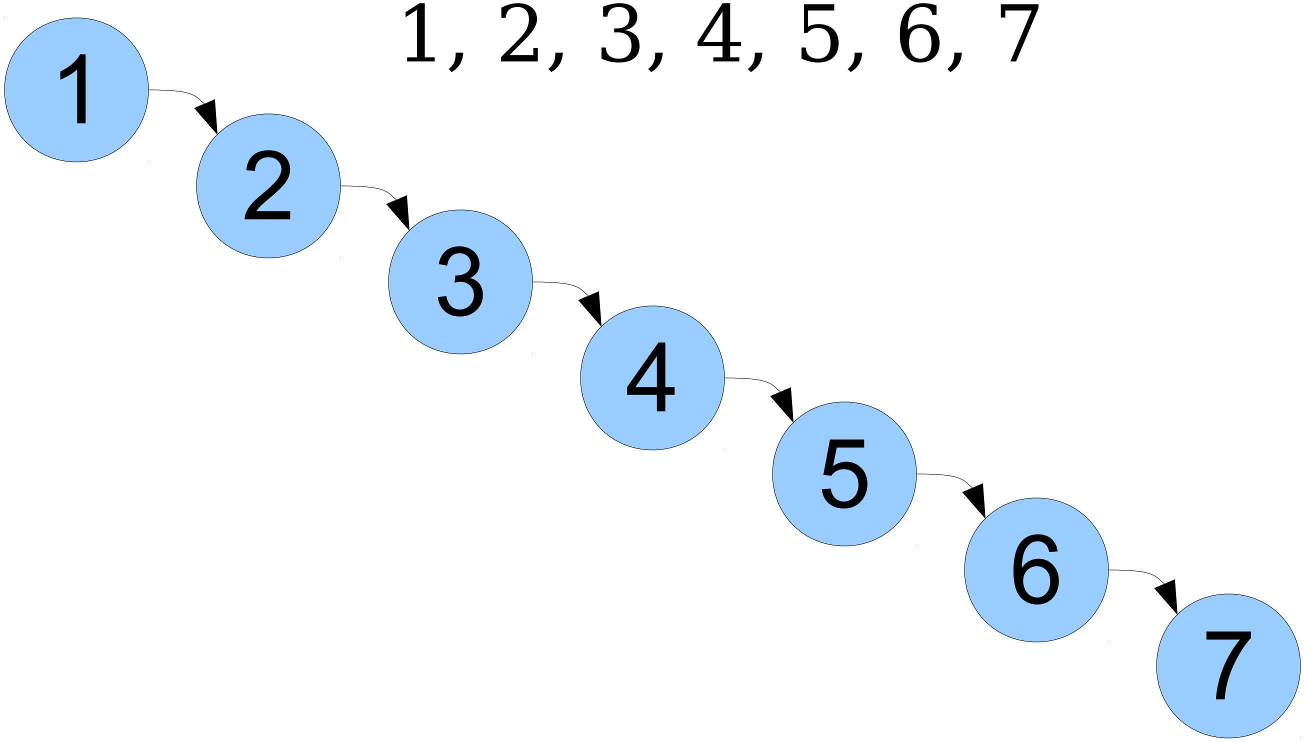
4, 2, 1, 3, 6, 5, 7



Insertion Order Matters

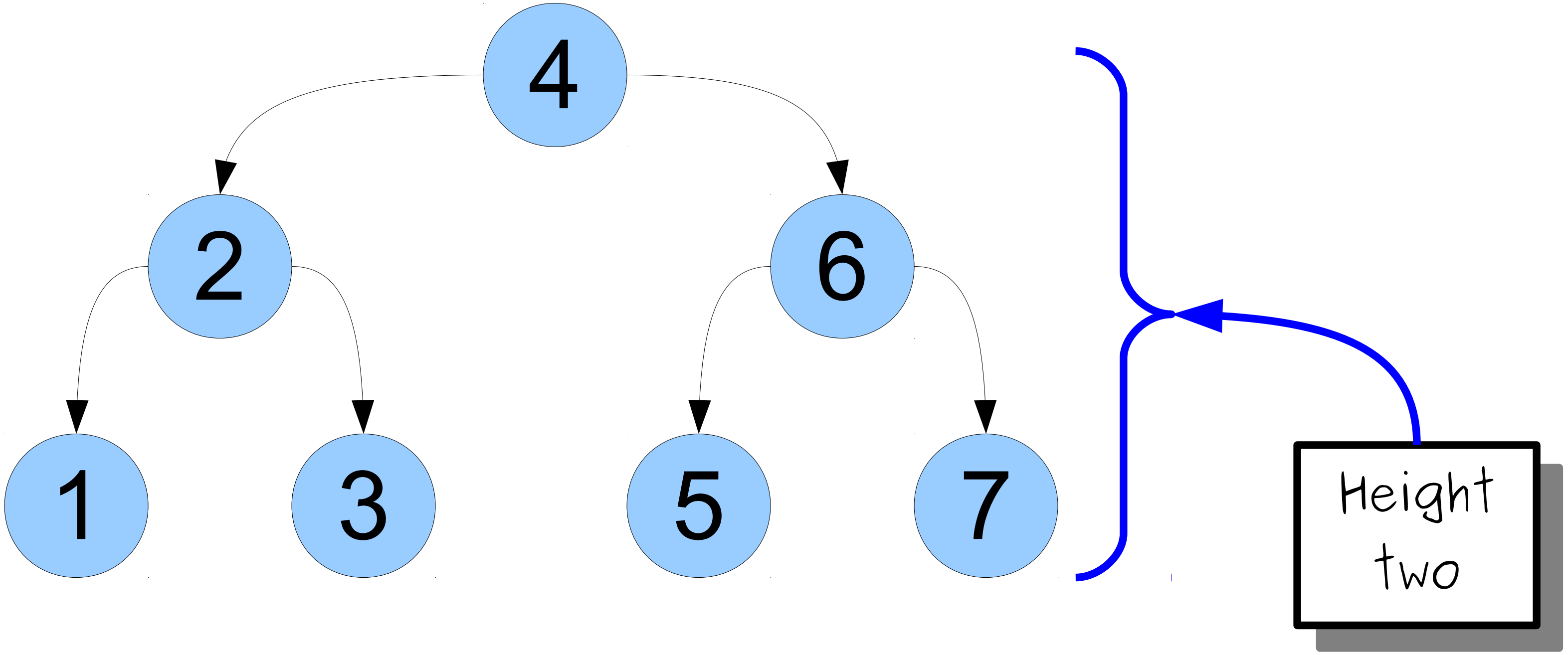
- You can have multiple BSTs holding the same elements
- Here's the BST we get by inserting these elements in this order:

1, 2, 3, 4, 5, 6, 7



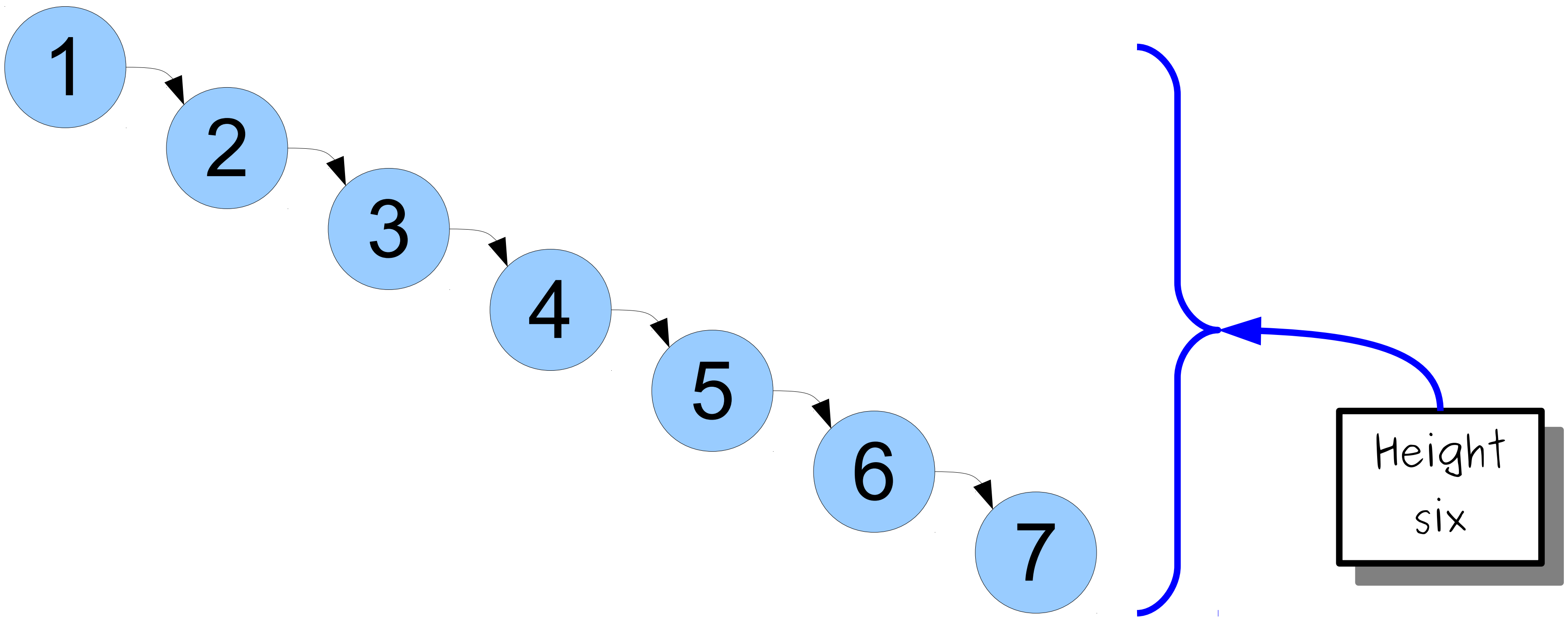
Tree Terminology

- The ***height*** of a tree is the number of nodes in the longest path from the root to a leaf.



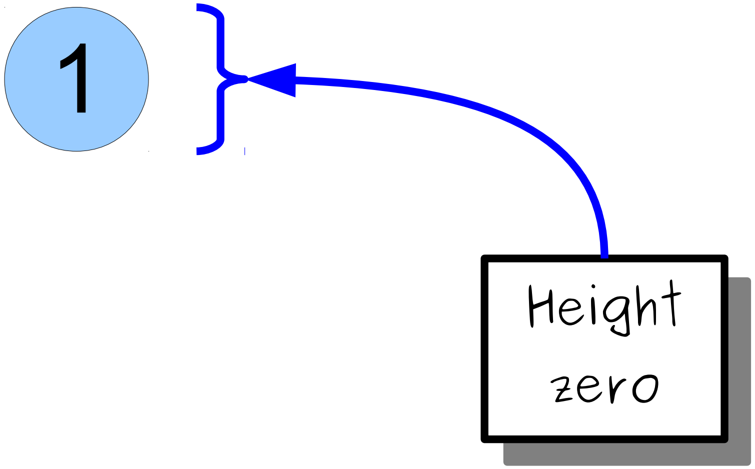
Tree Terminology

- The ***height*** of a tree is the number of nodes in the longest path from the root to a leaf.



Tree Terminology

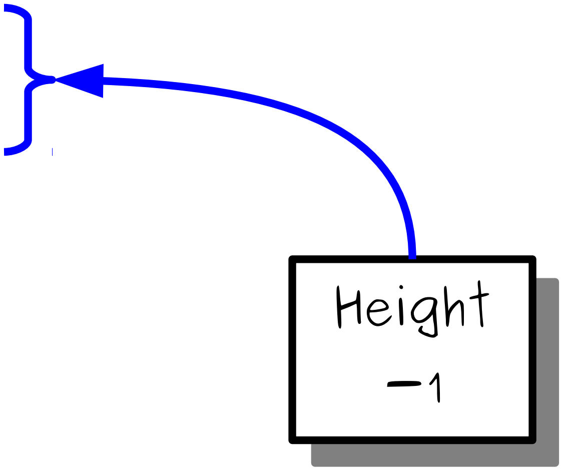
- The ***height*** of a tree is the number of nodes in the longest path from the root to a leaf.



Height zero

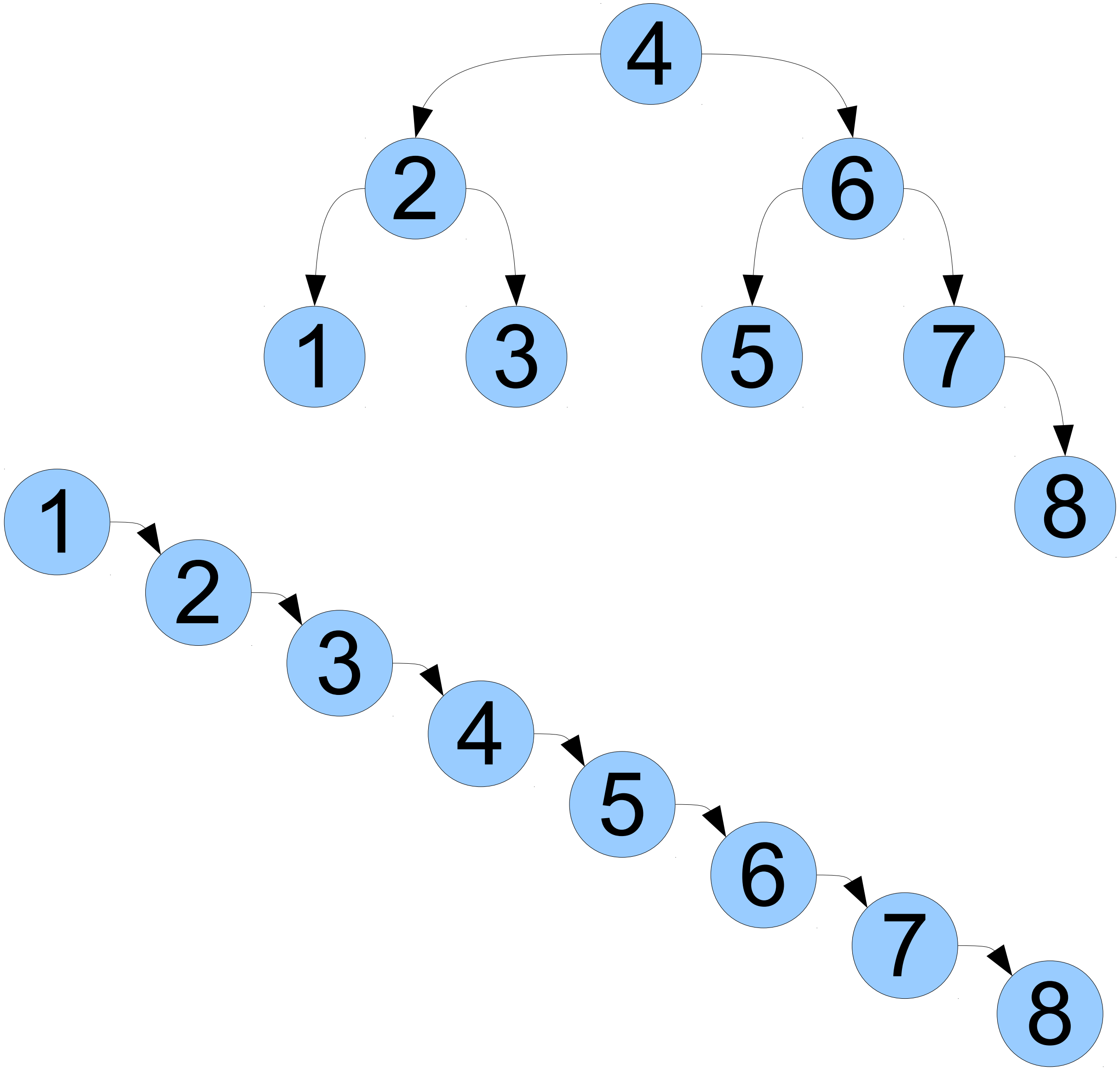
Tree Terminology

- The ***height*** of a tree is the number of nodes in the longest path from the root to a leaf.
- By convention, an empty tree has height -1.



Efficiency Questions

- The time to add an element to a BST (or look up an element in a BST) depends on the height of the tree.
- The runtime is **O(*h*)**, where *h* is the height of the tree.



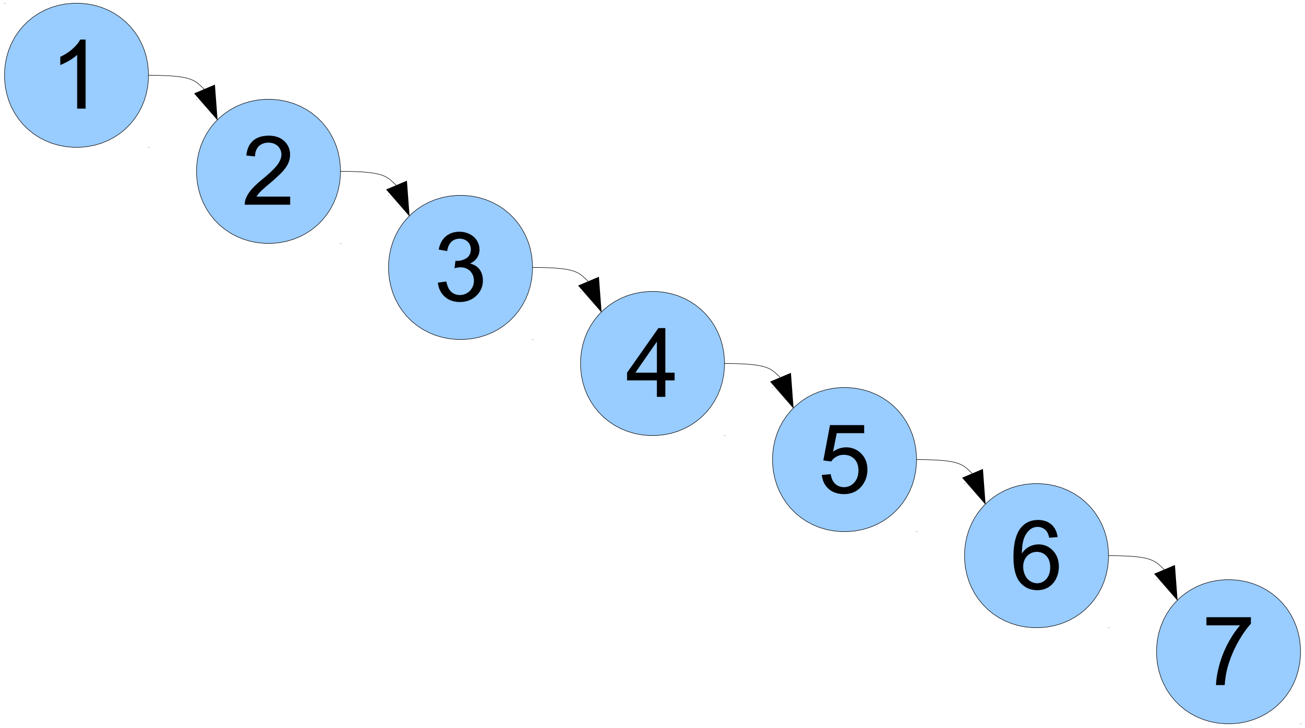
The cost of an operation on a BST is O(h), where h is the height of the tree.

Is there a connection between h, the tree height, and n, the number of nodes?

Balanced Trees

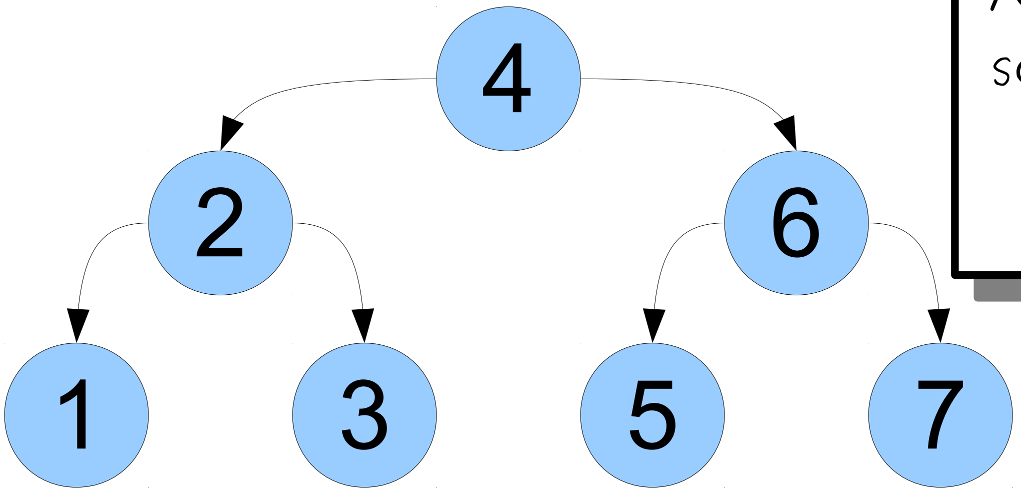
Tree Heights

- What are the maximum and minimum heights of a tree with *n* nodes?
- Maximum height: all nodes in a chain. Height is O(*n*).



Tree Heights

- What are the maximum and minimum heights of a tree with *n* nodes?
- Maximum height: all nodes in a chain. Height is O(*n*).
- Minimum height: Tree is as complete as possible. Height is O(log *n*).



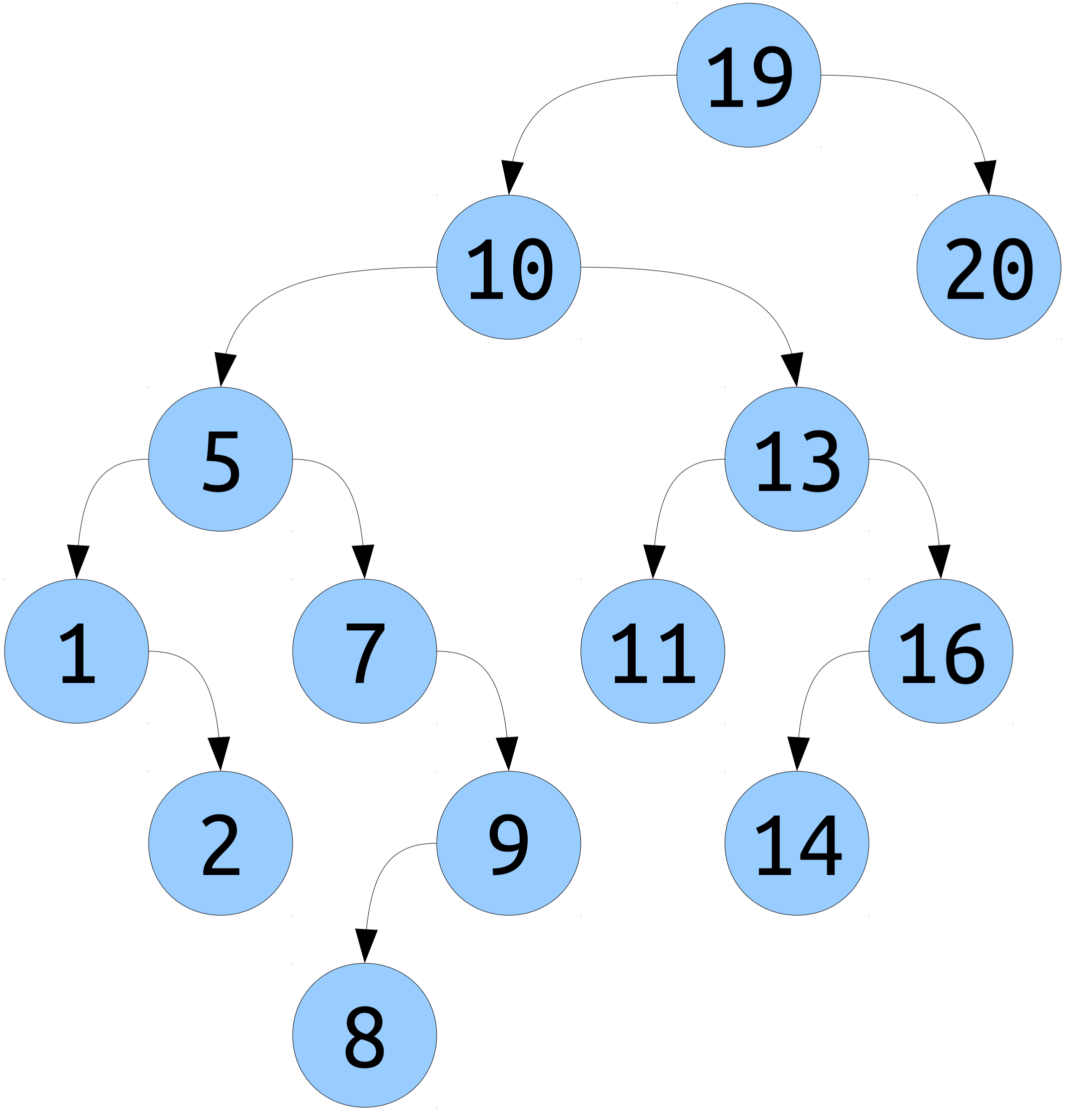
You can only double something O(log *n*) times before it exceeds *n*.

Balanced Trees

- A binary search tree is called ***balanced*** if its height is O(log *n*), where *n* is the number of nodes in the tree.
- Balanced trees are extremely efficient:
  - Lookups take time O(log *n*).
  - Insertions take time O(log *n*).
  - Deletions take time O(log *n*).
- ***Question:*** How do you balance a tree?

Balanced Trees

- ***Theorem:*** If you start with an empty tree and add in random values, then, with high probability, the tree is balanced.
- ***Proof:*** Take CS161!
- ***Takeaway:*** If you're adding elements to a BST and their values are actually random, then your tree is likely to be balanced.



Balanced Trees

- A ***self-balancing tree*** is a BST that reshapes itself on insertions and deletions to stay balanced.
- There are many strategies for doing this. They're beautiful. They're clever. And they're beyond the scope of CS106B.
- Some suggested topics to read up on, if you're curious:
  - Red/black trees (take CS161 or CS166!)
  - AVL trees (covered in the textbook)
  - Splay trees (trees that reshape on lookups)
  - Scapegoat trees (yes, that's what they're called)
  - Treaps (half binary heap, half binary search tree!)

Balanced Trees

- If you’re given a collection of values to put in a BST, and they’re already sorted, you can construct a perfectly-balanced tree from them.
- Things to think about:
    - Which element would you put up at the root?
    - What would the children of that element be?
- These are great questions to think through.

Time-Out for Announcements!

Midterm Graded

- We’ve finished grading the midterm exam. Exam scores were sent out over email last night.
  - Didn’t hear from us? Let us know!
- Be sure to read the solutions handout. It contains several solutions for each problem, information about why we asked each question, common mistakes, and strategies for improving.
- There is still plenty of time to improve with these concepts. Talk to your section leader or to Kate or me if you have any questions. We’re here to help out!

Some Resources

- Code Step by Step has a bunch of practice problems to work through on a variety of topics:

  https://codestepbystep.com/

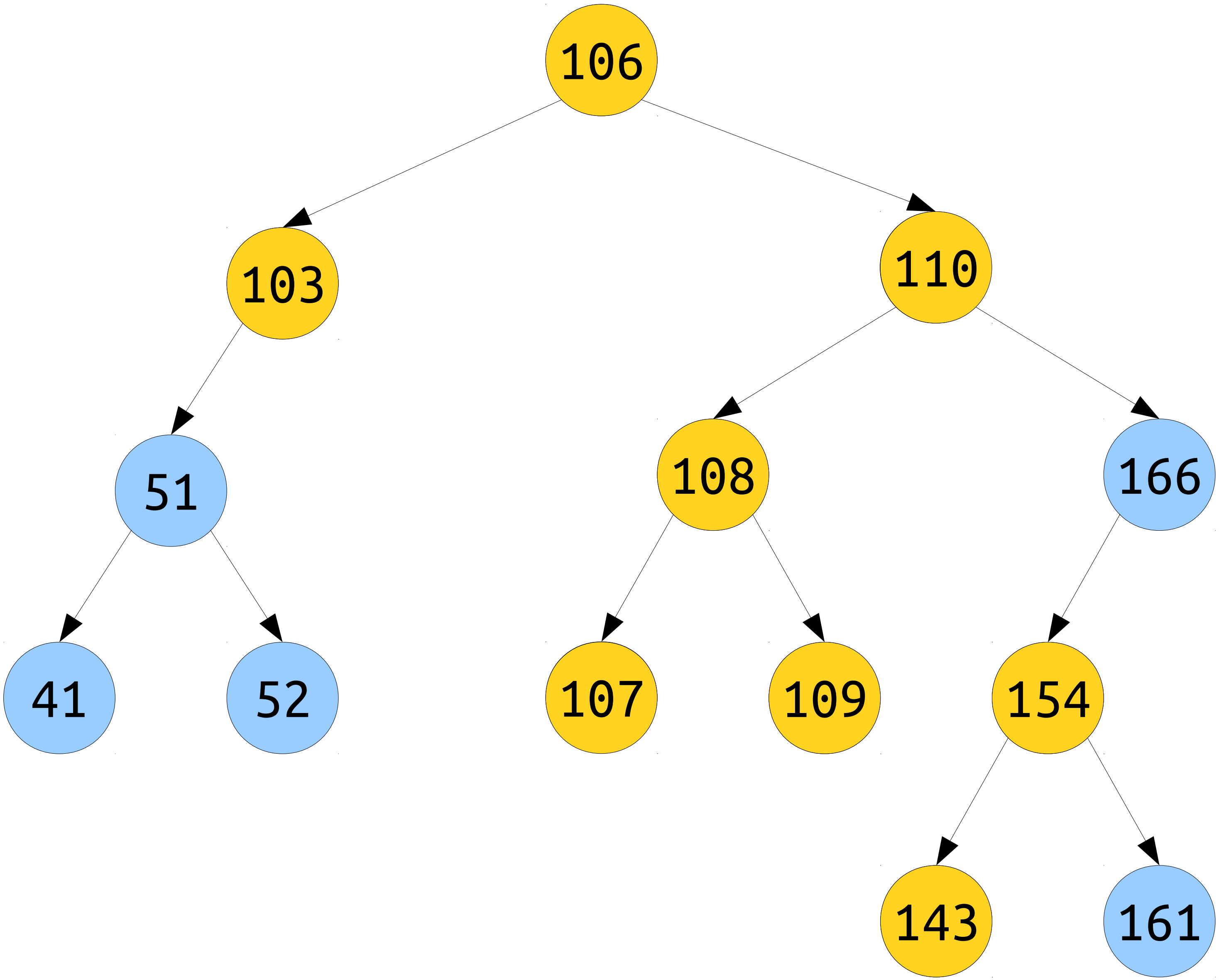
- It’s maintained by Marty Stepp, who is a fantastic CS106B instructor.

Assignment 5

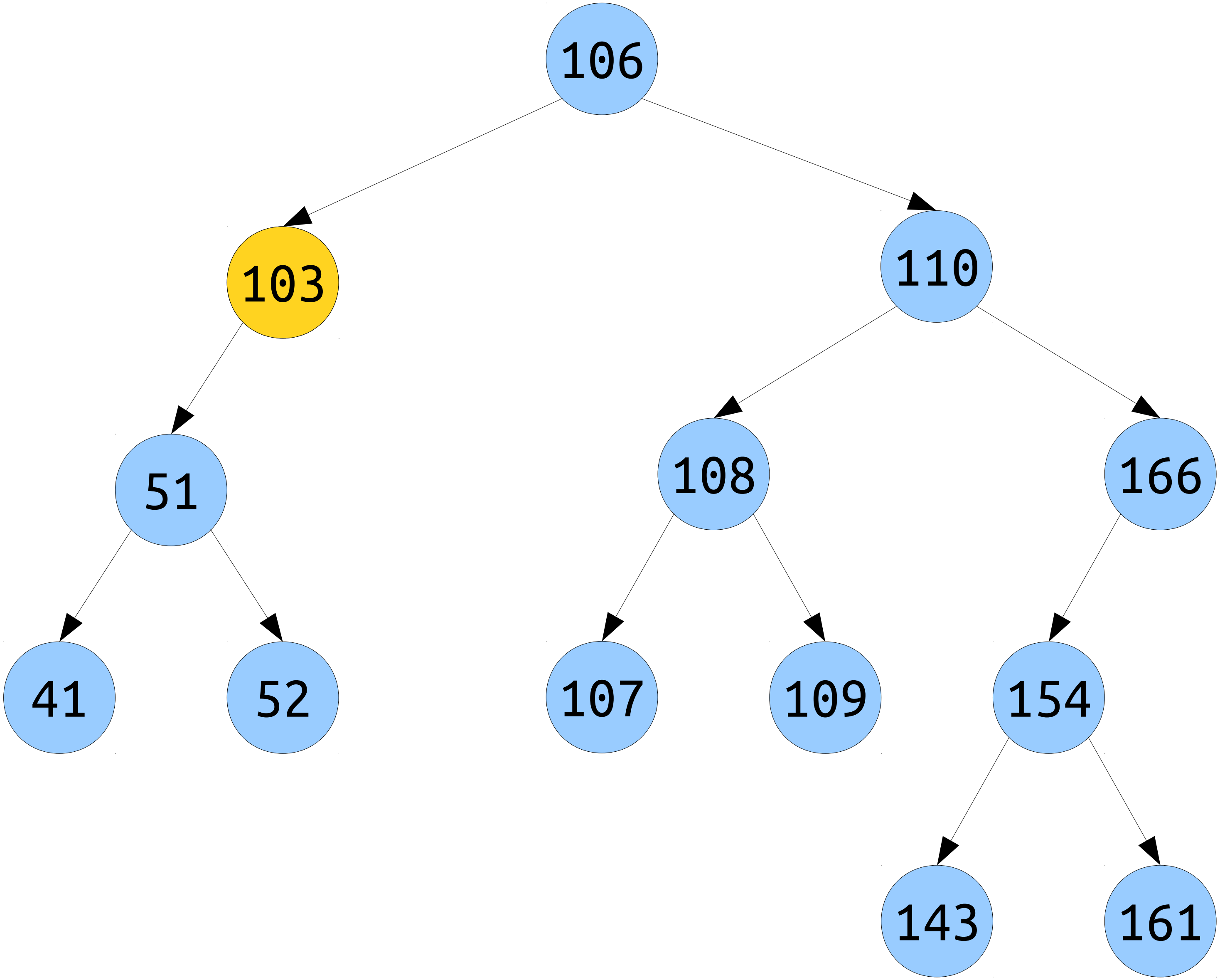
- Assignment 5 is due this Wednesday at the start of class.
- You know the drill – stop by the LaIR or CLaIR with questions, or email your section leader!

Back to CS106B!

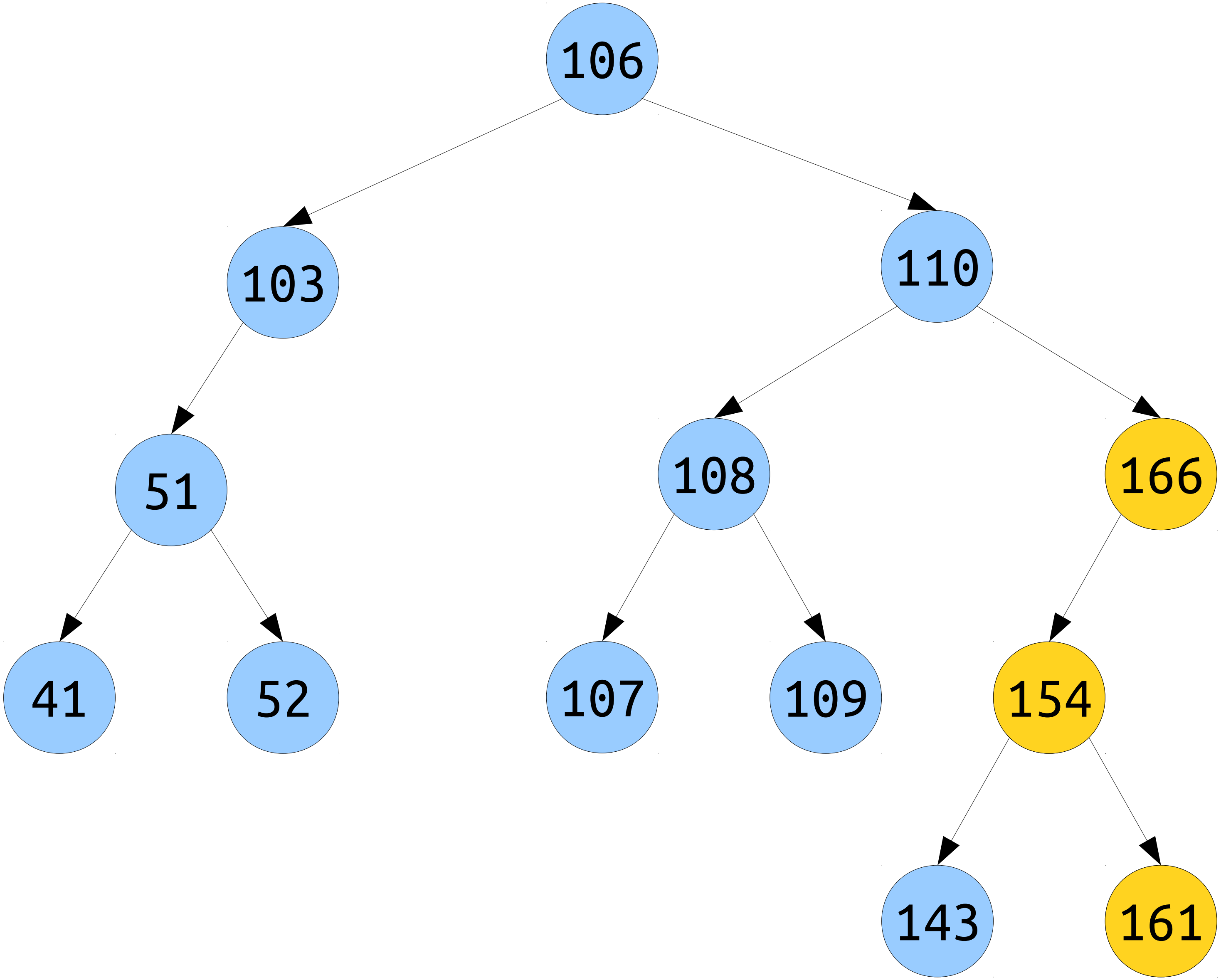
Range Searches



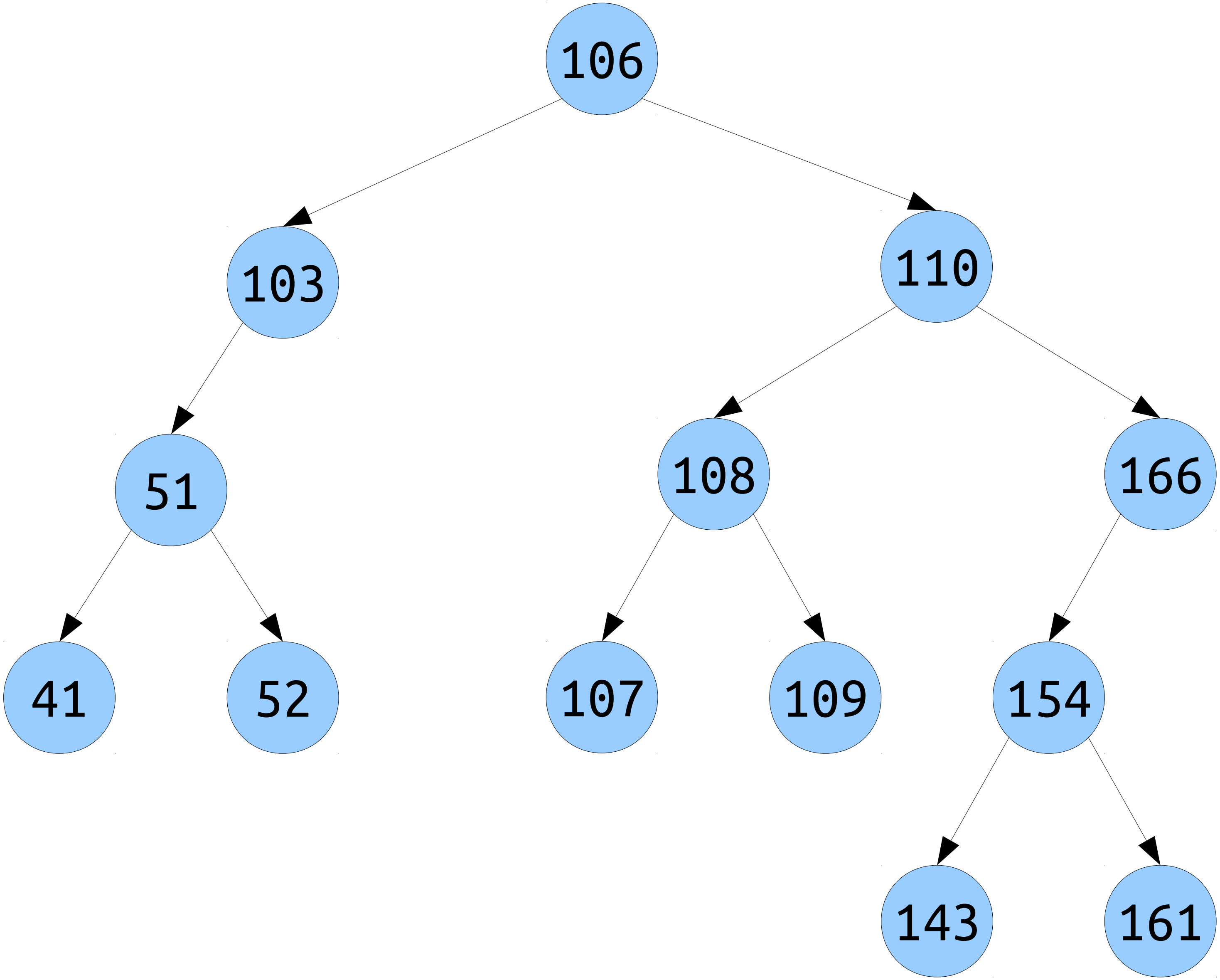
Find all elements in this tree in the range **[103, 154]**.



