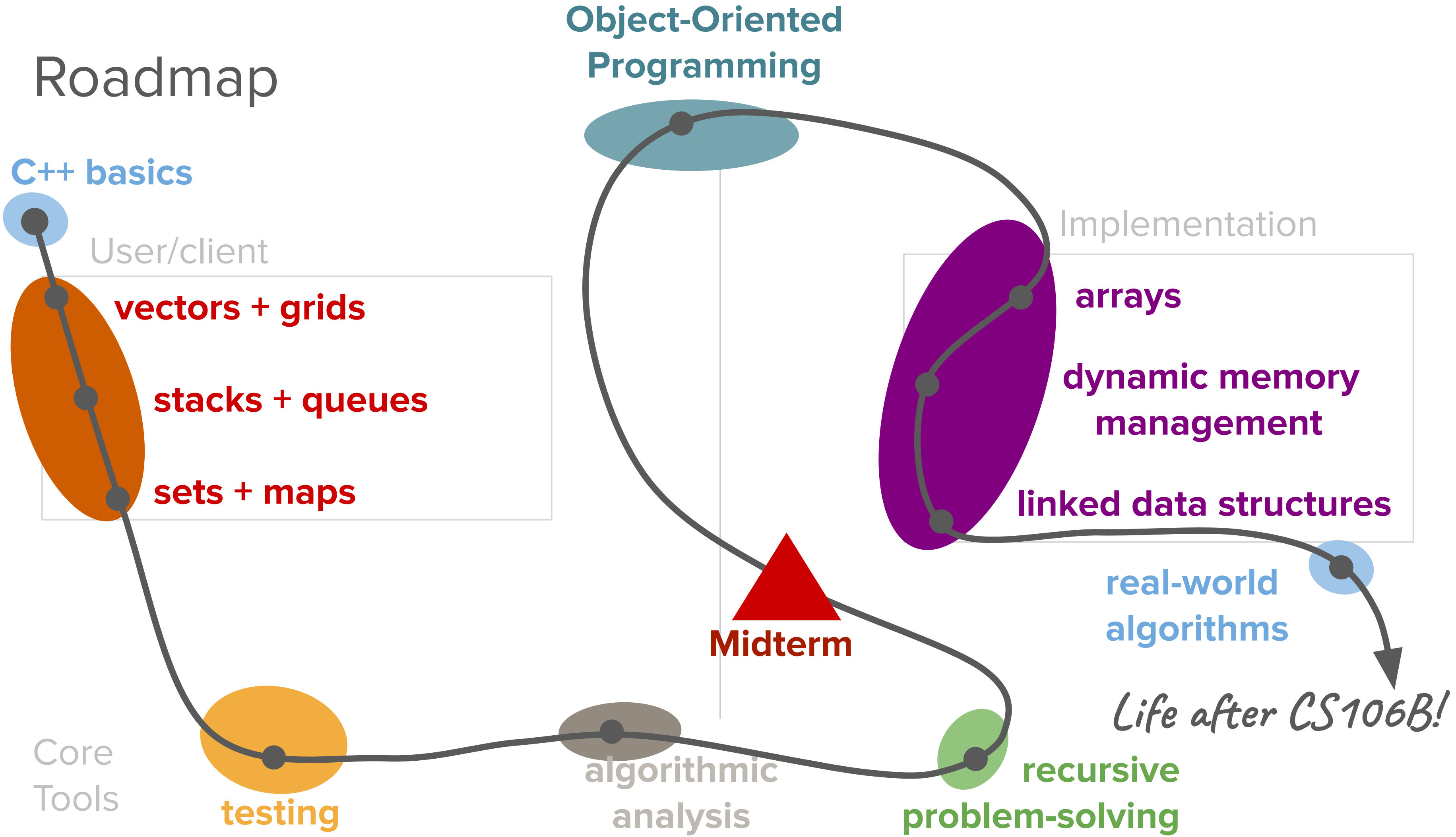
Binary Search Trees

What is your favorite type of tree?

(e.g. oak, redwood, maple, etc.)

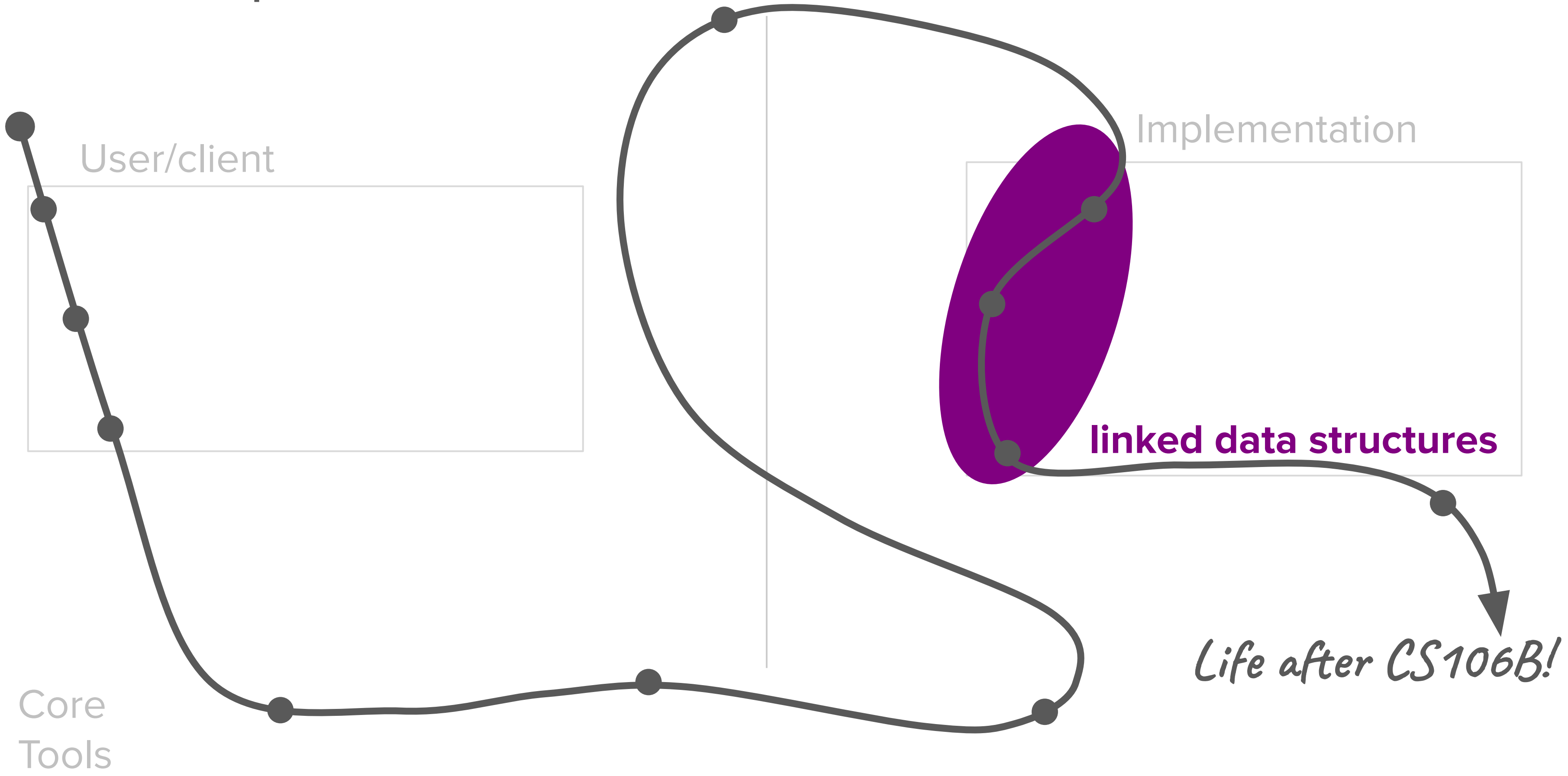
pollev.com/cs106bpolls

Roadmap



C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Core Tools
testing
algorithmic analysis
recursive problem-solving
Object-Oriented Programming
Midterm
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Life after CS106B!

Roadmap


User/client
Implementation
linked data structures
Life after CS106B!
Core
Tools

Today's question

How can we take advantage of trees to structure and efficiently manipulate data?

Today's topics

1. What is a binary search tree (BST)?
2. Building efficient BSTs
3. Implementing Sets with BSTs

Review

[trees]

Definition

tree

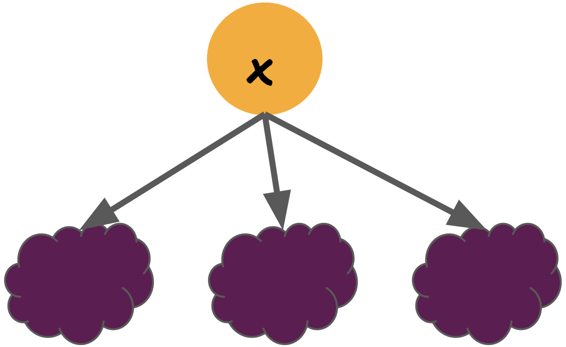
A tree is hierarchical data organization structure composed of a root value linked to zero or more non-empty subtrees.

What is a tree?

A tree is either...

An empty data structure, or...

A single node (parent), with zero or more non-empty subtrees (children)



Tree terminology

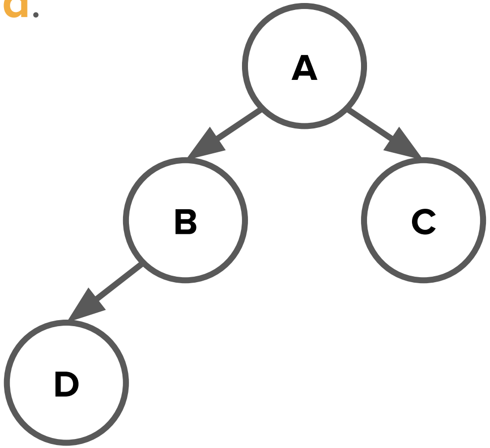
- Types of nodes
  - The **root** node defines the "top" of the tree.
  - Every node has 0 or more **children** nodes descended from it.
  - Nodes with no children are called **leaf nodes**.
  - Every node in a tree has exactly one **parent** node (except for the root node).

- Terminology for quantifying trees
  - A **path** *between two nodes* traverses edges between parents and their children, and **length** *of a path* is the number of edges between the two nodes.
  - The **depth** *of a node* is the length of the path (# of edges) between the root and that node.
  - The **height** *of a tree* is the number of nodes in the longest path through the tree (i.e. the number of **levels** in the tree).

Binary trees

- A **binary tree** is a tree where every node has either 0, 1, or 2 children. No node in a binary tree can have more than 2 children.

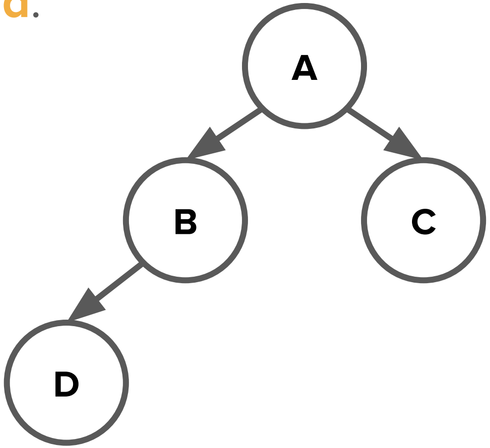
- Typically, the two children of a node in a binary tree are referred to as the **left child** and the **right child**.



Binary trees

- A **binary tree** is a tree where every node has either 0, 1, or 2 children. No node in a binary tree can have more than 2 children.

- Typically, the two children of a node in a binary tree are referred to as the **left child** and the **right child**.



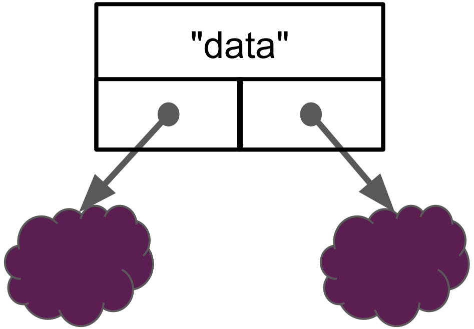

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```

What is a tree in C++?

A tree is either...

An empty tree represented by **`nullptr`**, or...

A single **`TreeNode`**, with 0, 1, or 2 non-null pointers to other **`TreeNodes`**



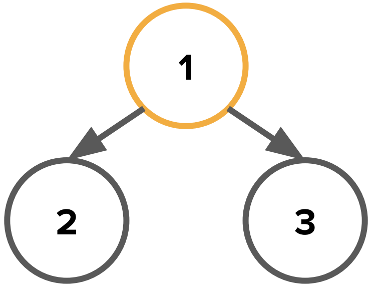
Building a tree

- Building a tree is very similar to the process of building a linked list.
- We create new nodes of the tree by dynamically allocating memory.
- We start by first creating the leaf nodes and then creating their parents.
- We integrate the parents into the tree by rewiring their **`left`** and **`right`** pointers to the already-created children.

Traversing a tree - recursively!

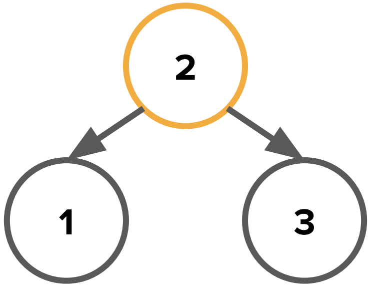
Pre-order

1. **"Do something" with the current node**
2. Traverse the left subtree
3. Traverse the right subtree



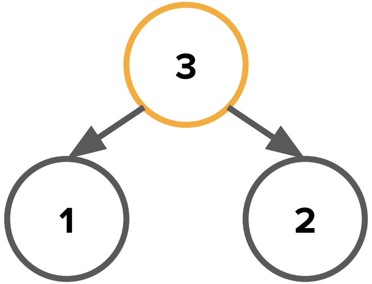
In-order

1. Traverse the left subtree
2. **"Do something" with the current node**
3. Traverse the right subtree



Post-order

1. Traverse the left subtree
2. Traverse the right subtree
3. **"Do something" with the current node**

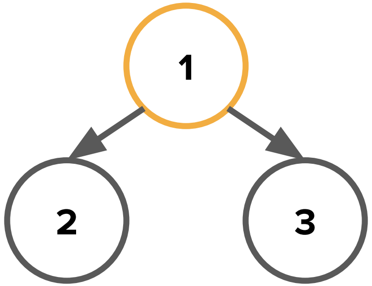


> *Poll:* pollev.com/cs106bpolls
>
> Which type of traversal should you use to free a tree?

Traversing a tree - recursively!

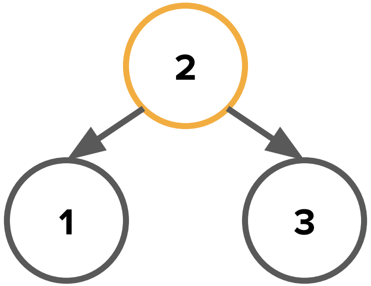
Pre-order

1. **"Do something" with the current node**
2. Traverse the left subtree
3. Traverse the right subtree



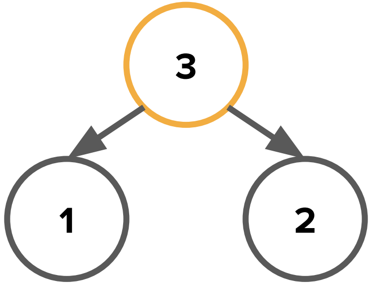
In-order

1. Traverse the left subtree
2. **"Do something" with the current node**
3. Traverse the right subtree



Post-order

1. Traverse the left subtree
2. Traverse the right subtree
3. **"Do something" with the current node**



Let’s code it:

Freeing a tree!