Find all elements in this tree in the range **[99, 105]**.



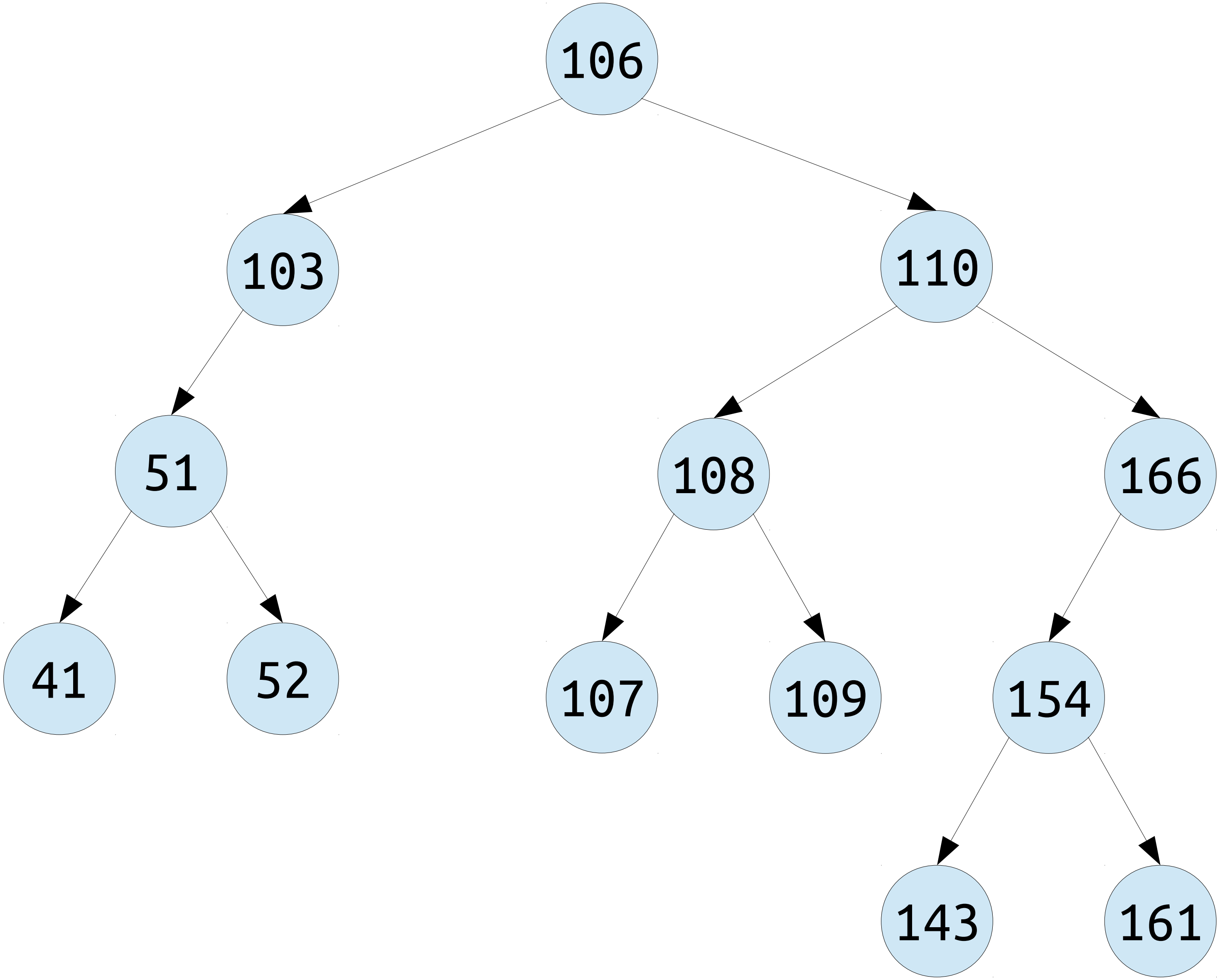
Find all elements in this tree in the range **[150, 170]**.



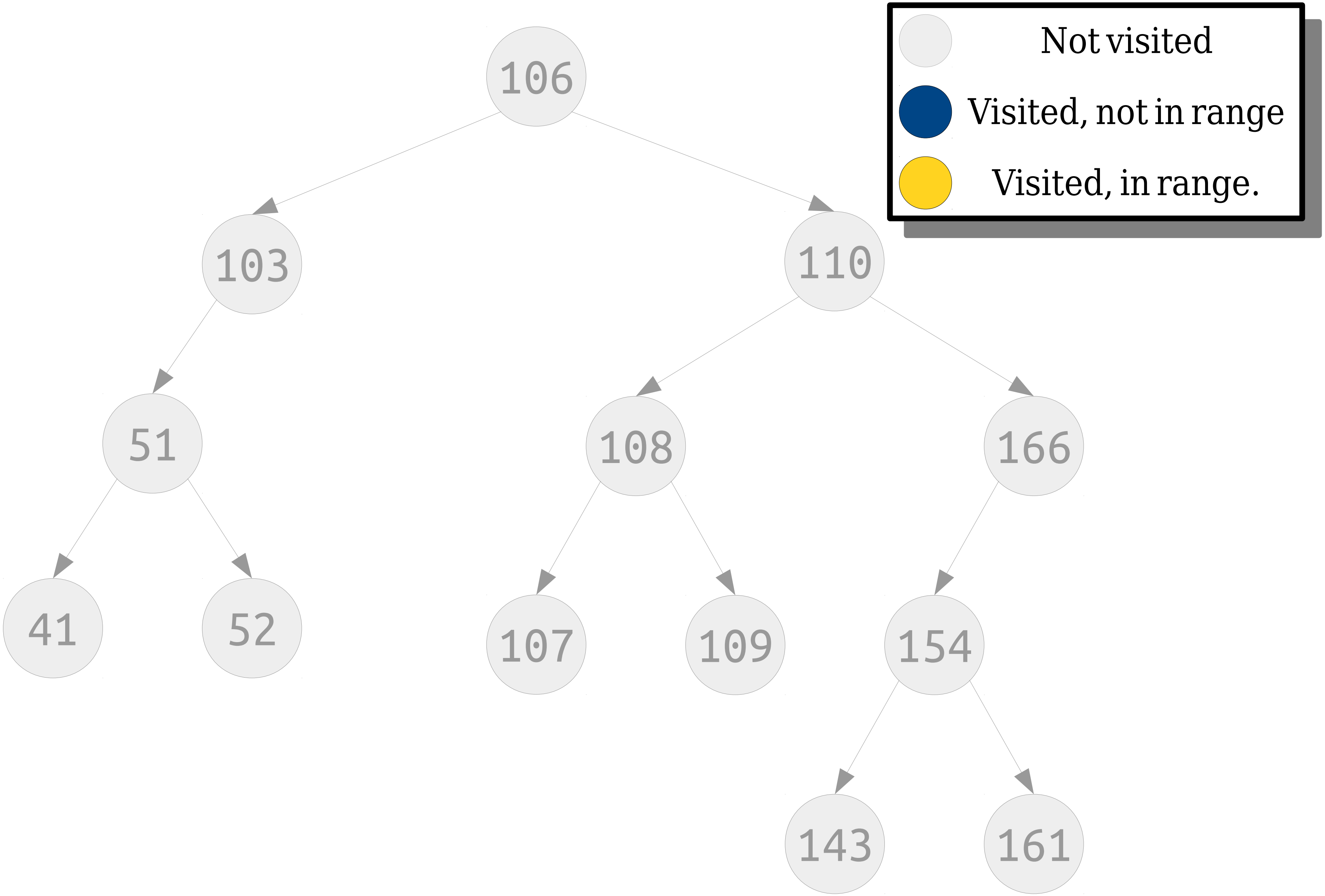
Find all elements in this tree in the range **[137, 138]**.

Range Searches

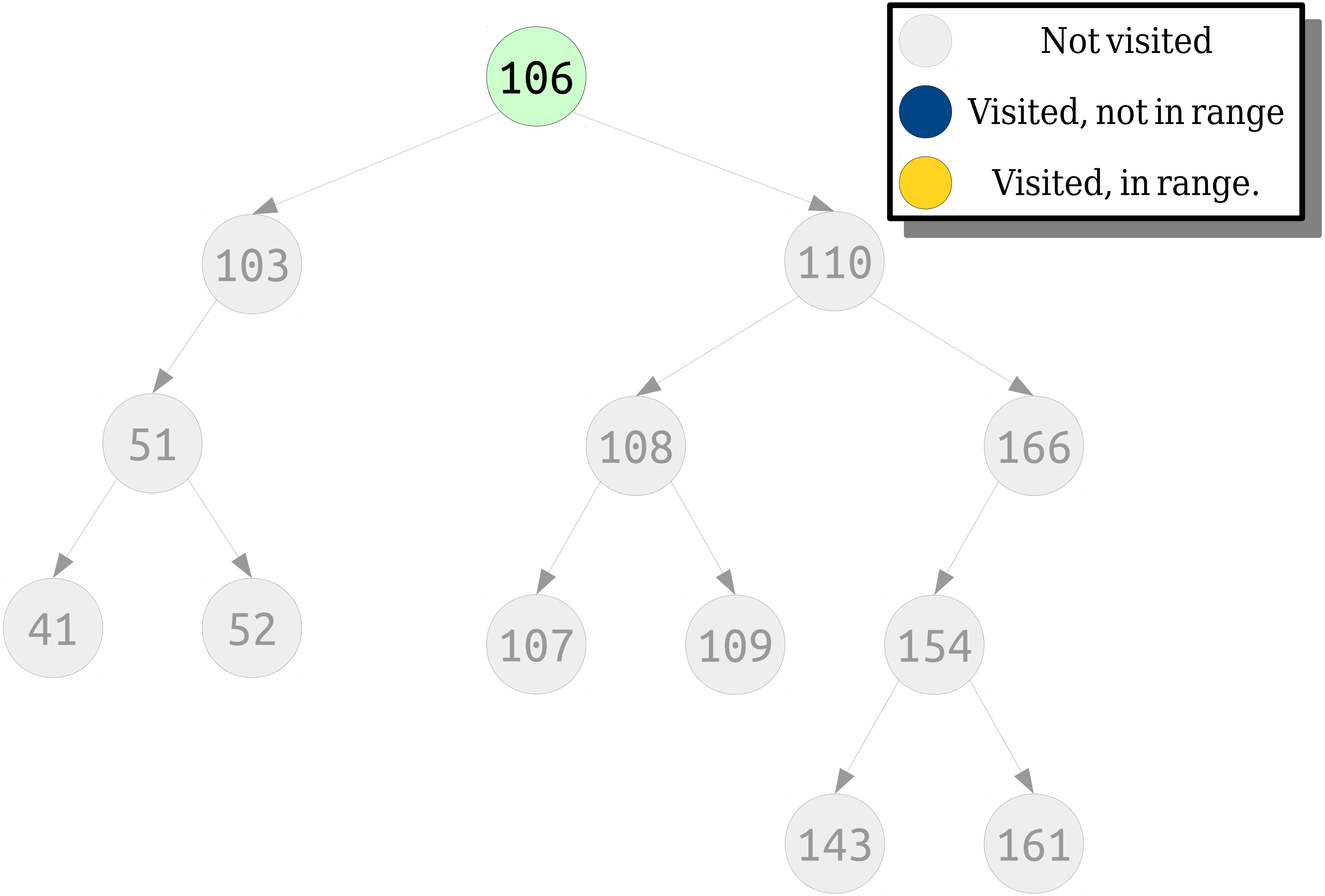
- We can use BSTs to do ***range searches***, in which we find all values in the BST within some range.
- For example:
  - If the values in the BST are dates, we can find all events that occurred within some time window.
  - If the values in the BST are number of diagnostic scans ordered, we can find all doctors who order a disproportionate number of scans.



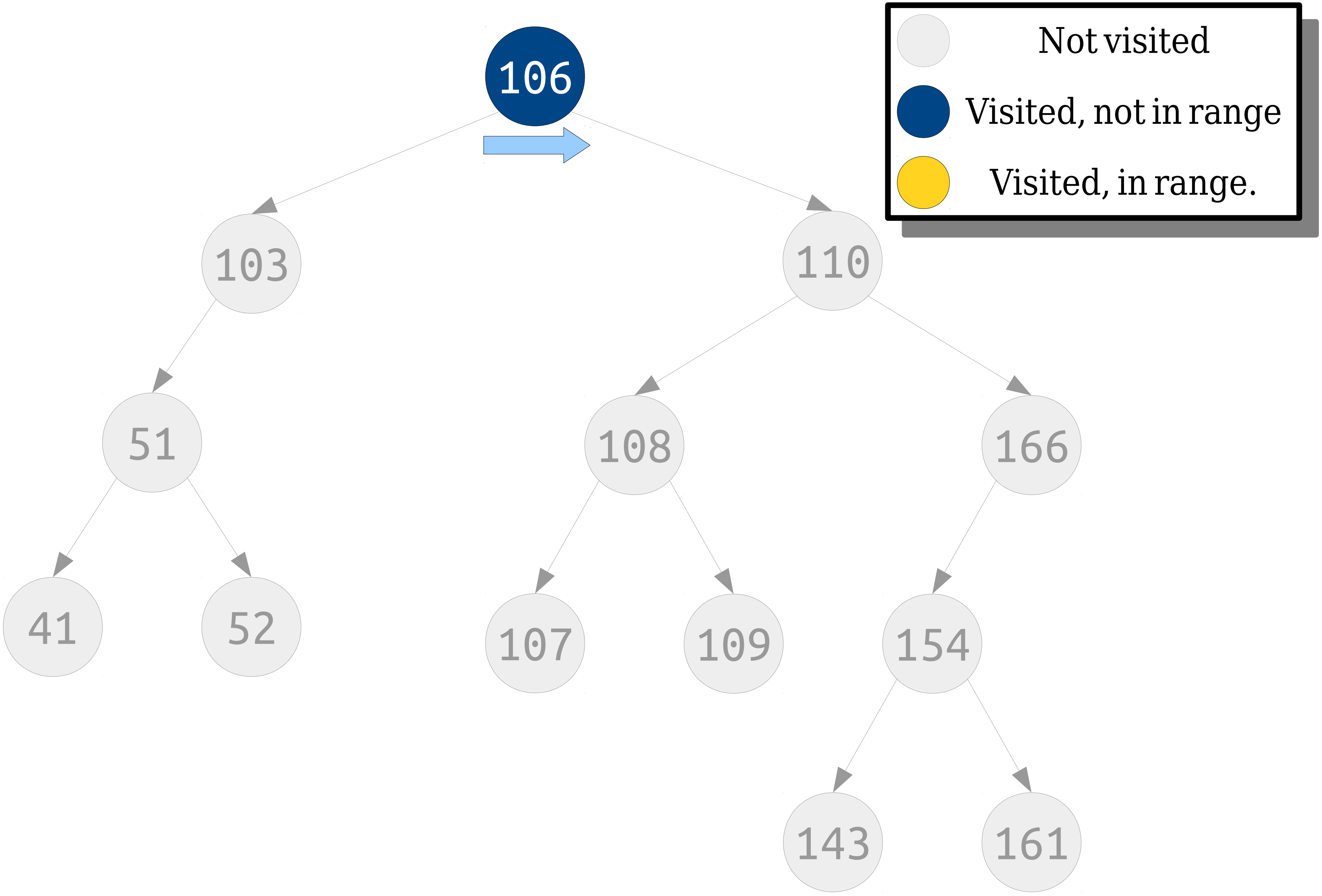
Find all elements in this tree in the range **[109, 163]**.



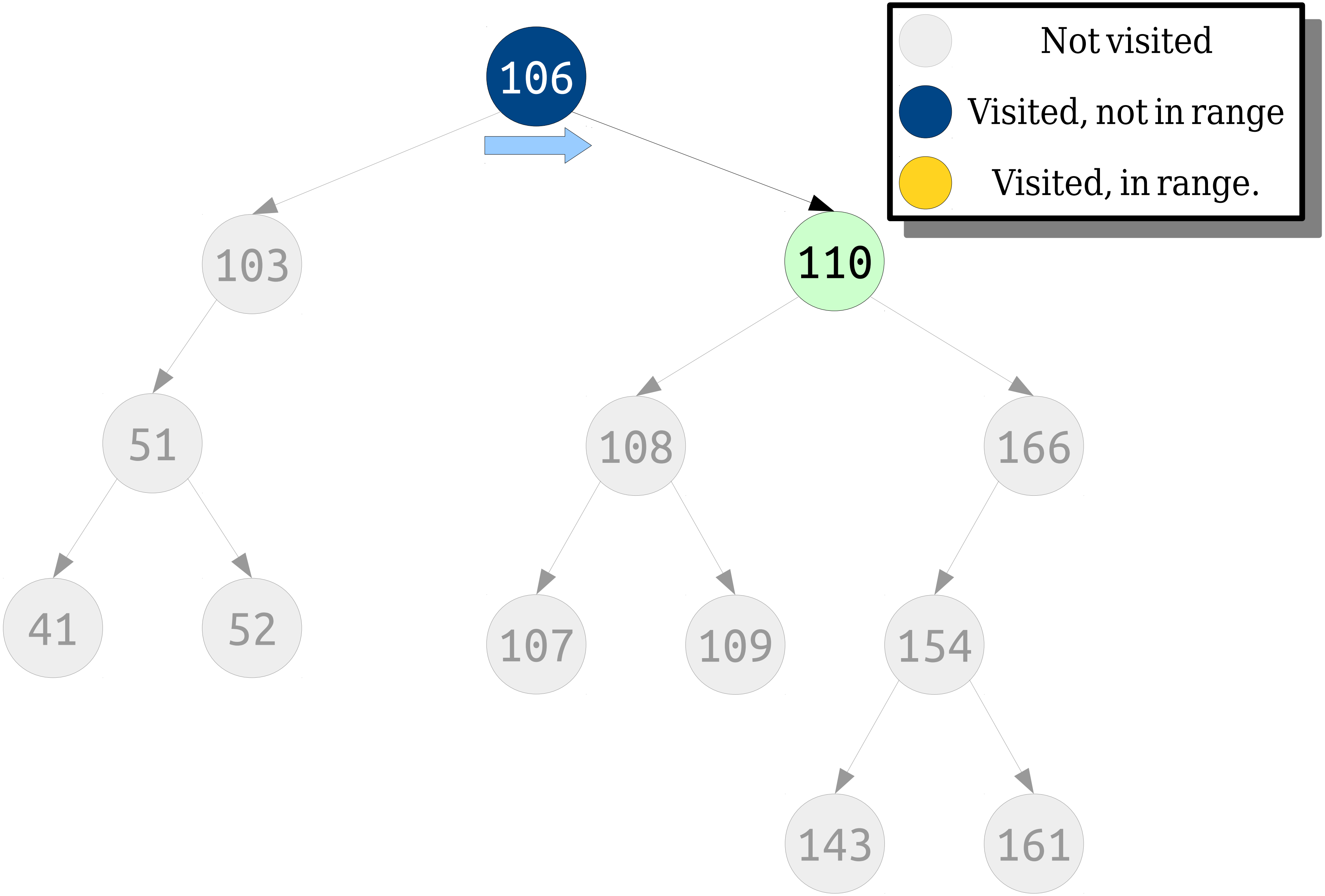
Find all elements in this tree in the range **[109, 163]**.



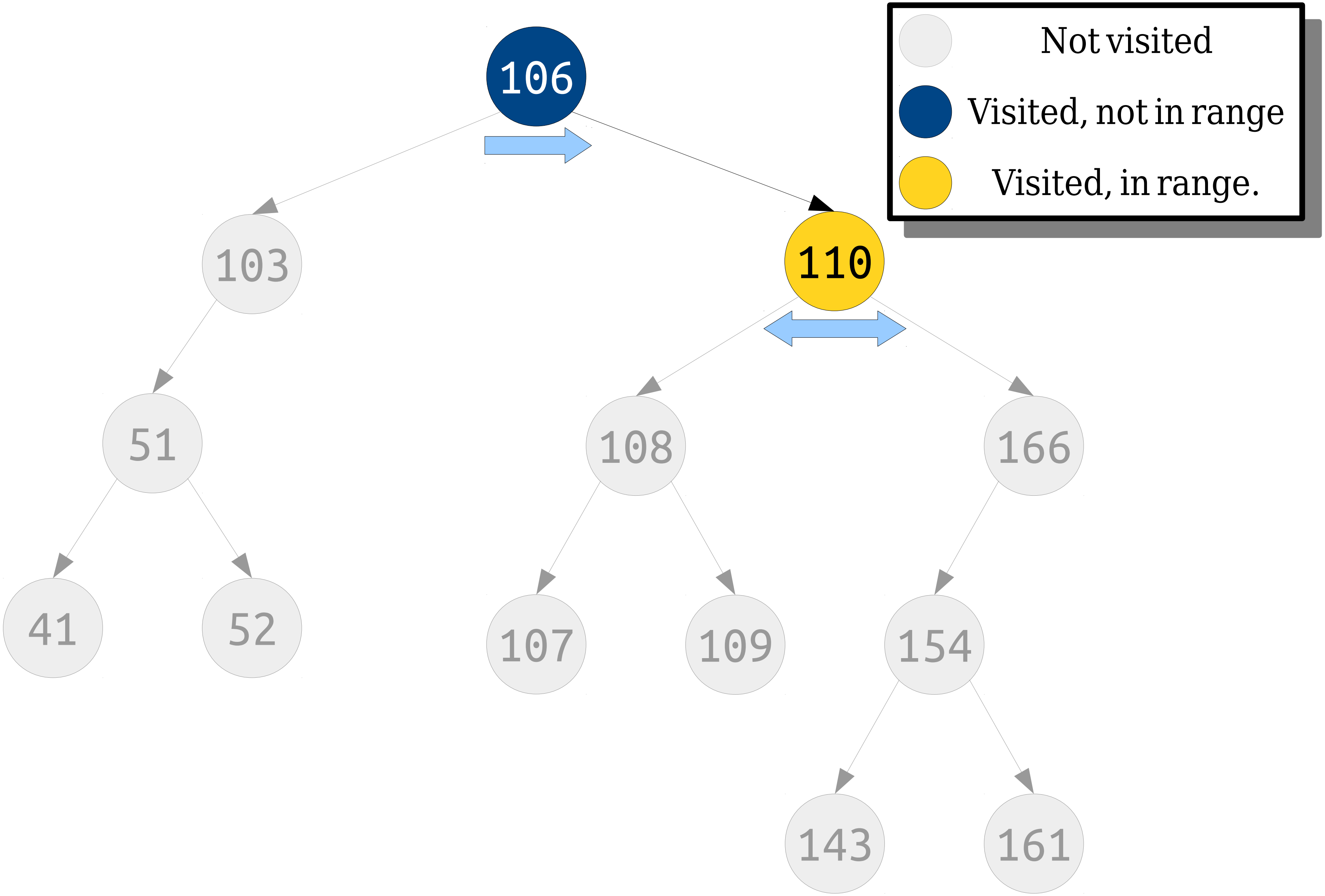
Find all elements in this tree in the range **[109, 163]**.



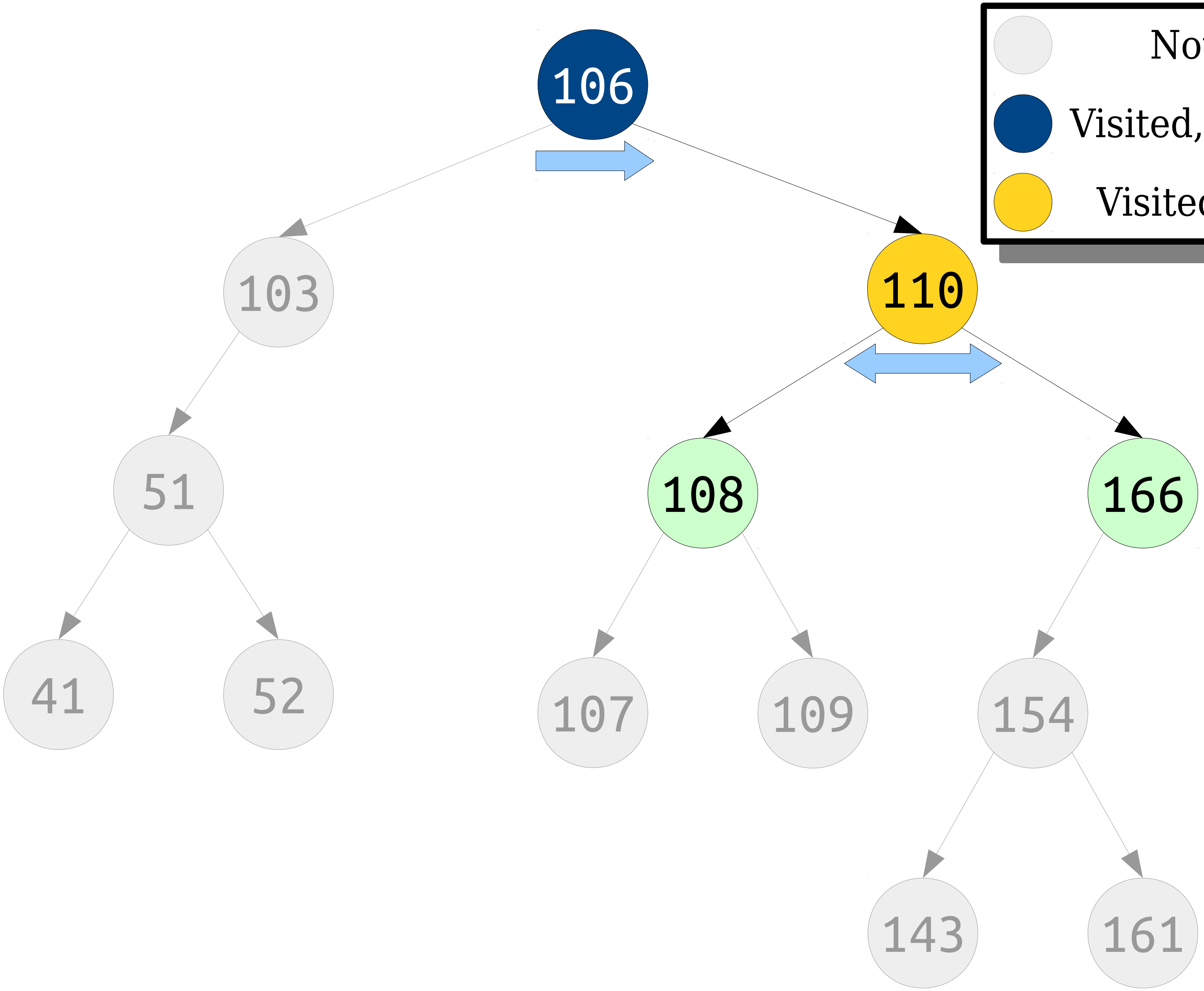
Find all elements in this tree in the range **[109, 163]**.



Find all elements in this tree in the range **[109, 163]**.



Find all elements in this tree in the range **[109, 163]**.

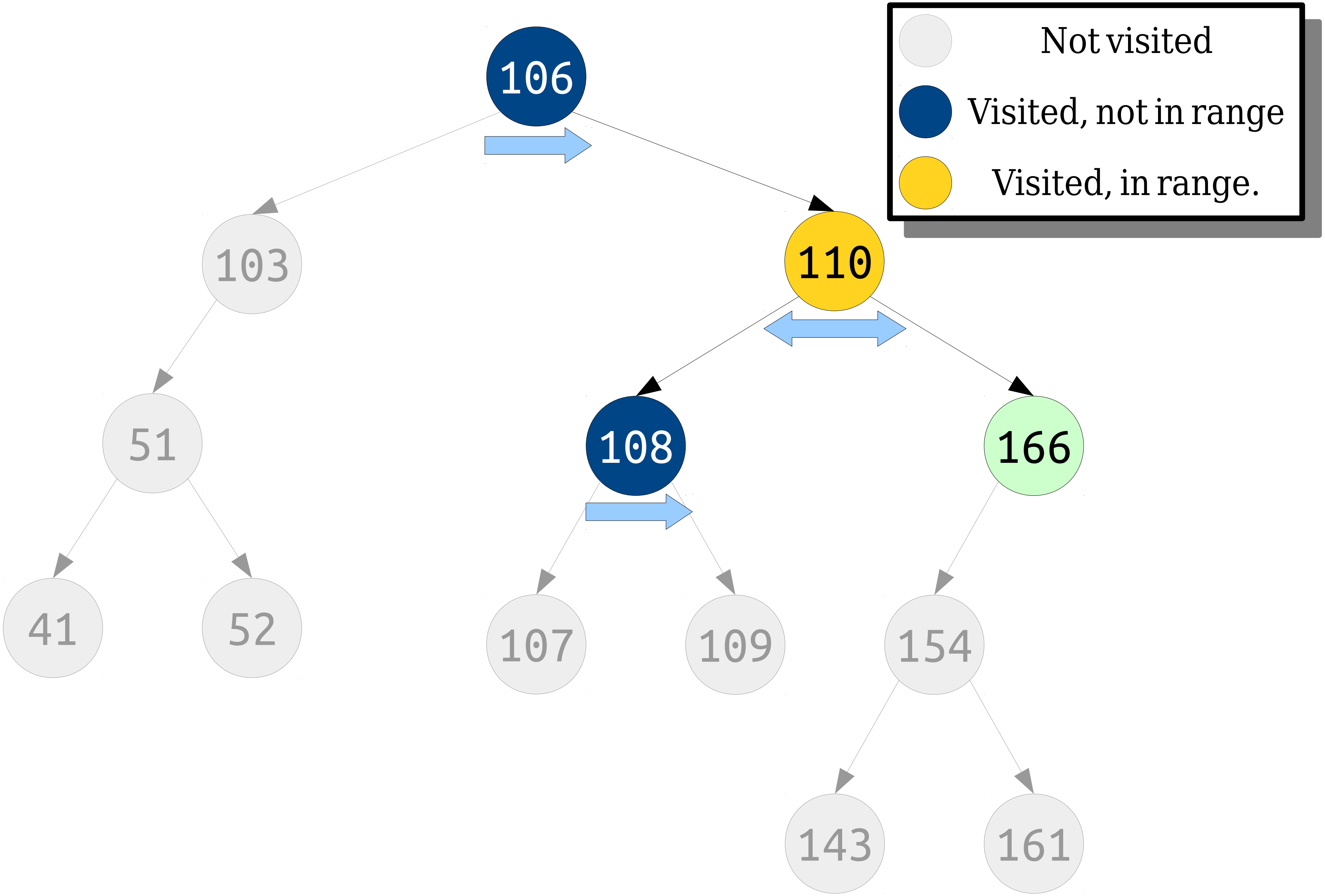


⚪ Not visited

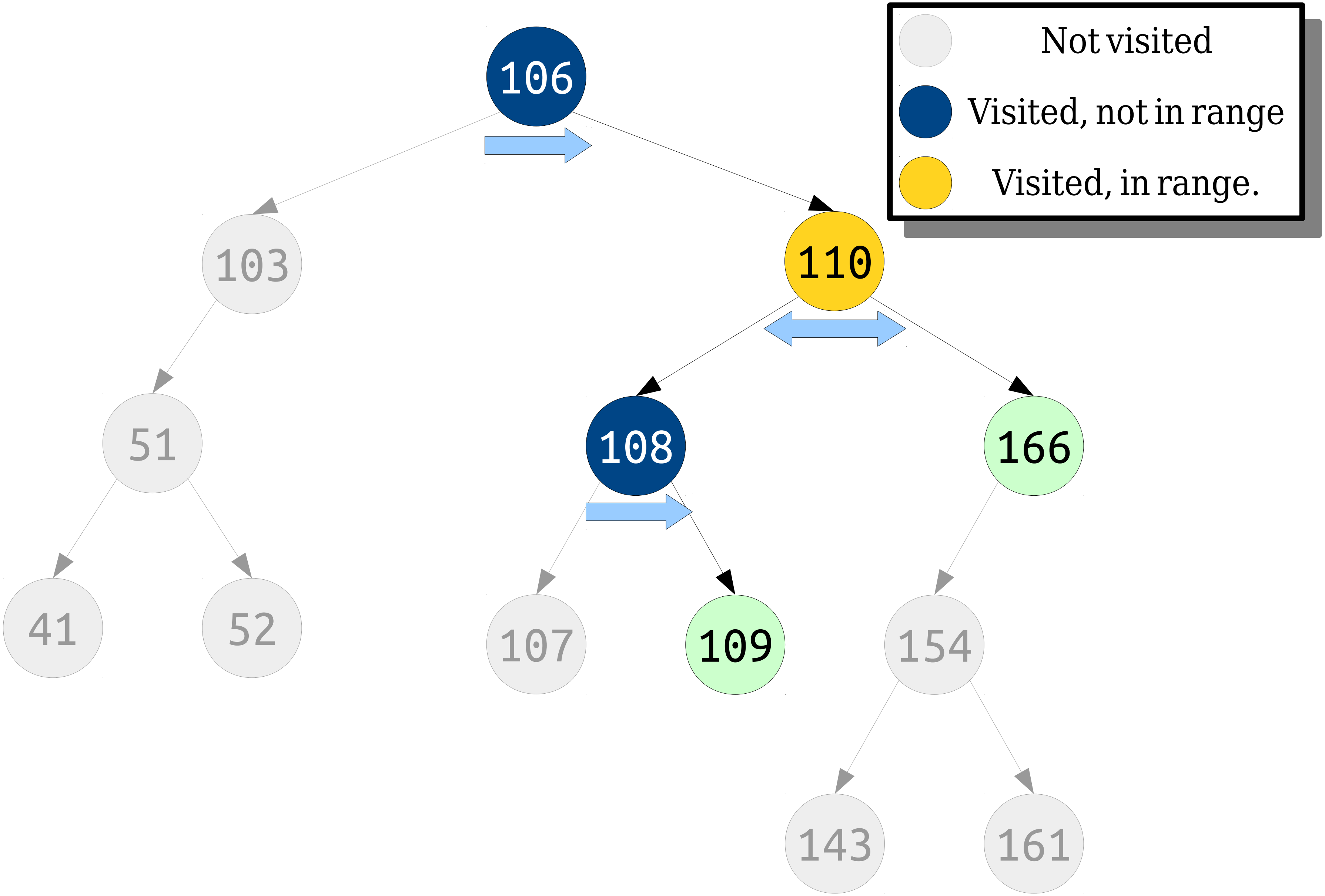
🔵 Visited, not in range

🟡 Visited, in range.

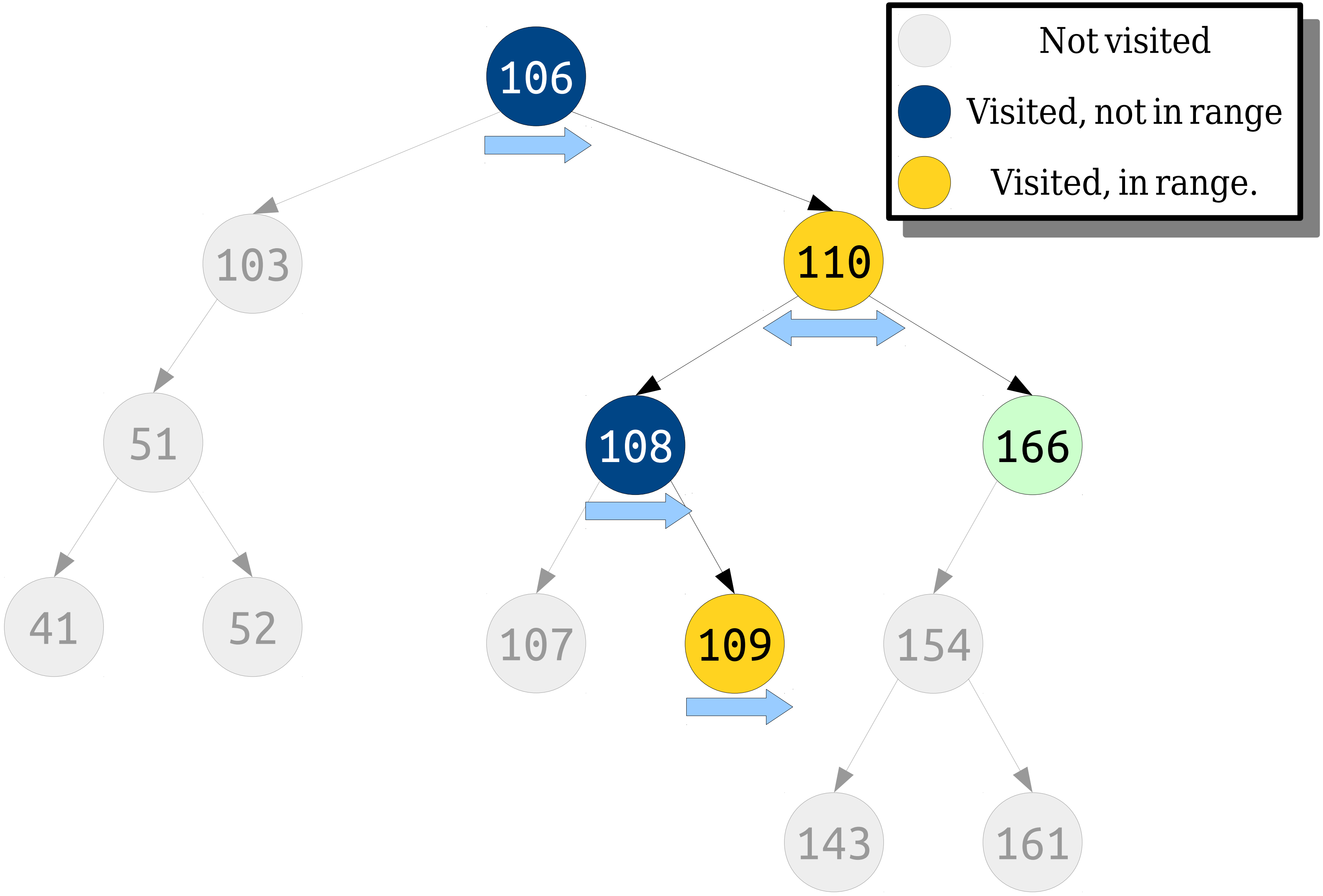
Find all elements in this tree in the range **[109, 163]**.



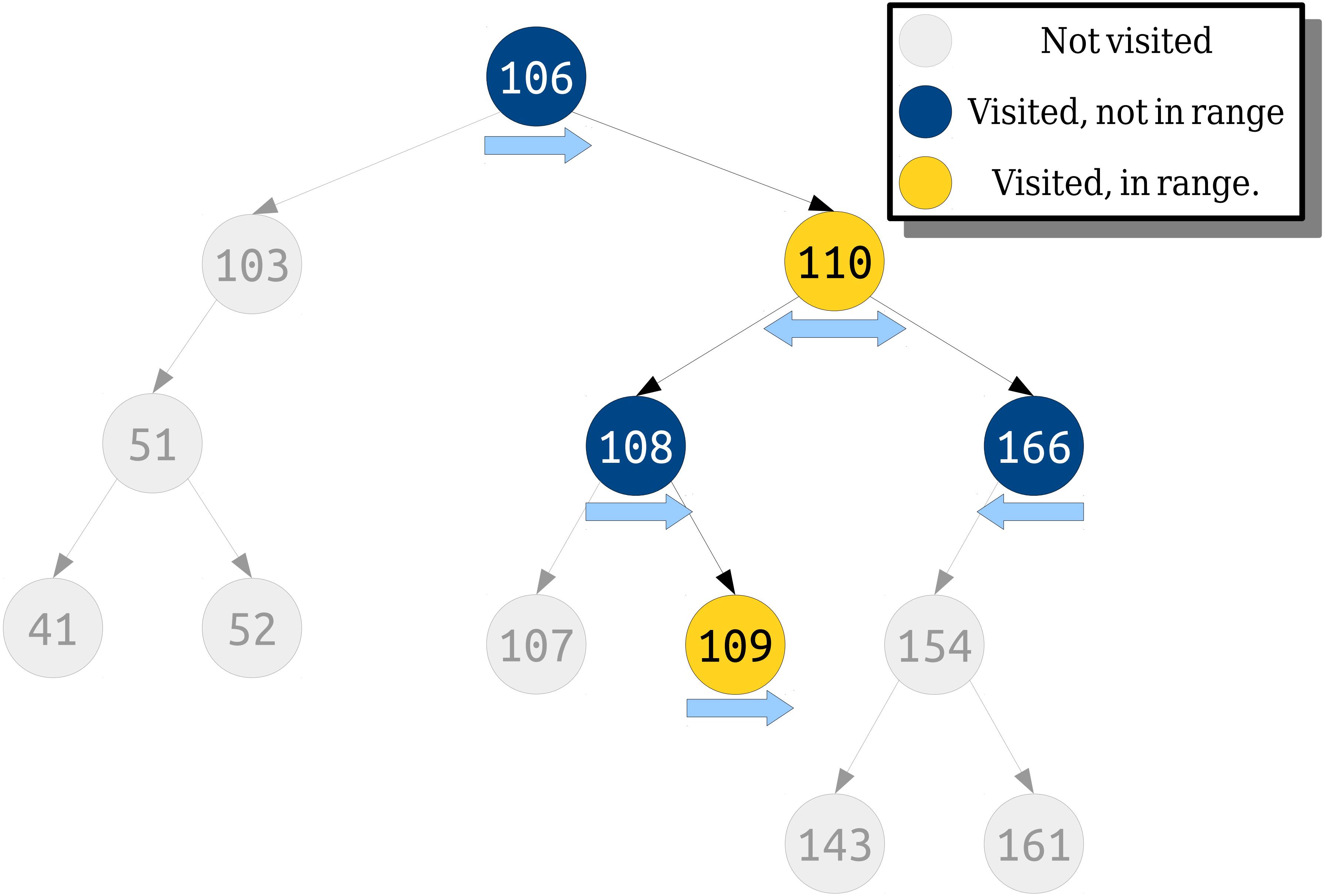
Find all elements in this tree in the range **[109, 163]**.