Key Idea: The distance from each element (node) in a tree to the top of the tree (the root) is small, even if there are many elements.

Key Idea: The distance from each element (node) in a tree to the top of the tree (the root) is small, even if there are many elements.

How can we take advantage of trees to structure and efficiently manipulate data?

Revisiting our levels of abstraction…

Levels of abstraction

What is the interface for the user?

How is our data organized?
(binary heaps, BSTs, Huffman trees)

What stores our data?
(arrays, linked lists, **trees**)

How is data represented electronically?
(RAM)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Computer Hardware

Levels of abstraction

What is the interface for the user?

How is our data organized?
(binary heaps, **BSTs**, Huffman trees)

What stores our data?
(arrays, linked lists, **trees**)

How is data represented electronically?
(RAM)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Computer Hardware

ADT Big-O Matrix

- Vectors
  - `.size() – O(1)`
  - `.add() – O(1)`
  - `v[i] – O(1)`
  - `.insert() – O(n)`
  - `.remove() – O(n)`
  - `.clear() - O(n)`
  - `traversal – O(n)`
- Grids
  - `.numRows()/.numCols() – O(1)`
  - `g[i][j] – O(1)`
  - `.inBounds() – O(1)`
  - `traversal` – $O(n^2)$
- Queues
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.enqueue() – O(1)`
  - `.dequeue() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Stacks
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.push() – O(1)`
  - `.pop() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Sets
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `.add() – O(log(n))`
  - `.remove() – O(log(n))`
  - `.contains() – O(log(n))`
  - `traversal – O(n)`
- Maps
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `m[key] – O(log(n))`
  - `.contains() – O(log(n))`
  - `traversal – O(n)`

ADT Big-O Matrix

- Vectors
  - .size() – O(1)
  - .add() – O(1)
  - v[i] – O(1)
  - .insert() – O(n)
  - .remove() – O(n)
  - .clear() - O(n)
  - traversal – O(n)
- Grids
  - .numRows()/.numCols() – O(1)
  - g[i][j] – O(1)
  - .inBounds() – O(1)
  - traversal – $O(n^2)$
- Queues
  - .size() – O(1)
  - .peek() – O(1)
  - .enqueue() – O(1)
  - .dequeue() – O(1)
  - .isEmpty() – O(1)
  - traversal – O(n)
- Stacks
  - .size() – O(1)
  - .peek() – O(1)
  - .push() – O(1)
  - .pop() – O(1)
  - .isEmpty() – O(1)
  - traversal – O(n)
- Sets
  - .size() – O(1)
  - .isEmpty() – O(1)
  - .add() – O(log(n))
  - .remove() – O(log(n))
  - .contains() – O(log(n))
  - traversal – O(n)
- Maps
  - .size() – O(1)
  - .isEmpty() – O(1)
  - m[key] – O(log(n))
  - .contains() – O(log(n))
  - traversal – O(n)

Levels of abstraction

What is the interface for the user?
(**Sets**, Maps, etc.)

How is our data organized?
(binary heaps, **BSTs**, Huffman trees)

What stores our data?
(arrays, linked lists, **trees**)

How is data represented electronically?
(RAM)

Abstract Data Structures

Data Organization Strategies

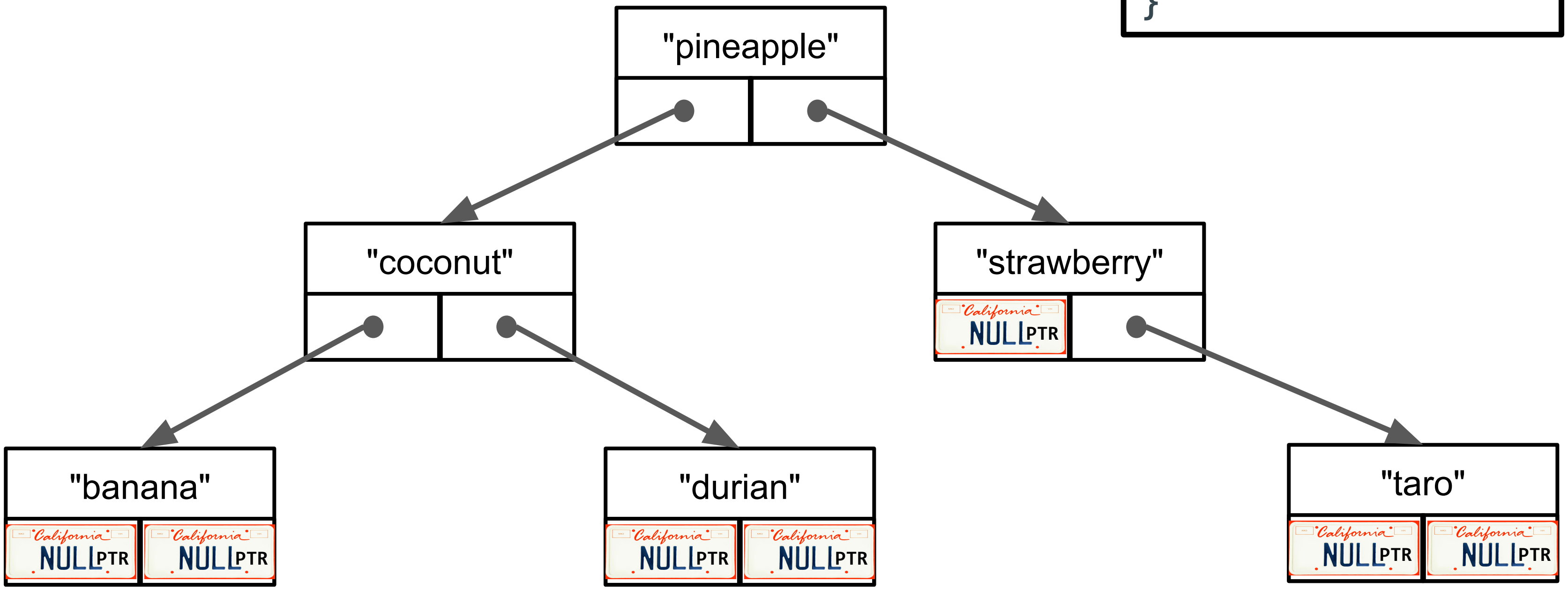
Fundamental C++ Data Storage

Computer Hardware

What is a binary search tree (BST)?