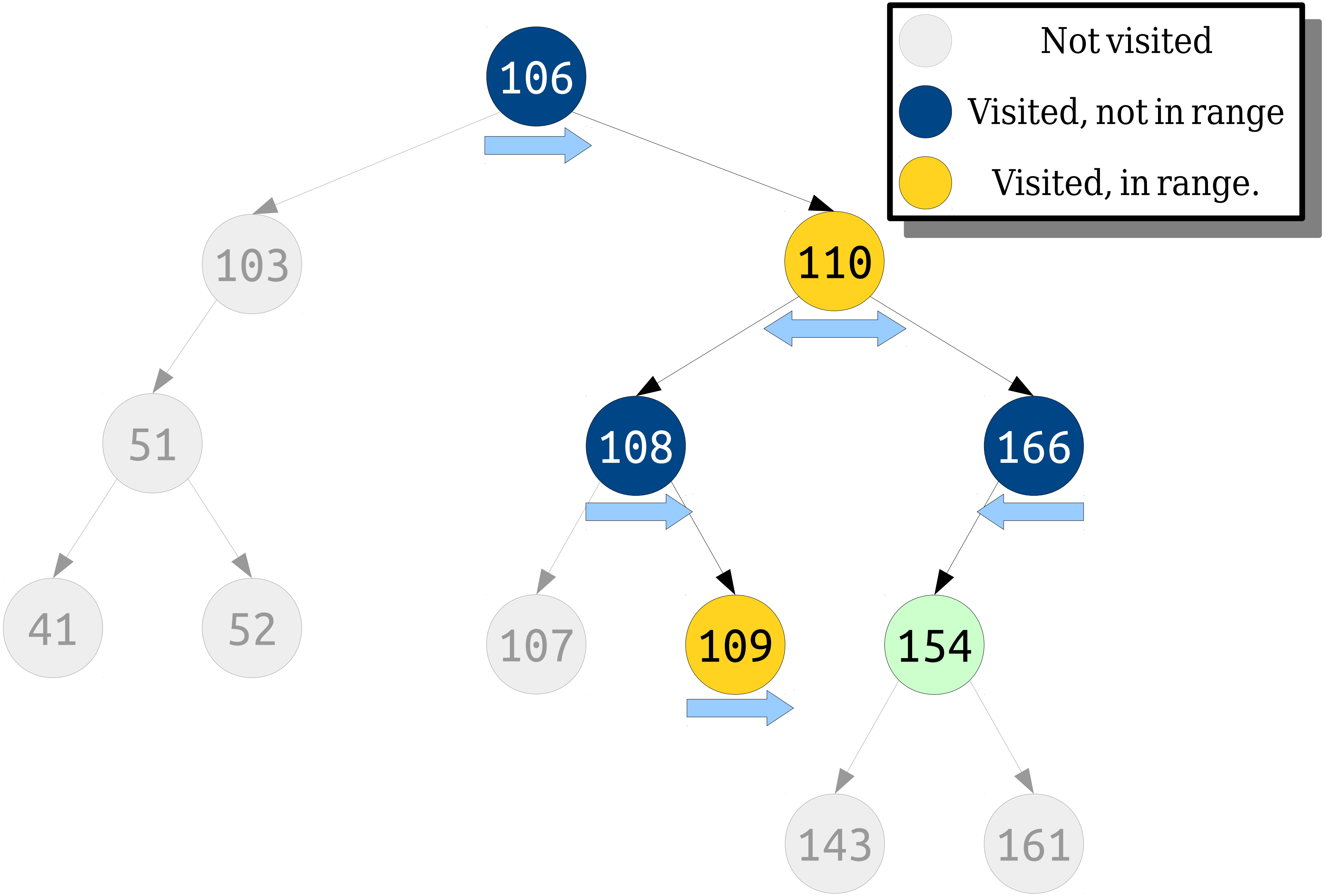
Find all elements in this tree in the range **[109, 163]**.



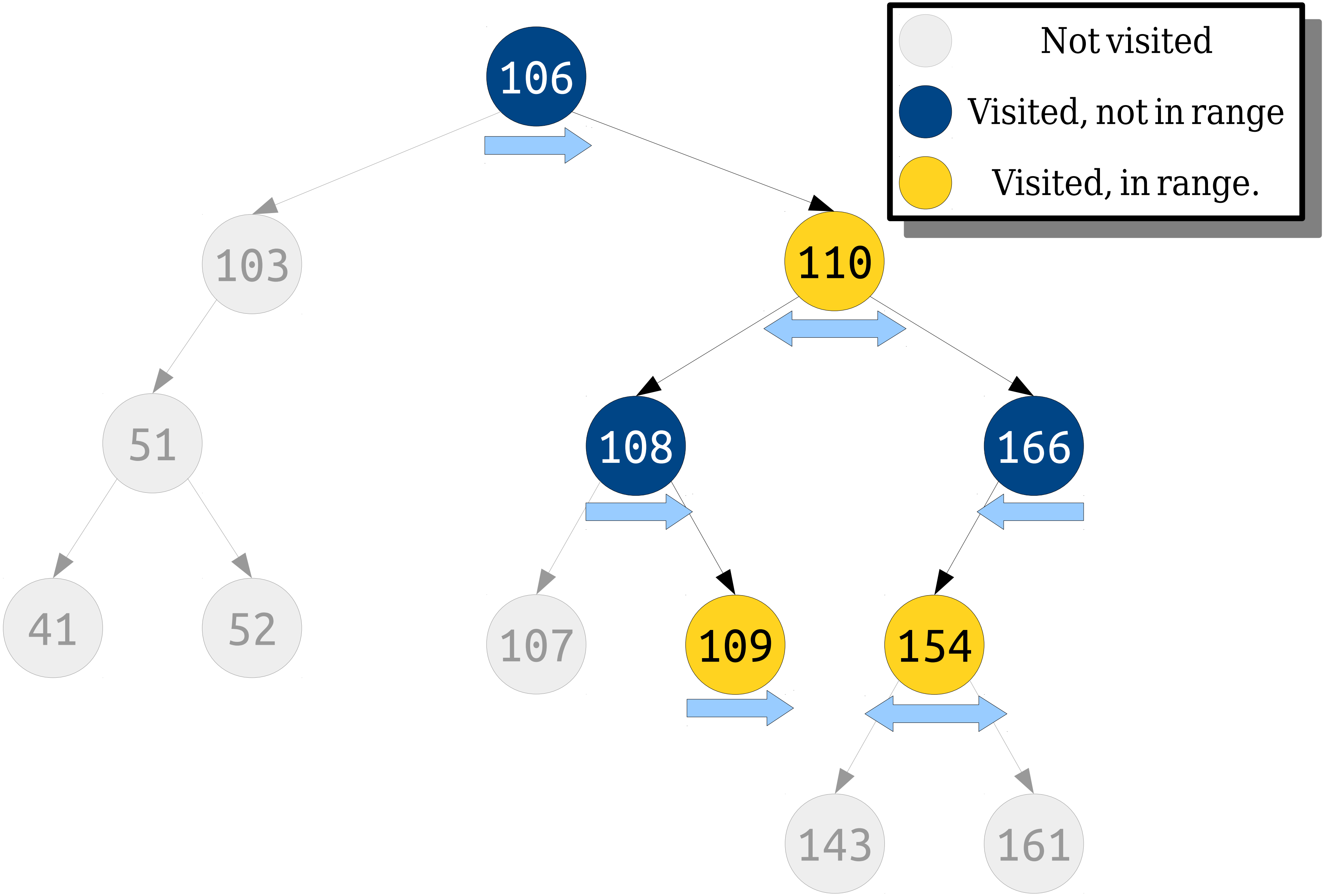
Find all elements in this tree in the range **[109, 163]**.



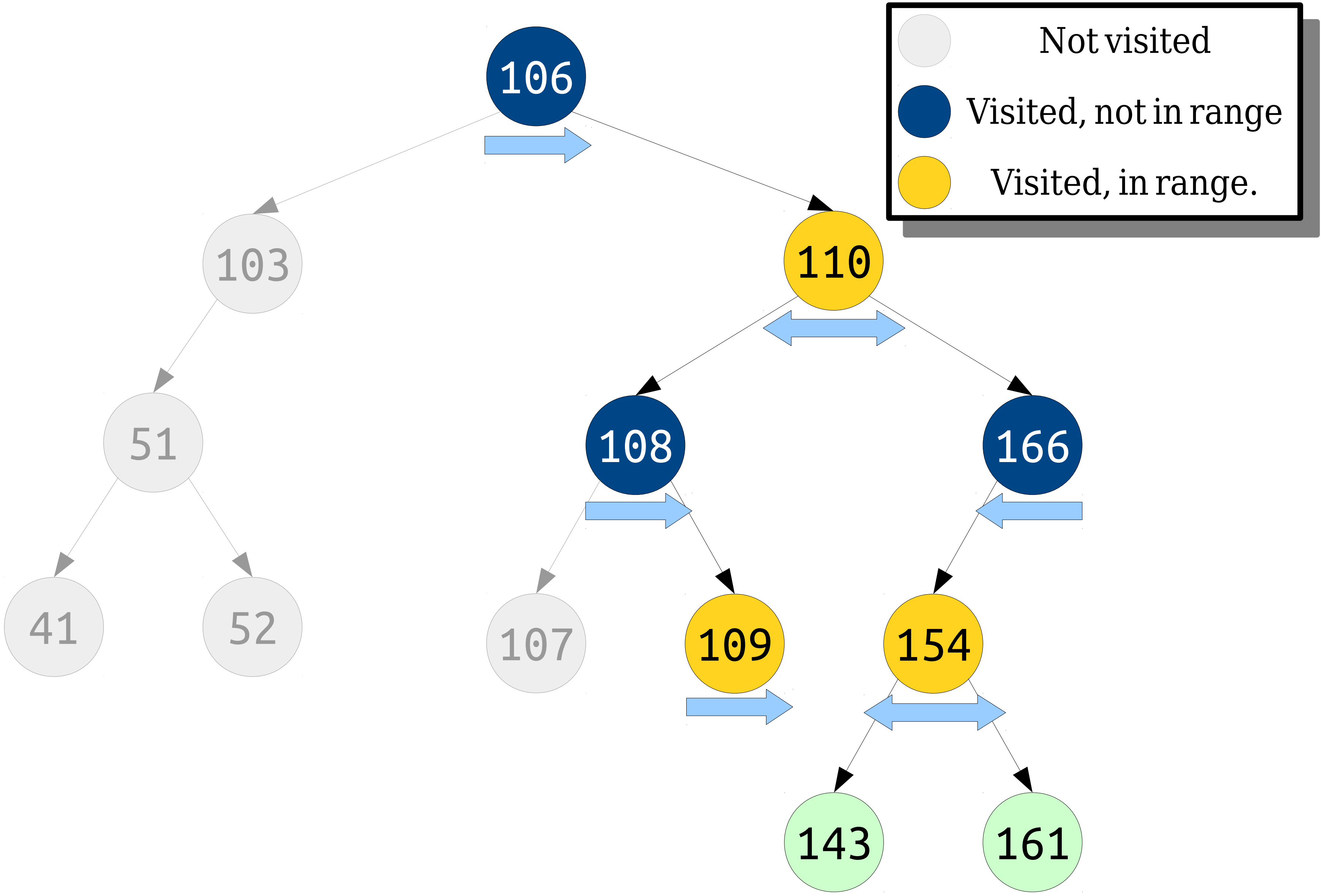
Find all elements in this tree in the range **[109, 163]**.



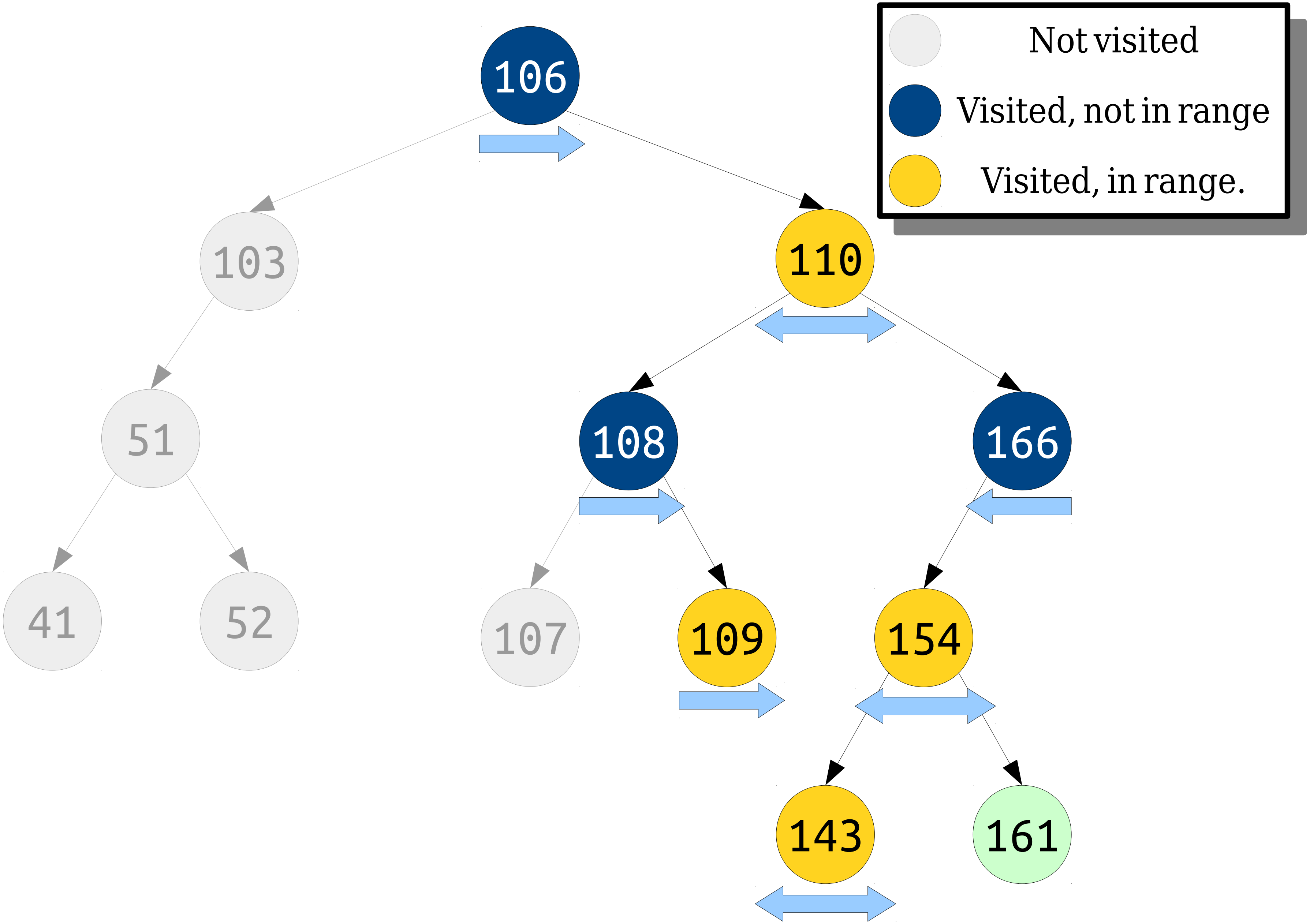
Find all elements in this tree in the range **[109, 163]**.



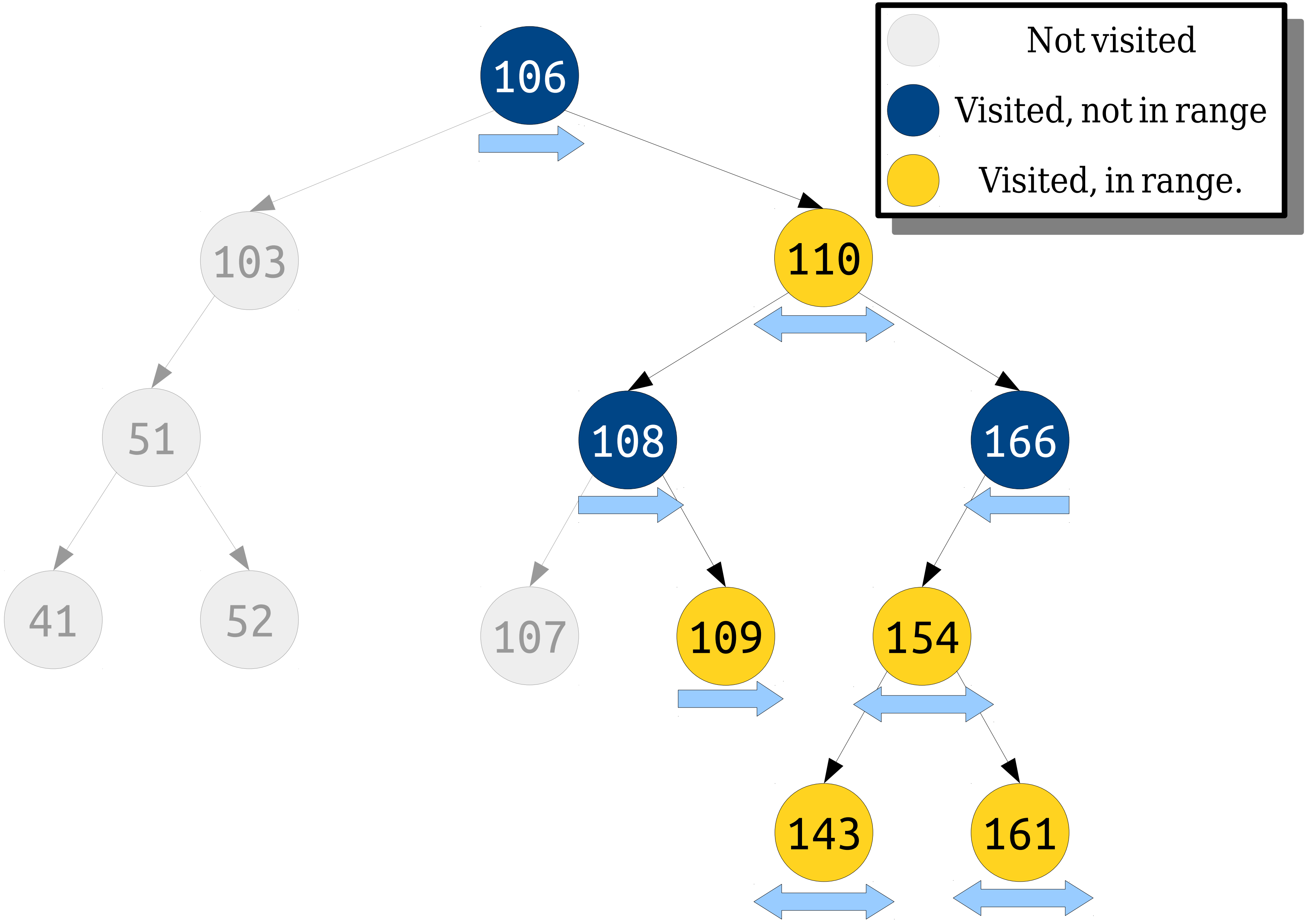
Find all elements in this tree in the range **[109, 163]**.



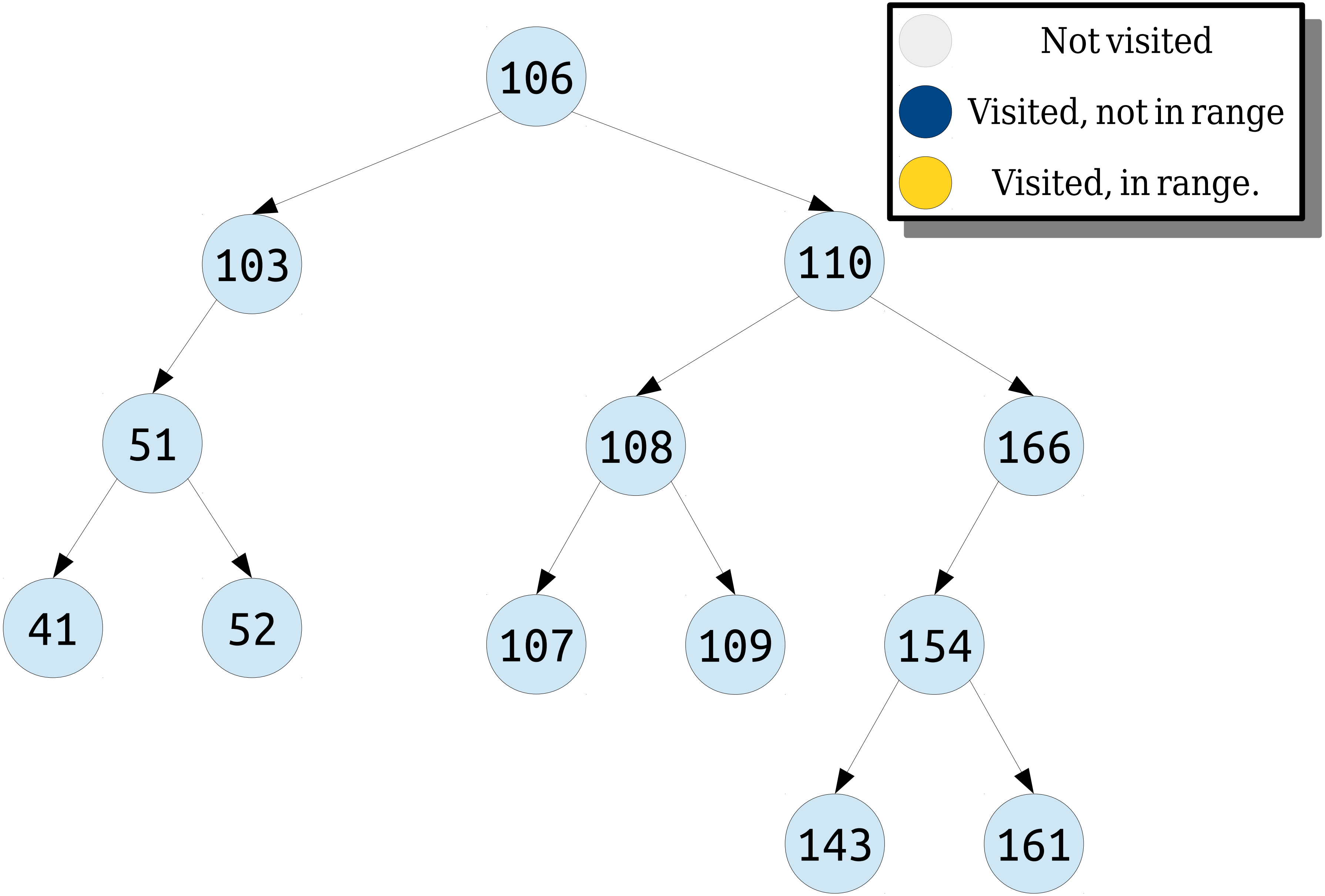
Find all elements in this tree in the range **[109, 163]**.



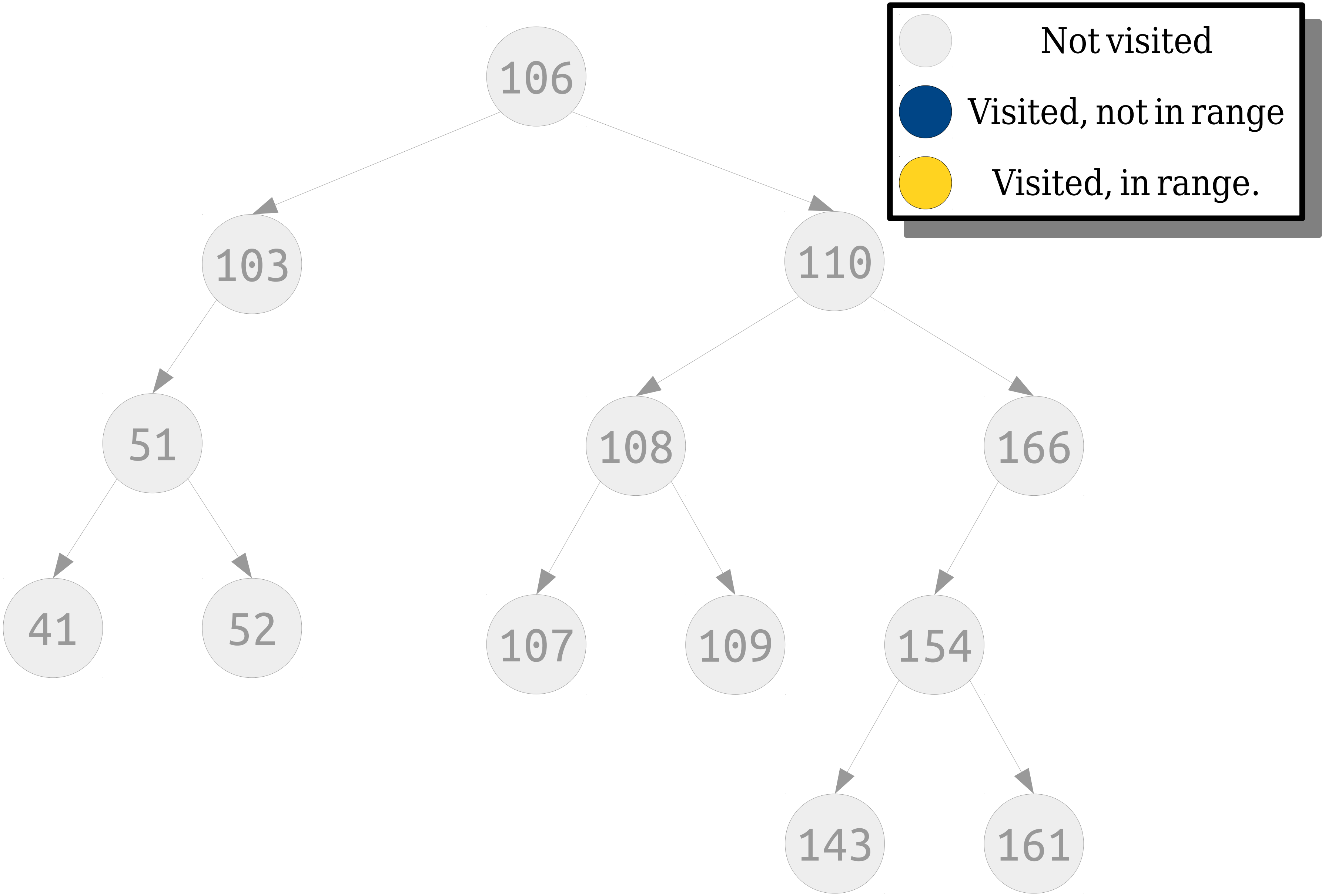
Find all elements in this tree in the range **[109, 163]**.



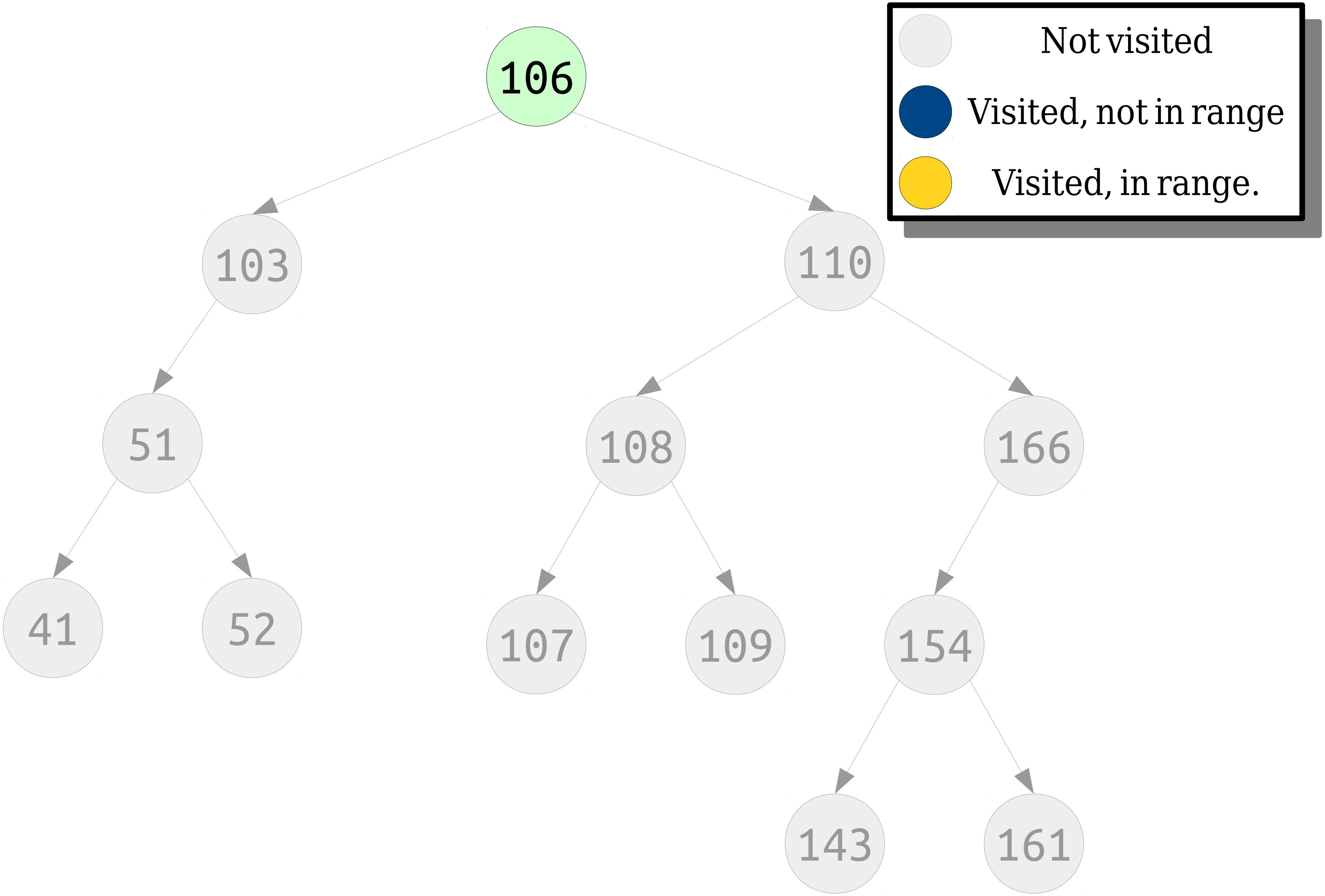
Find all elements in this tree in the range **[109, 163]**.



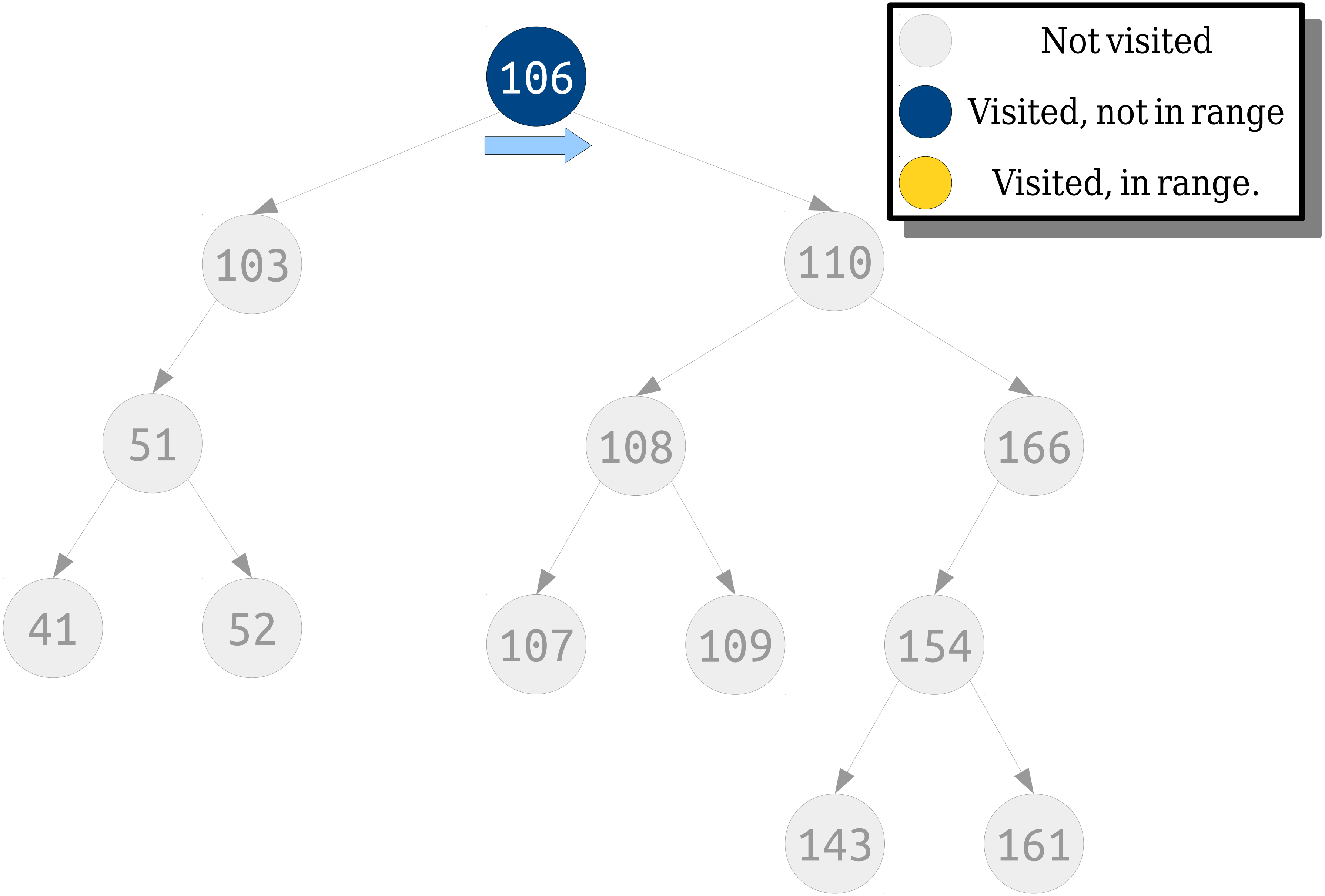
Find all elements in this tree in the range **[124, 155]**.



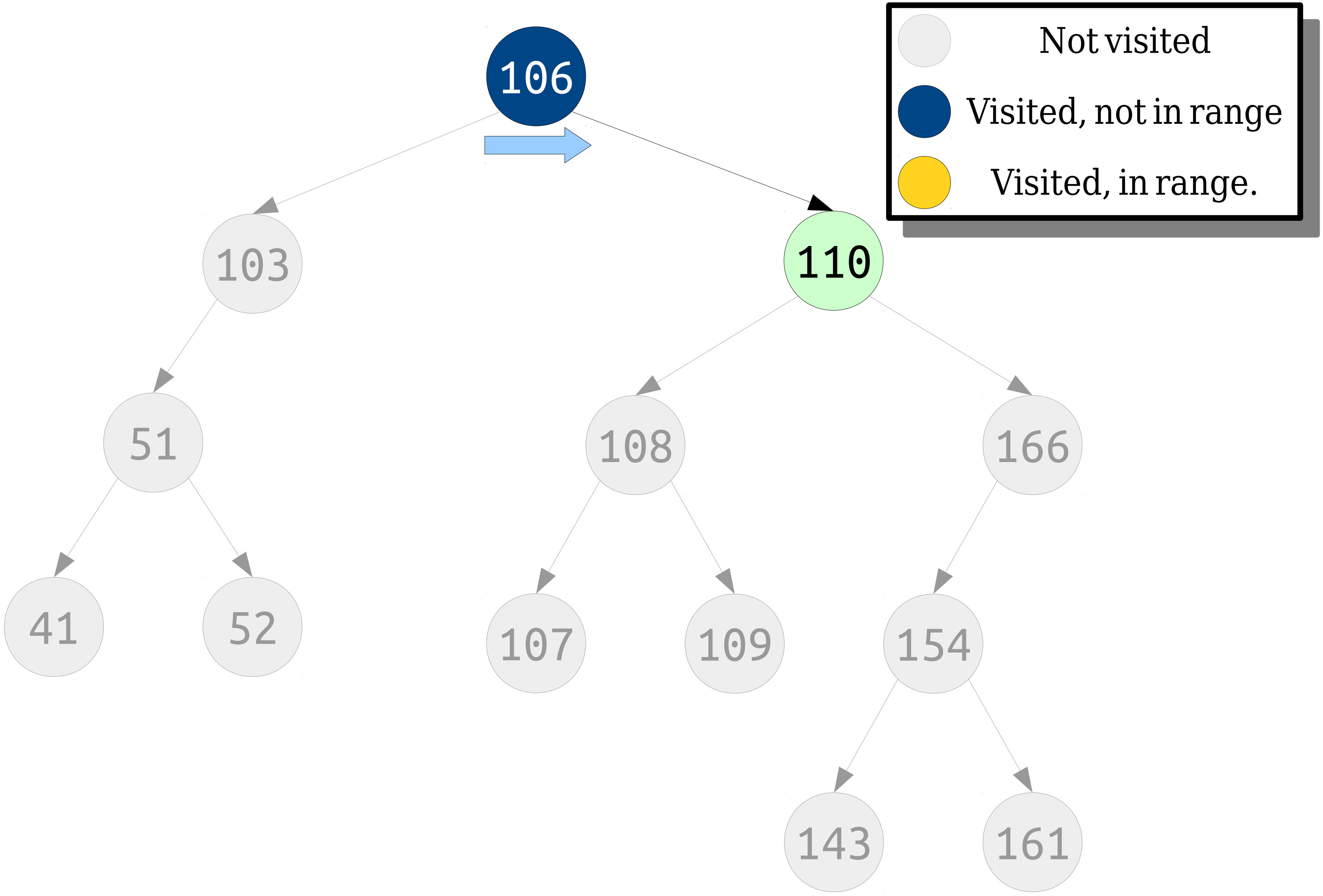
Find all elements in this tree in the range **[124, 155]**.



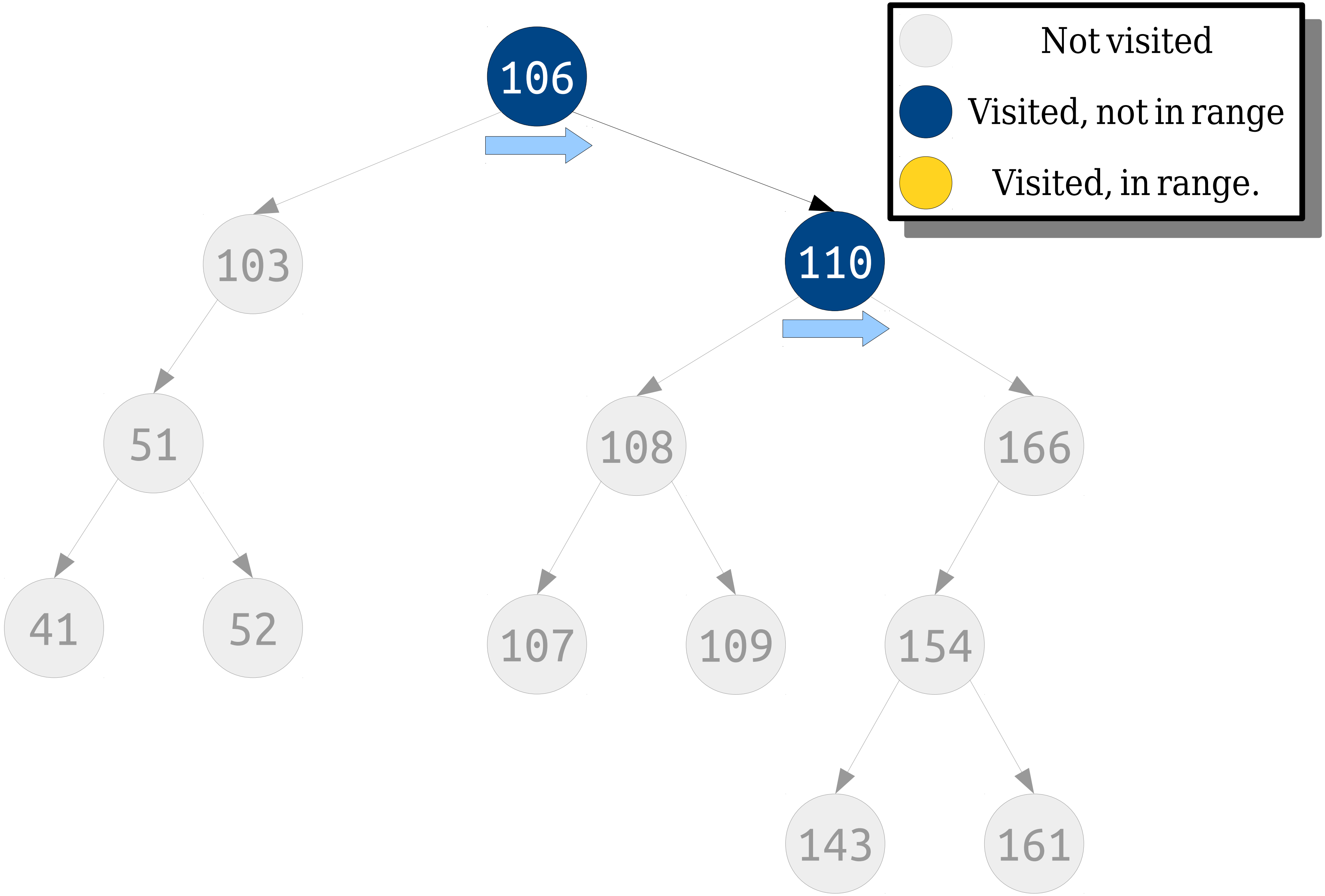
Find all elements in this tree in the range **[124, 155]**.



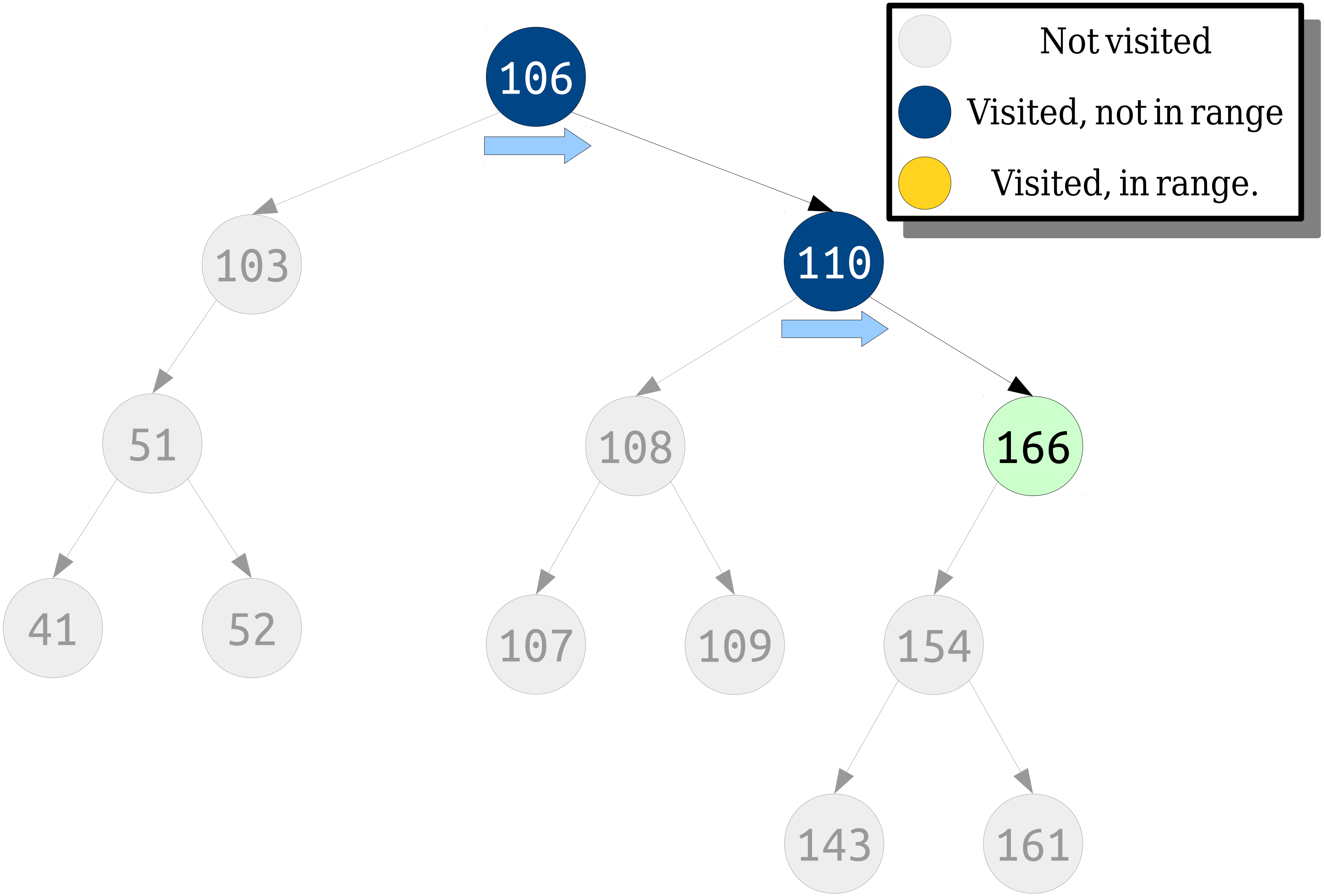
Find all elements in this tree in the range **[124, 155]**.



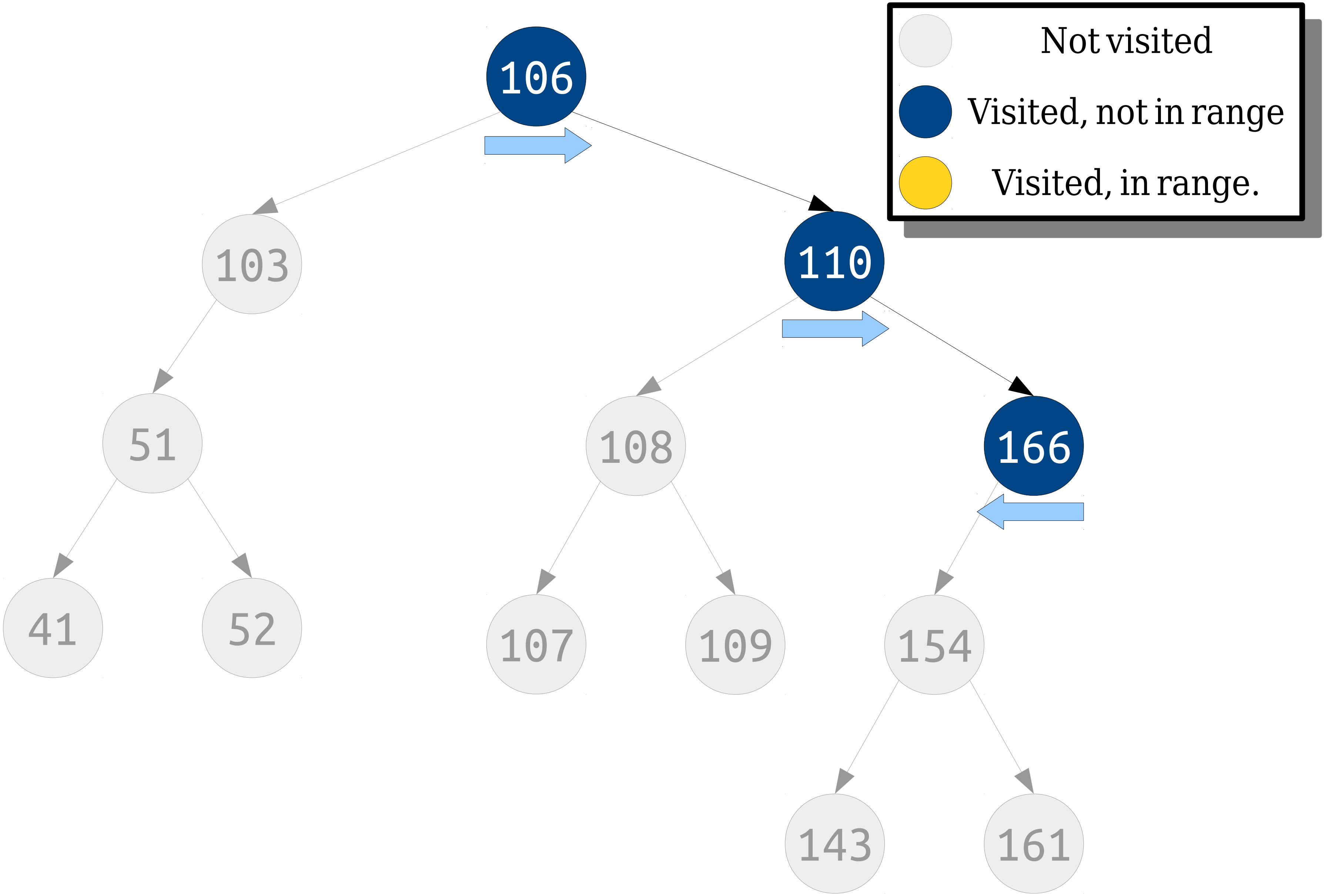
Find all elements in this tree in the range **[124, 155]**.



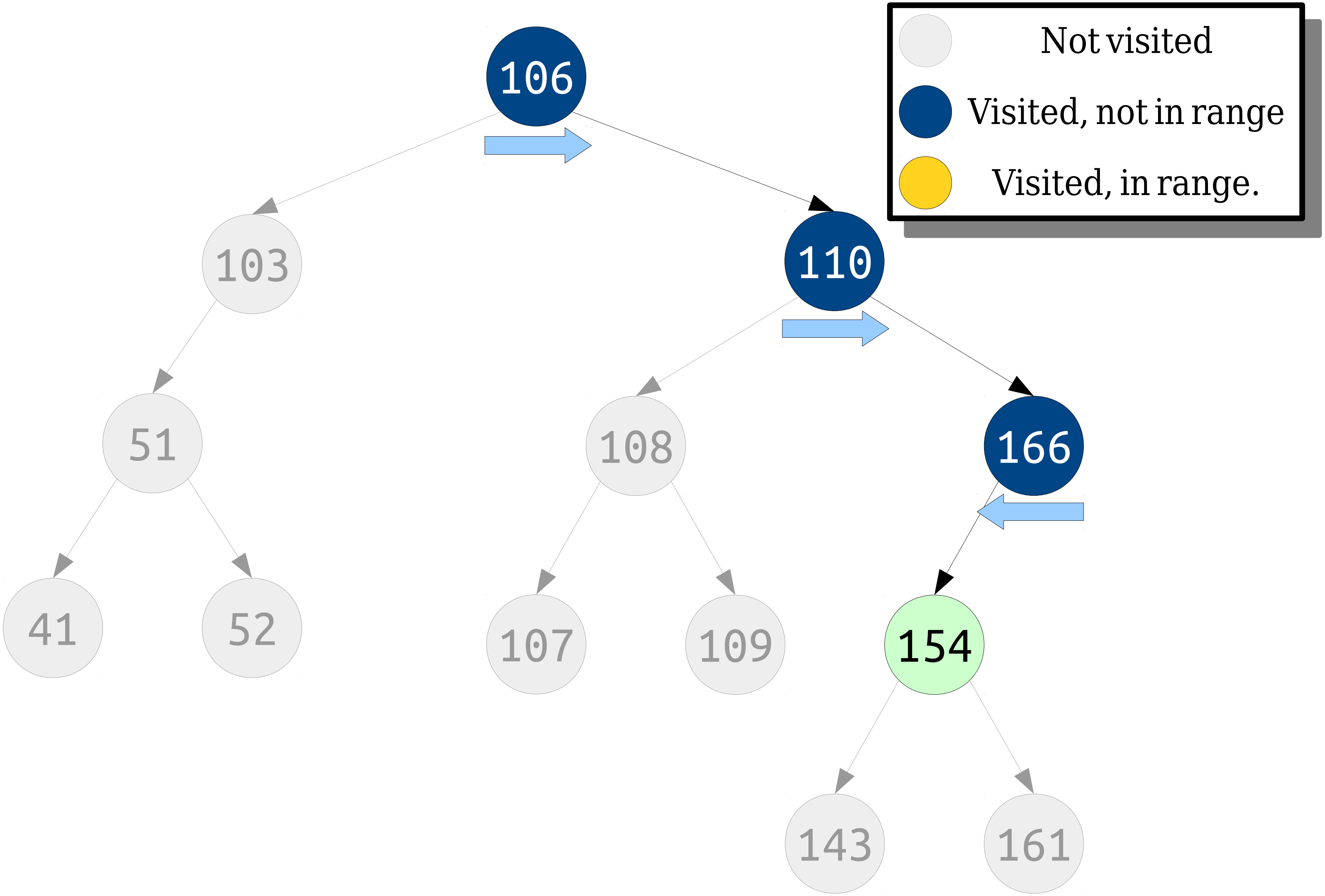
Find all elements in this tree in the range **[124, 155]**.



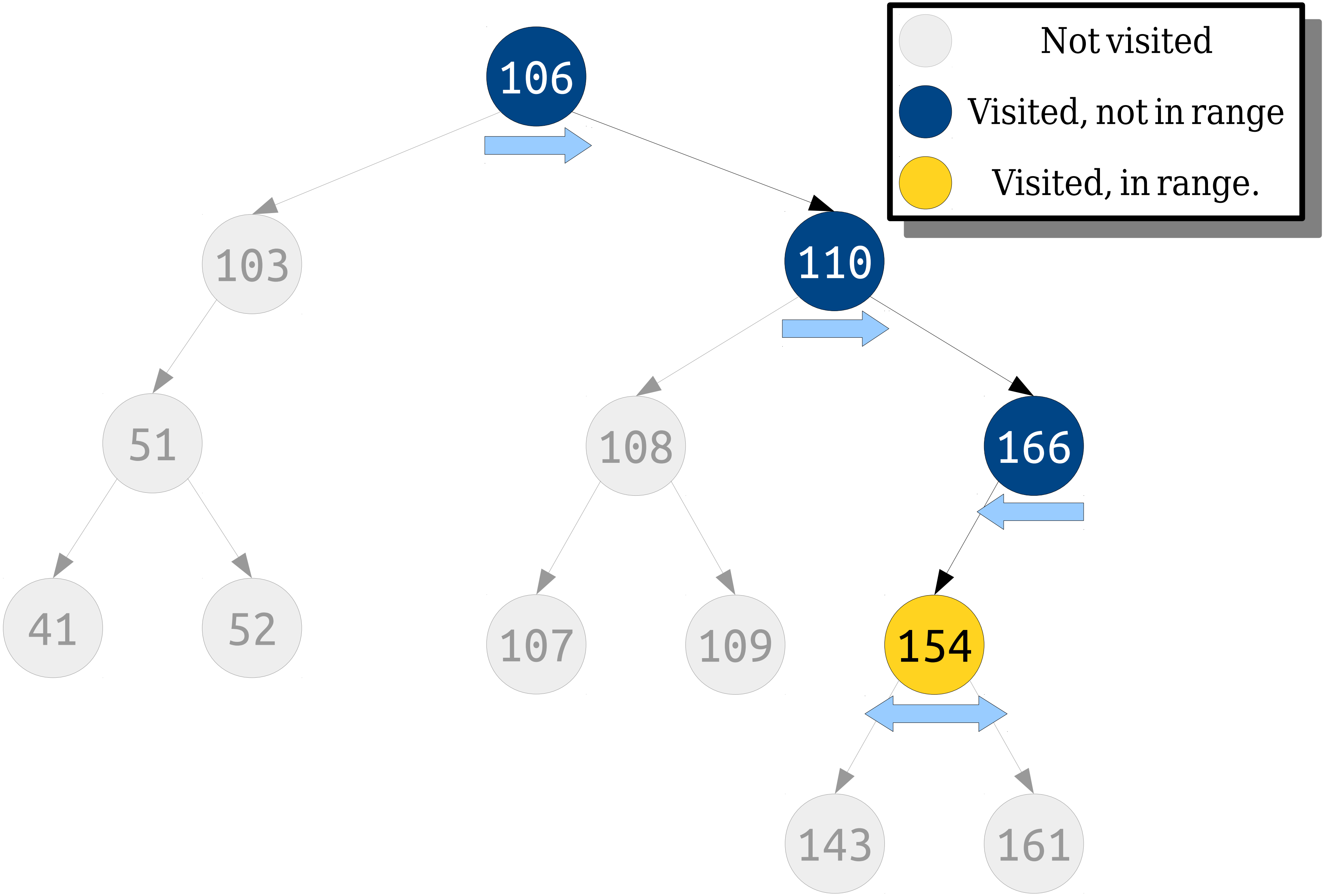
Find all elements in this tree in the range **[124, 155]**.



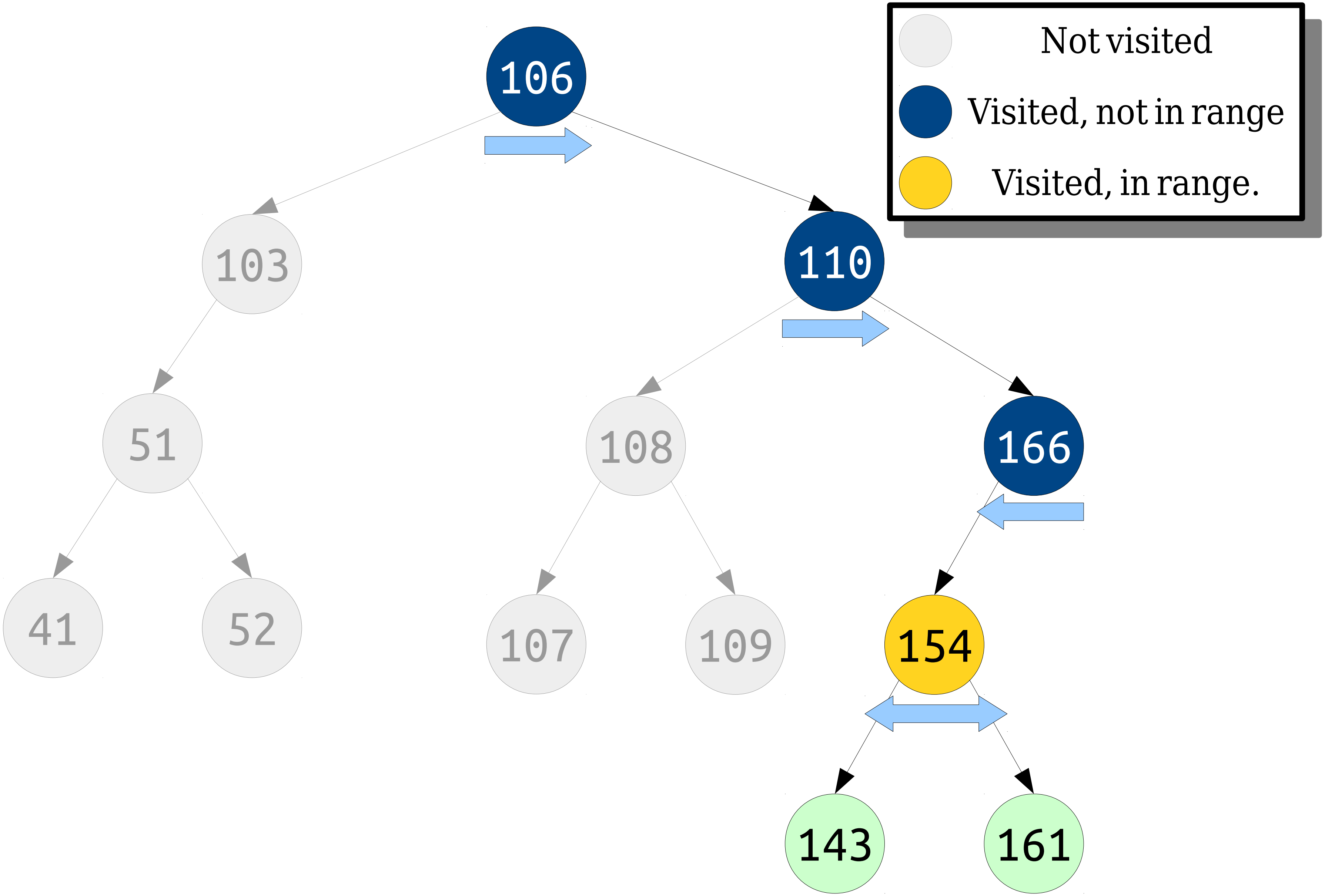
Find all elements in this tree in the range **[124, 155]**.



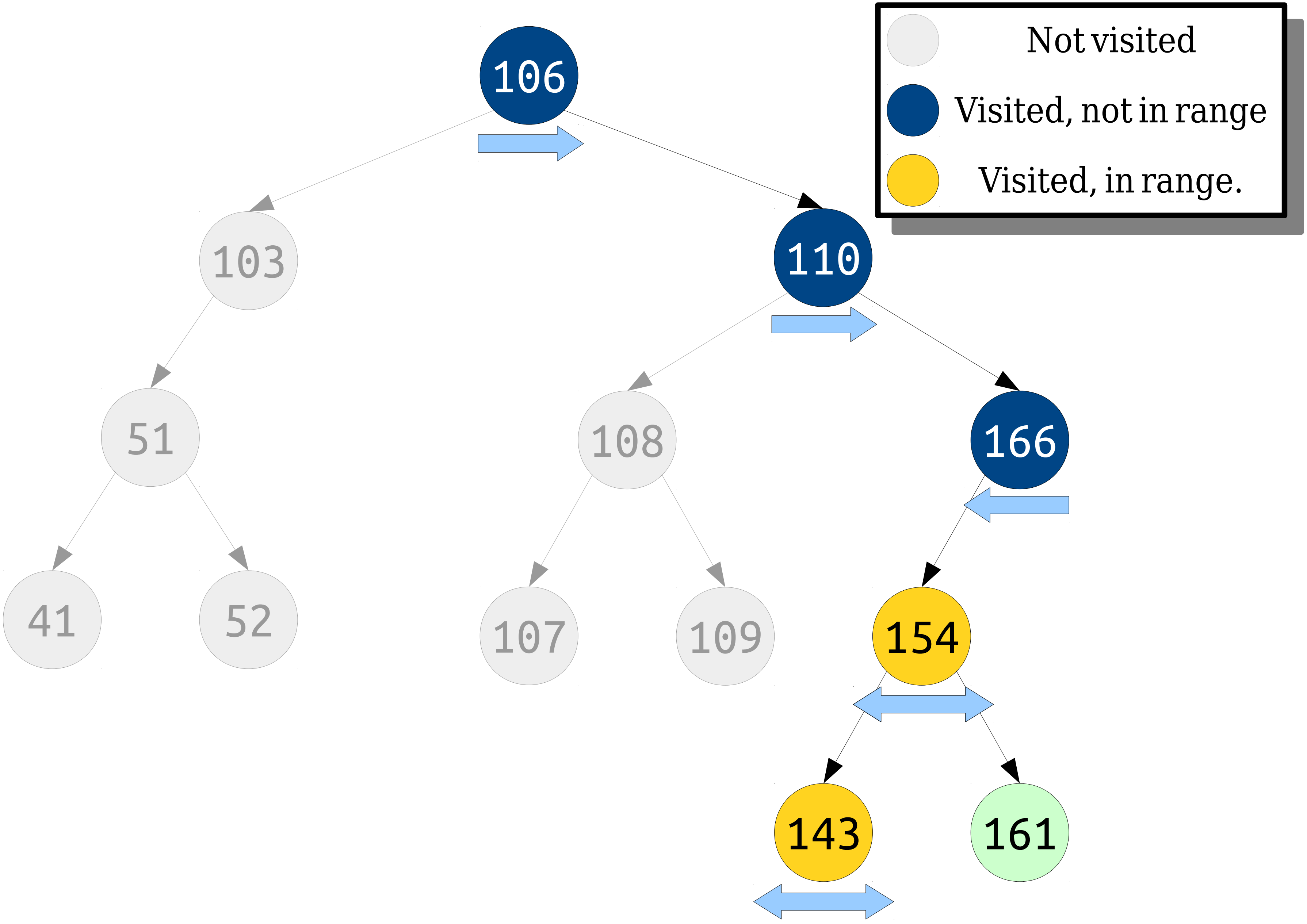
Find all elements in this tree in the range **[124, 155]**.



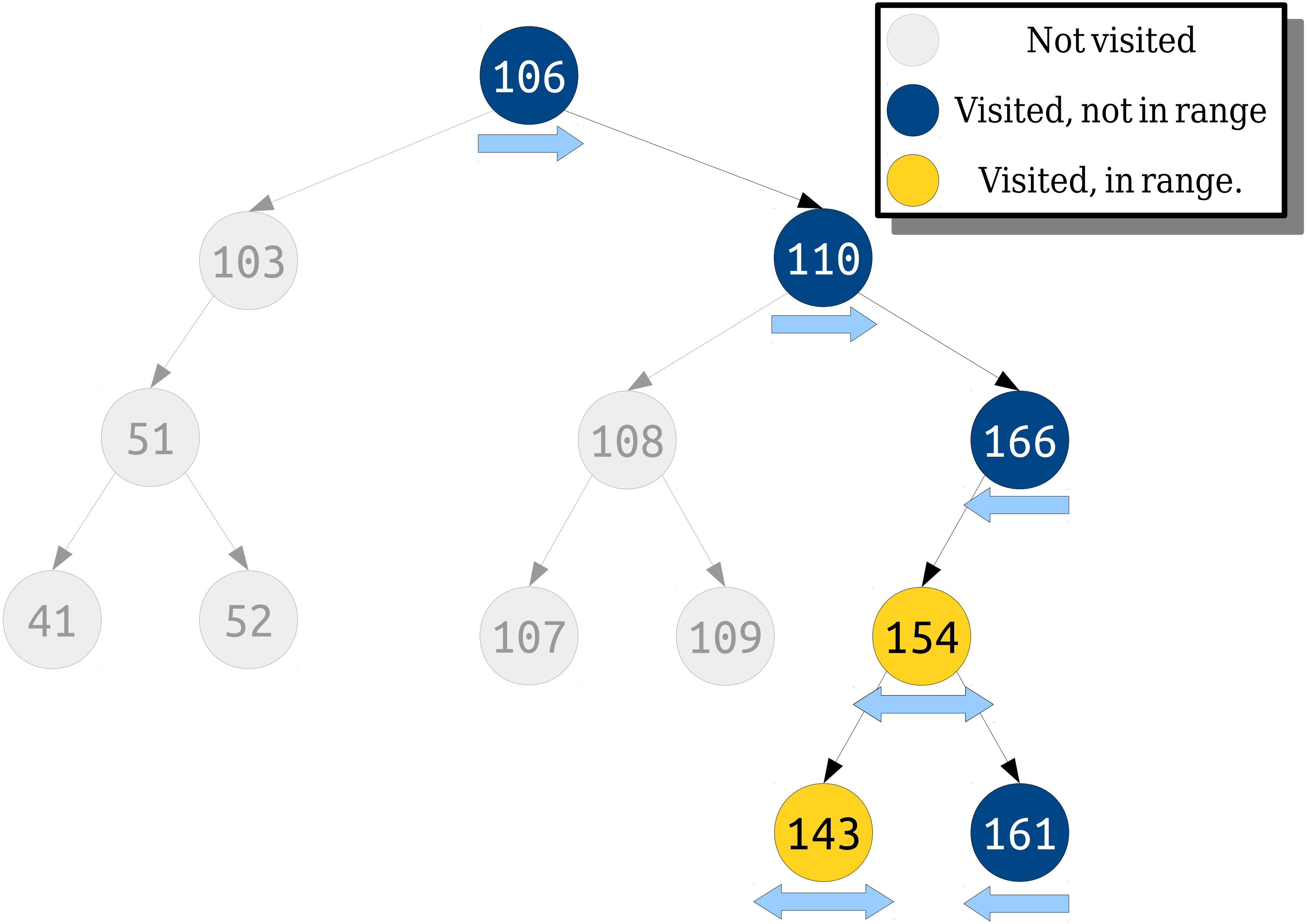
Find all elements in this tree in the range **[124, 155]**.



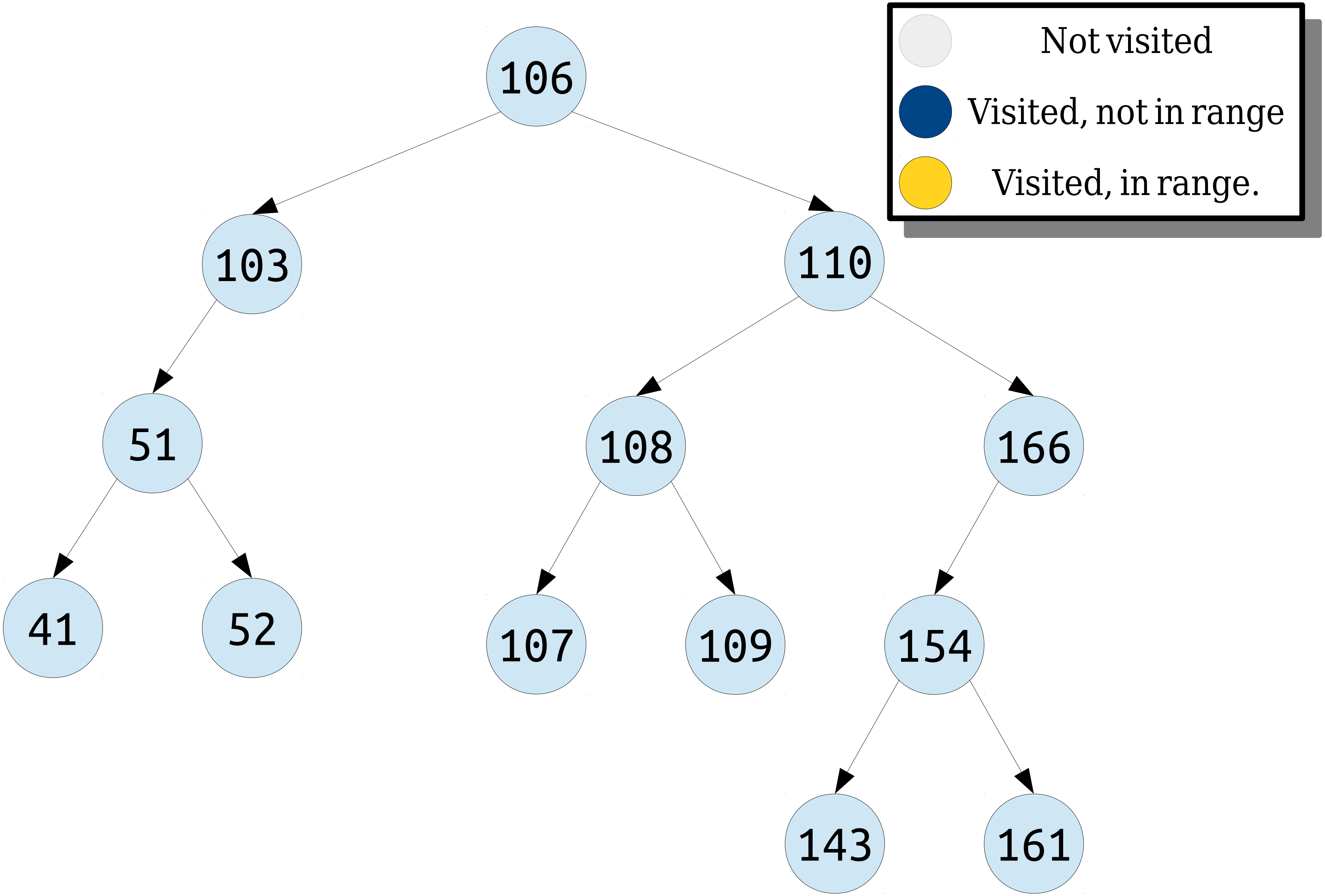
Find all elements in this tree in the range **[124, 155]**.



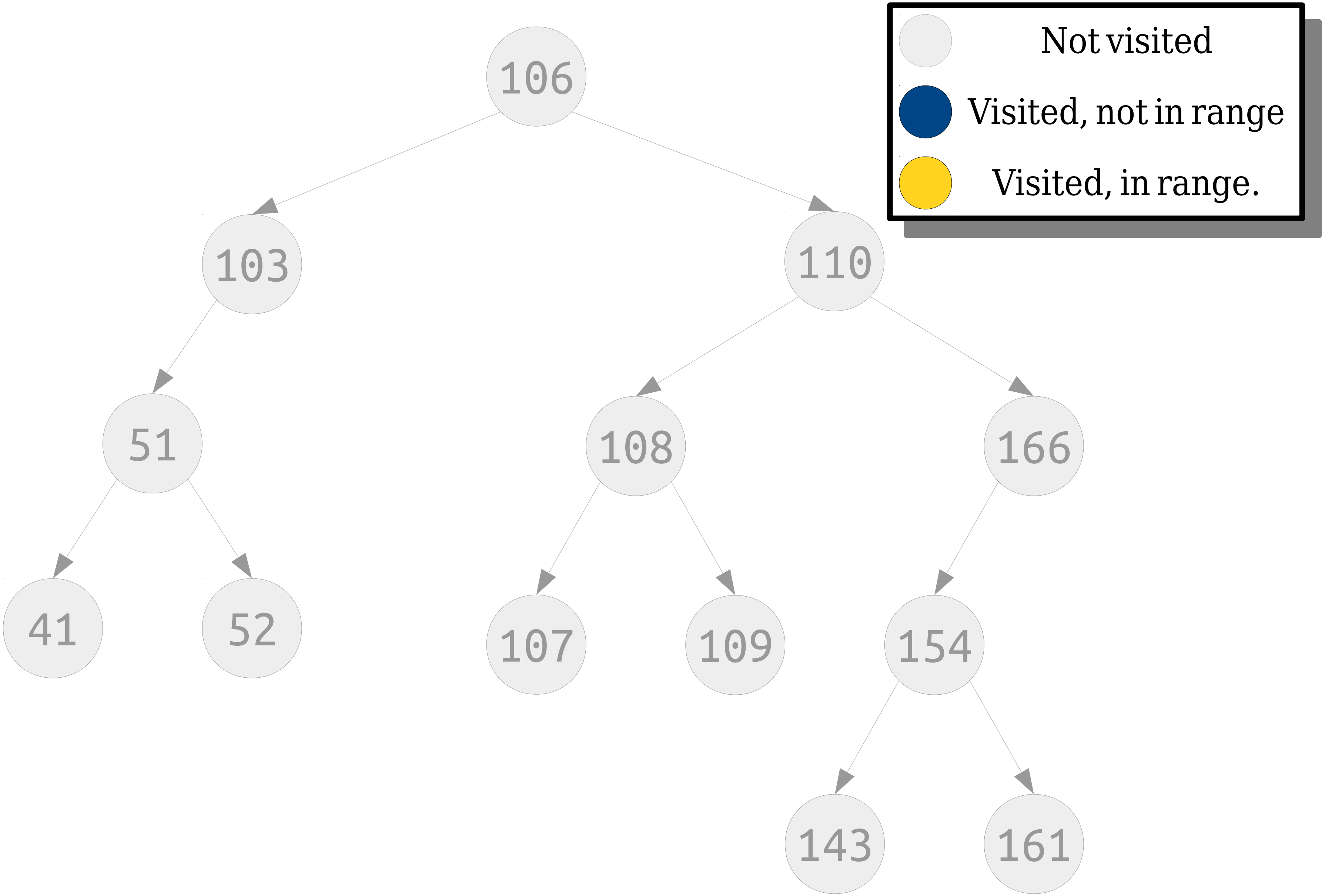
Find all elements in this tree in the range **[124, 155]**.



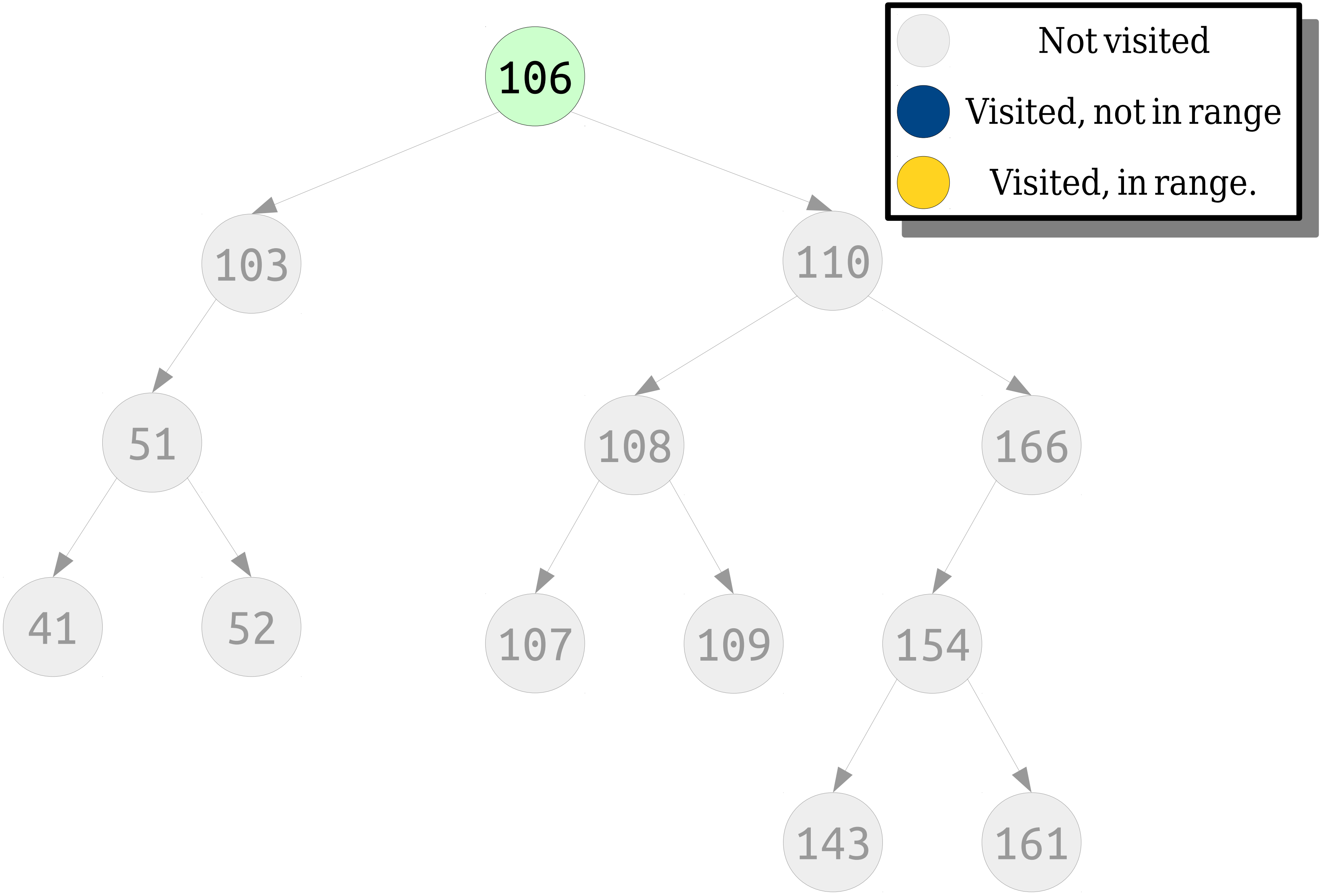
Find all elements in this tree in the range **[124, 155]**.



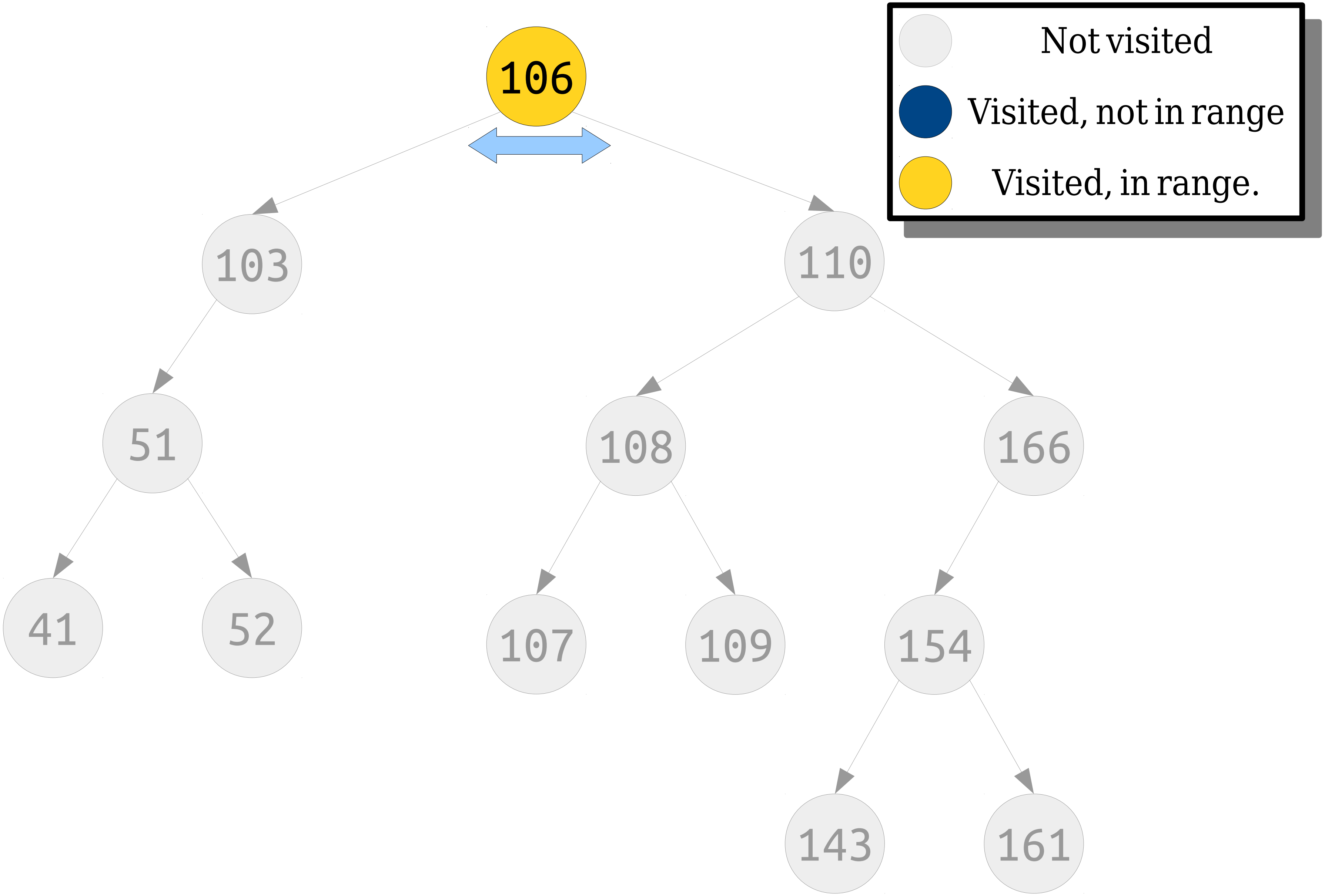
Find all elements in this tree in the range **[42, 165]**.