Building Trees Programmatically

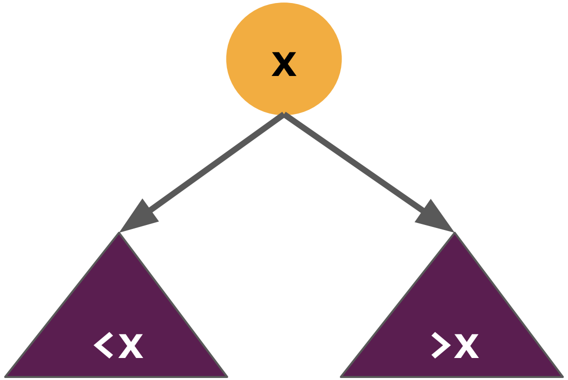
```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```



A binary search tree is either...

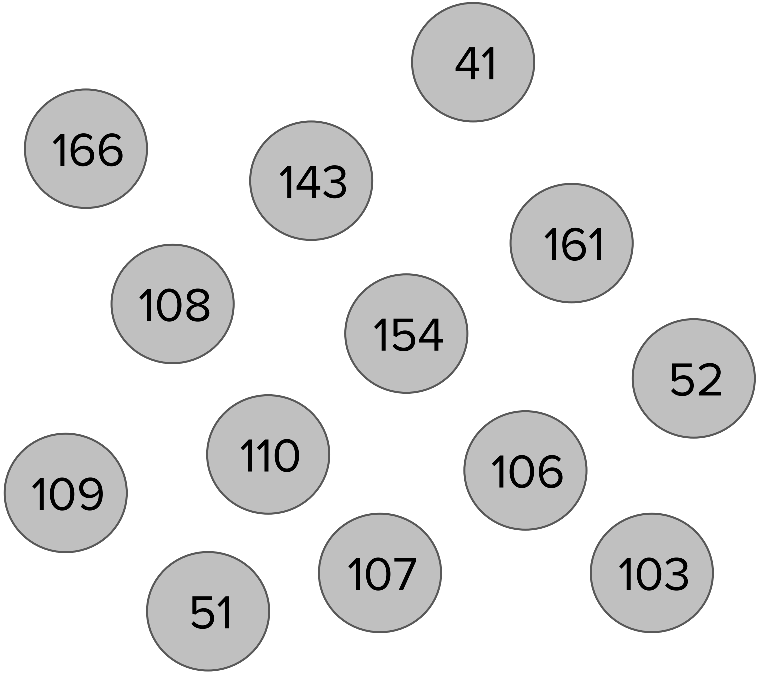
an empty data structure represented by nullptr or...

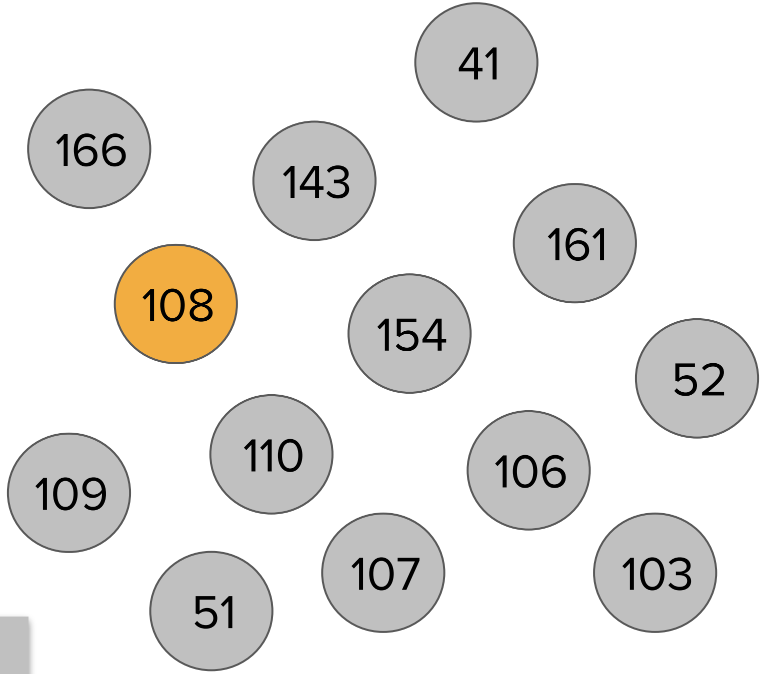
a single node, whose left subtree is a BST of smaller values than **x**...



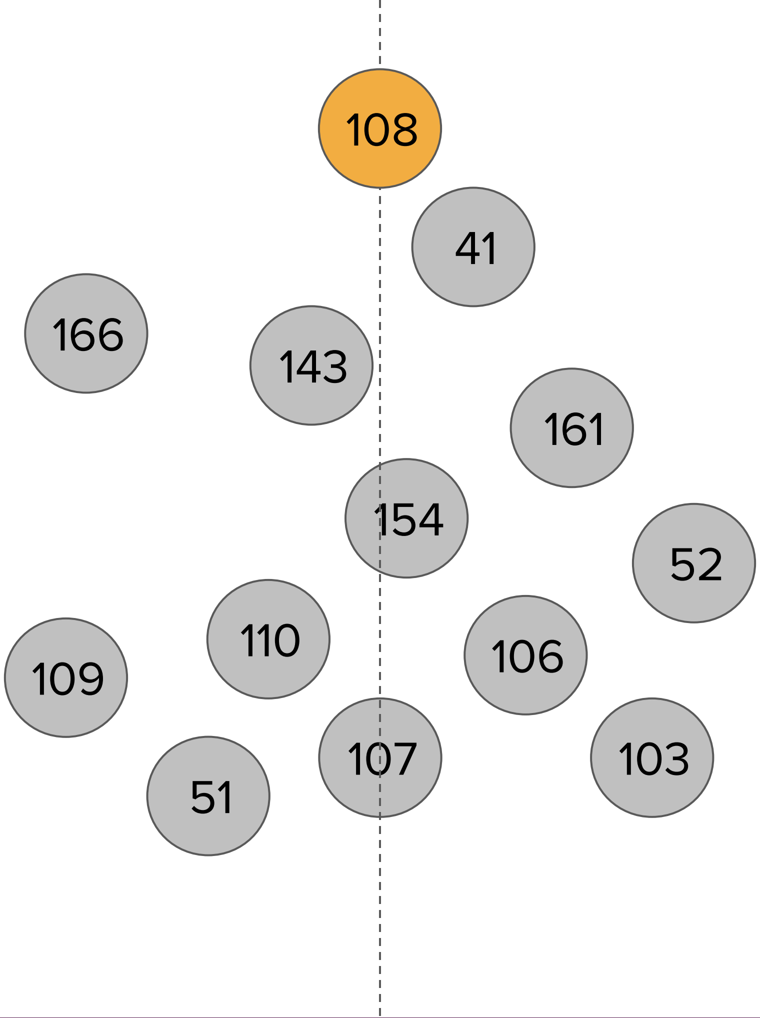
and whose right subtree is a BST of larger values than **x**.

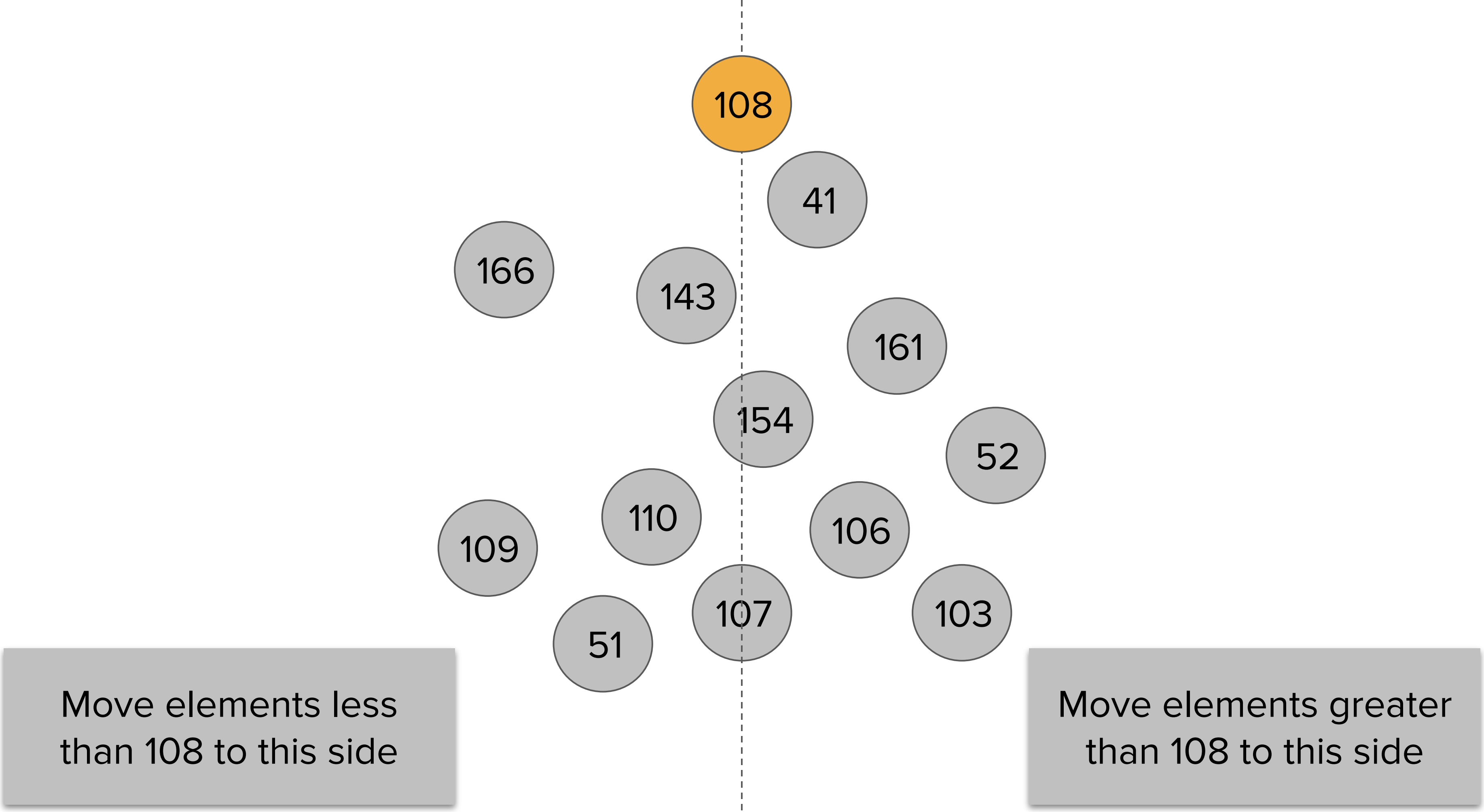
Building a BST

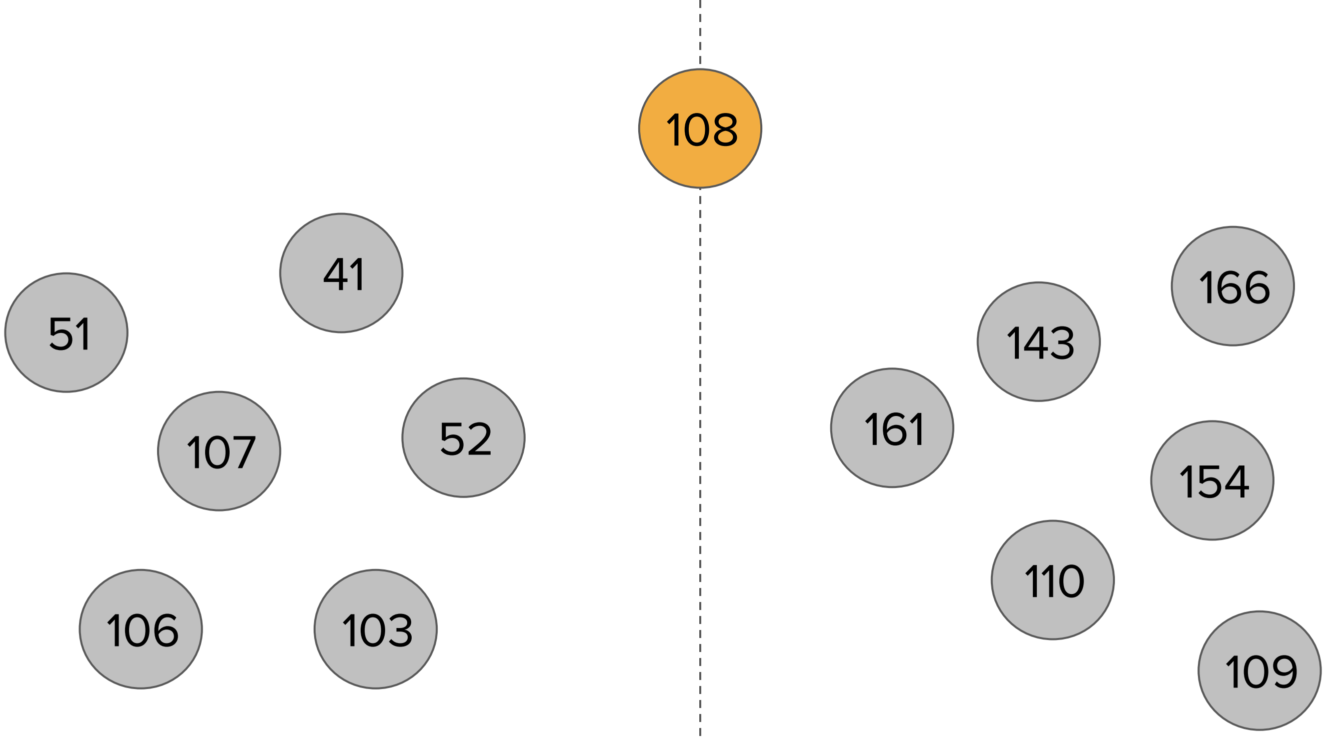

41
166
143
161
108
154
52
110
106
109
107
103
51



Pick the median element

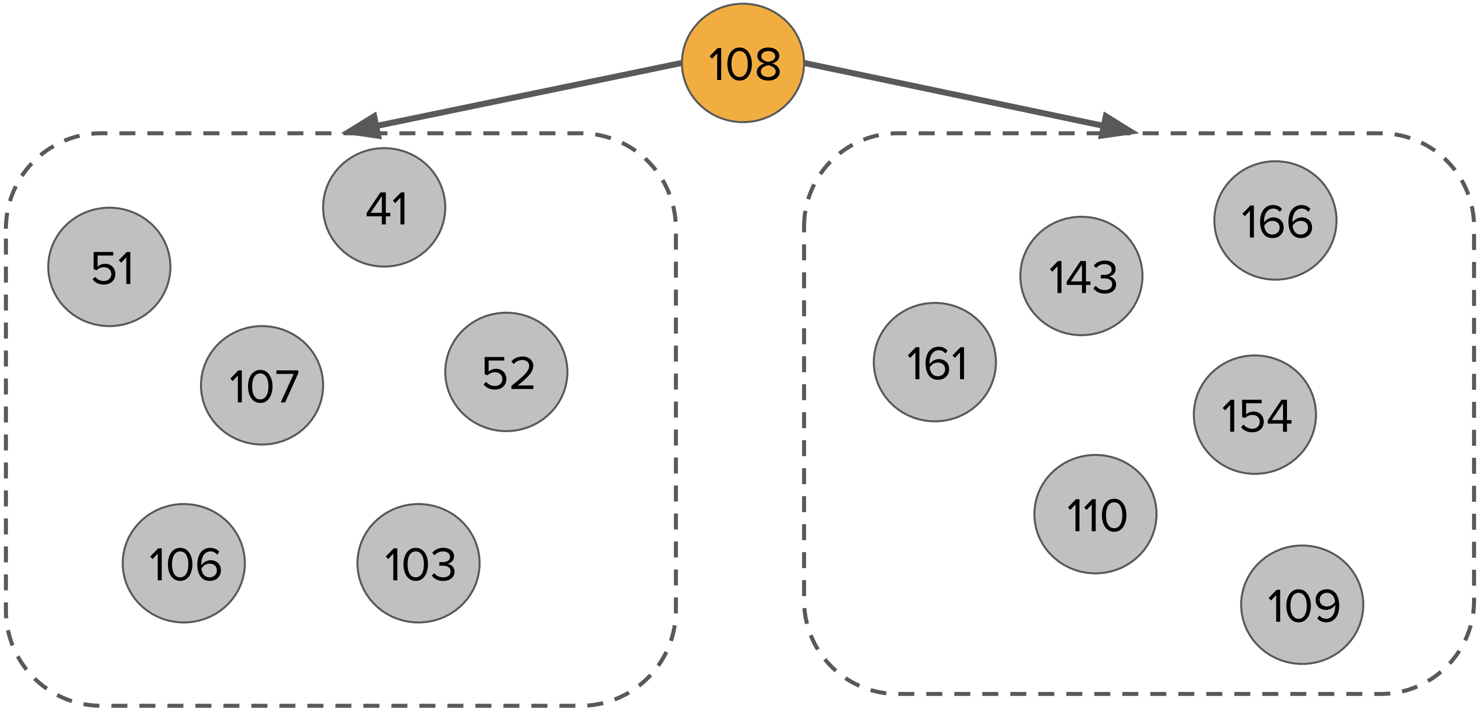

108
41
166
143
161
154
52
110
106
109
107
103
51

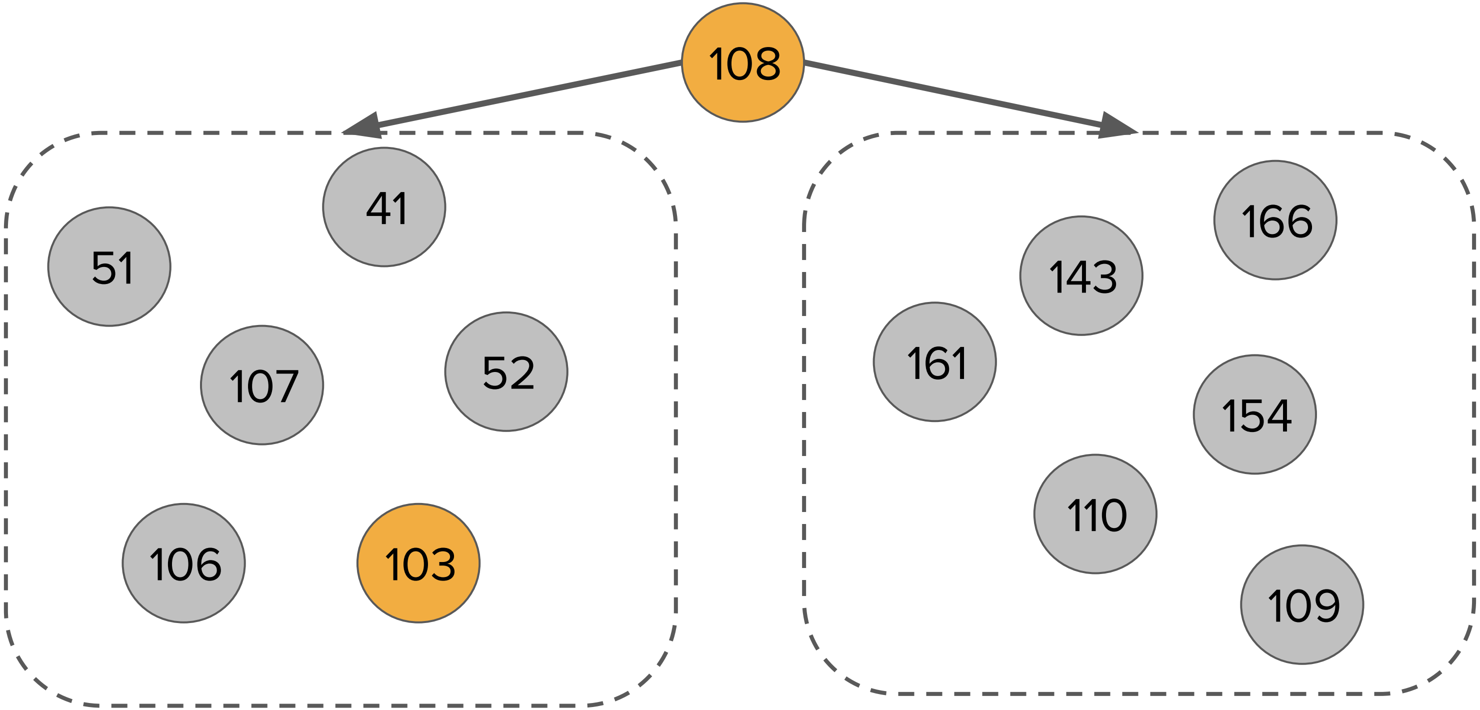

108
41
166
143
161
154
52
110
106
109
107
103
51
Move elements less than 108 to this side
Move elements greater than 108 to this side



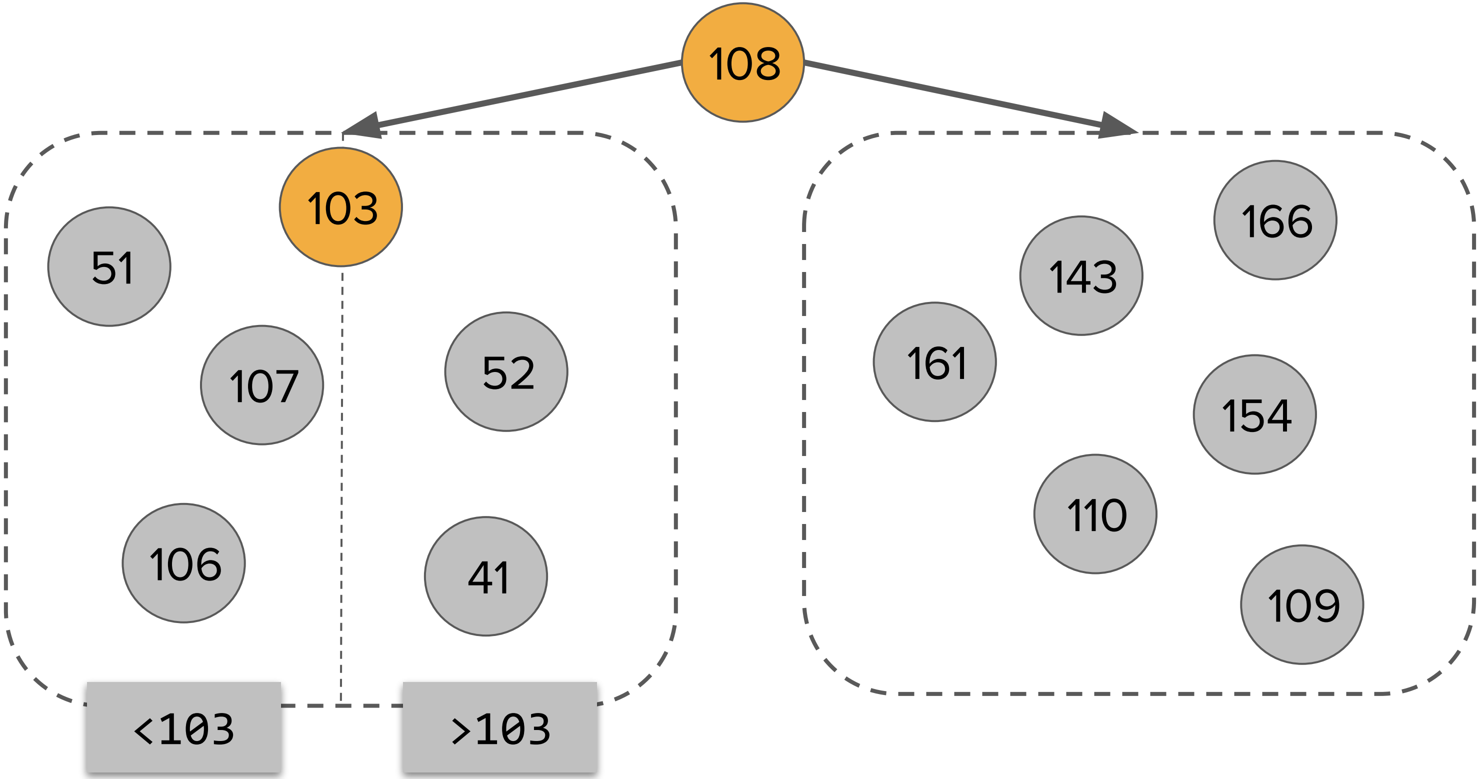
Move elements less than 108 to this side

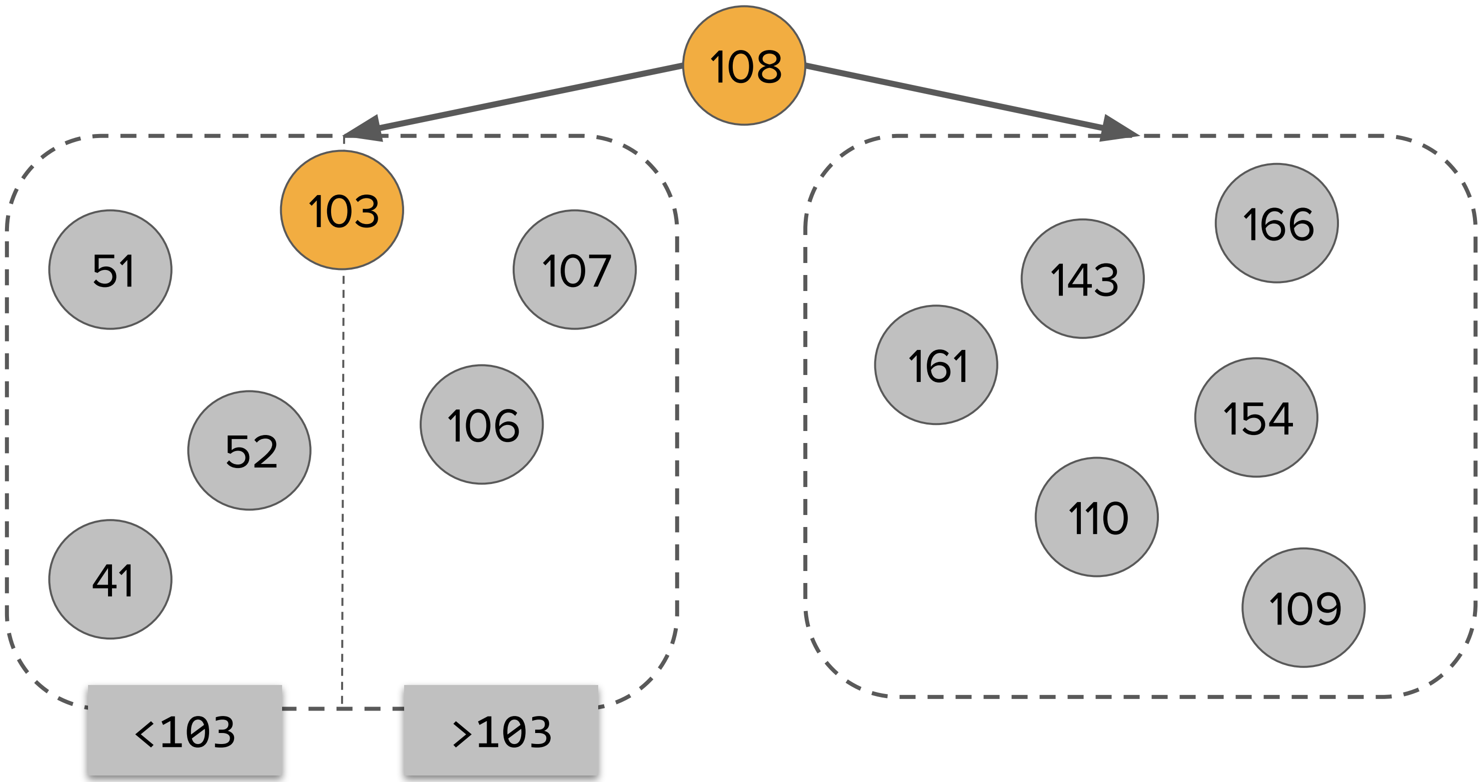
Move elements greater than 108 to this side

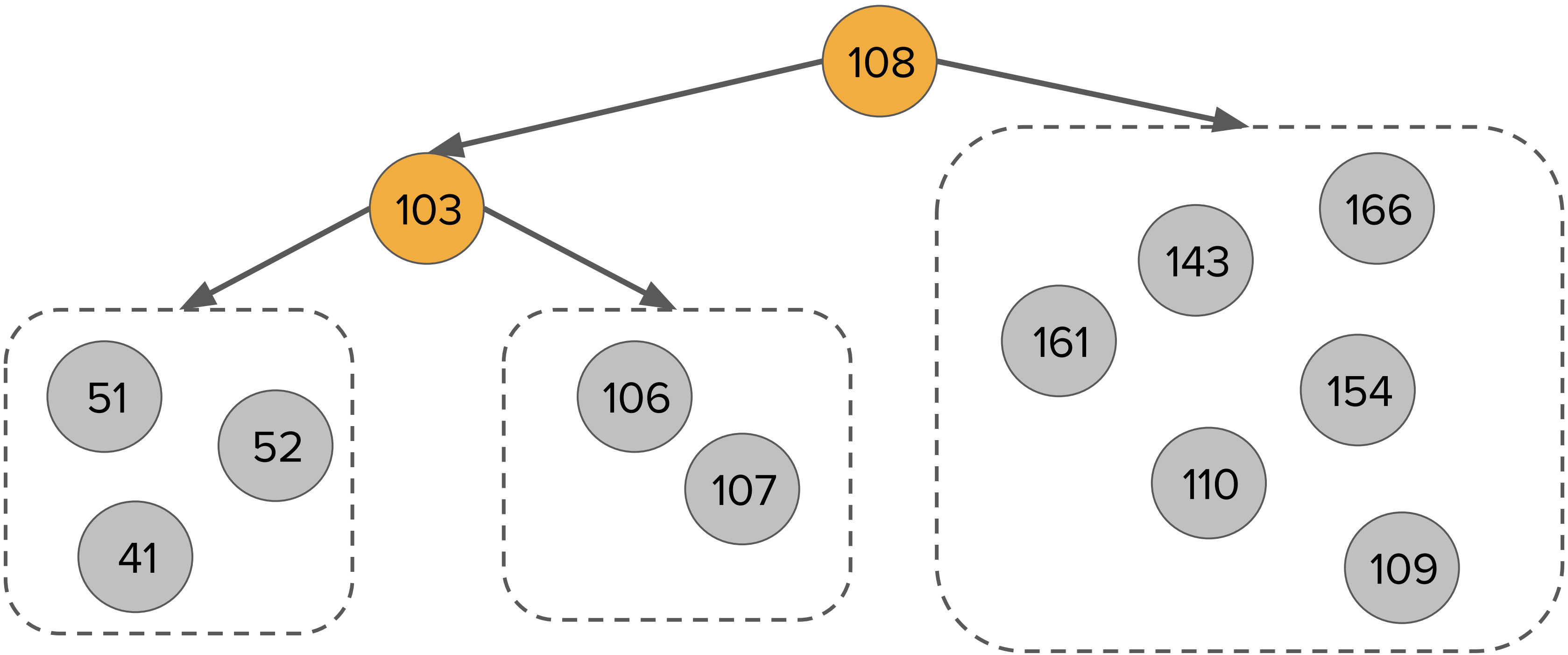

108
41
51
107
52
106
103
166
143
161
154
110
109

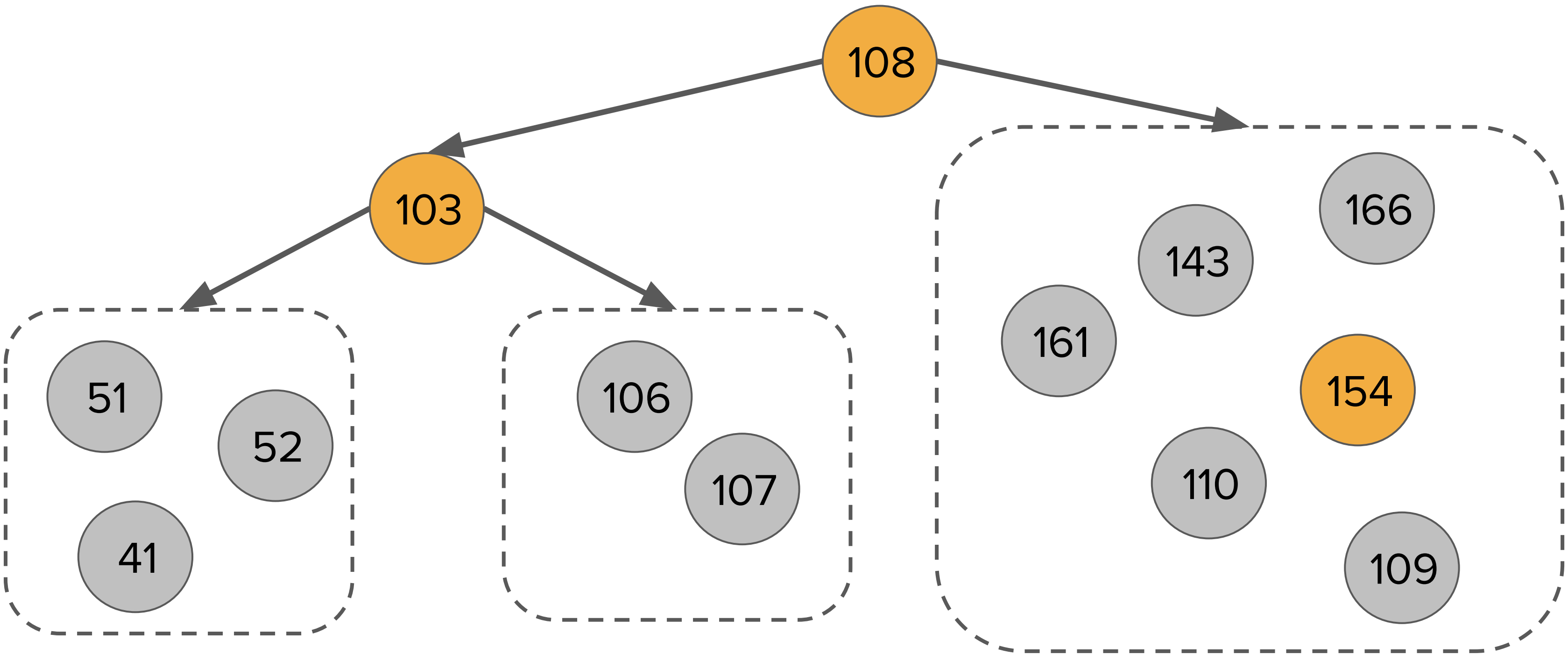


Pick the median element of the left side

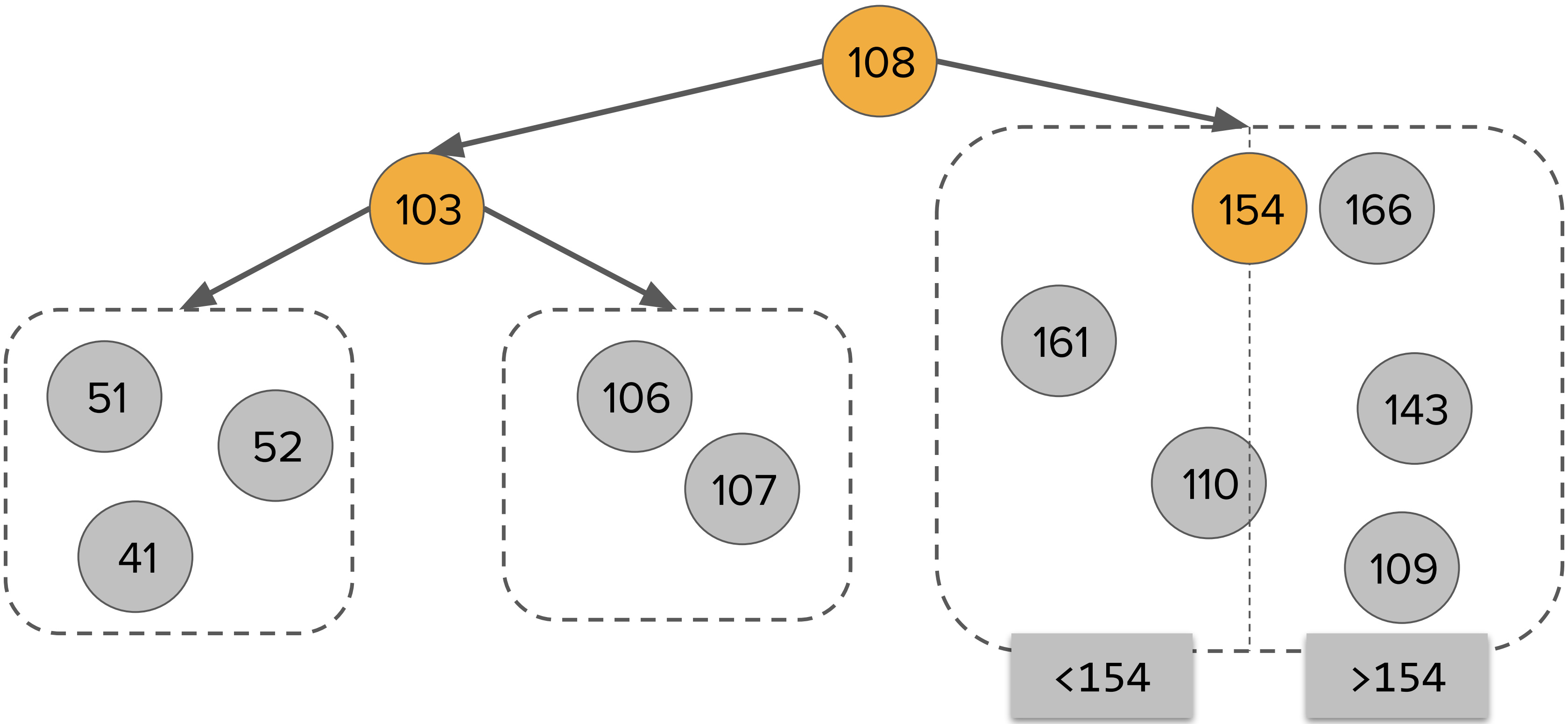

108
103
51
107
106
52
41
<103
>103
166
143
161
154
110
109

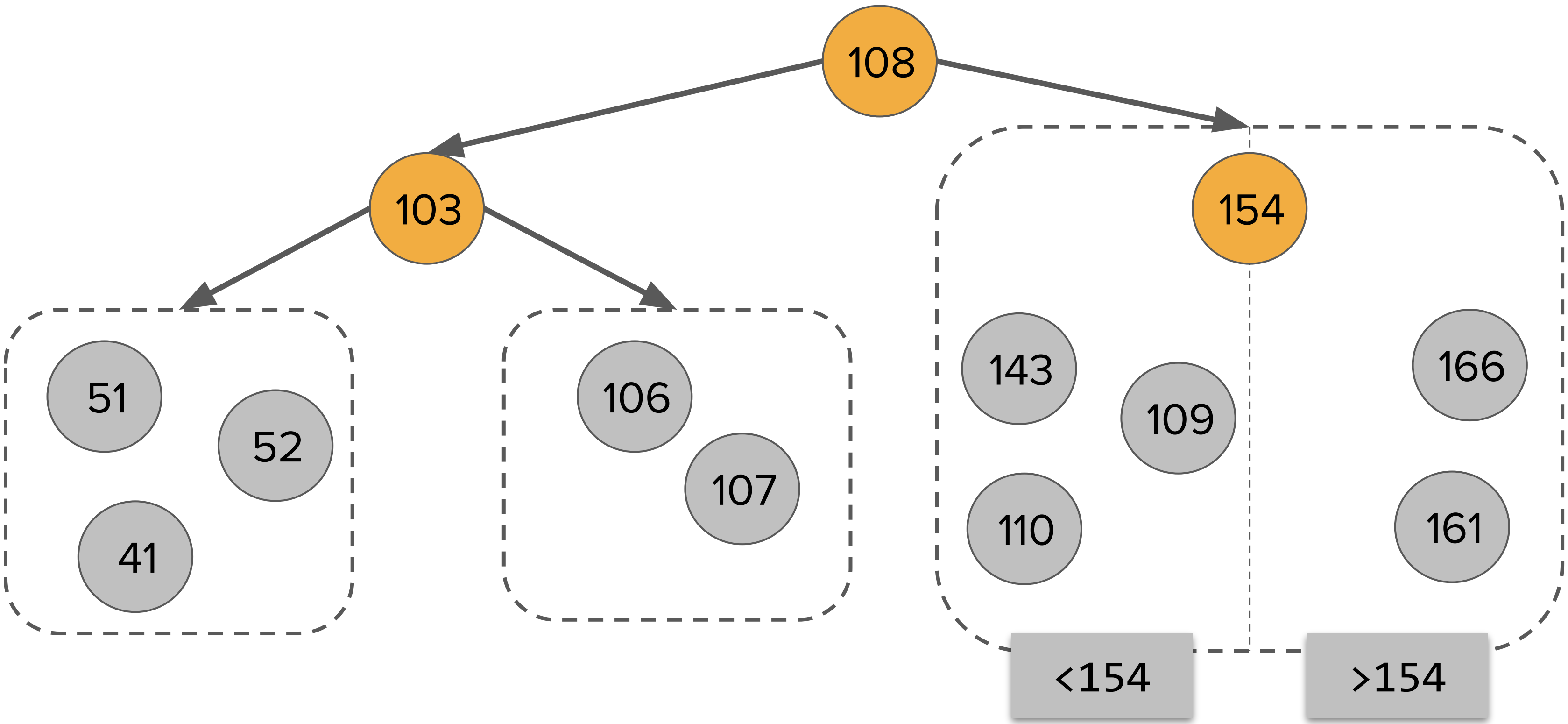

108
103
51
107
52
106
41
<103
>103
166
143
161
154
110
109

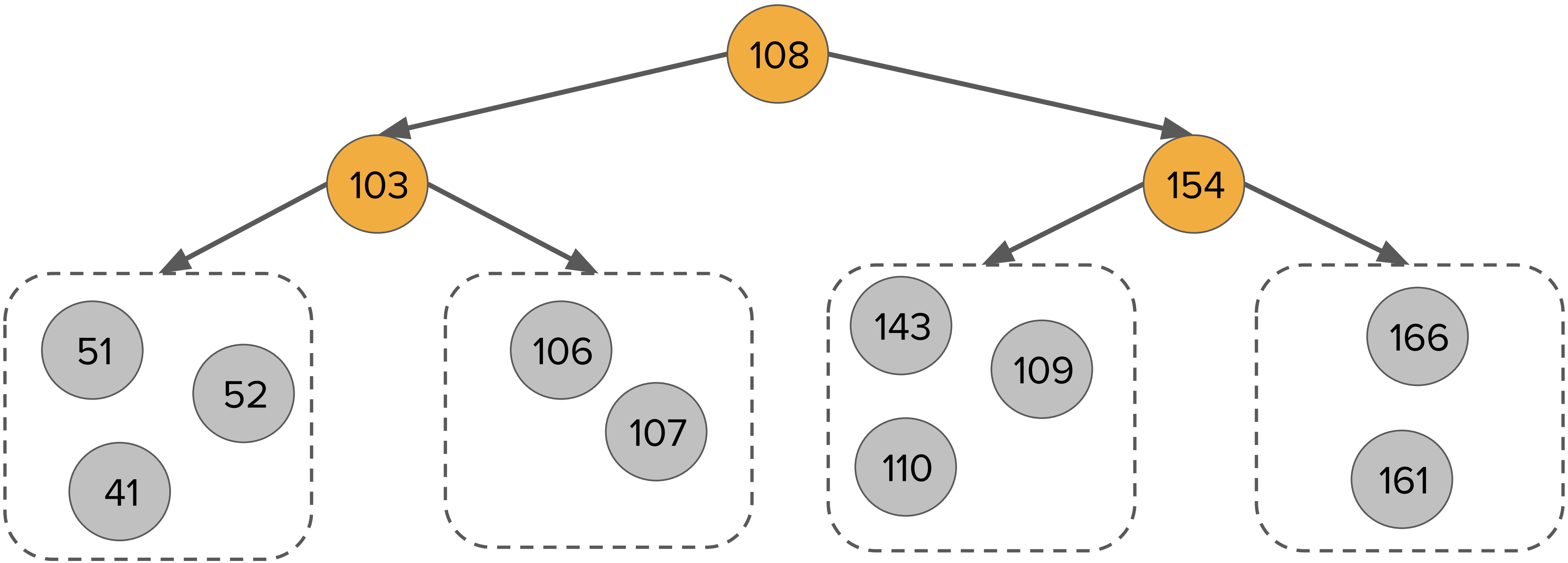

108
103
51
52
41
106
107
166
143
161
154
110
109

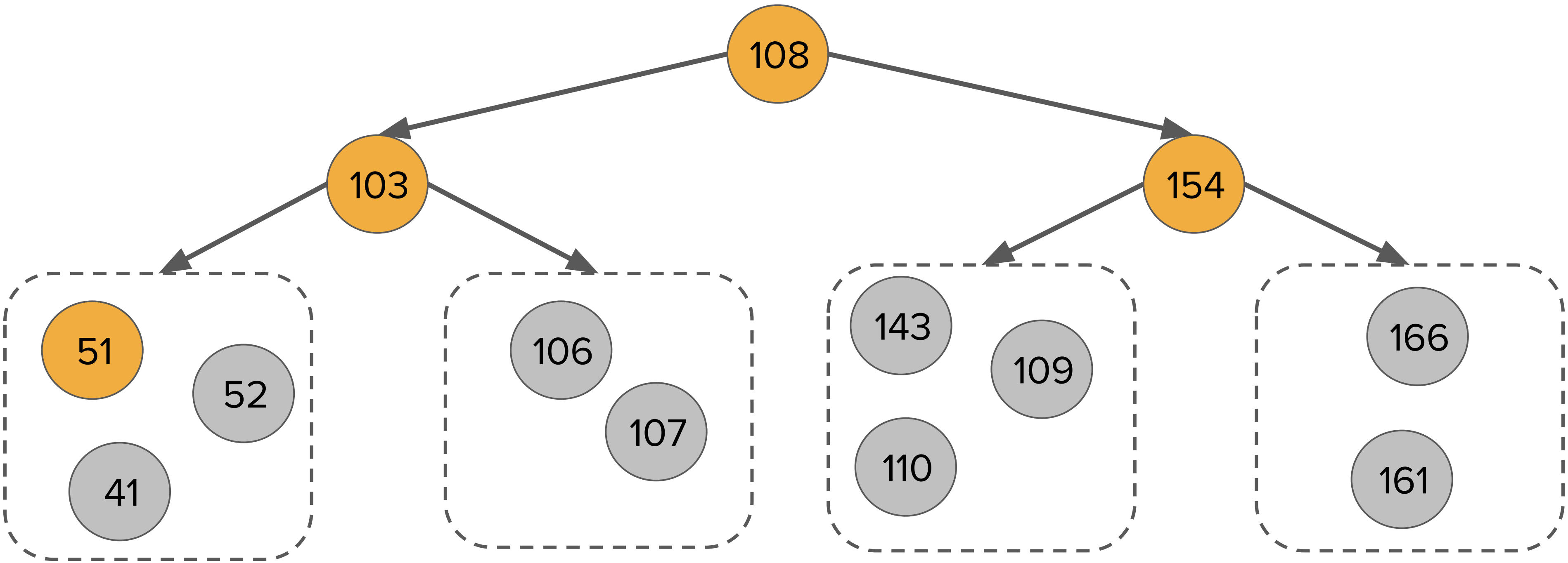


Pick the median element of the right side

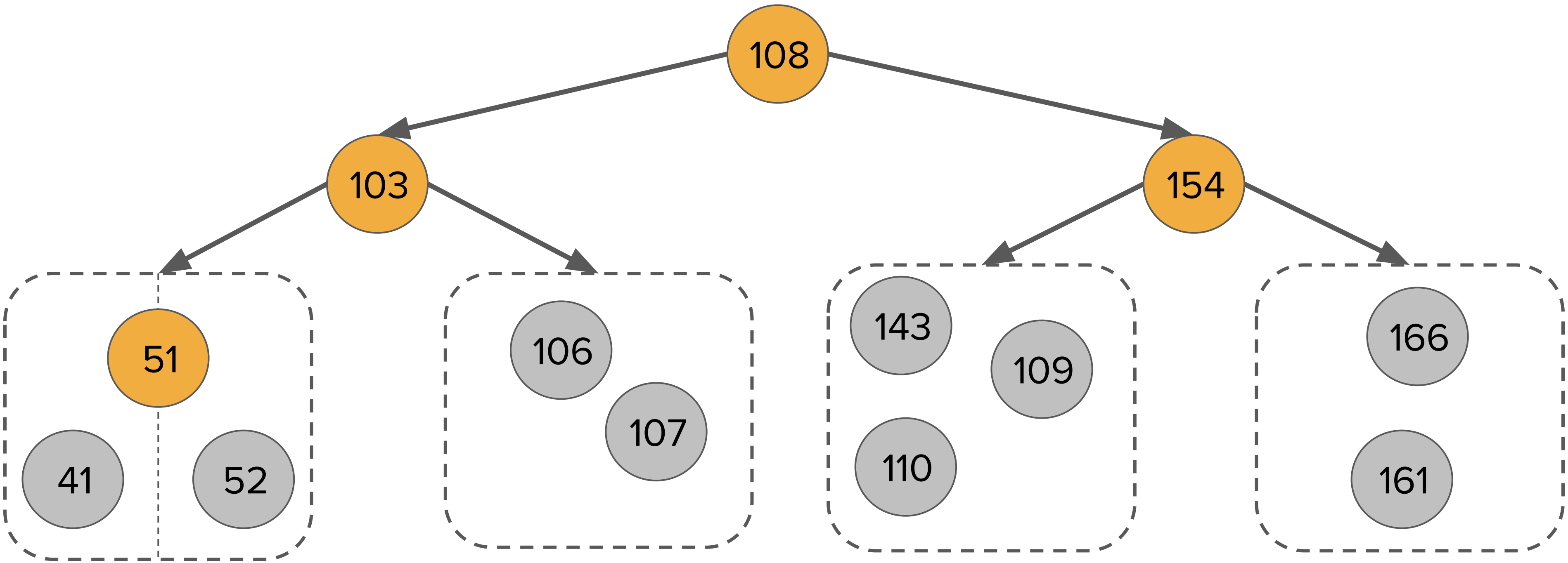

108
103
154
166
51
52
41
106
107
161
110
143
109
<154
>154

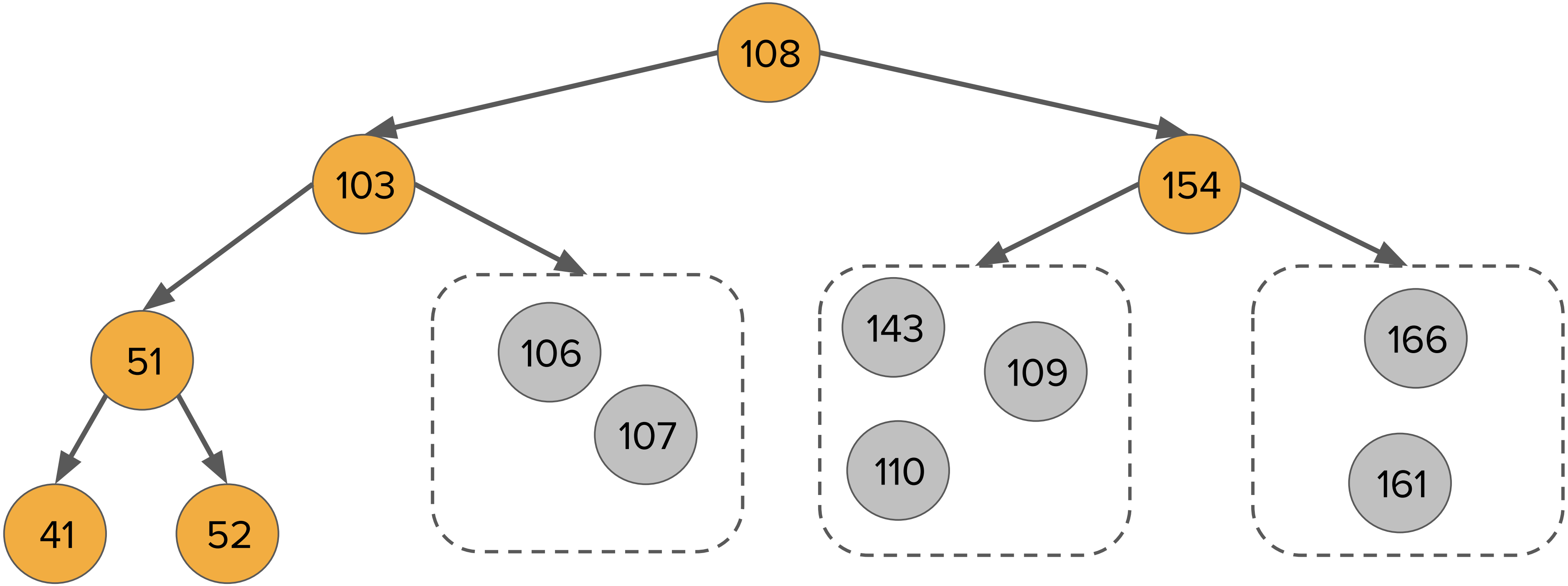

108
103
154
51
52
41
106
107
143
109