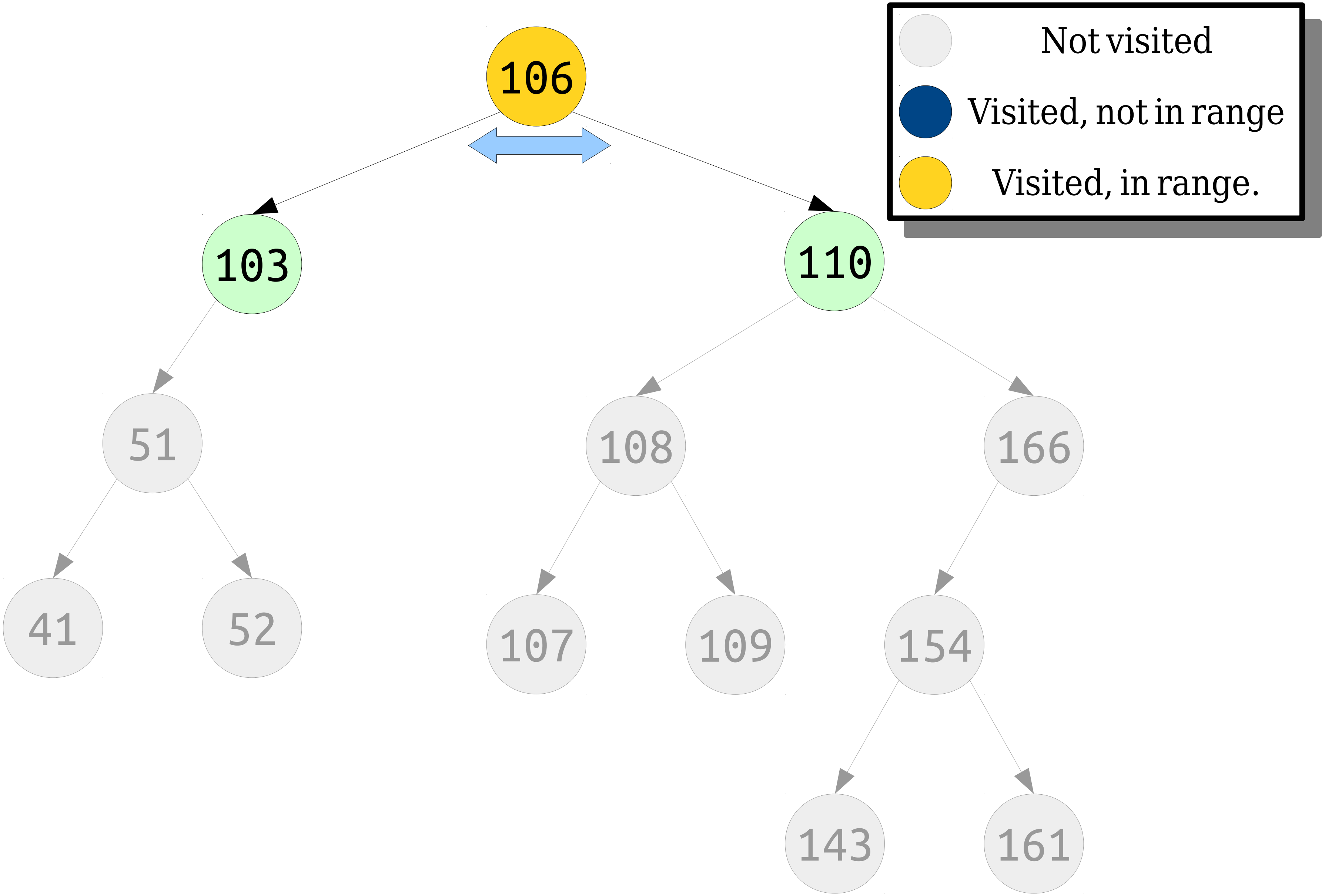
Find all elements in this tree in the range **[42, 165]**.



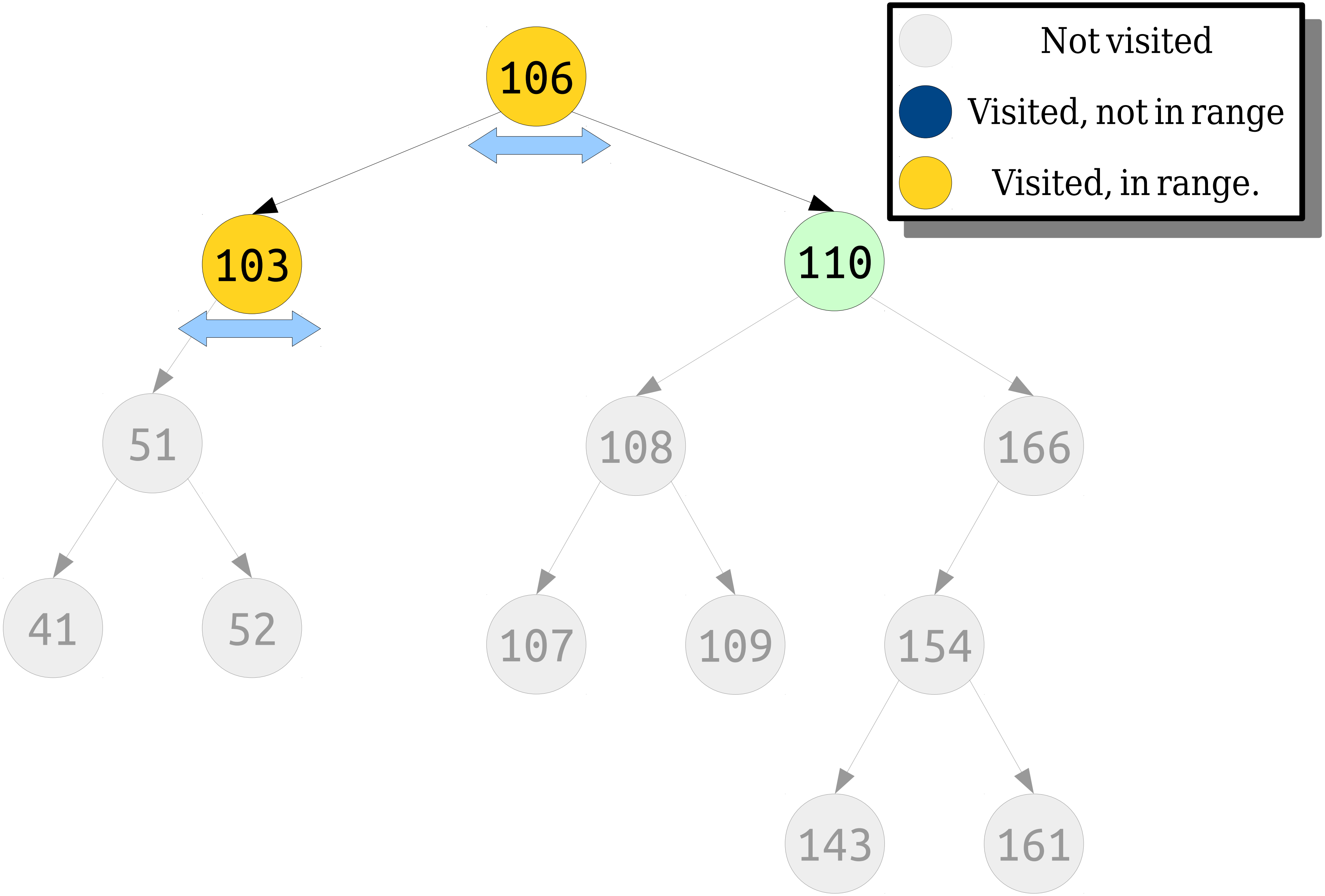
Find all elements in this tree in the range **[42, 165]**.



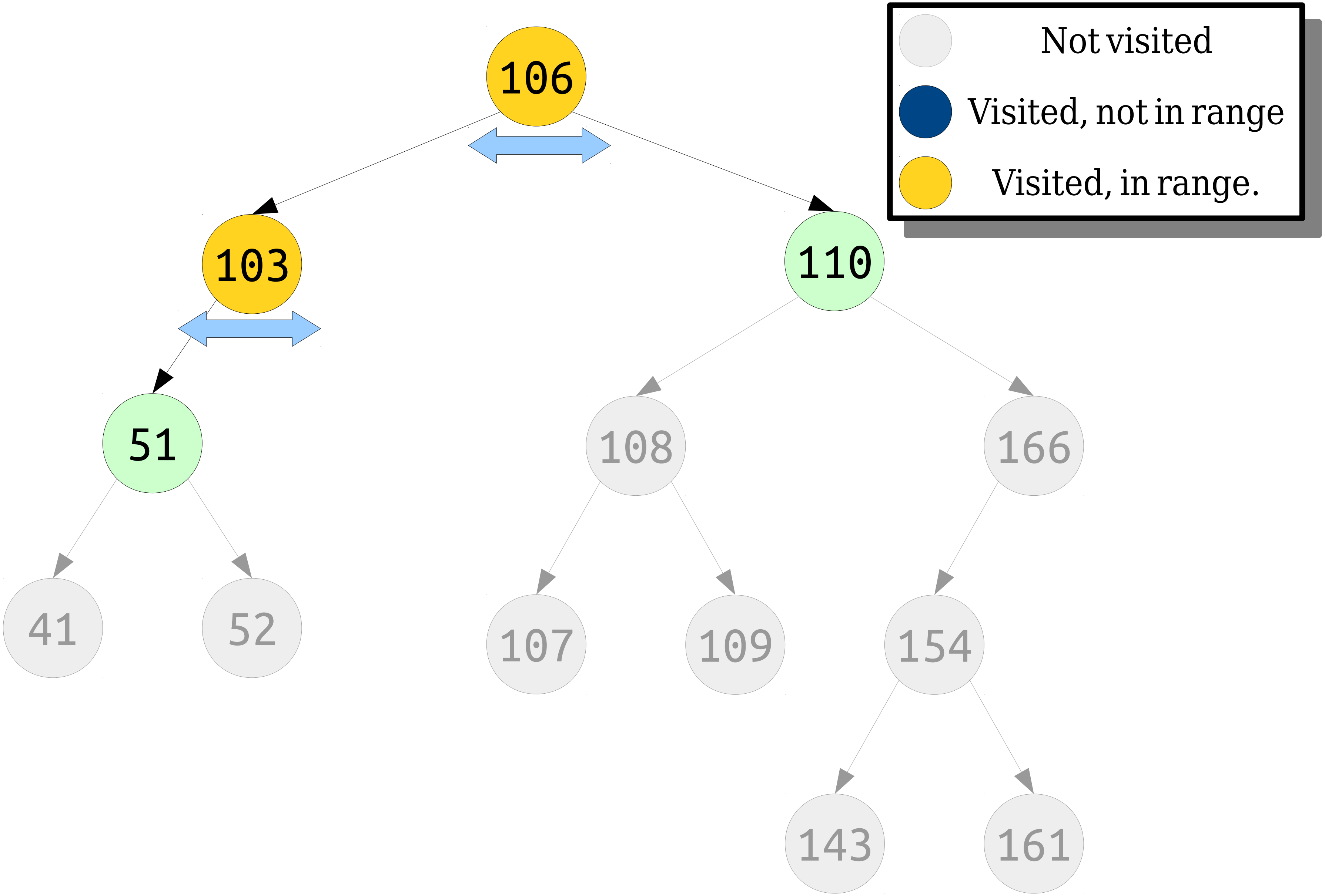
Find all elements in this tree in the range **[42, 165]**.



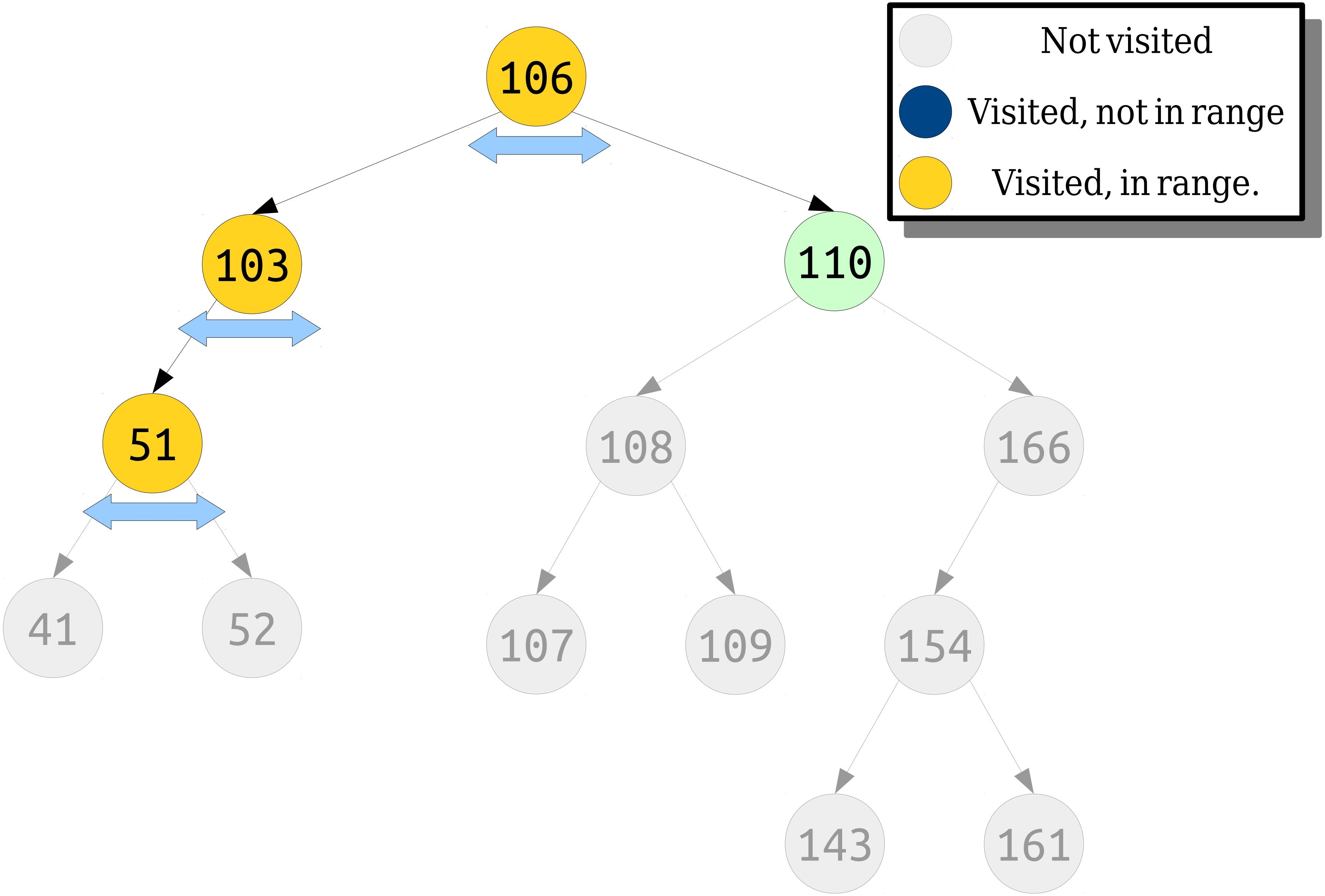
Find all elements in this tree in the range **[42, 165]**.



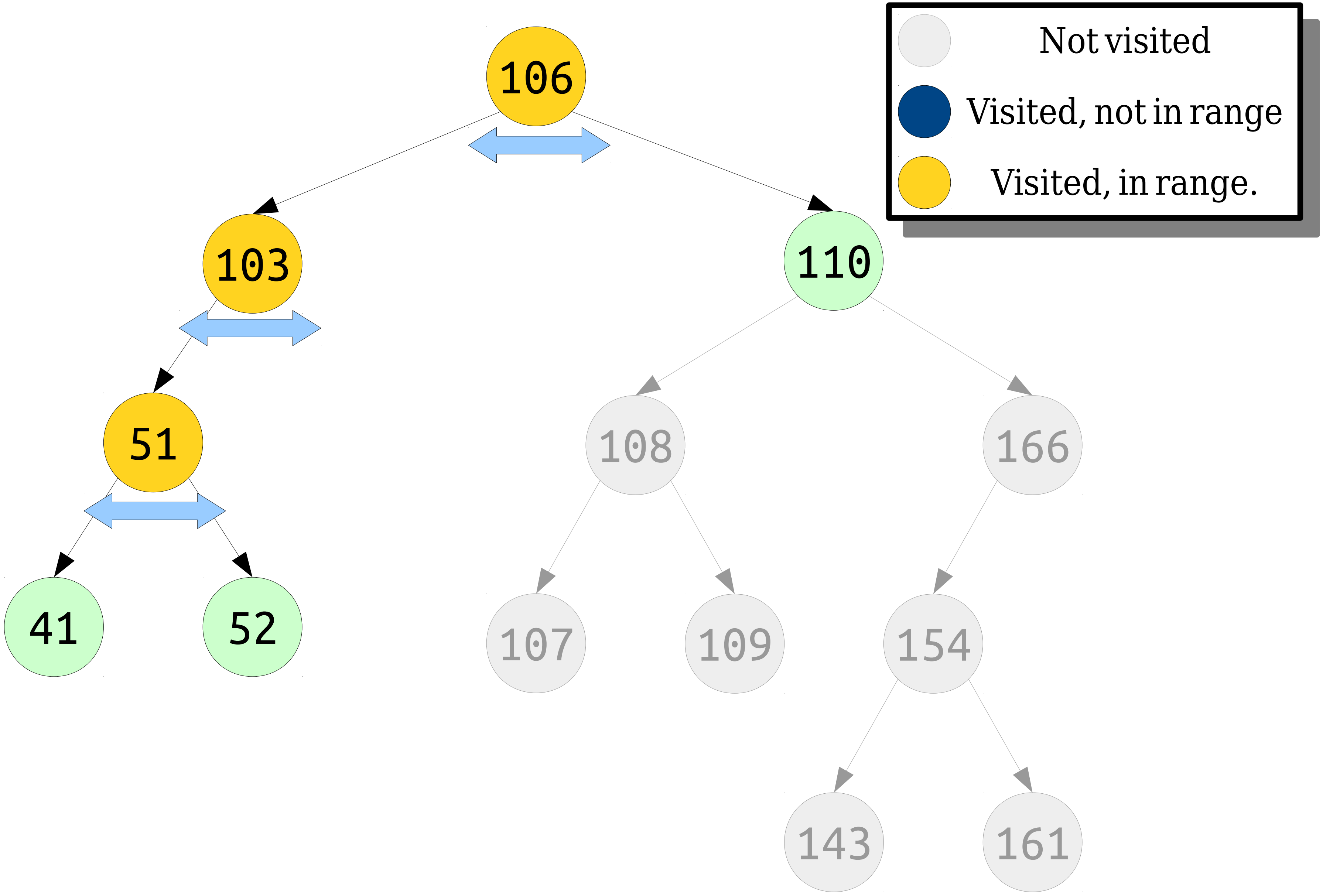
Find all elements in this tree in the range **[42, 165]**.



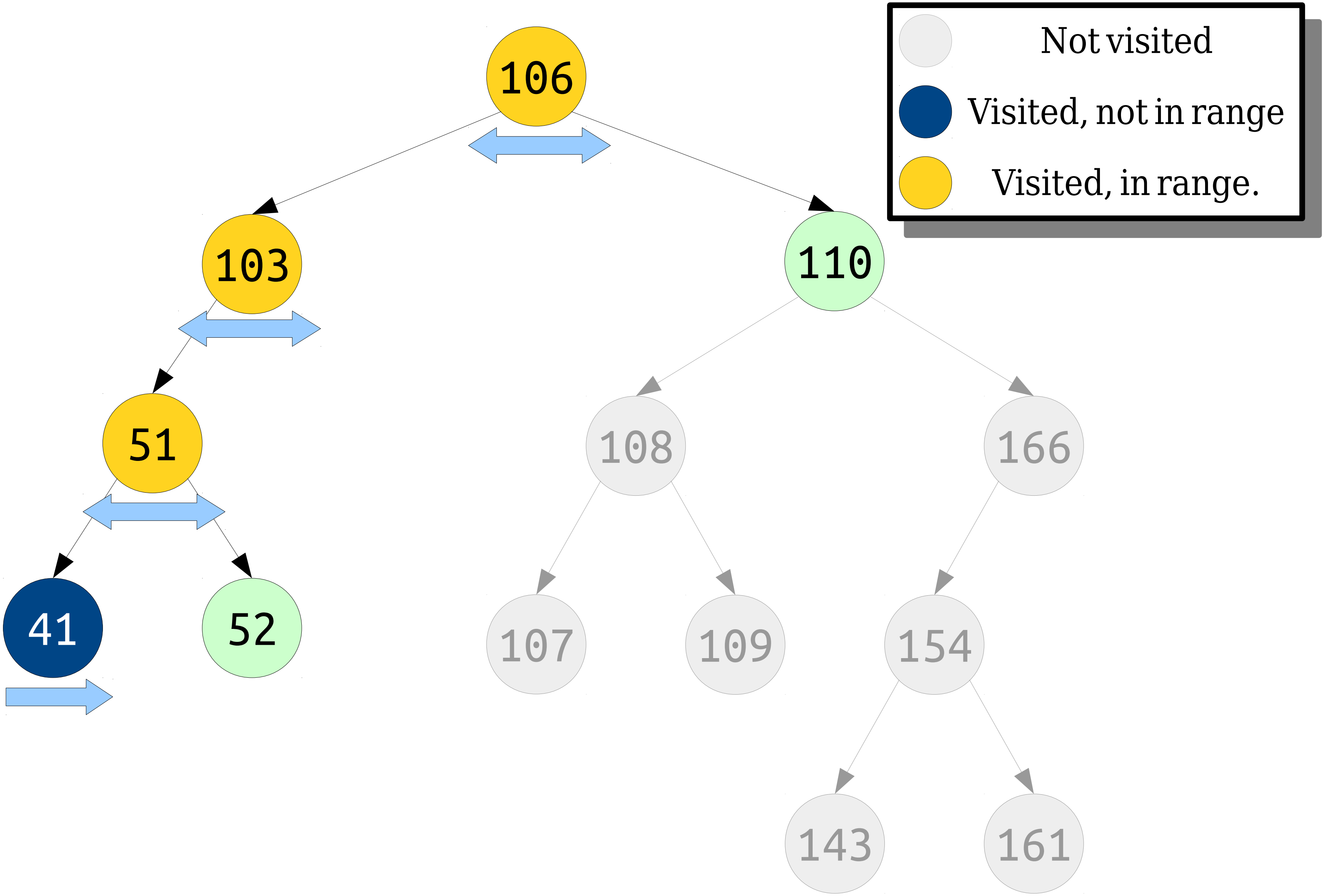
Find all elements in this tree in the range **[42, 165]**.



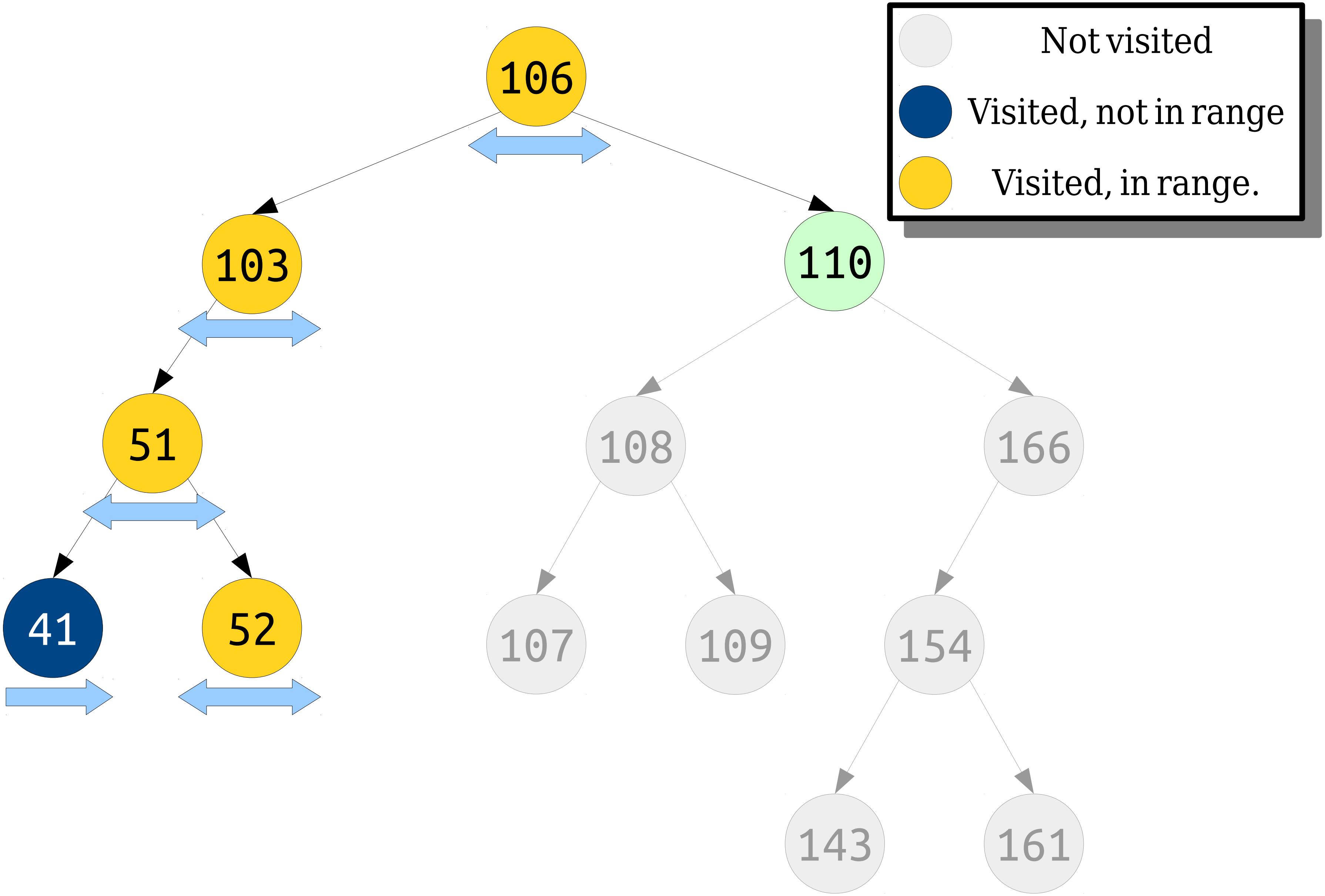
Find all elements in this tree in the range **[42, 165]**.



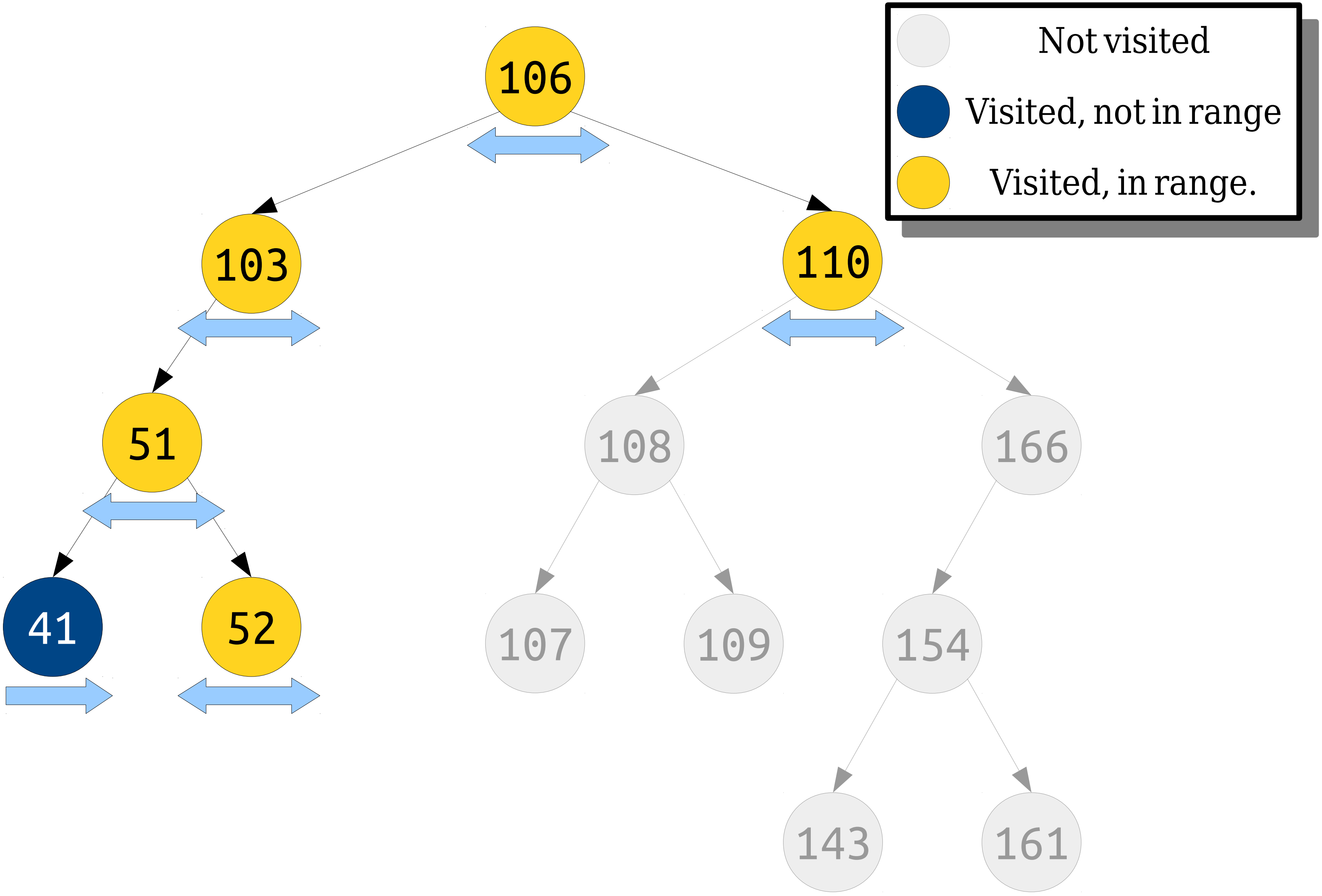
Find all elements in this tree in the range **[42, 165]**.



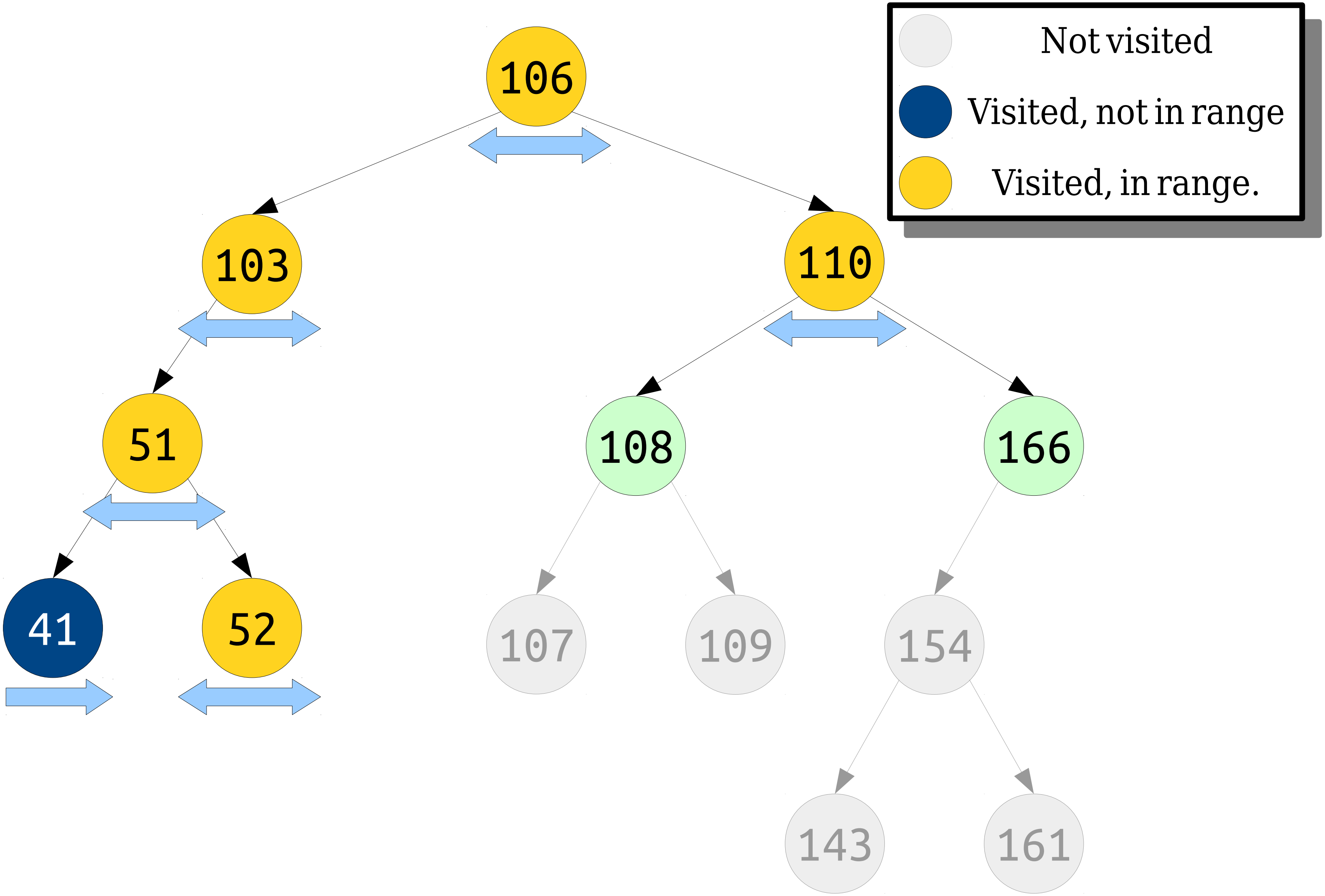
Find all elements in this tree in the range **[42, 165]**.



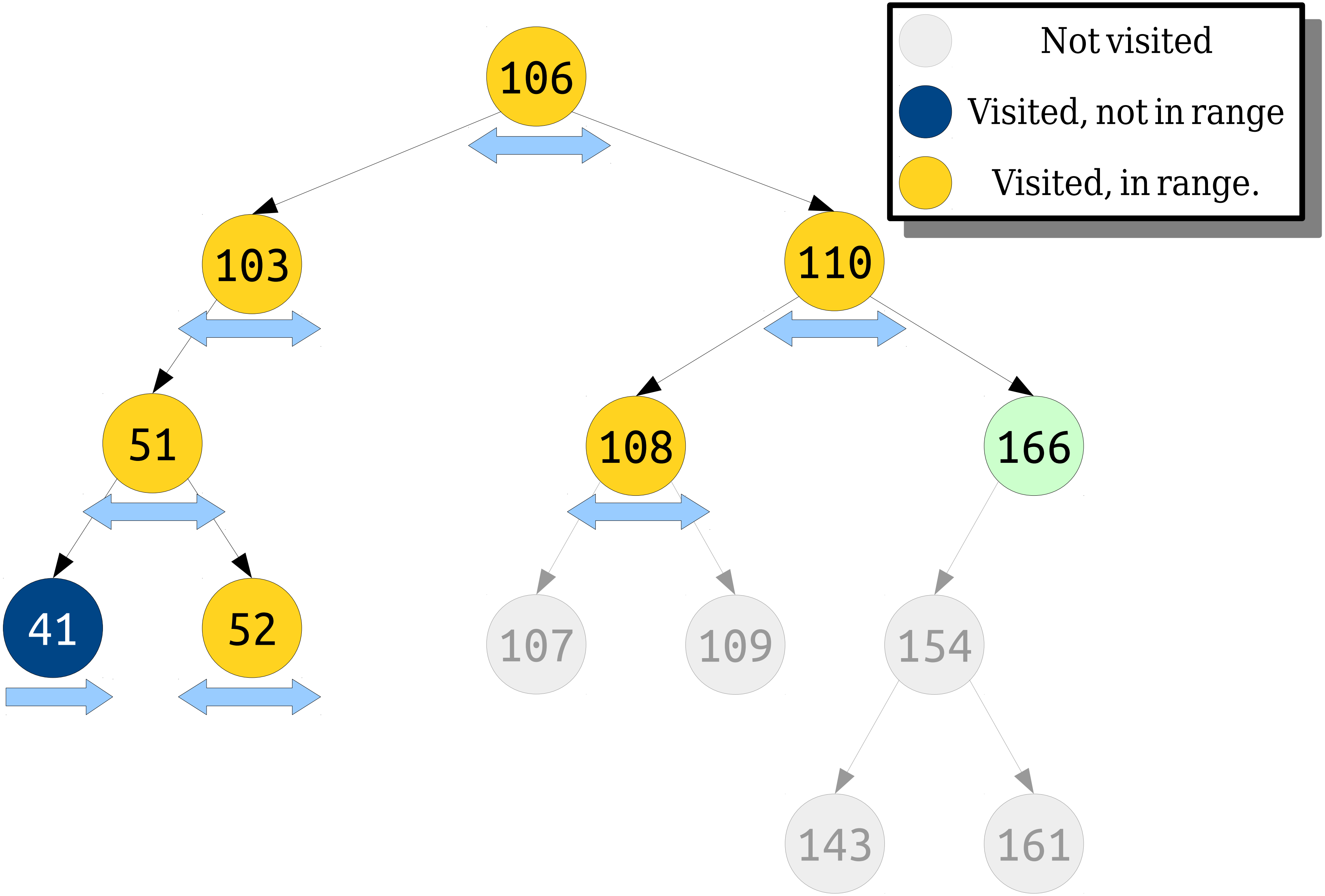
Find all elements in this tree in the range **[42, 165]**.



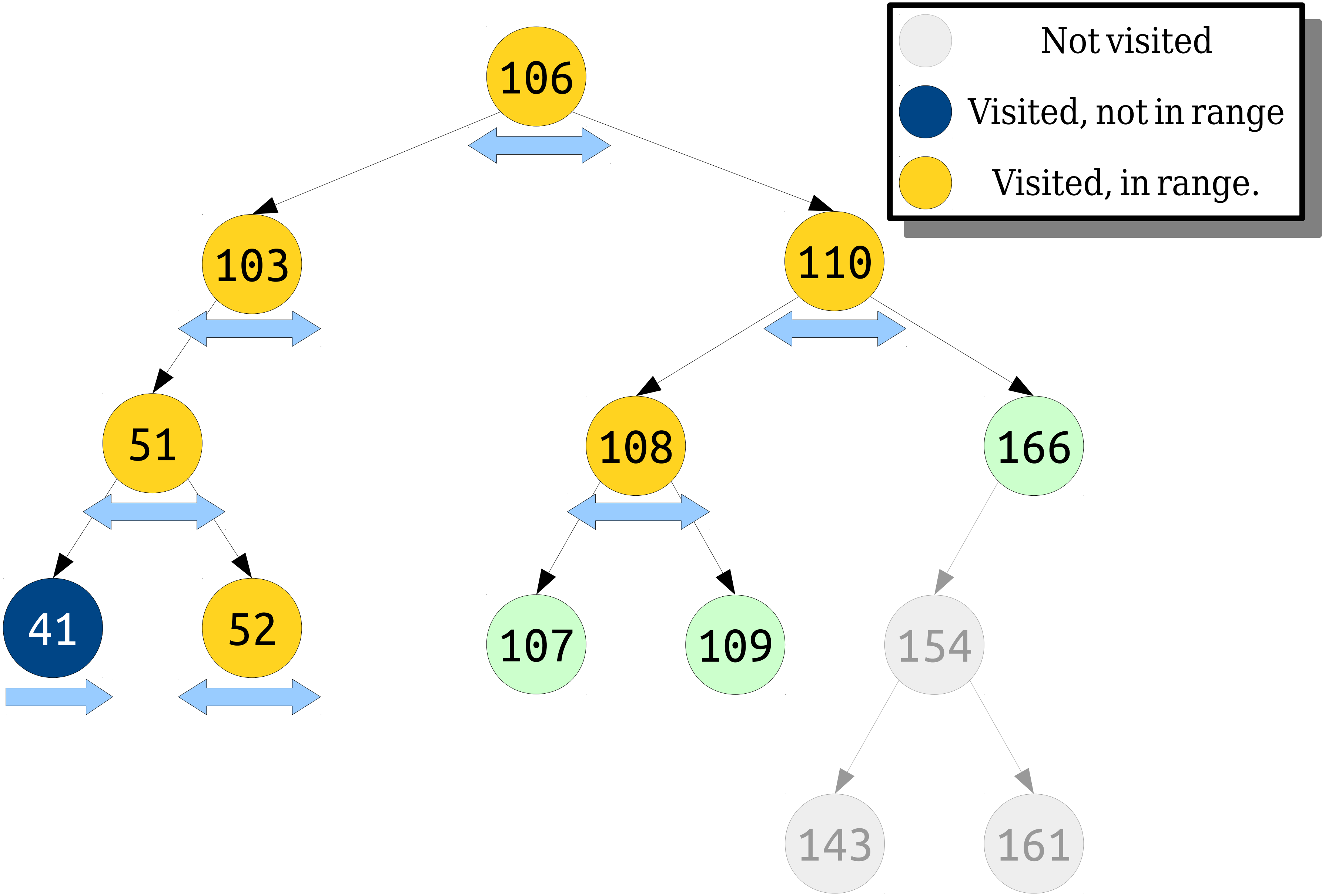
Find all elements in this tree in the range **[42, 165]**.



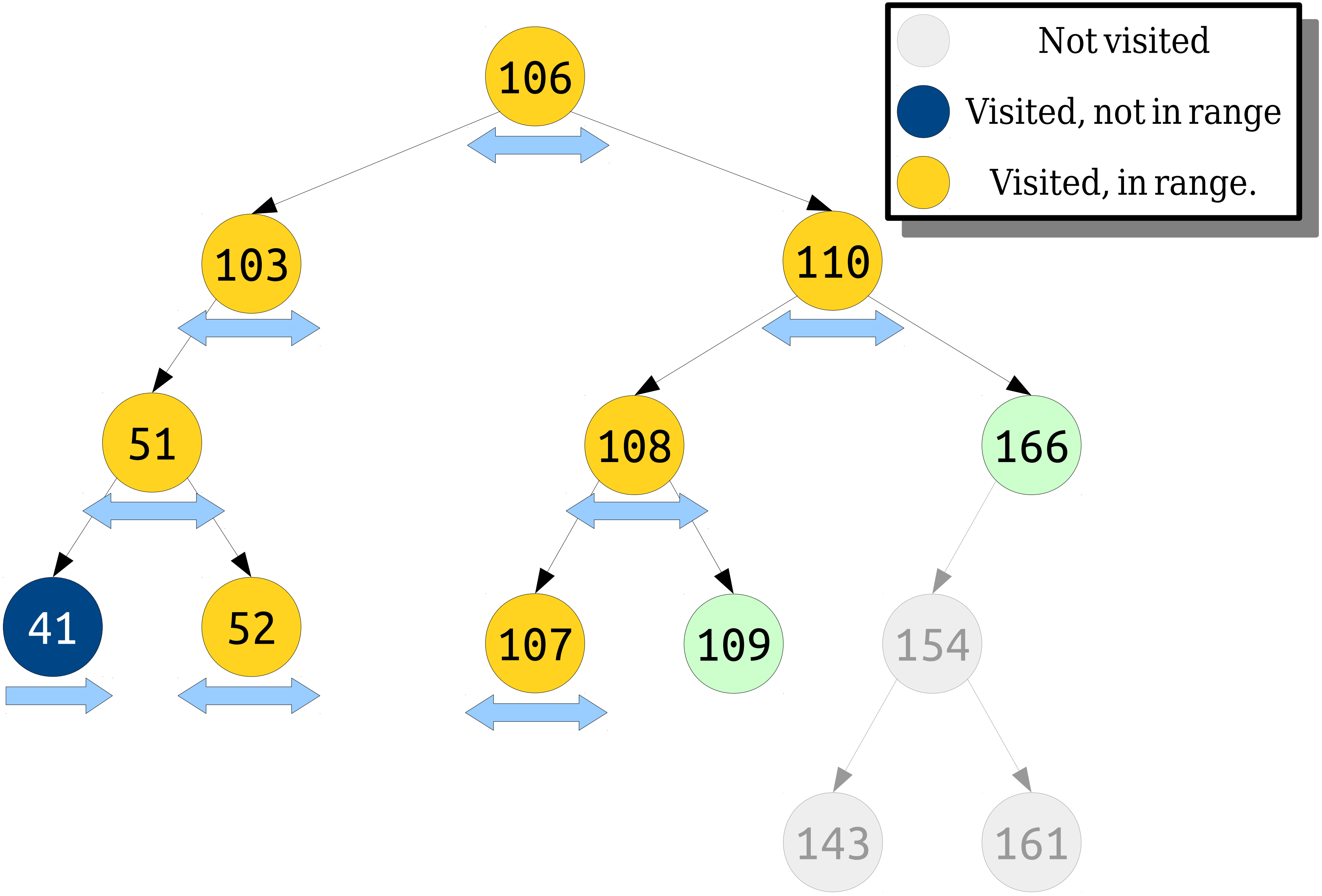
Find all elements in this tree in the range **[42, 165]**.



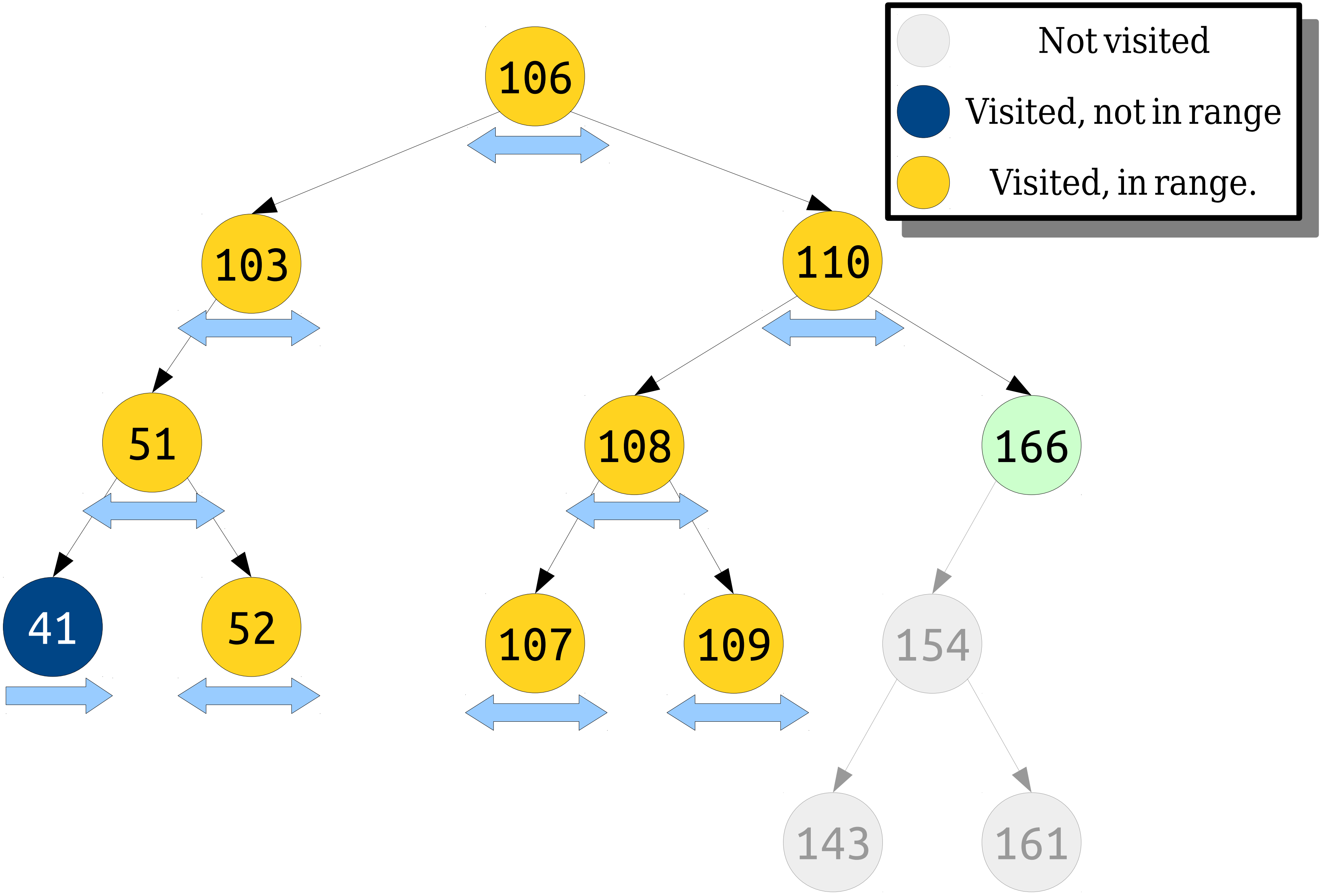
Find all elements in this tree in the range **[42, 165]**.



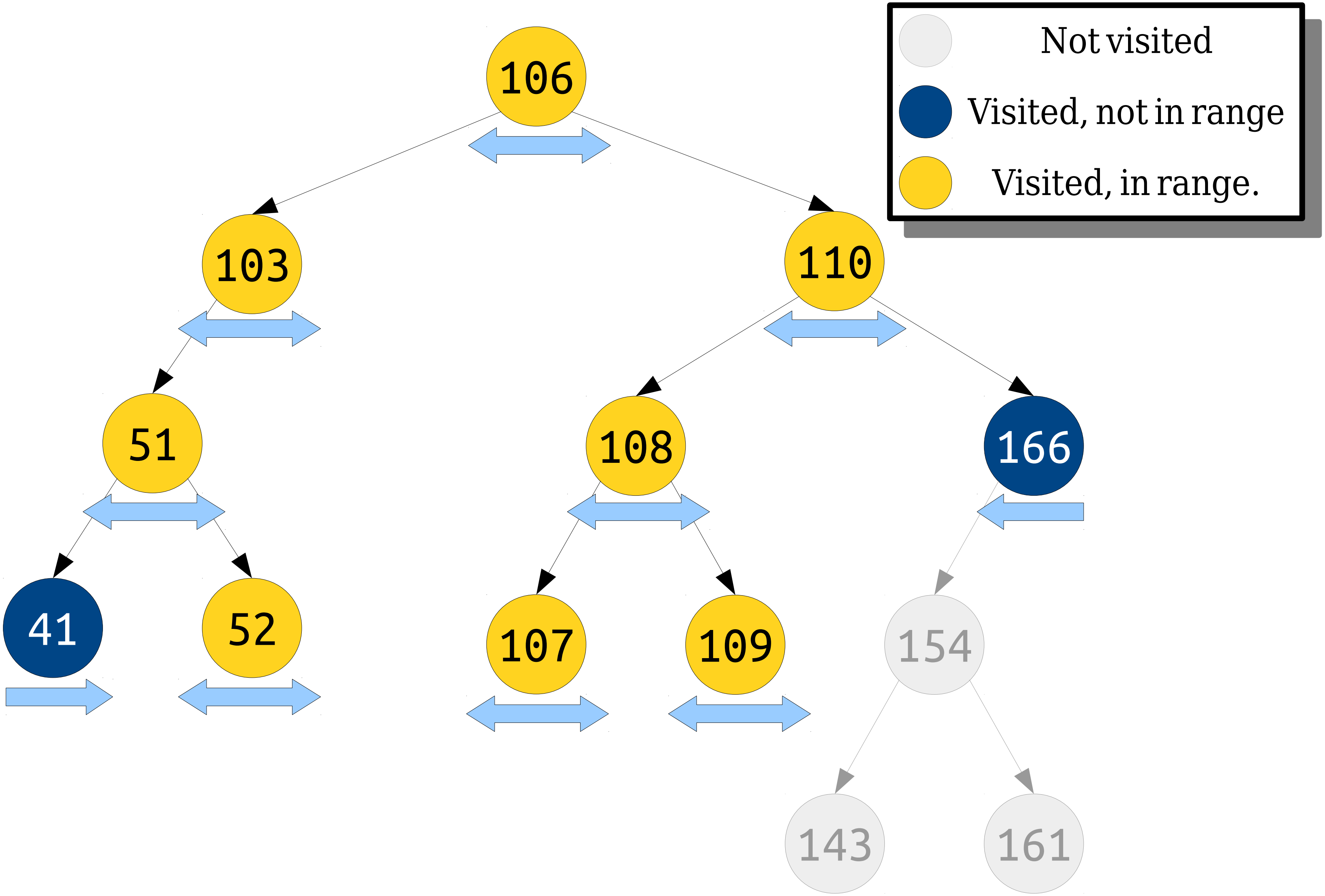
Find all elements in this tree in the range **[42, 165]**.



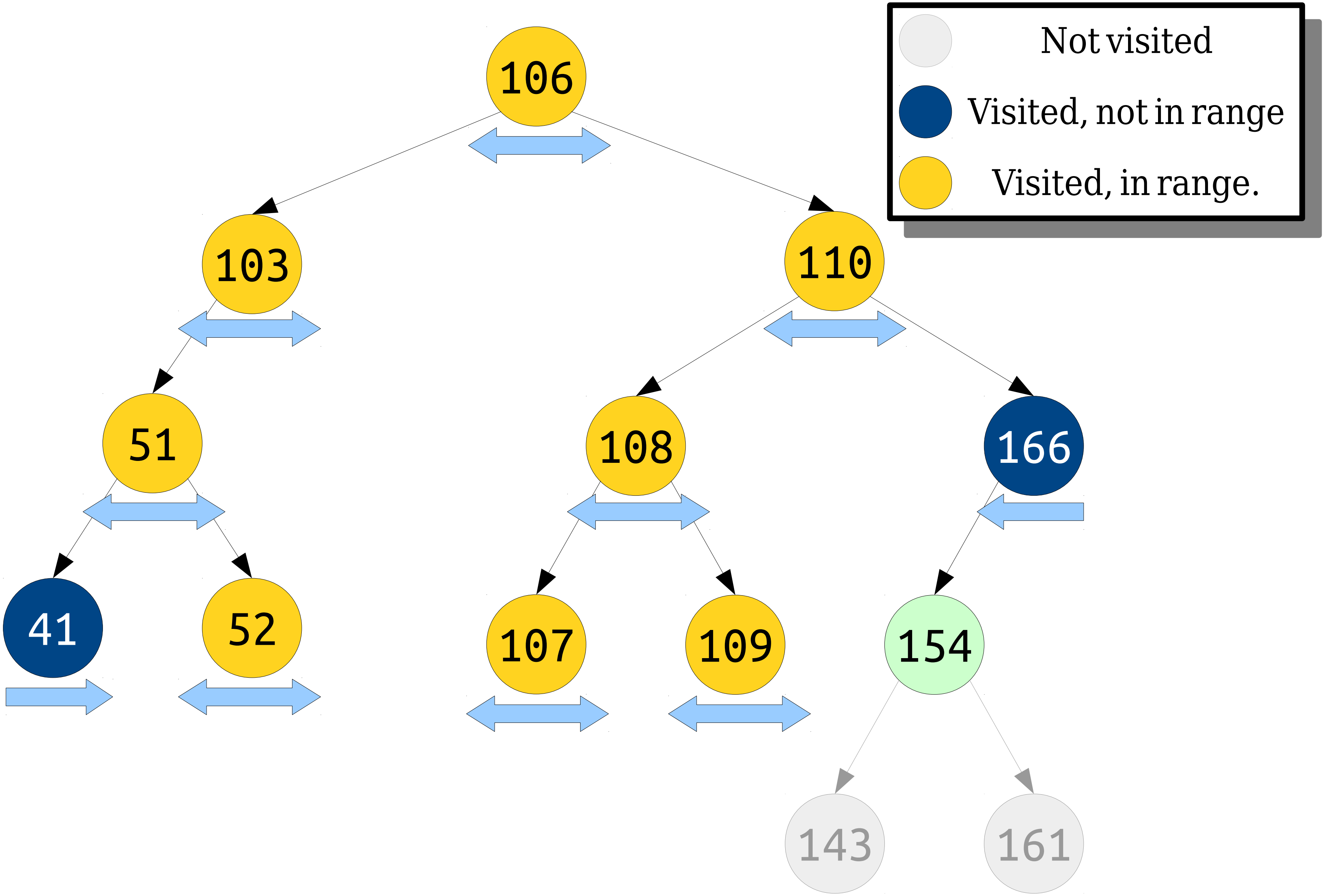
Find all elements in this tree in the range **[42, 165]**.



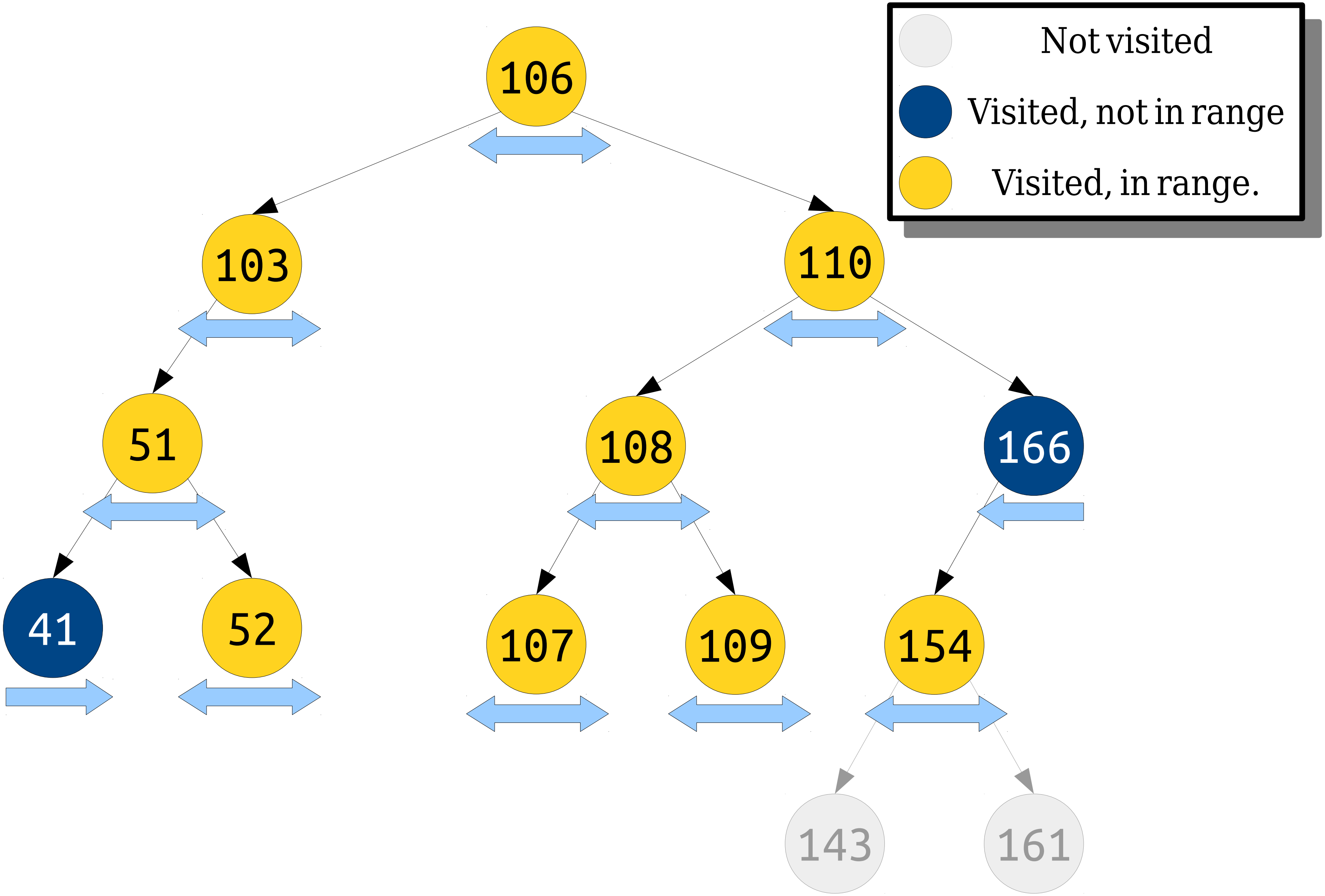
Find all elements in this tree in the range **[42, 165]**.



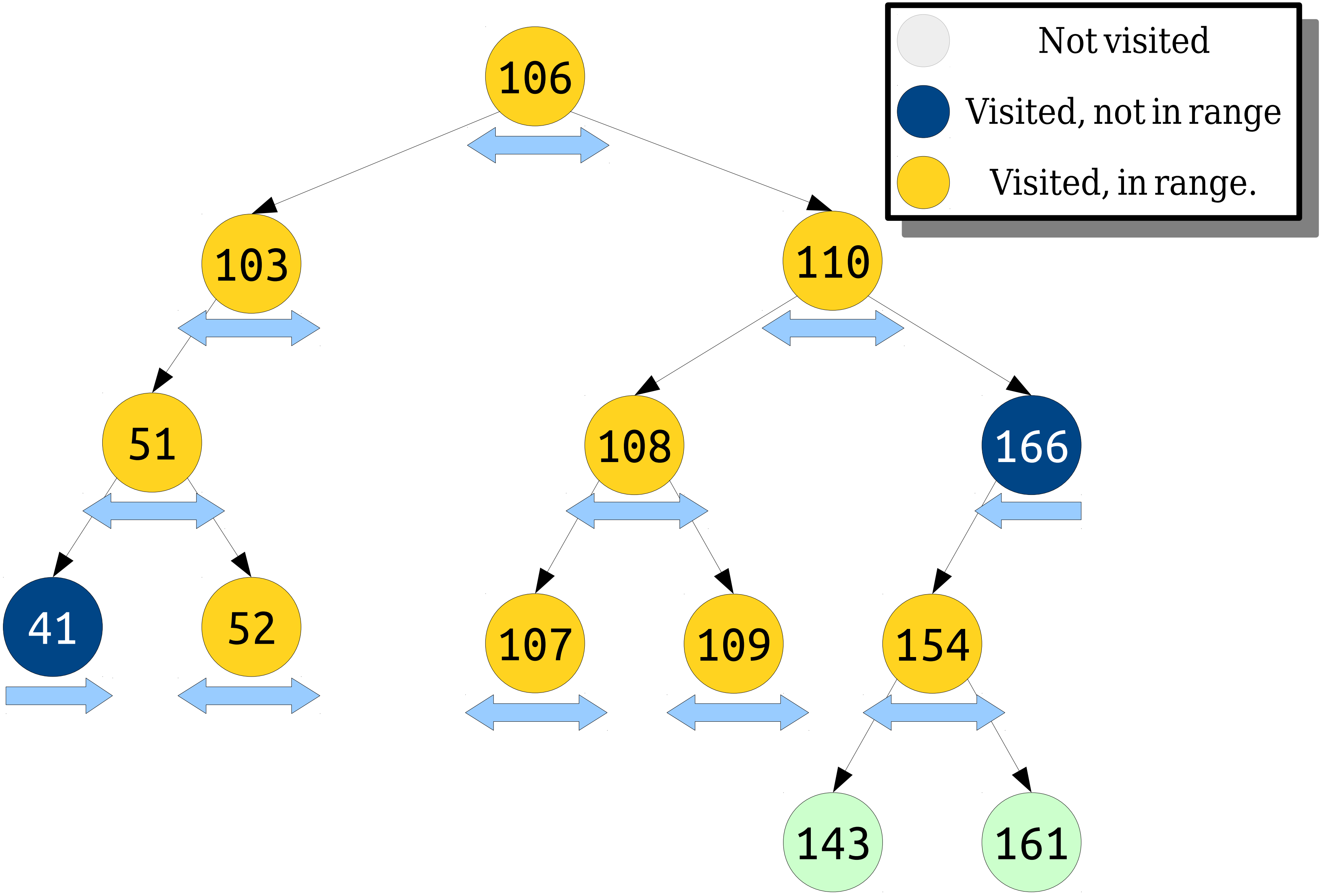
Find all elements in this tree in the range **[42, 165]**.



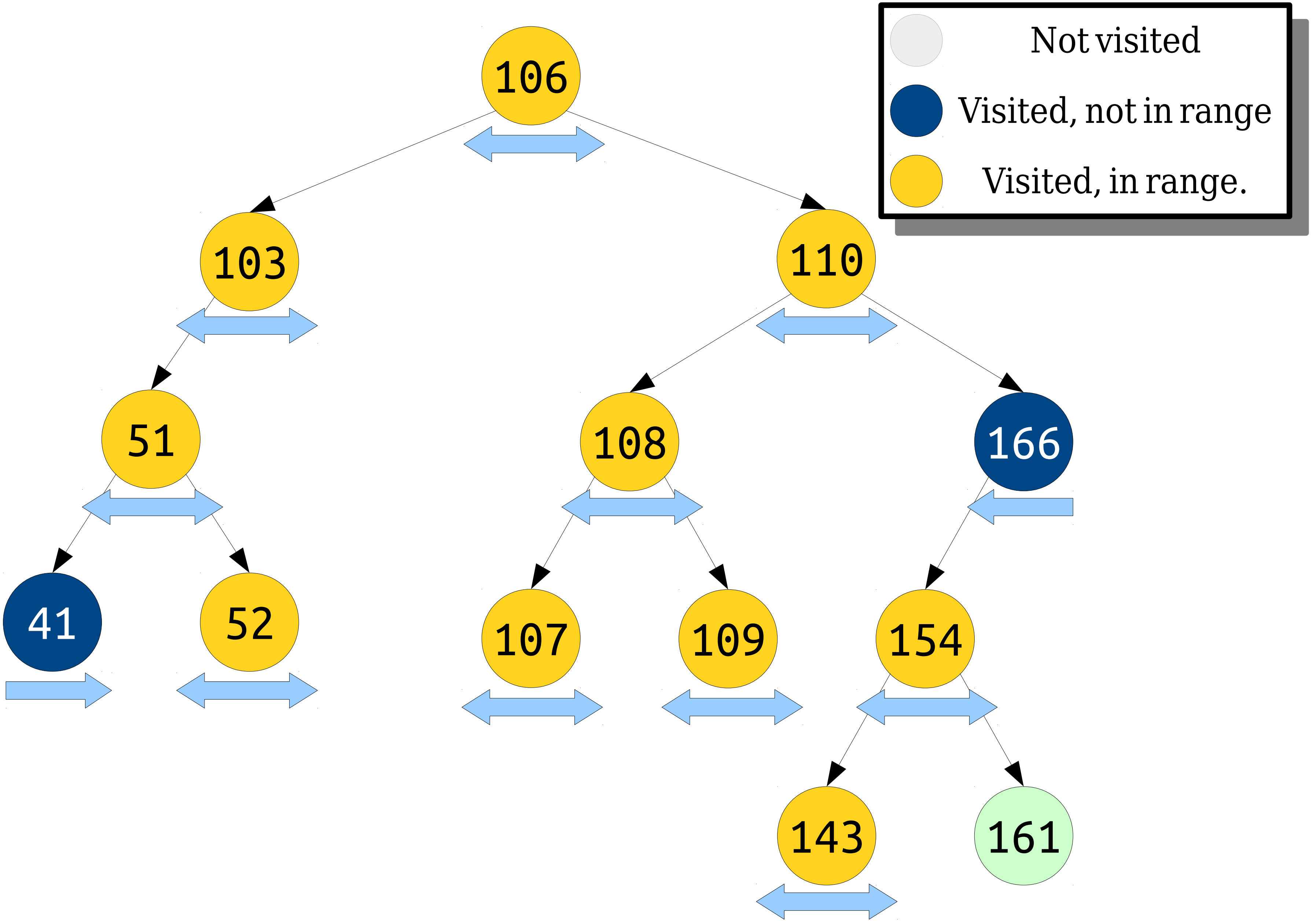
Find all elements in this tree in the range **[42, 165]**.



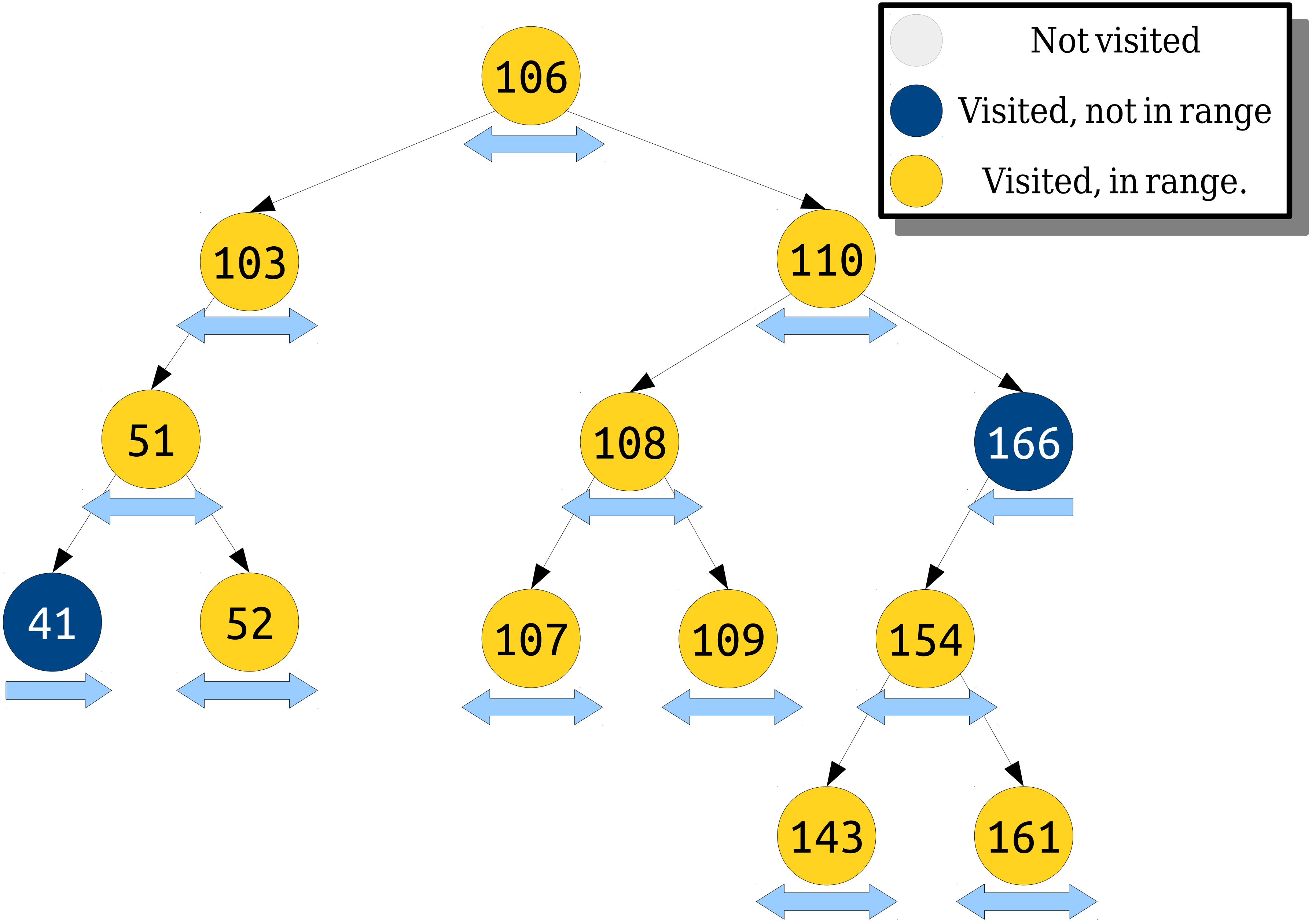
Find all elements in this tree in the range **[42, 165]**.



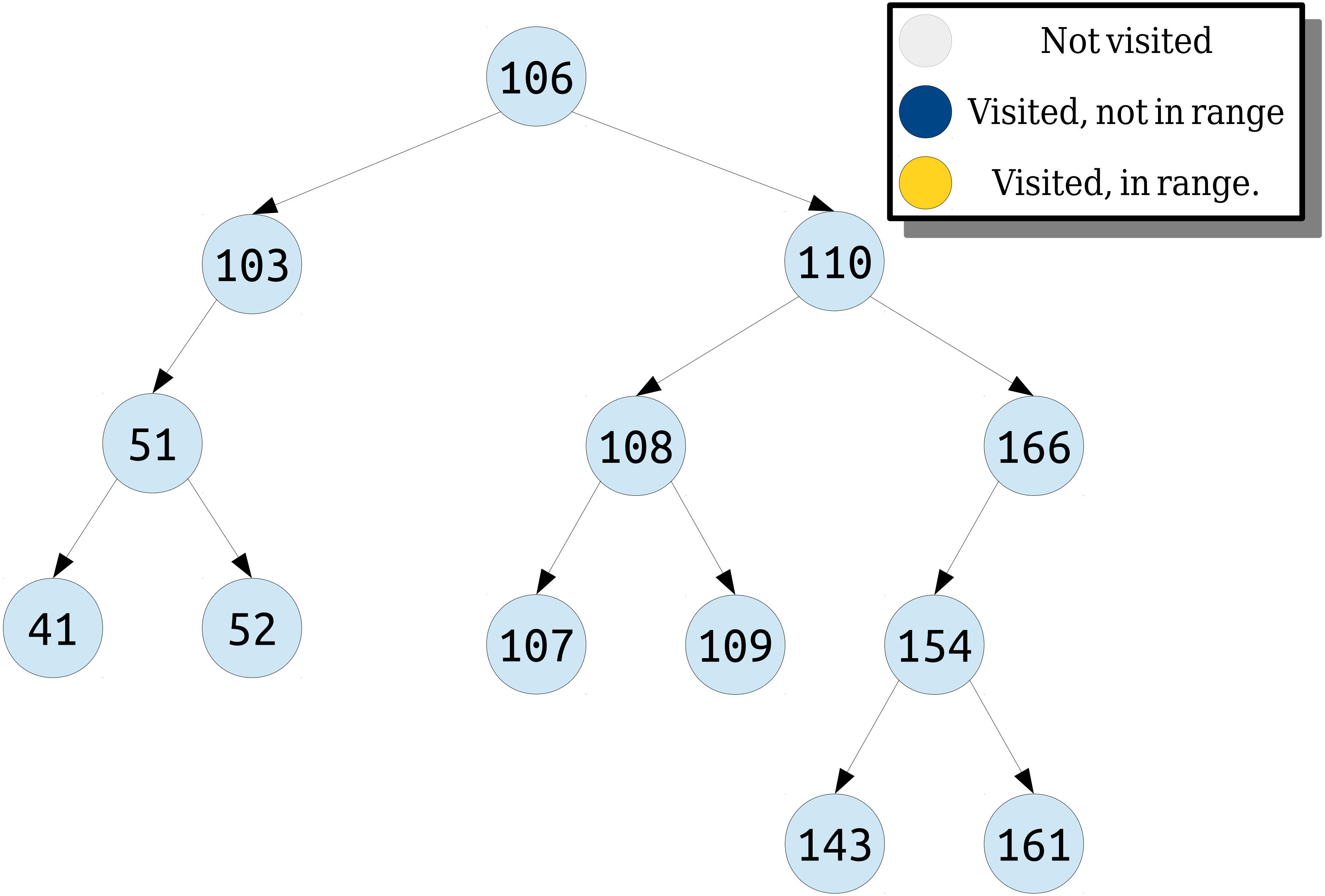
Find all elements in this tree in the range **[42, 165]**.



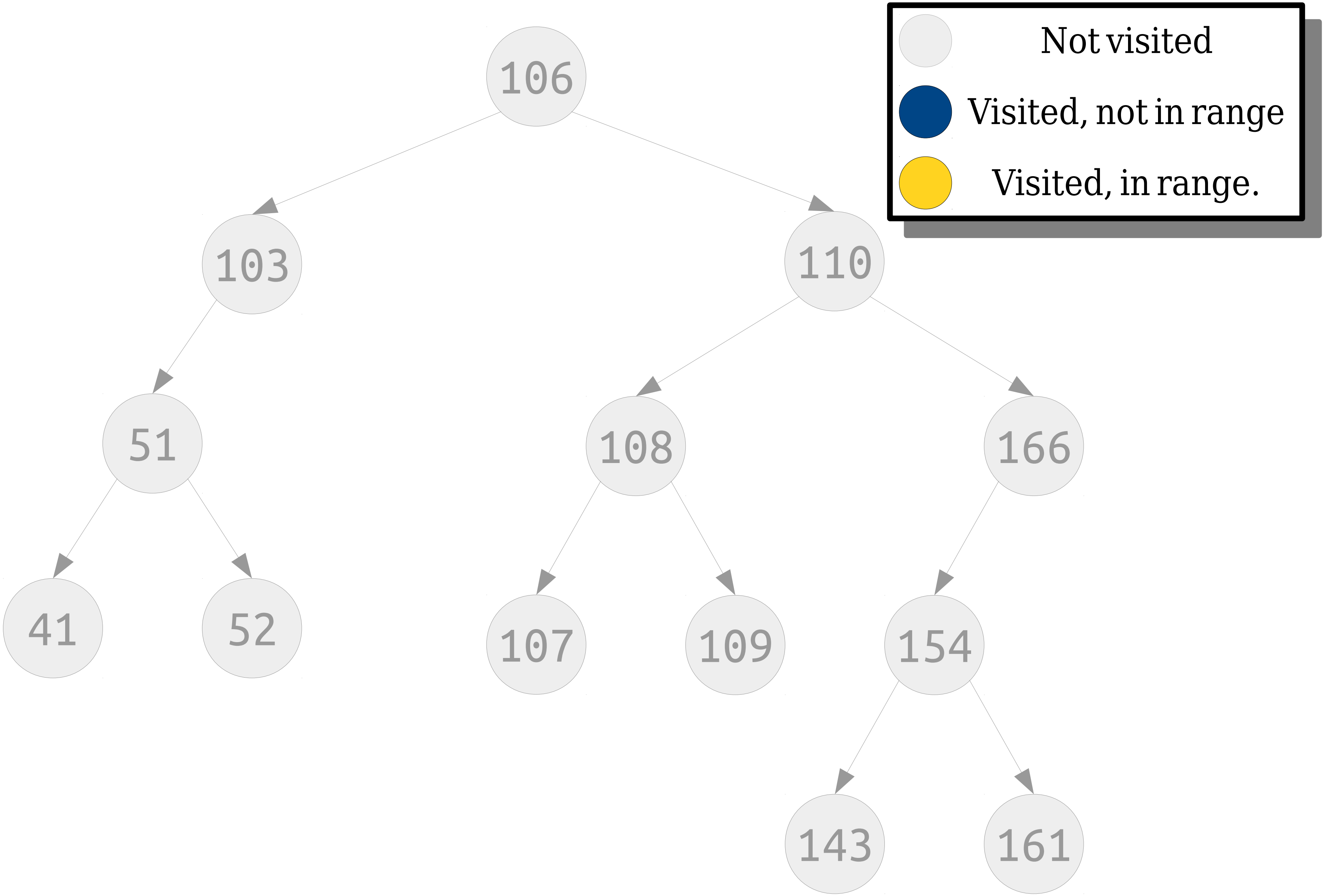
Find all elements in this tree in the range **[42, 165]**.