110
166
161
<154
>154

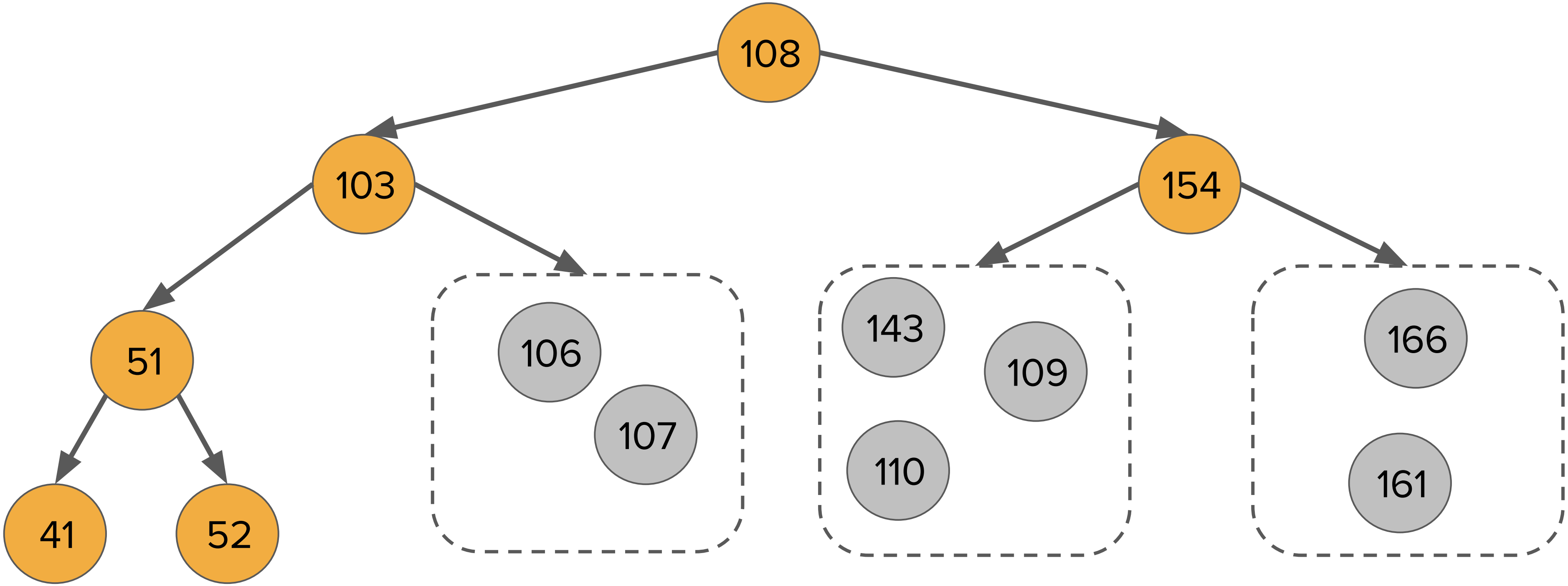

108
103
154
51
52
41
106
107
143
109
110
166
161



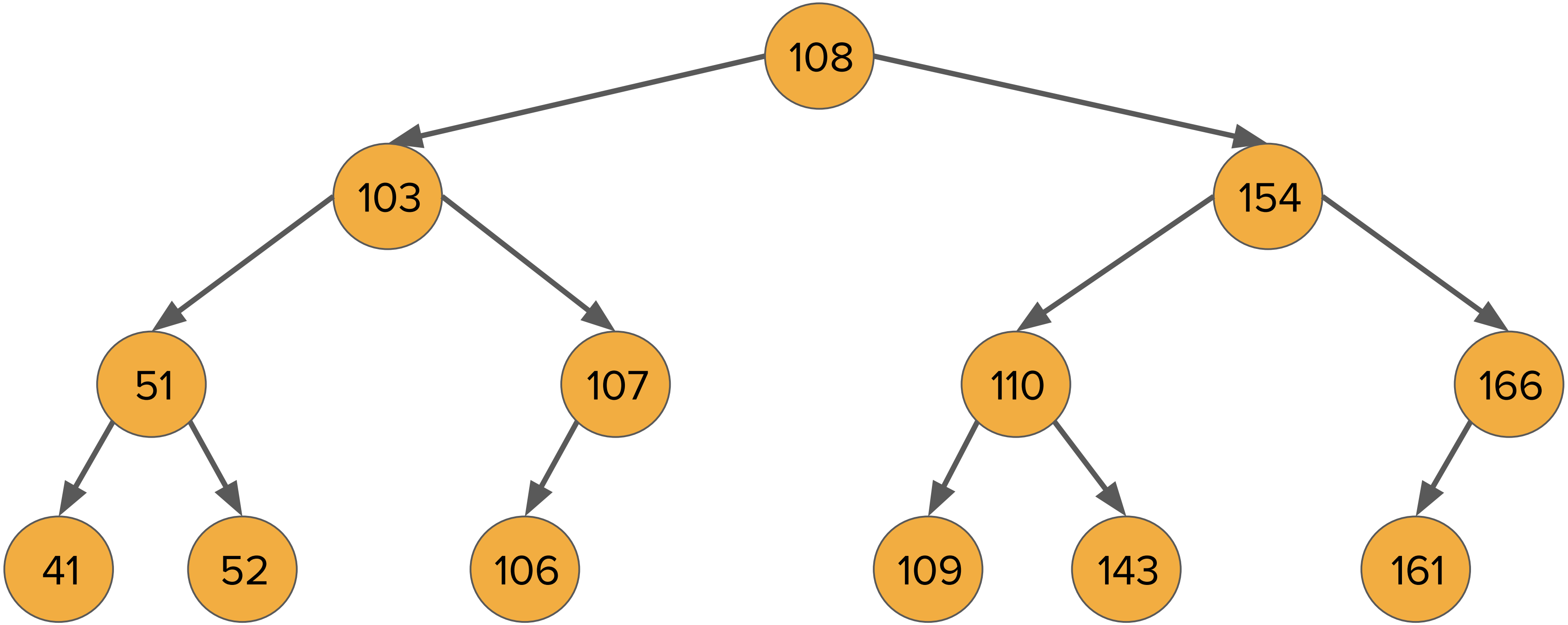
Pick the median element of the left side

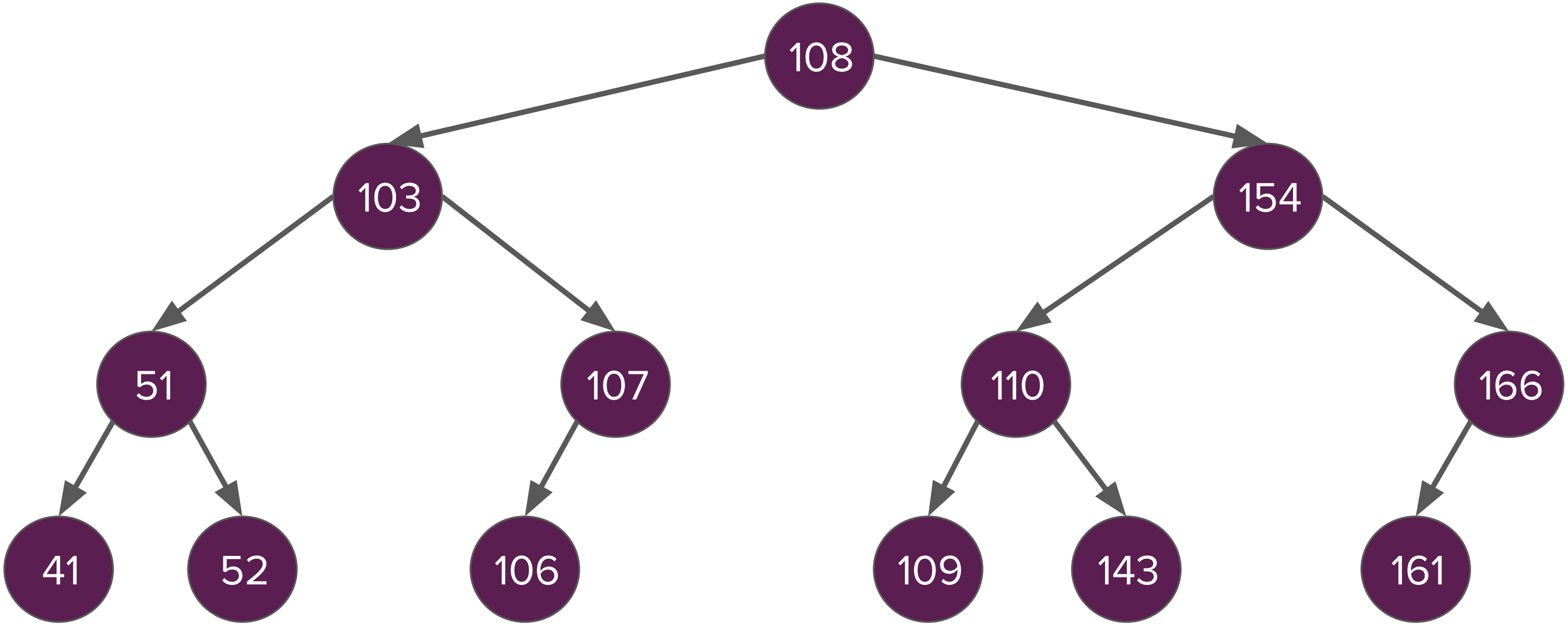

108
103
154
51
41
52
106
107
143
109
110
166
161


108
103
154
51
106
107
143
109
110
166
161
41
52

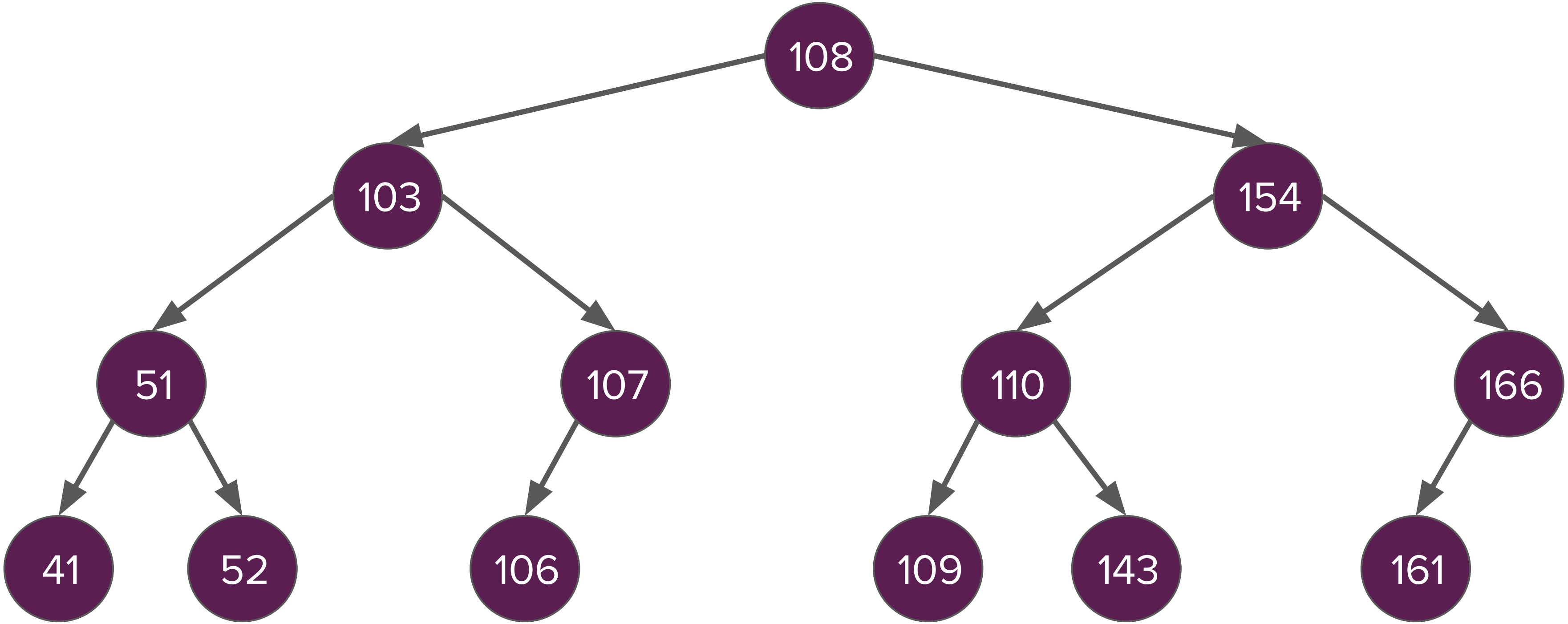


Keep repeating this process
for all the subtrees!


108
103
154
51
107
110
166
41
52
106
109
143
161

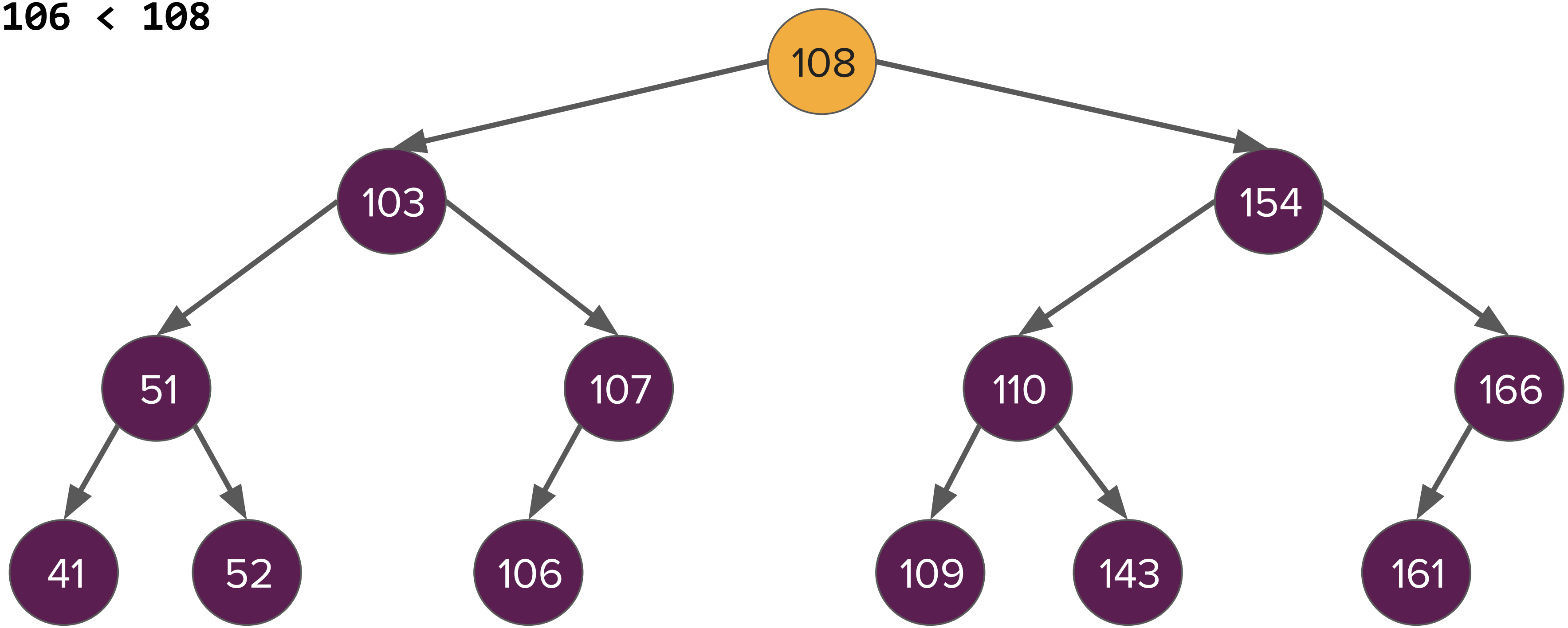


There are 13 nodes in the tree, but the path to each node is short (~`O(log 13)`)!



How could we check if **106** is in this tree?

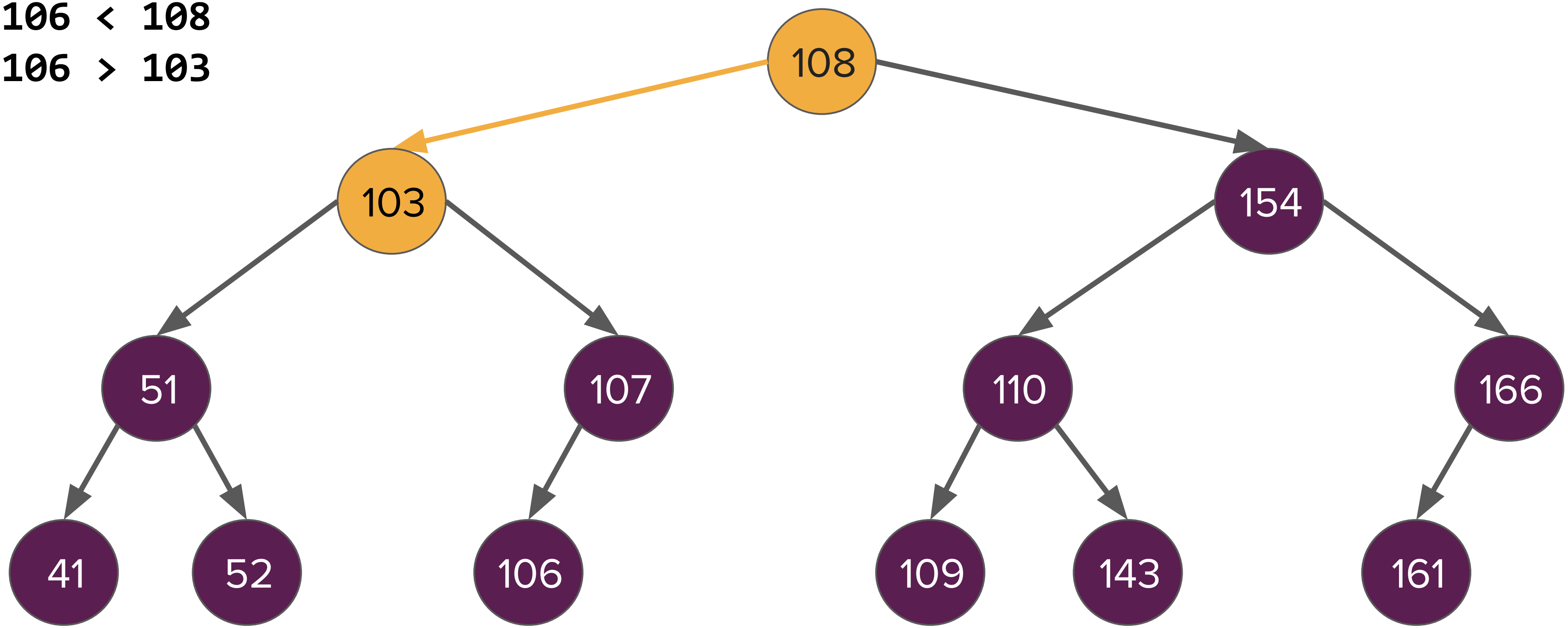
`106 < 108`



How could we check if **`106`** is in this tree?

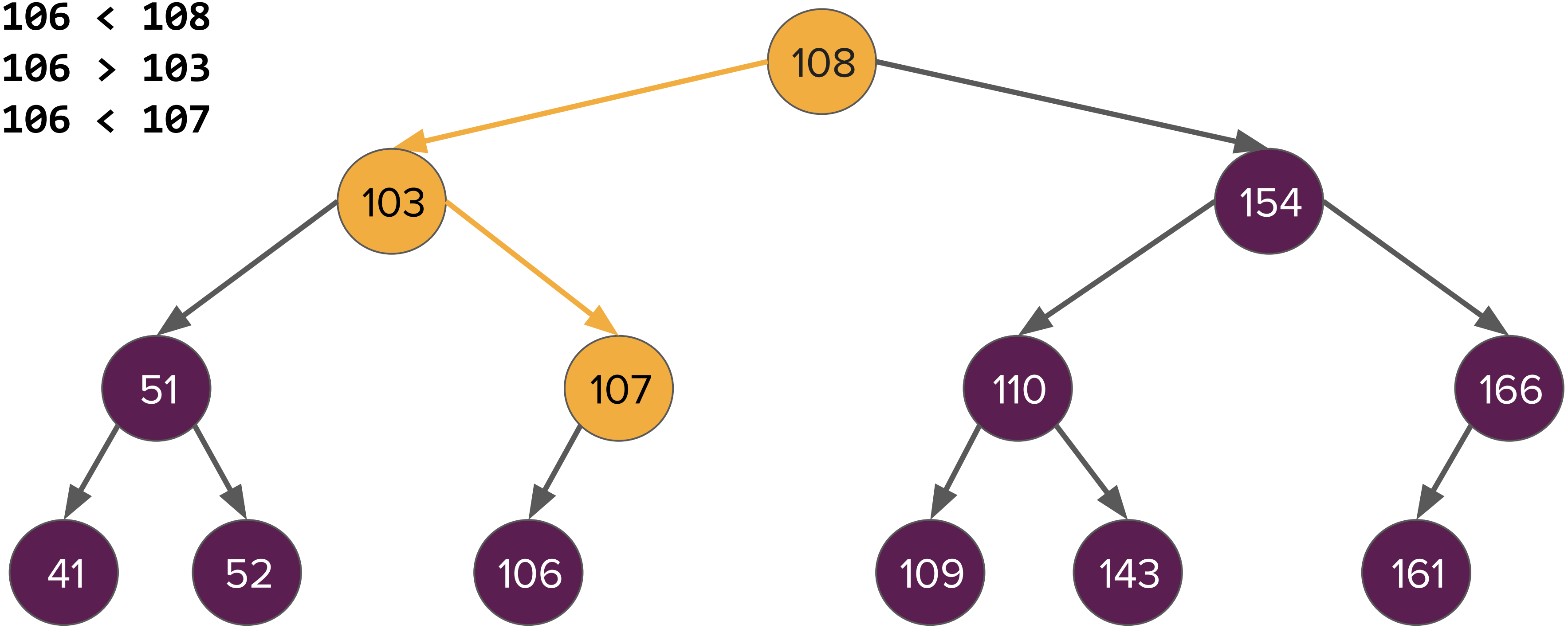
106 < 108

106 > 103



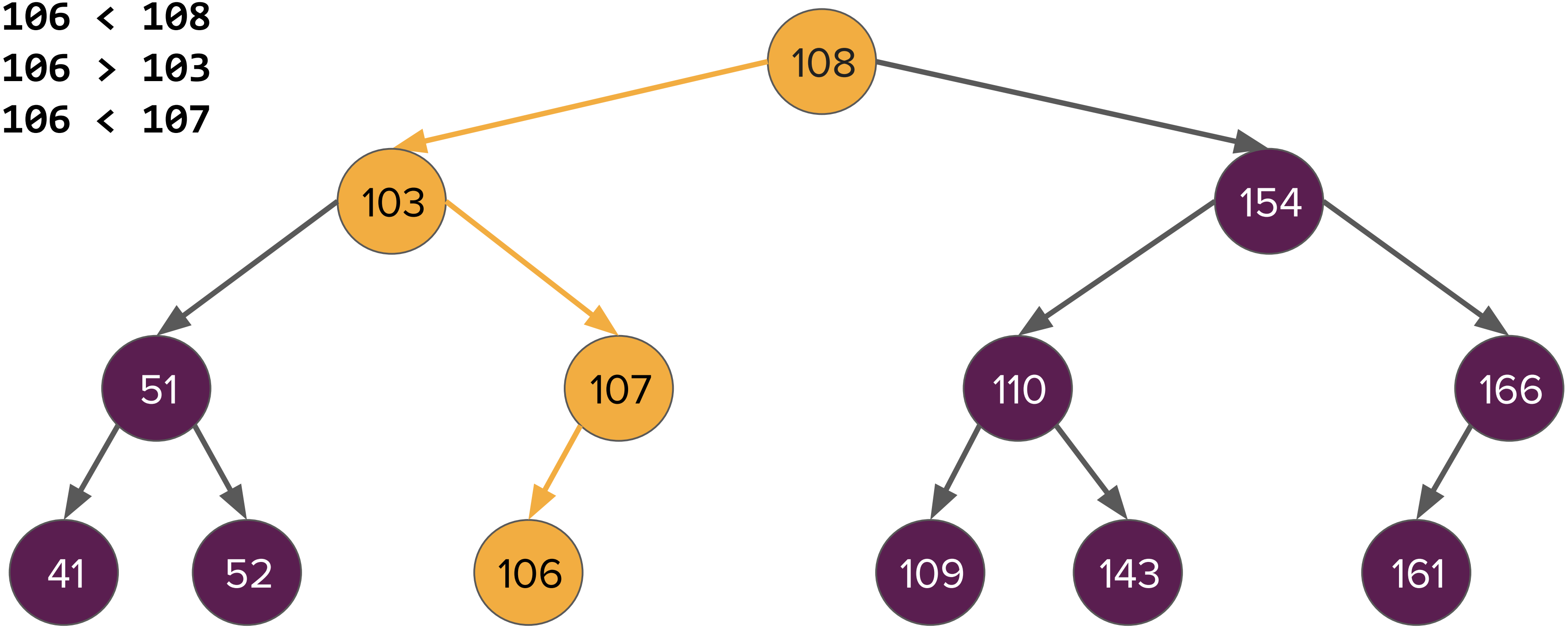
How could we check if **106** is in this tree?