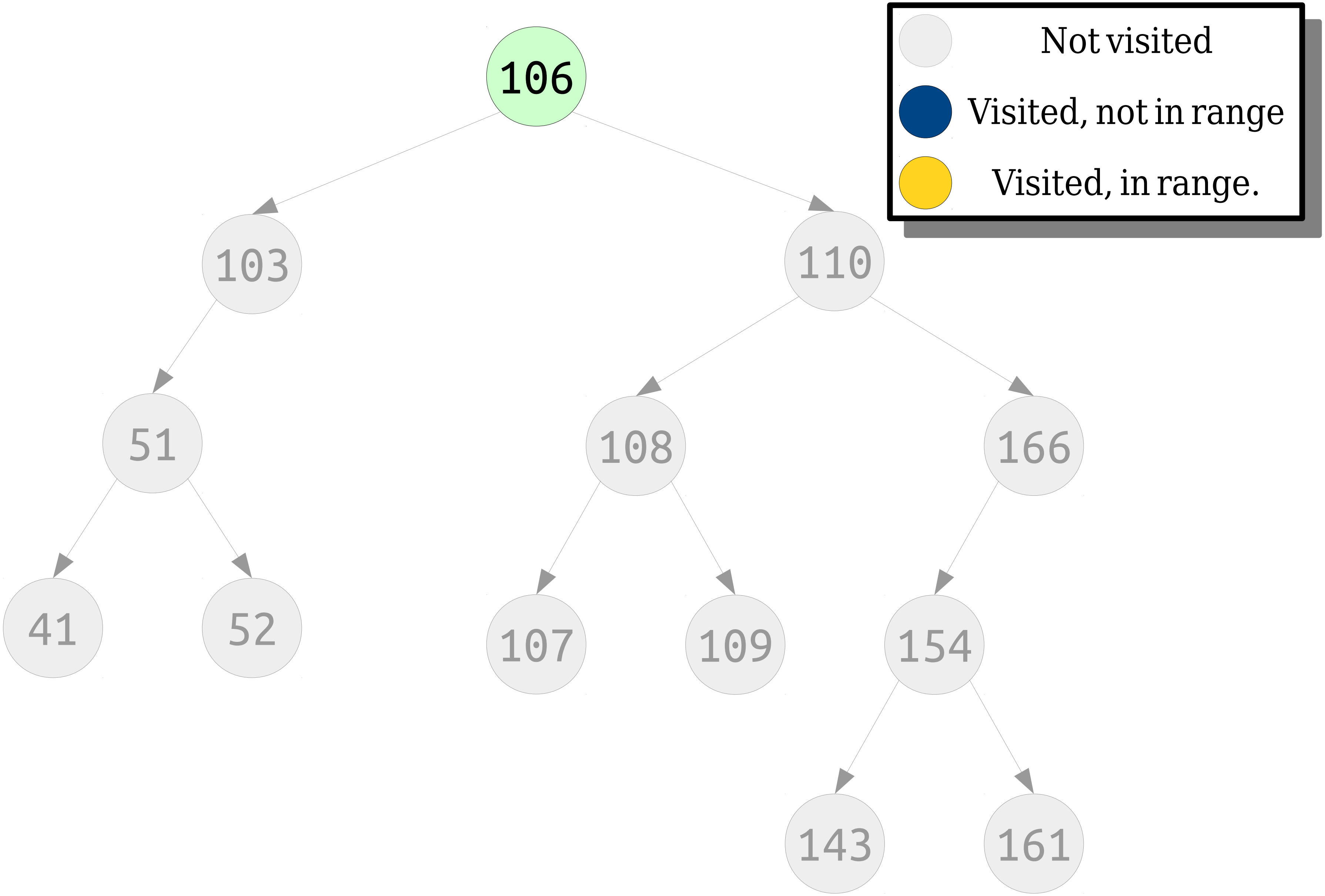
Find all elements in this tree in the range **[42, 165]**.



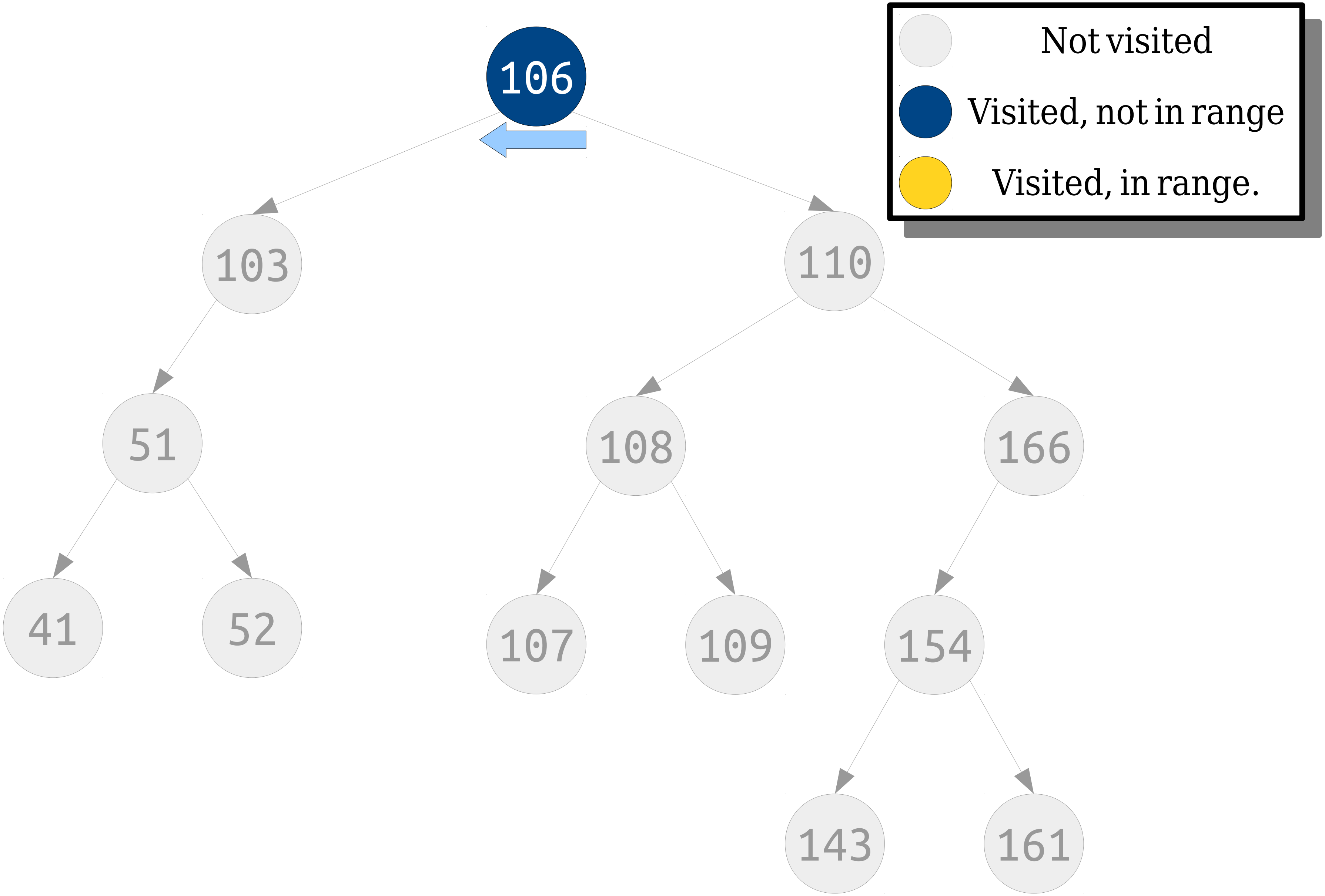
Find all elements in this tree in the range **[49, 50]**.



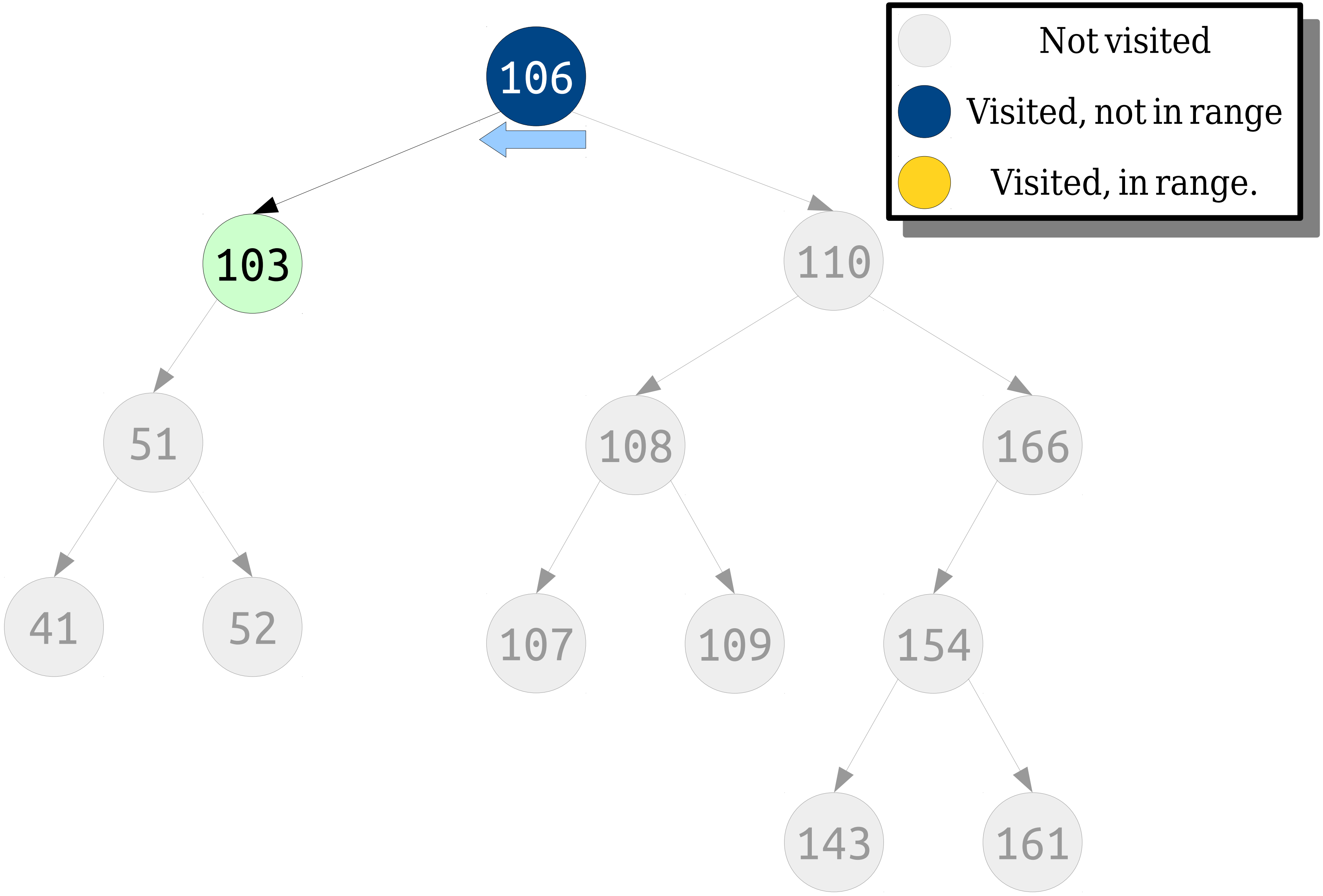
Find all elements in this tree in the range **[49, 50]**.



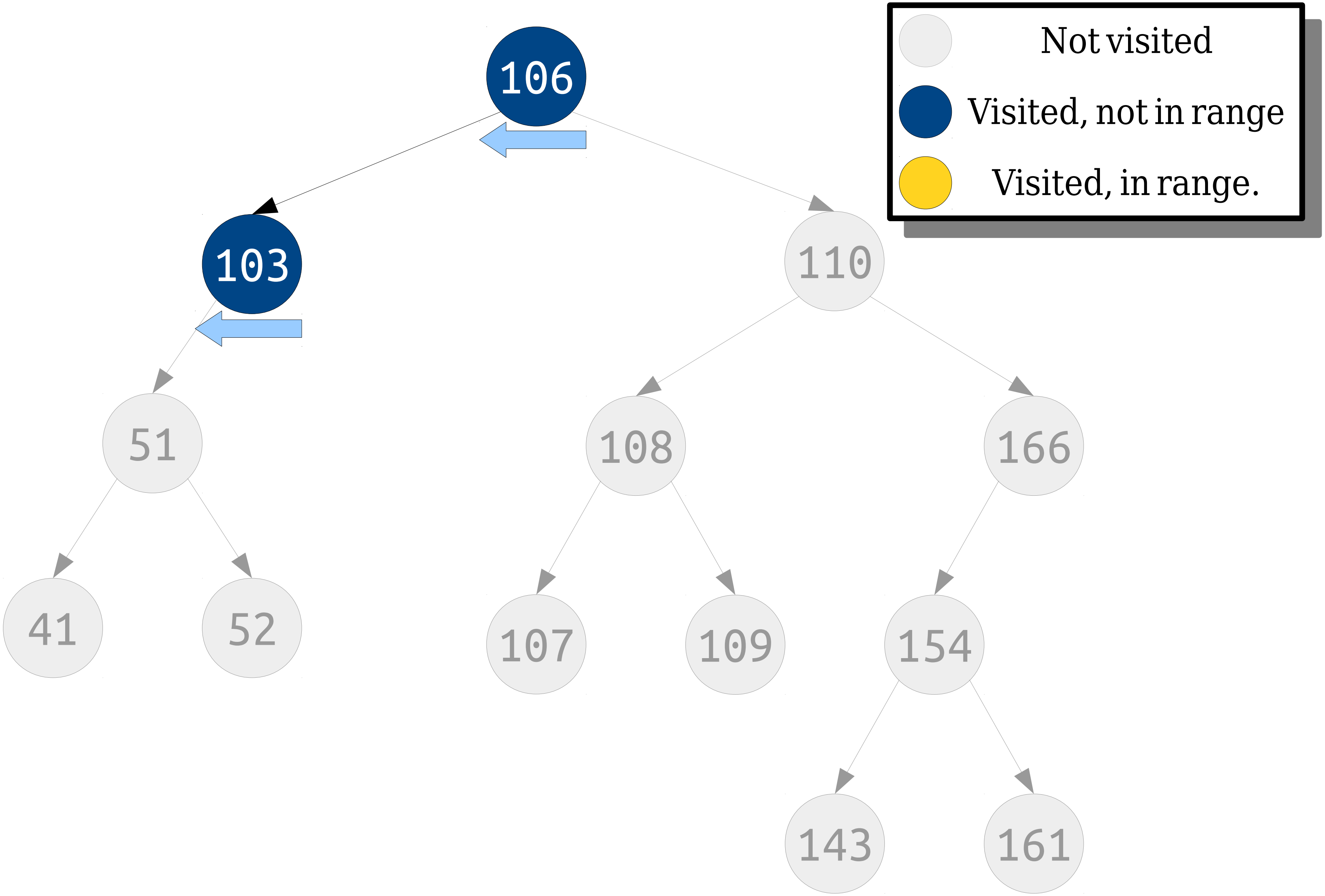
Find all elements in this tree in the range **[49, 50]**.



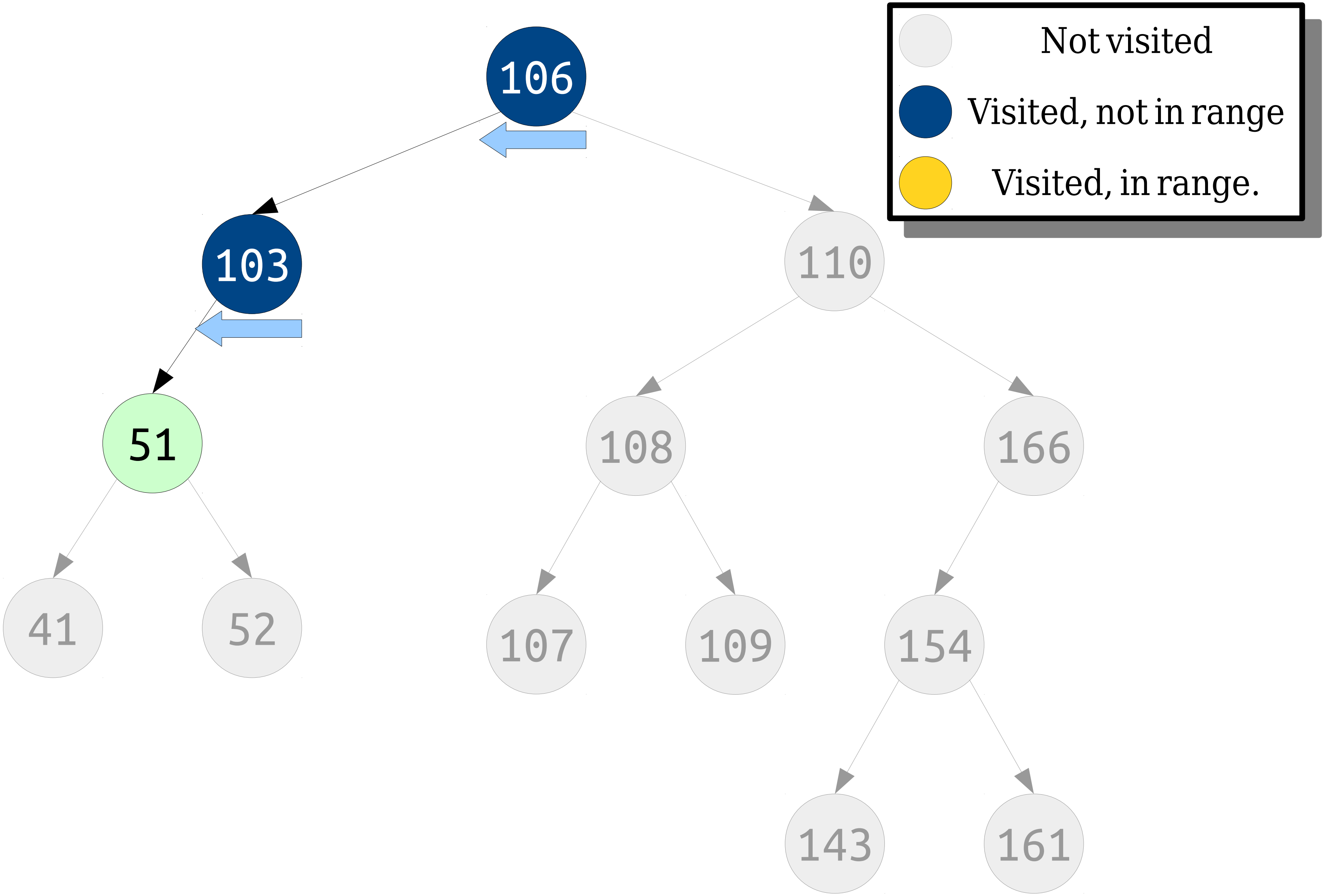
Find all elements in this tree in the range **[49, 50]**.



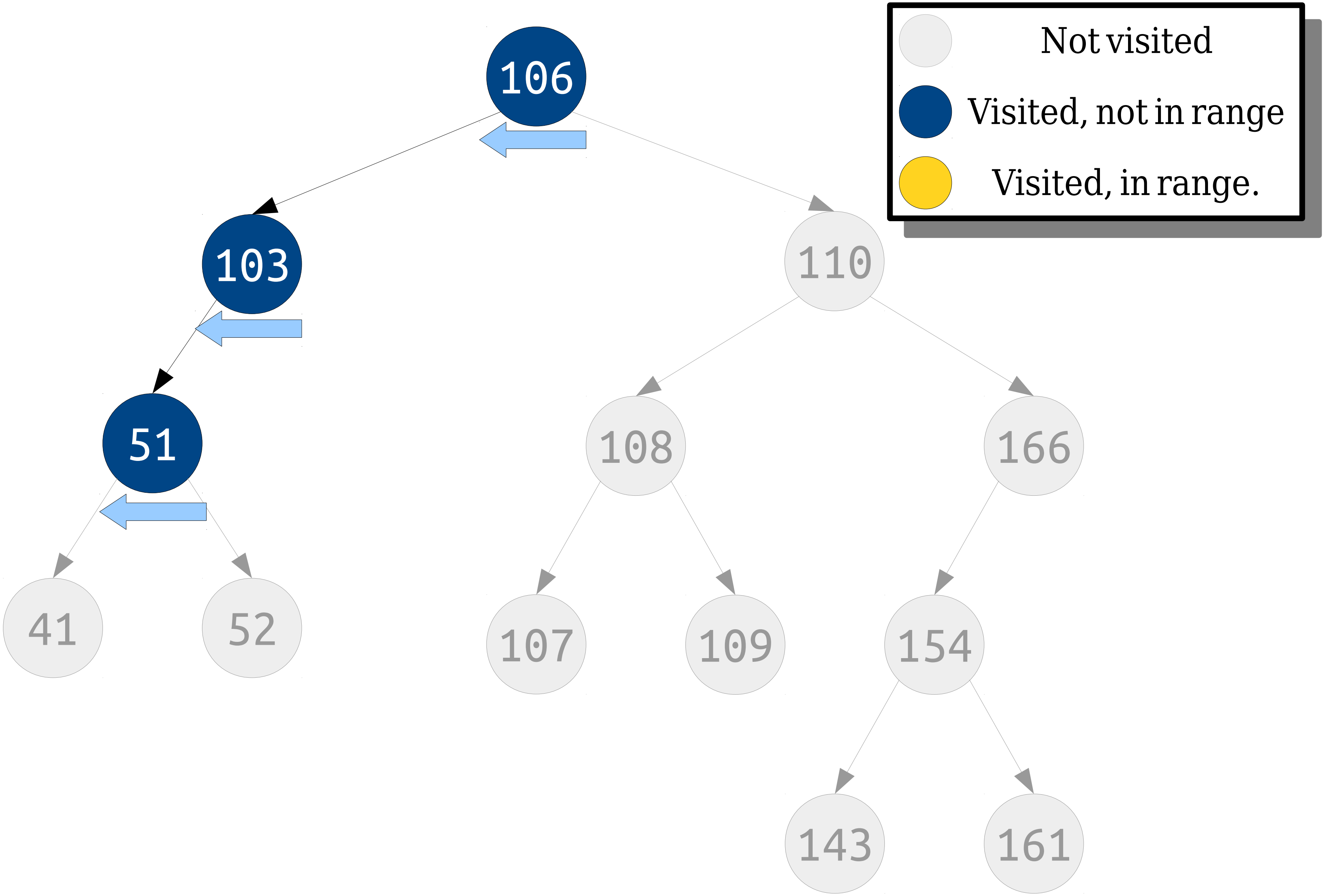
Find all elements in this tree in the range **[49, 50]**.



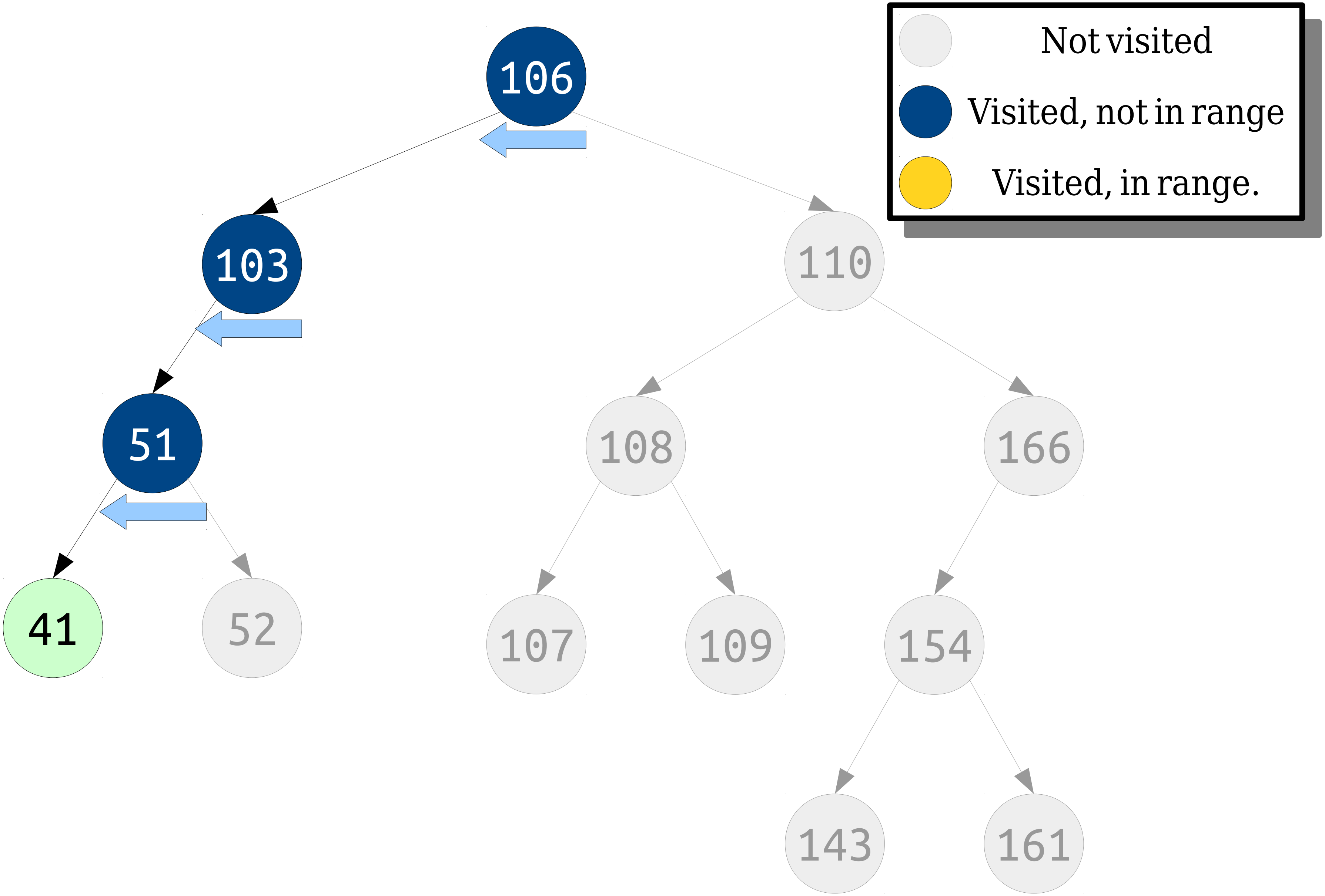
Find all elements in this tree in the range **[49, 50]**.



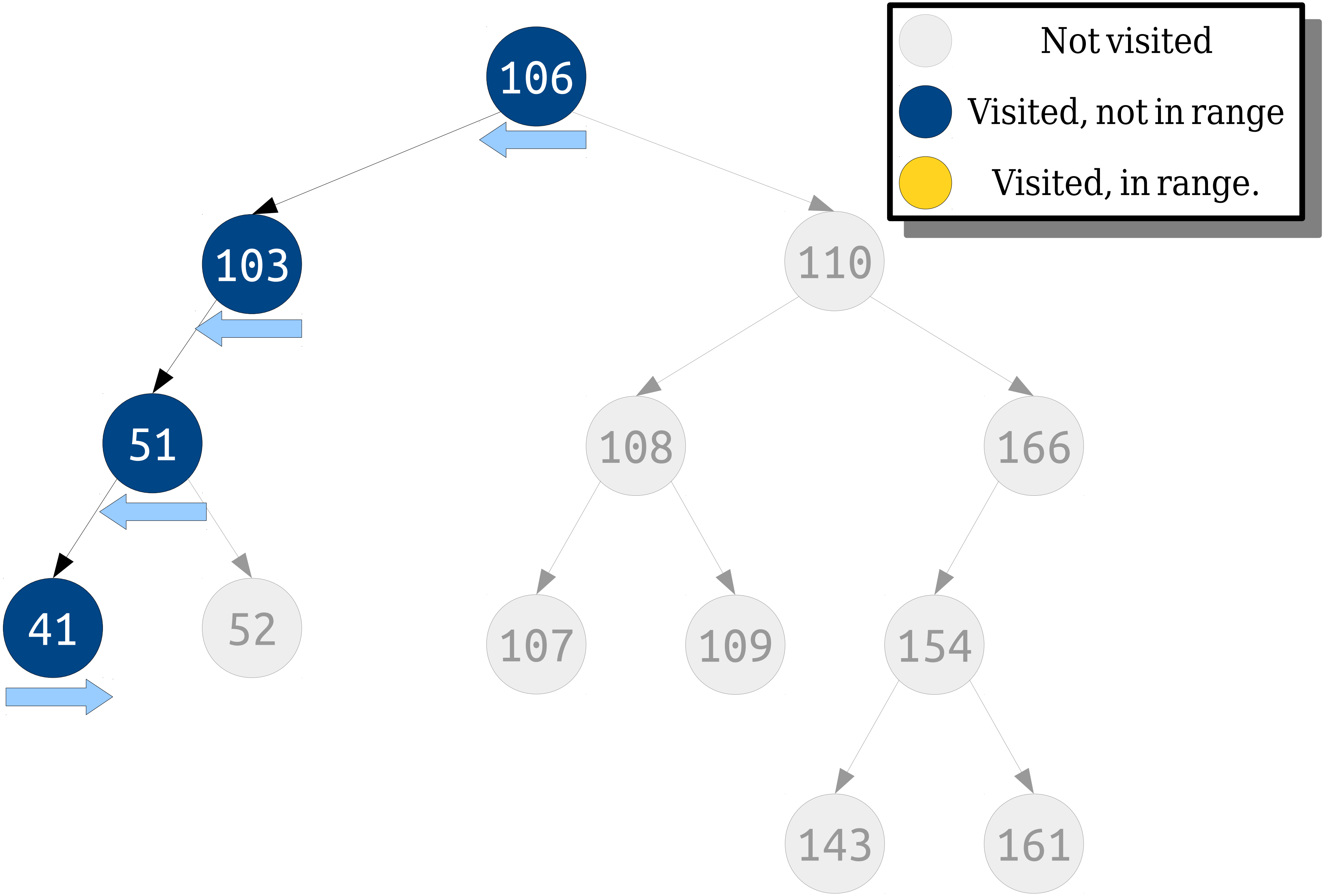
Find all elements in this tree in the range **[49, 50]**.



Find all elements in this tree in the range **[49, 50]**.



Find all elements in this tree in the range **[49, 50]**.

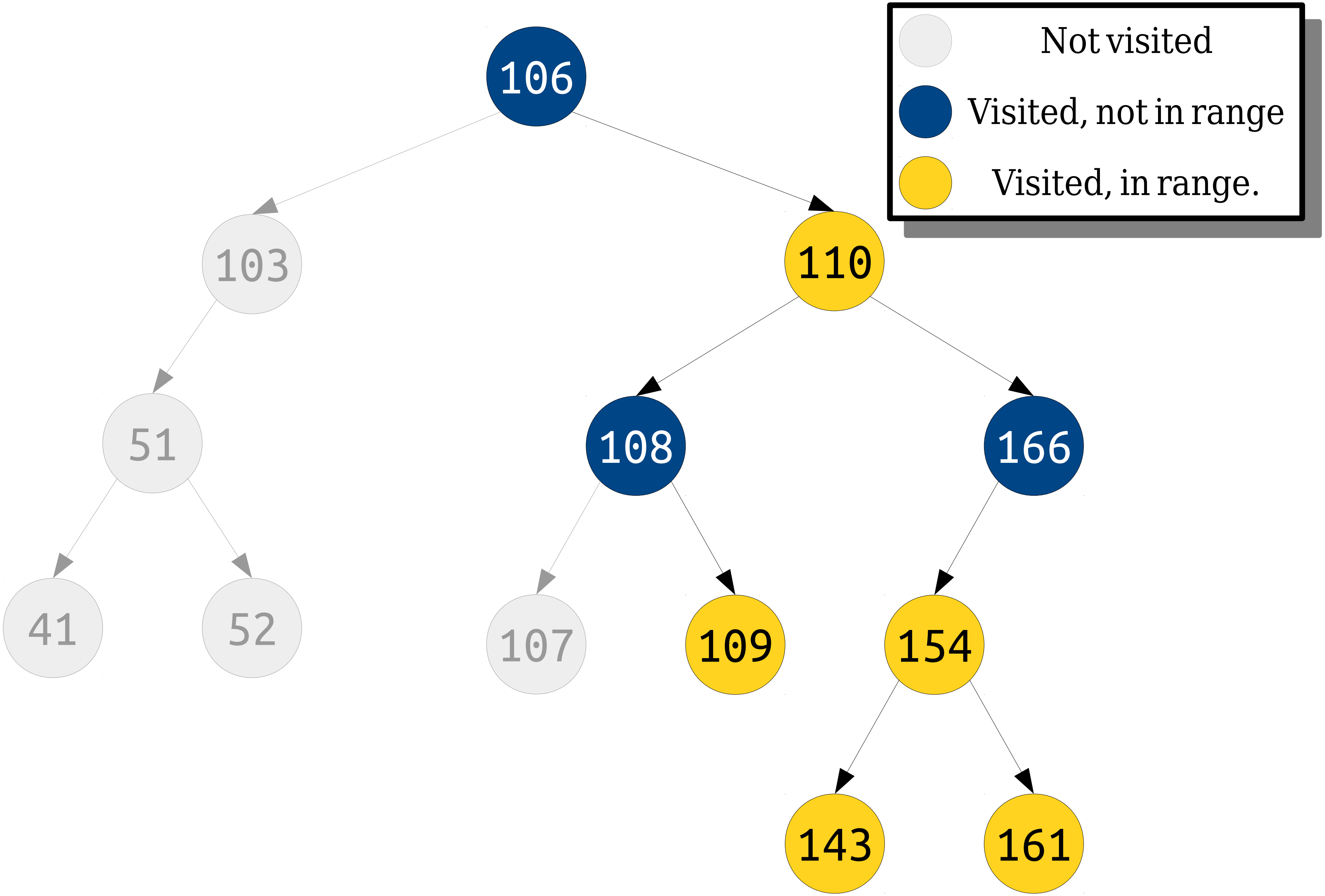


Find all elements in this tree in the range **[49, 50]**.

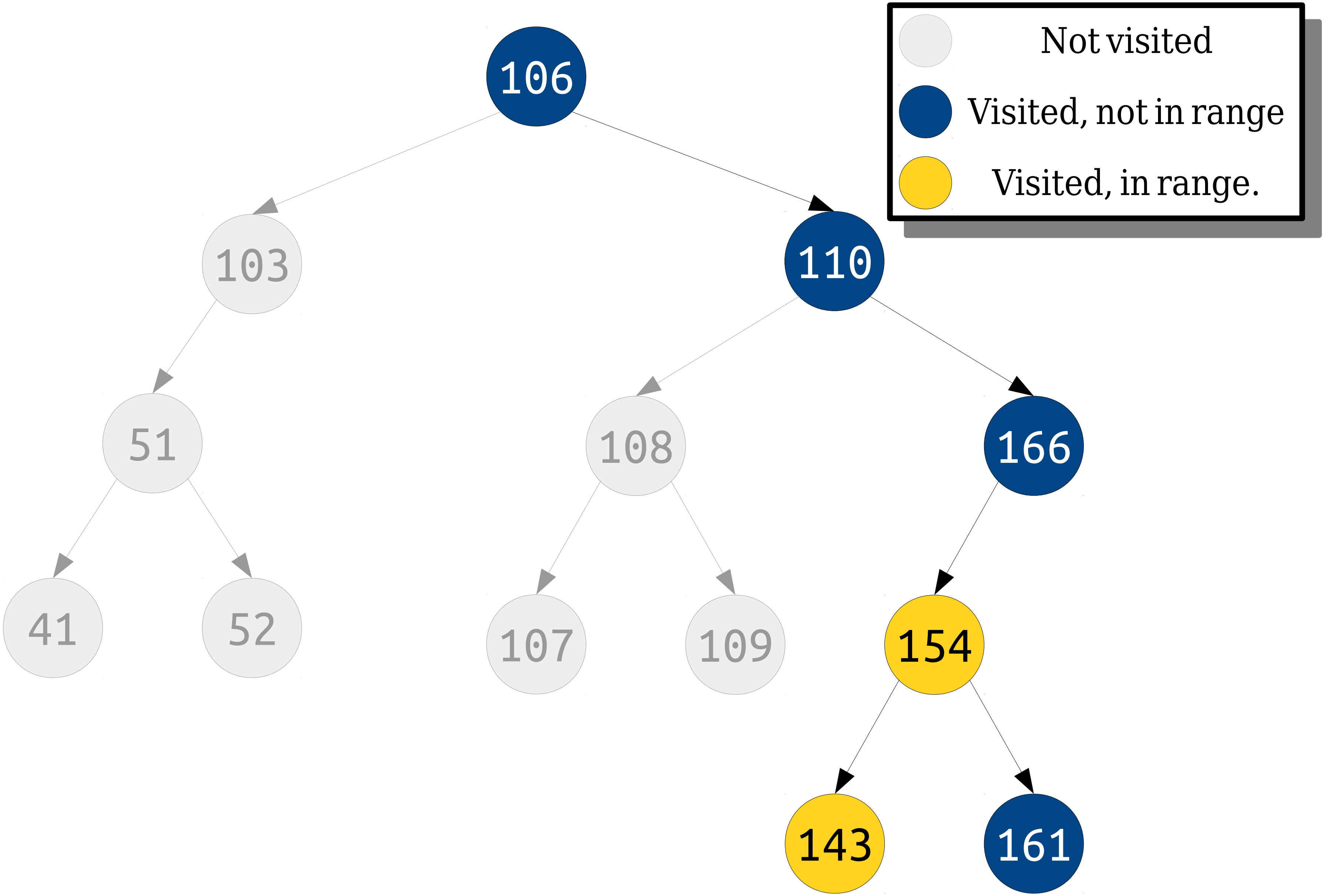
Range Searches

- A hybrid between an inorder traversal and a regular BST lookup!
- The idea:
  - If the node is in the range being searched, add it to the result.
  - Recursively explore each subtree that could potentially overlap with the range.
- ***Question to Ponder:*** Given how an inorder traversal works, how would you code this up if you wanted the values back in sorted order?

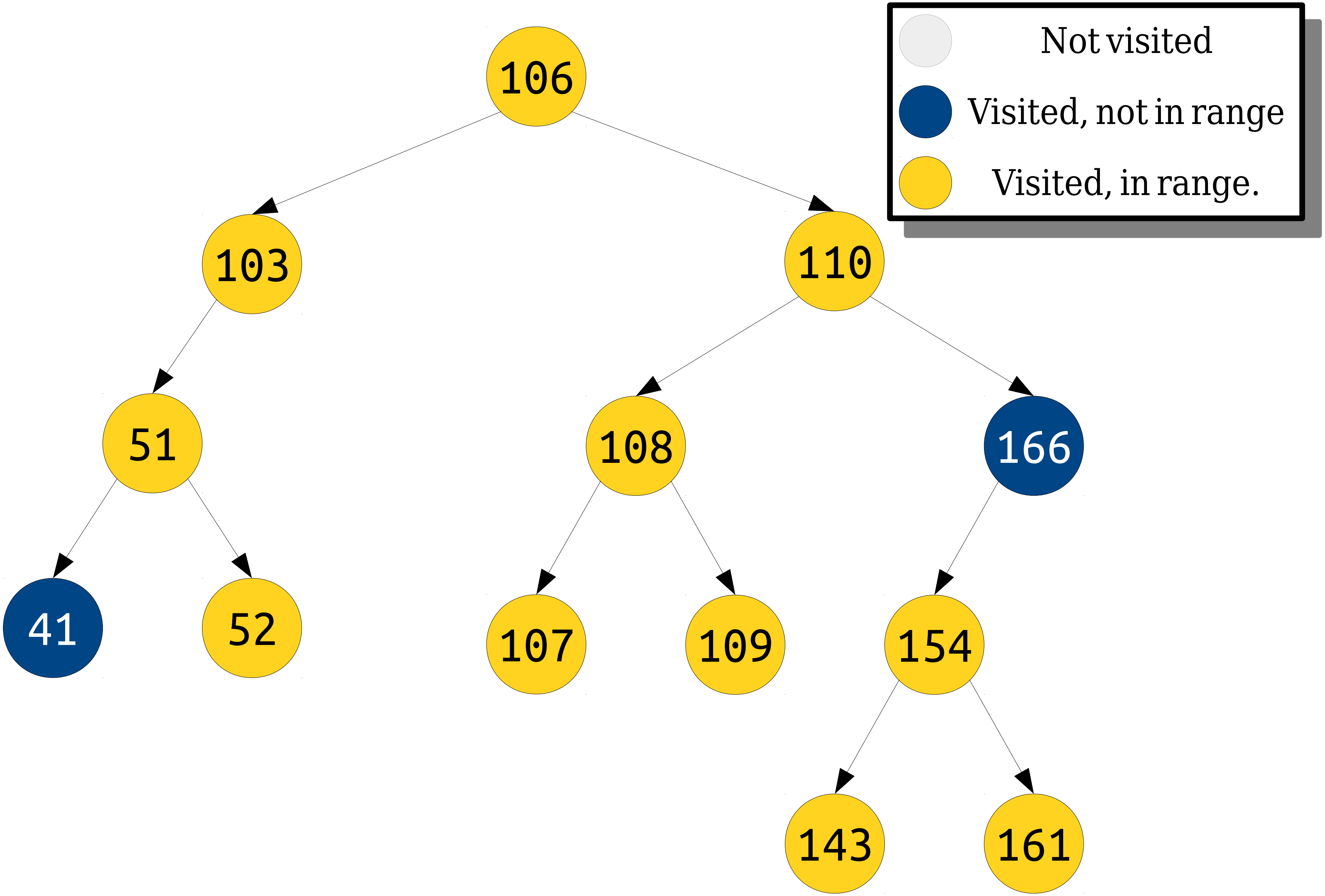
How efficient is a range search?



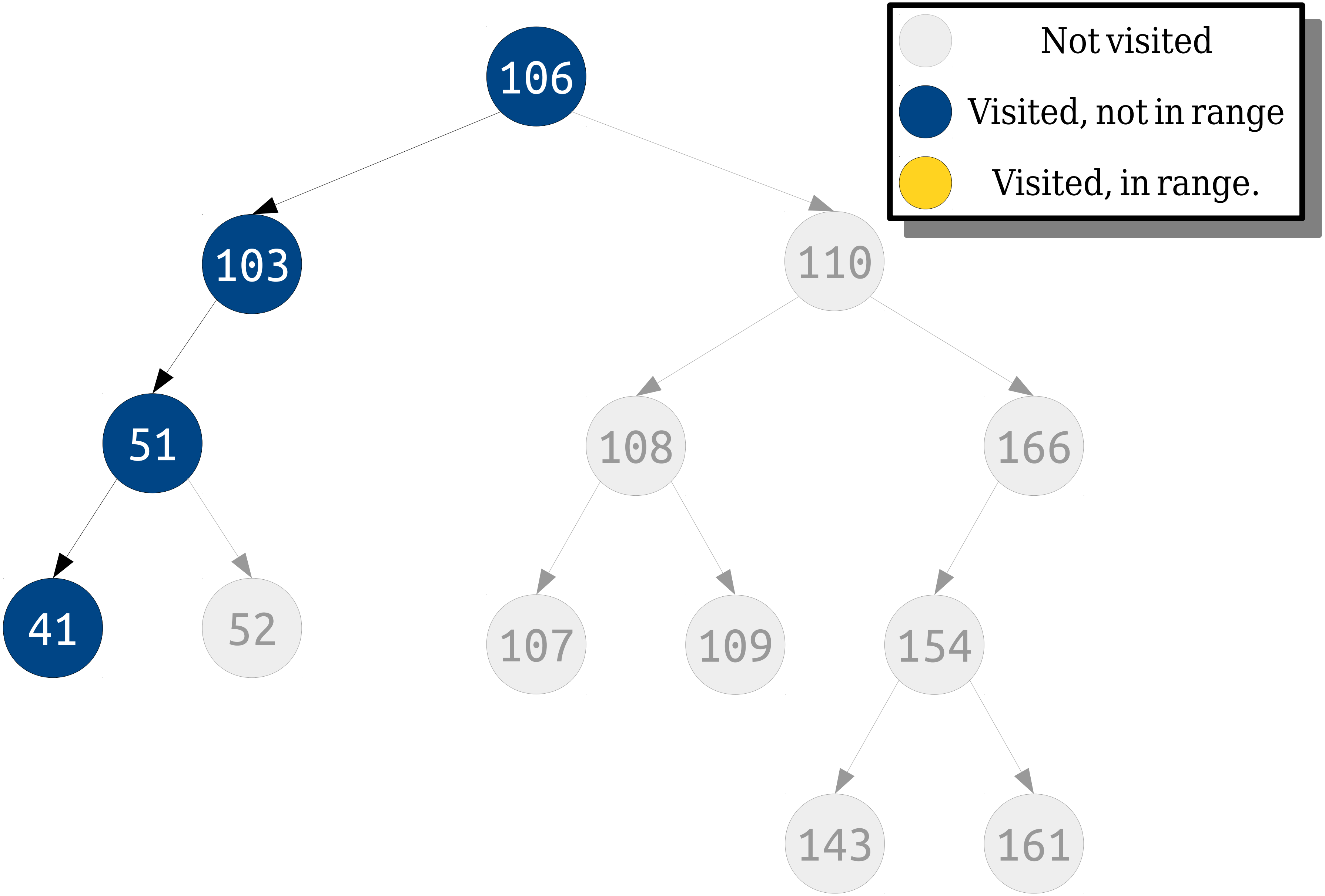
Find all elements in this tree in the range **[109, 163]**.



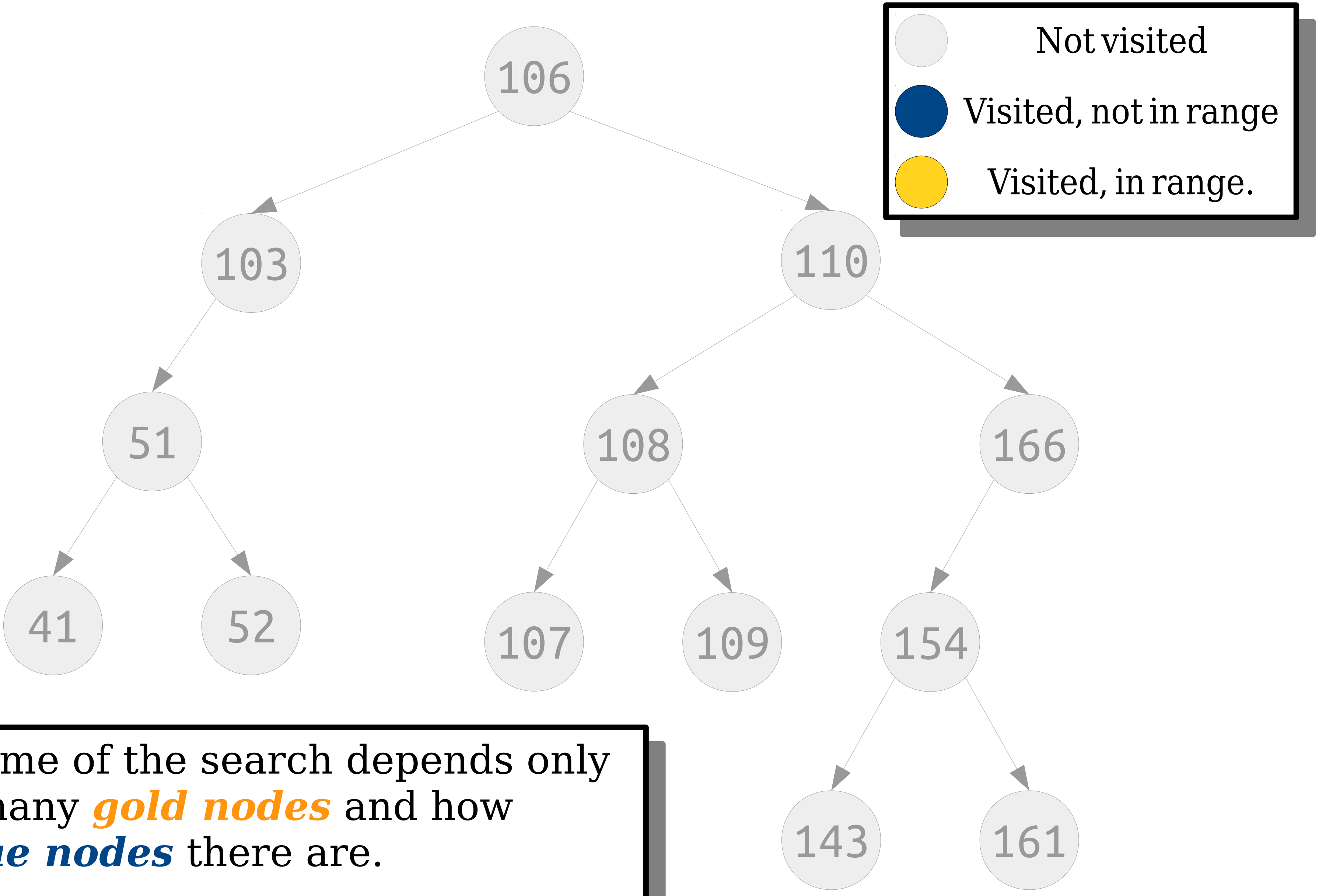
Find all elements in this tree in the range **[124, 155]**.



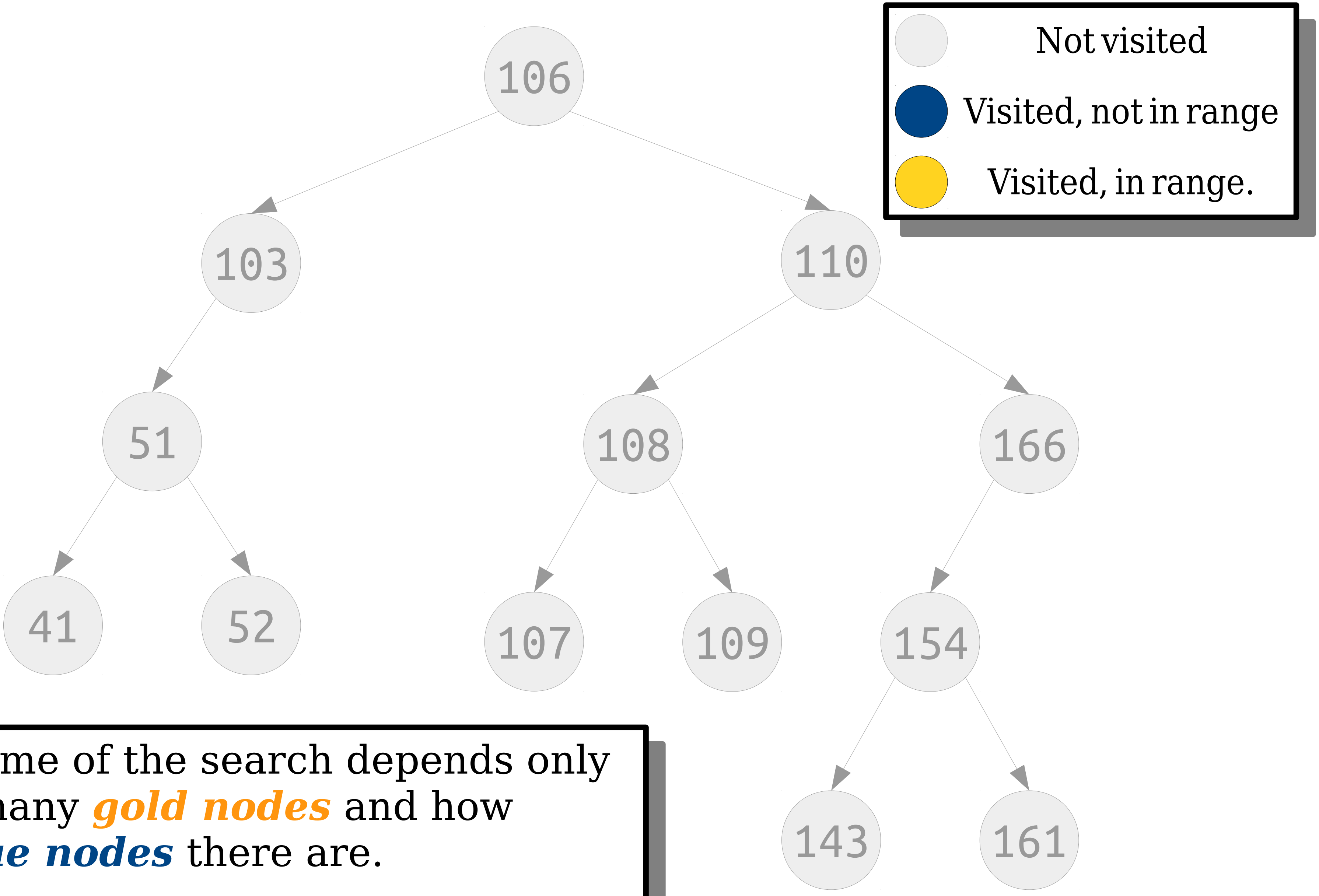
Find all elements in this tree in the range **[42, 165]**.



Find all elements in this tree in the range **[49, 50]**.

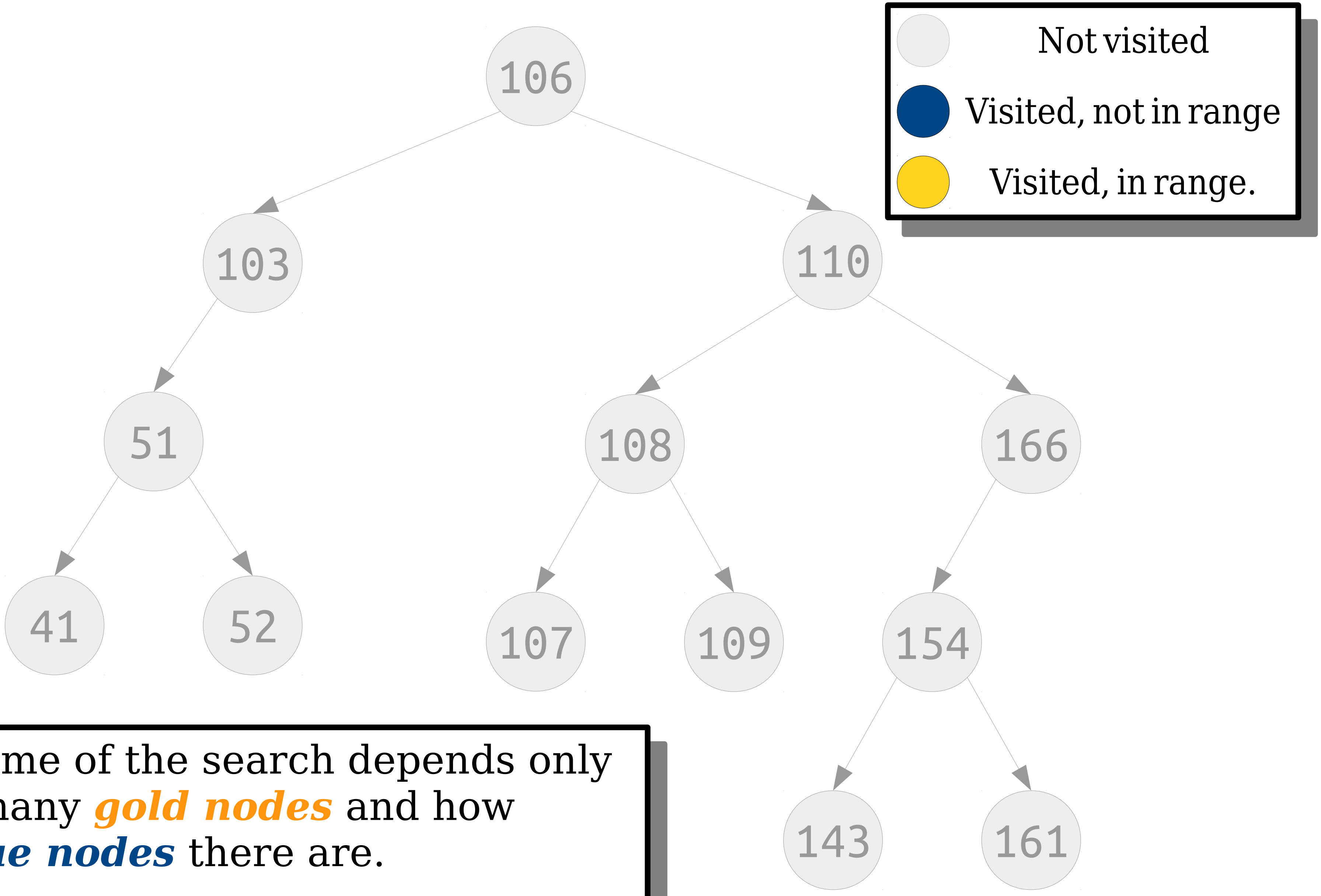


The runtime of the search depends only on how many ***gold nodes*** and how many ***blue nodes*** there are.



The runtime of the search depends only on how many ***gold nodes*** and how many ***blue nodes*** there are.

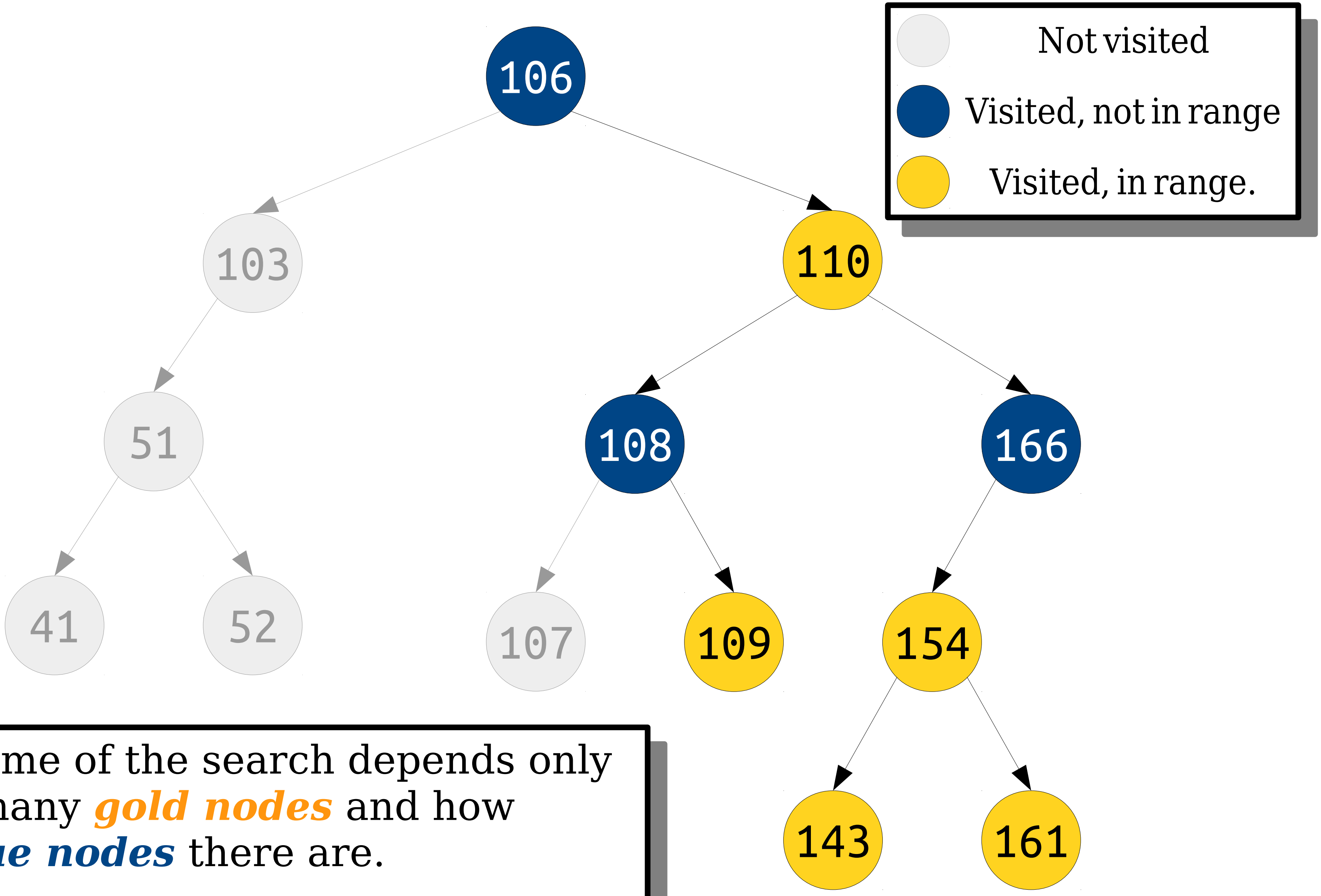
The number ***gold nodes*** is equal to the number of elements in the range.



The runtime of the search depends only on how many ***gold nodes*** and how many ***blue nodes*** there are.

The number ***gold nodes*** is equal to the number of elements in the range.

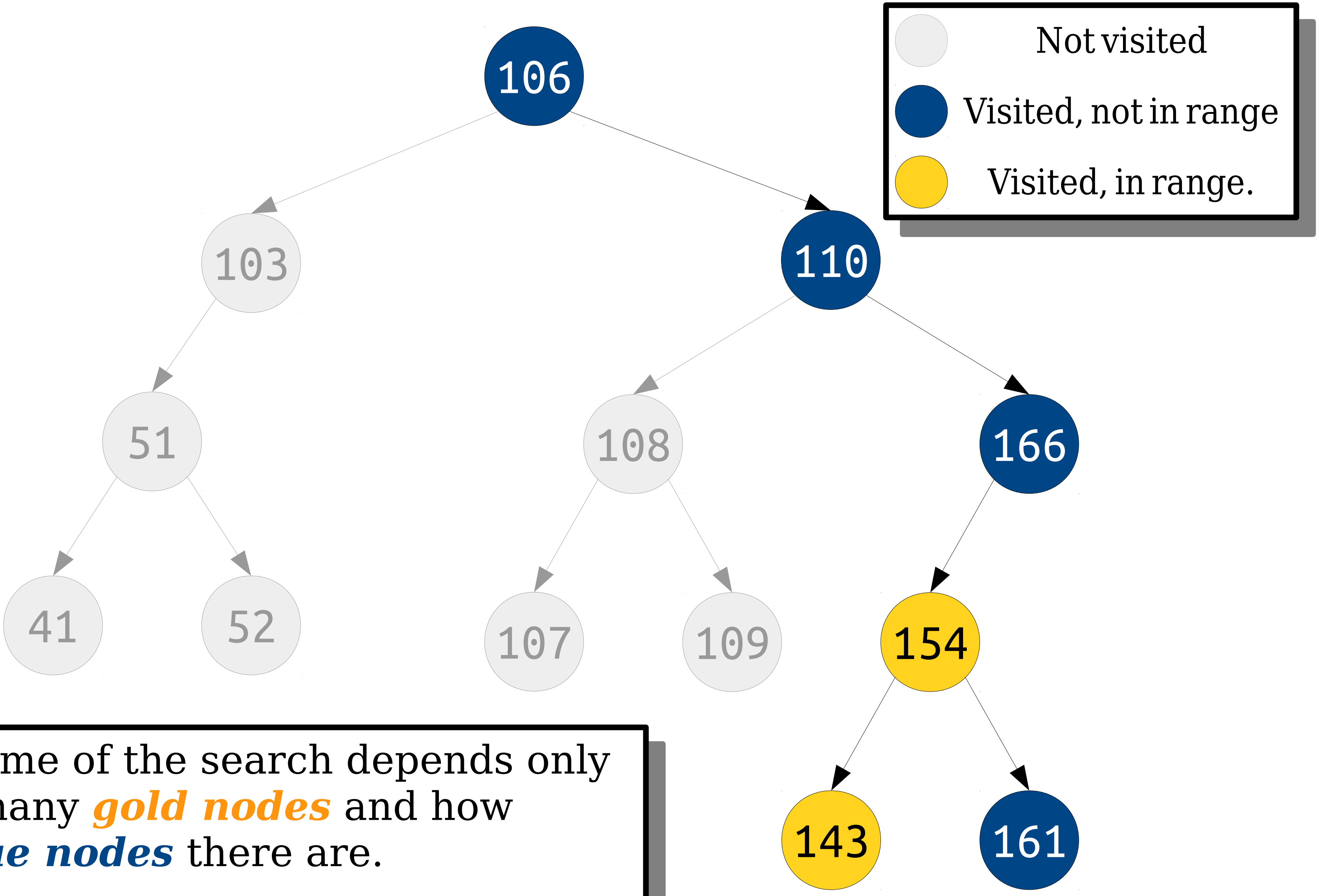
The number of ***blue nodes*** in each layer of the tree is at most two.



The runtime of the search depends only on how many ***gold nodes*** and how many ***blue nodes*** there are.

The number ***gold nodes*** is equal to the number of elements in the range.

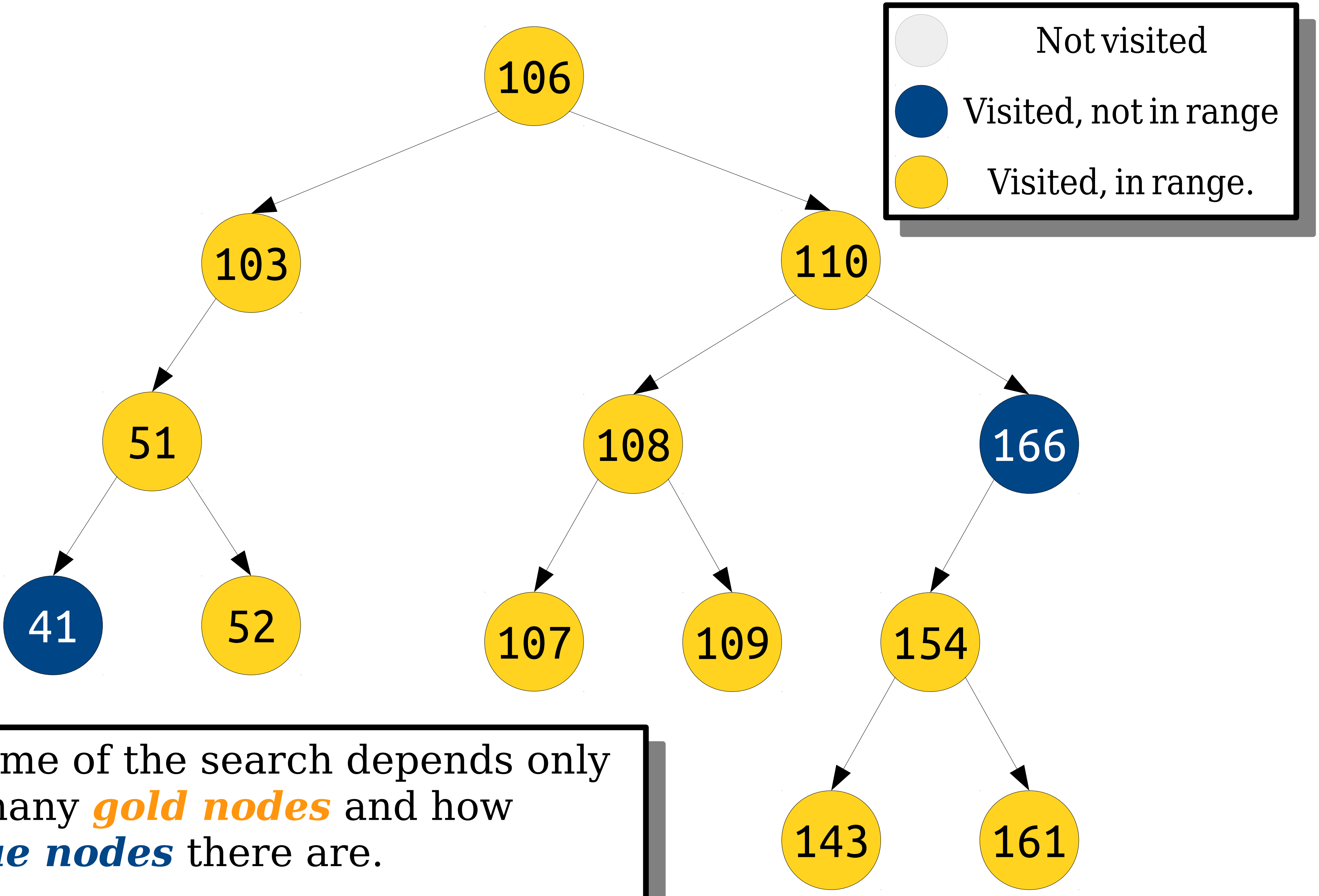
The number of ***blue nodes*** in each layer of the tree is at most two.



The runtime of the search depends only on how many ***gold nodes*** and how many ***blue nodes*** there are.

The number ***gold nodes*** is equal to the number of elements in the range.

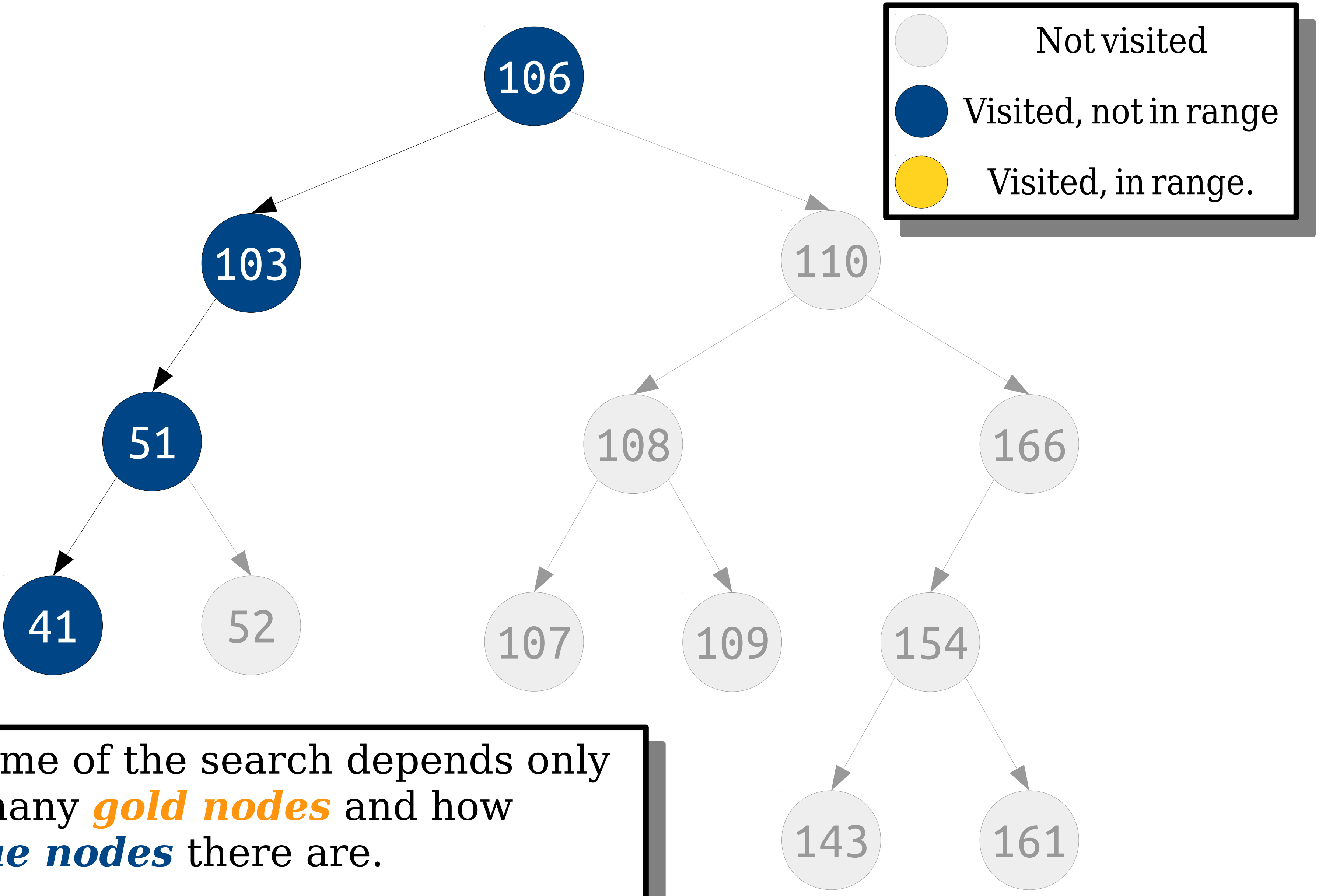
The number of ***blue nodes*** in each layer of the tree is at most two.



The runtime of the search depends only on how many ***gold nodes*** and how many ***blue nodes*** there are.

The number ***gold nodes*** is equal to the number of elements in the range.

The number of ***blue nodes*** in each layer of the tree is at most two.



The runtime of the search depends only on how many ***gold nodes*** and how many ***blue nodes*** there are.

The number ***gold nodes*** is equal to the number of elements in the range.

The number of ***blue nodes*** in each layer of the tree is at most two.

Range Searches

- The cost of a range search in a BST of height h is

  $$\mathbf{O(}\boldsymbol{h + k}\mathbf{)},$$

  where k is the number of matches reported.
- Notice that
  - if k is low (close to 0), then the runtime is close to O(h), the cost of a single search; and
  - if k is high (close to n), then the runtime is close to O(n), the cost of an inorder traversal.
- This is called an ***output-sensitive algorithm***.

Operator Overloading

Has this ever happened to you?

What’s Going On?

- Internally, the `Map` and `Set` types are implemented using binary search trees.
- BSTs assume there’s a way to compare elements against one another using the relational operators.
- But you can’t compare two `struct`s using the less-than operator!
- “There’s got to be a better way!”

Defining Comparisons

- Most programming languages provide some mechanism to let you define how to compare two objects.
- C has comparison functions, Java has the `Comparator` interface, Python has `__cmp__`, etc.
- In C++, we can use a technique called ***operator overloading*** to tell it how to compare objects using the `<` operator.

```
Doctor zhivago = /*     …     */
Doctor acula   = /*     …     */

if (zhivago < acula) {
    /*     …     */
}
```

```
Doctor zhivago = /*     …     */
Doctor acula   = /*     …     */

if (zhivago < acula) {
    /*     …     */
}
```

This function is named “**operator<**”

```
bool operator< (const Doctor& lhs, const Doctor& rhs) {
    /*      …      */
}
```

Its arguments correspond to the left-hand and right-hand operands to the < operator.

```
Doctor zhivago = /*      …      */
Doctor acula   = /*      …      */

if (zhivago < acula) {
    /*      …      */
}
```

```
bool operator< (const Doctor& lhs, const Doctor& rhs) {
    /*     …     */
}
```

```
Doctor zhivago = /*     …     */
Doctor acula   = /*     …     */

if (zhivago < acula) {
    /*     …     */
}
```

C++ treats this as

```
operator< (zhivago, acula)
```

Overloading Less-Than

- To store custom types in `Map`s or `Set`s in C++, overload the less-than operator by defining a function like this one:

```
bool operator< (const Type& lhs, const Type& rhs);
```

- This function must obey four rules:
    - It is ***consistent:*** writing $x < y$ always returns the same result given x and y.
    - It is ***irreflexive:*** $x < x$ is always false.
    - It is ***transitive:*** If $x < y$ and $y < z$, then $x < z$.
    - It has ***transitivity of incomparability:*** If neither $x < y$ nor $y < x$ are true, then x and y behave indistinguishably.
- (These rules mean that < is a ***strict weak order***; take CS103 for details!)

Overloading Less-Than

A standard technique for implementing the less-than operator is to use a ***lexicographical comparison***, which looks like this:

```
bool operator< (const Type& lhs, const Type& rhs) {
    if (lhs.field1 != rhs.field1) {
        return lhs.field1 < rhs.field1;
    } else if (lhs.field2 != rhs.field2) {
        return lhs.field2 < rhs.field2;
    } else if (lhs.field3 != rhs.field3) {
        return lhs.field3 < rhs.field3;
    } … {
      …
    } else {
        return lhs.fieldN < rhs.fieldN;
    }
}
```

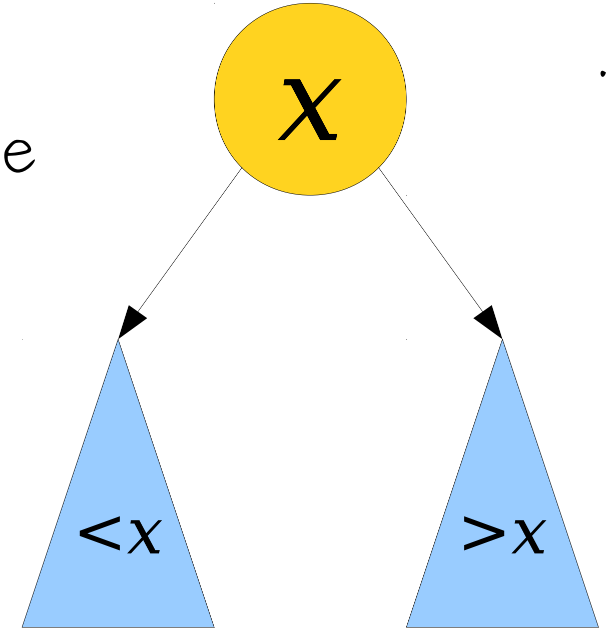
To Summarize:

A Binary Search Tree Is Either…

an empty tree,
represented by
nullptr, or…

⊘

… a single node,
whose left subtree
is a BST of
smaller values …

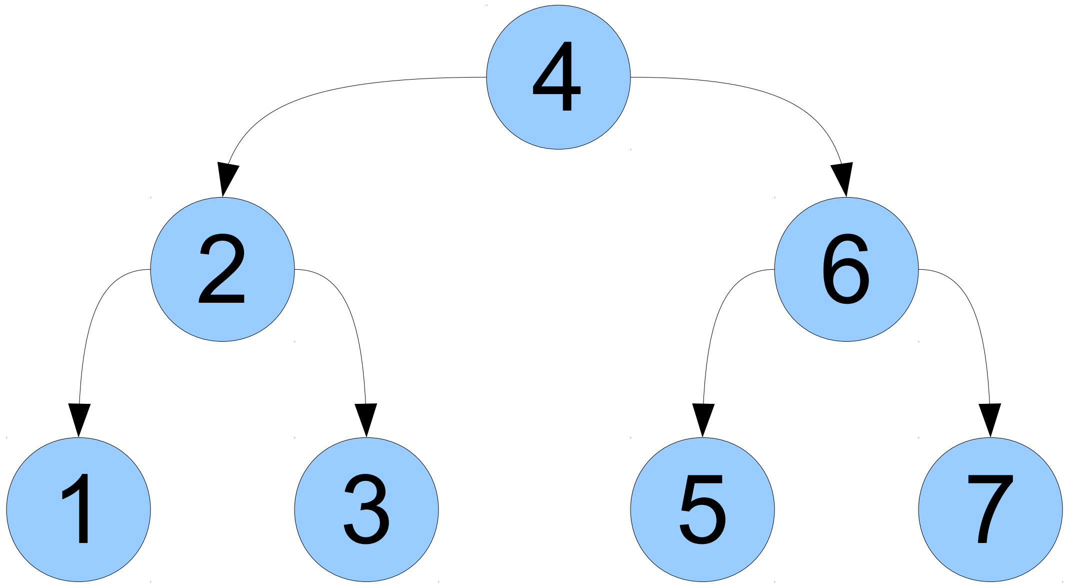


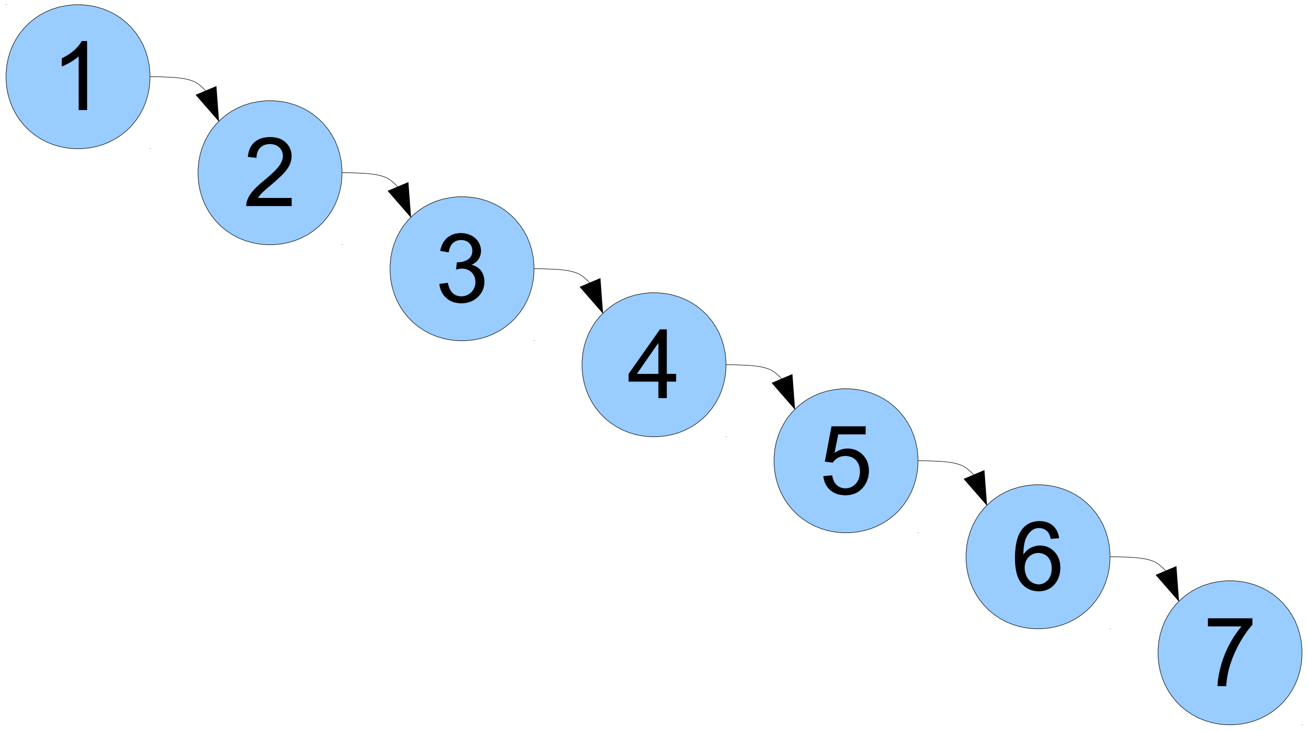
… and whose right
subtree is a BST
of larger values.

```
struct Node {
    int value;
    Node* left;  // Smaller values
    Node* right; // Bigger values
};
```

```
bool contains(Node* root, const string& key) {
    if (root == nullptr) return false;
    else if (key == root->value) return true;
    else if (key <  root->value) return contains(root->left,  key);
    else return contains(root->right, key);
}

void insert(Node*& root, const string& key) {
    if (root == nullptr) {
        root = new Node;
        node->value = key;
        node->left = node->right = nullptr;
    } else if (key < root->value) {
        insert(root→left, key);
    } else if (key > root->value) {
        insert(root->right, key);
    } else {
        // Already here!
    }
}
```


4
2
6
1
3
5
7


1
2
3
4
5
6
7

```
void printTree(Node* root) {
    if (root == nullptr) return;

    printTree(root->left);
    cout << root->value << endl;
    printTree(root->right);
}

void freeTree(Node* root) {
    if (root == nullptr) return;

    freeTree(root->left);
    freeTree(root->right);
    delete root;
}
```

```
bool operator< (const Type& lhs, const Type& rhs) {
    if (lhs.field1 != rhs.field1) {
        return lhs.field1 < rhs.field1;
    } else if (lhs.field2 != rhs.field2) {
        return lhs.field2 < rhs.field2;
    } else if (lhs.field3 != rhs.field3) {
        return lhs.field3 < rhs.field3;
    } … {
      …
    } else {
        return lhs.fieldN < rhs.fieldN;
    }
}
```

Next Time

- ***Tries***
  - How is the Lexicon implemented?
- ***More on Trees***
  - Where else are they used?