```
106 < 108
106 > 103
106 < 107
```

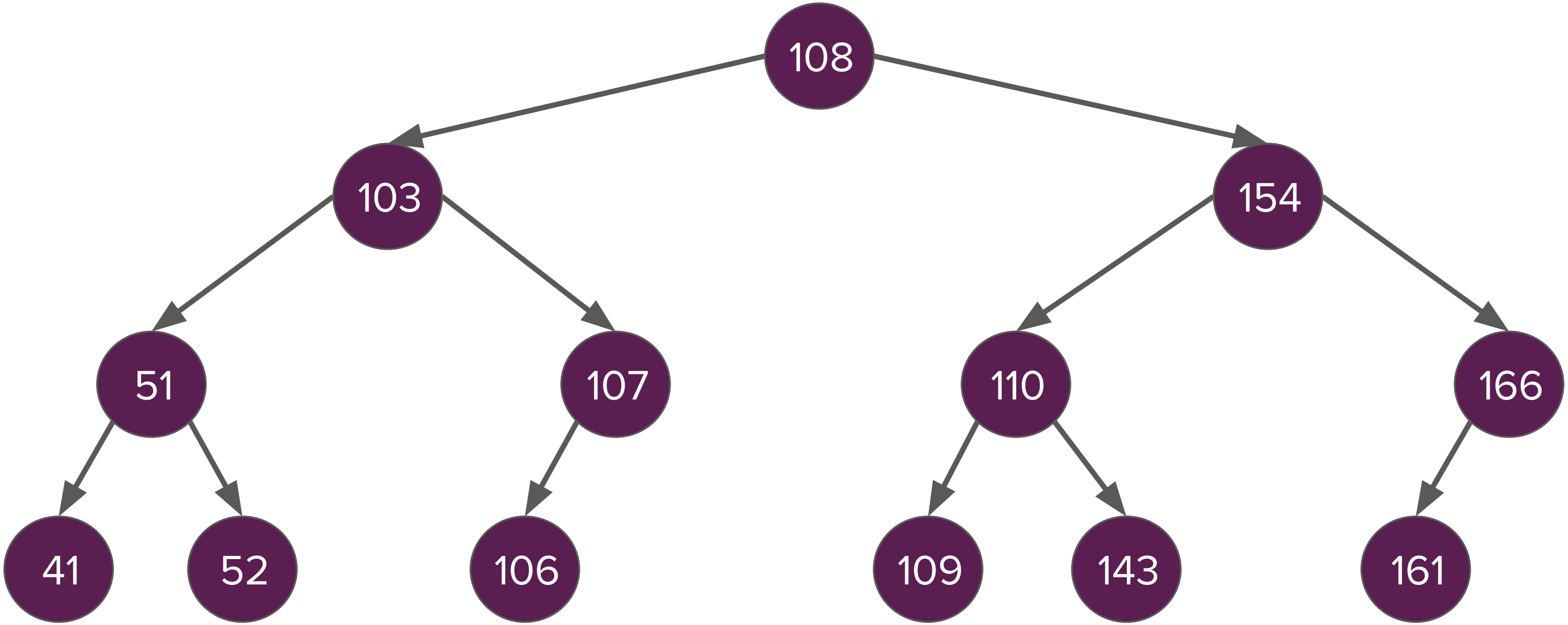


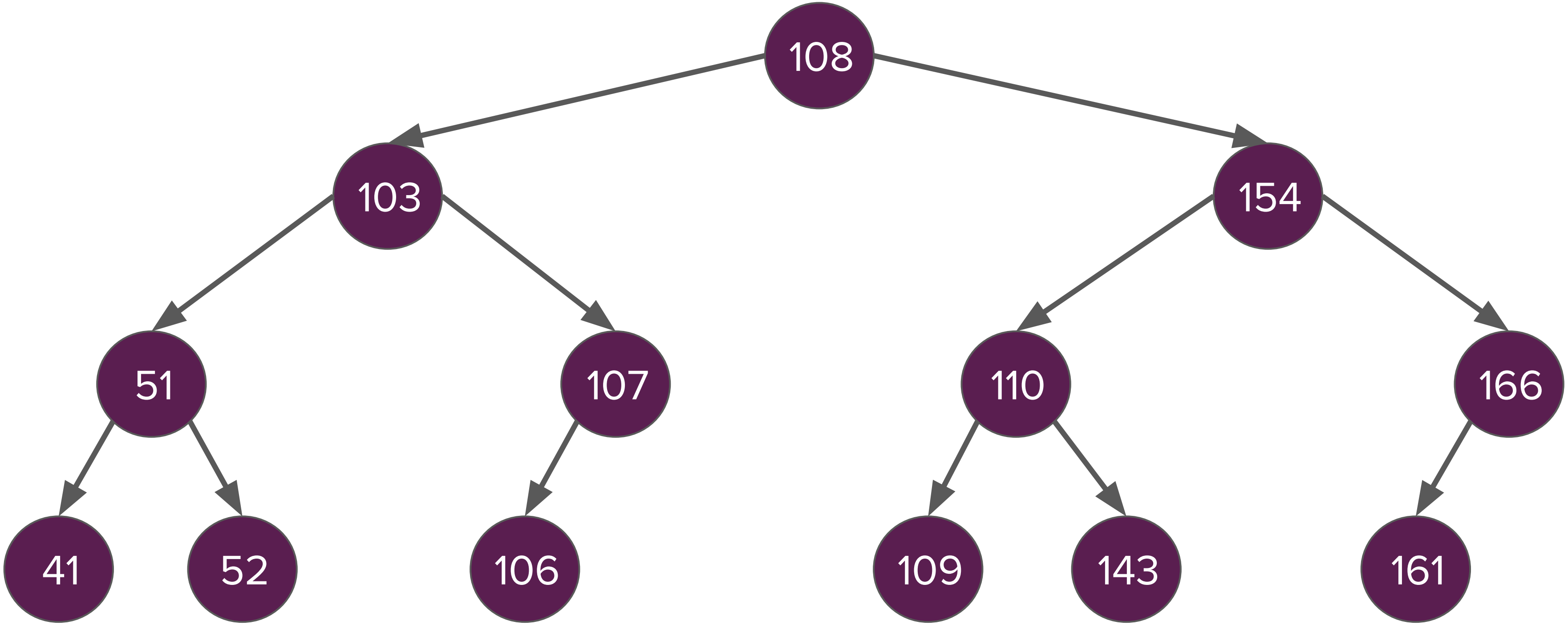
How could we check if **106** is in this tree?

```
106 < 108
106 > 103
106 < 107
```



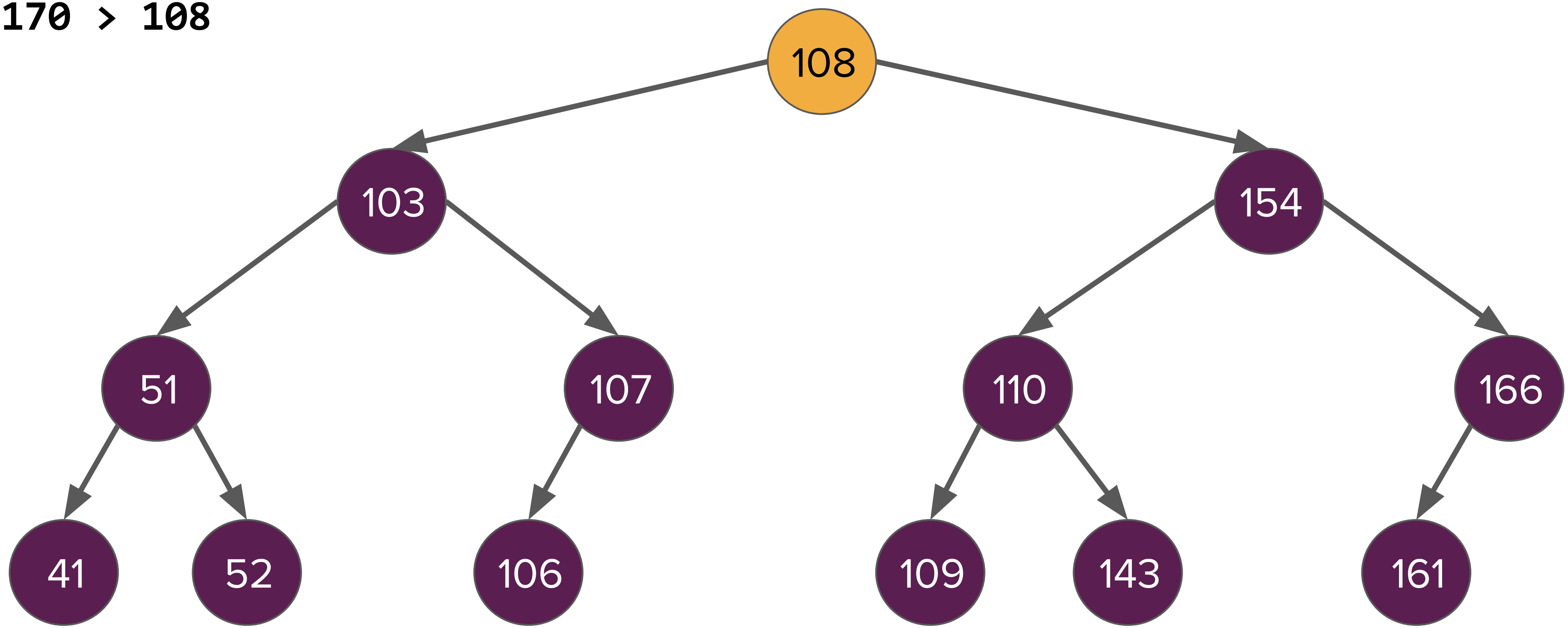
We found **106** so we're done!


108
103
154
51
107
110
166
41
52
106
109
143
161



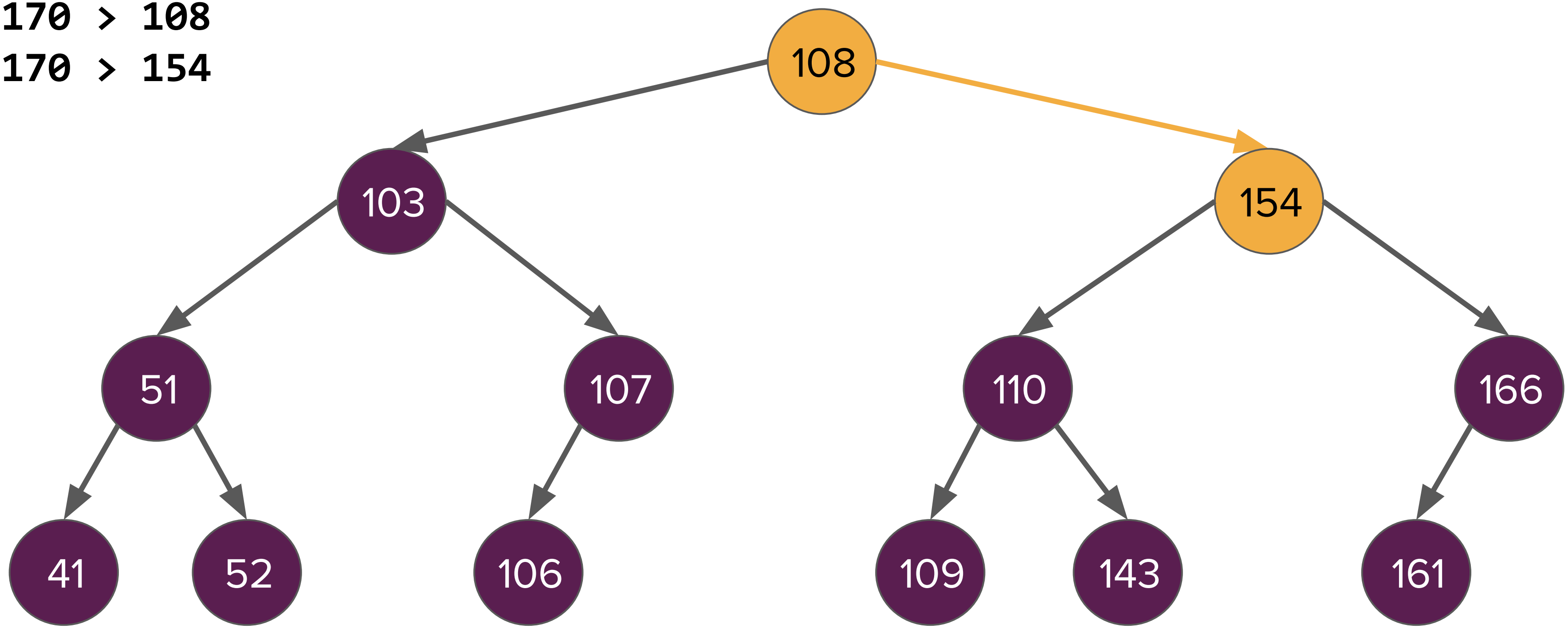
How could we check if **170** is in this tree?

`170 > 108`



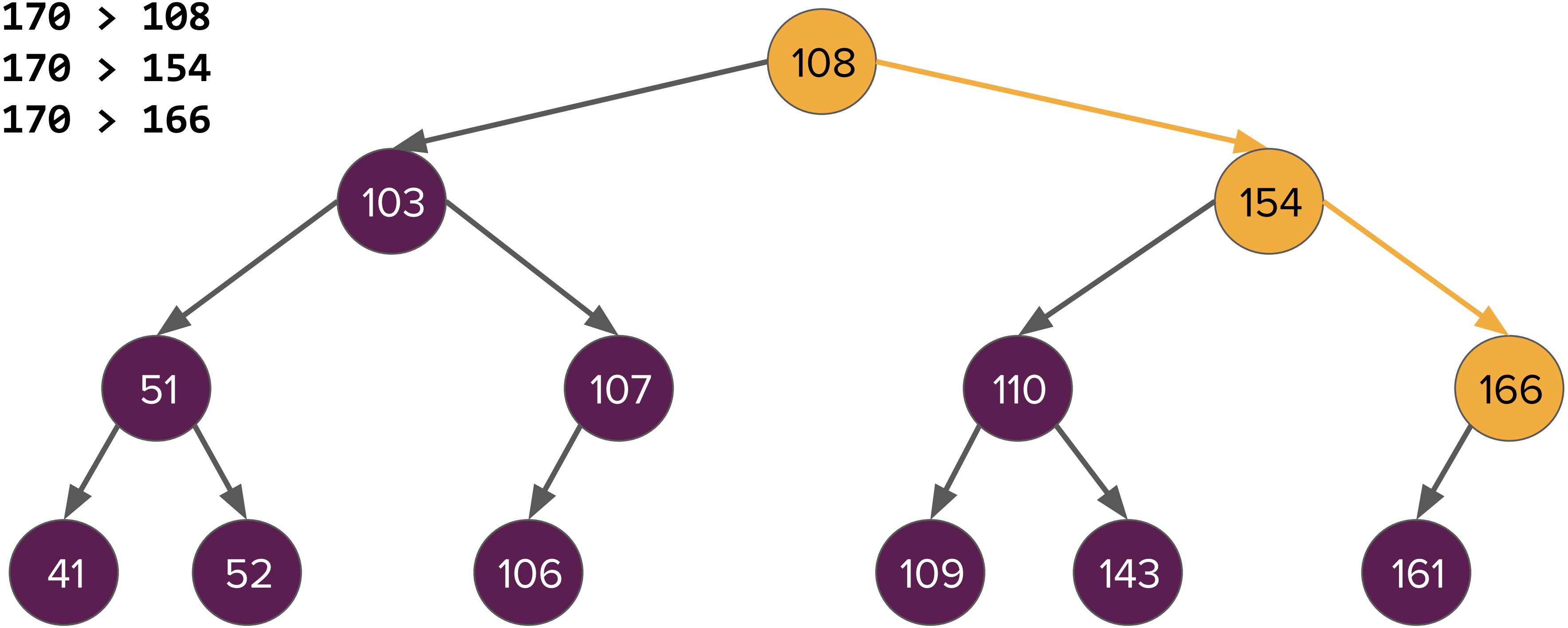
How could we check if **`170`** is in this tree?

`170 > 108`
`170 > 154`



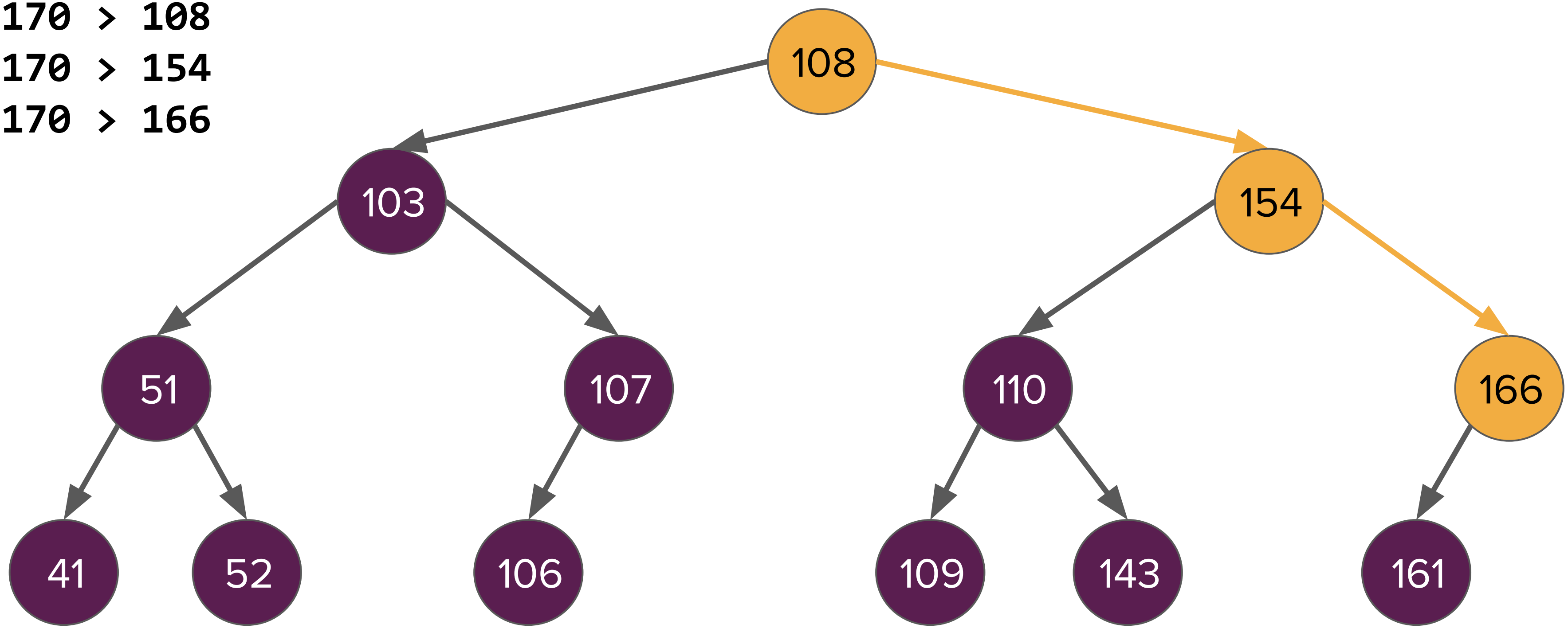
How could we check if **170** is in this tree?

170 > 108
170 > 154
170 > 166



How could we check if **170** is in this tree?

```
170 > 108
170 > 154
170 > 166
```



Right child is **`nullptr`** so we're done!

Building a BST

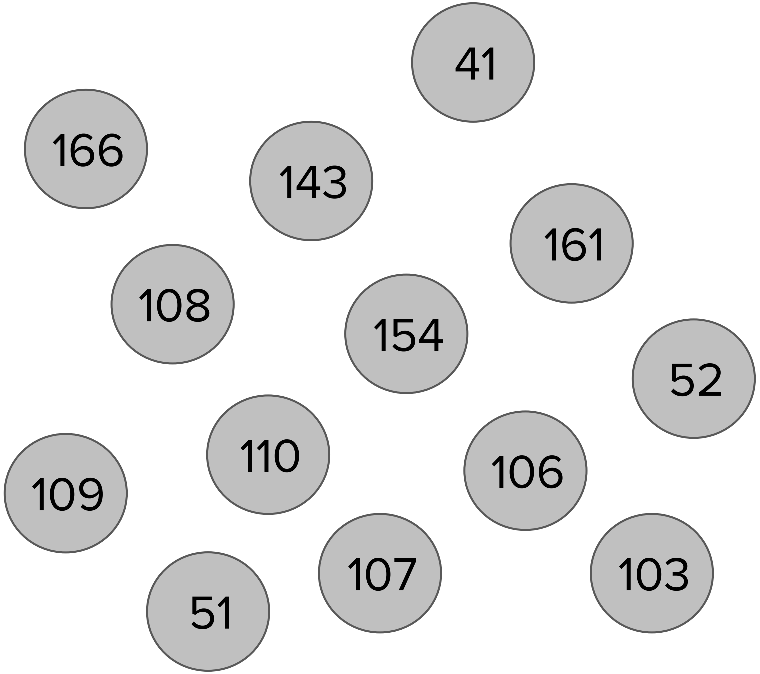
- An **optimal BST** is built by repeatedly choosing the median element as the root node of a given subtree and then separating elements into groups less than and greater than that median.

Building a BST

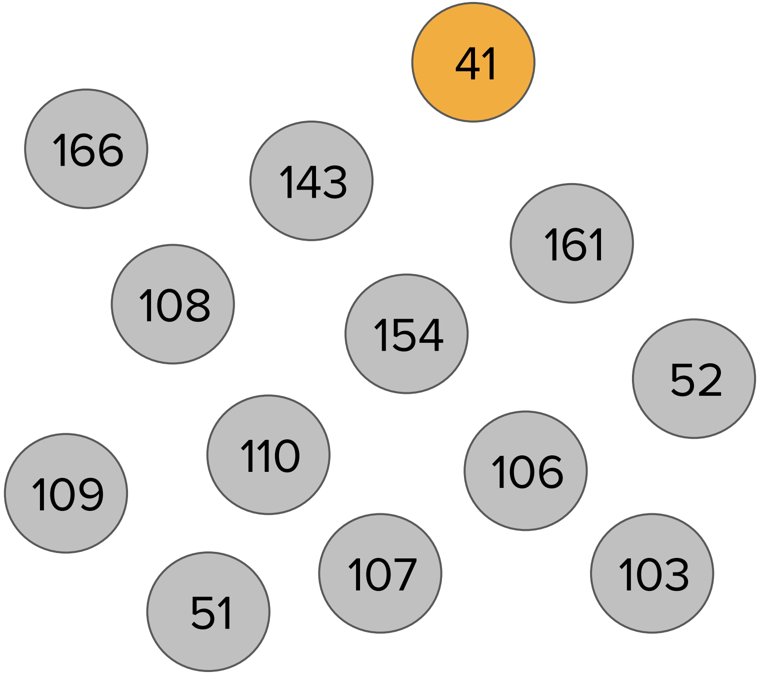
- An **optimal BST** is built by repeatedly choosing the median element as the root node of a given subtree and then separating elements into groups less than and greater than that median.

What does "optimal" mean?

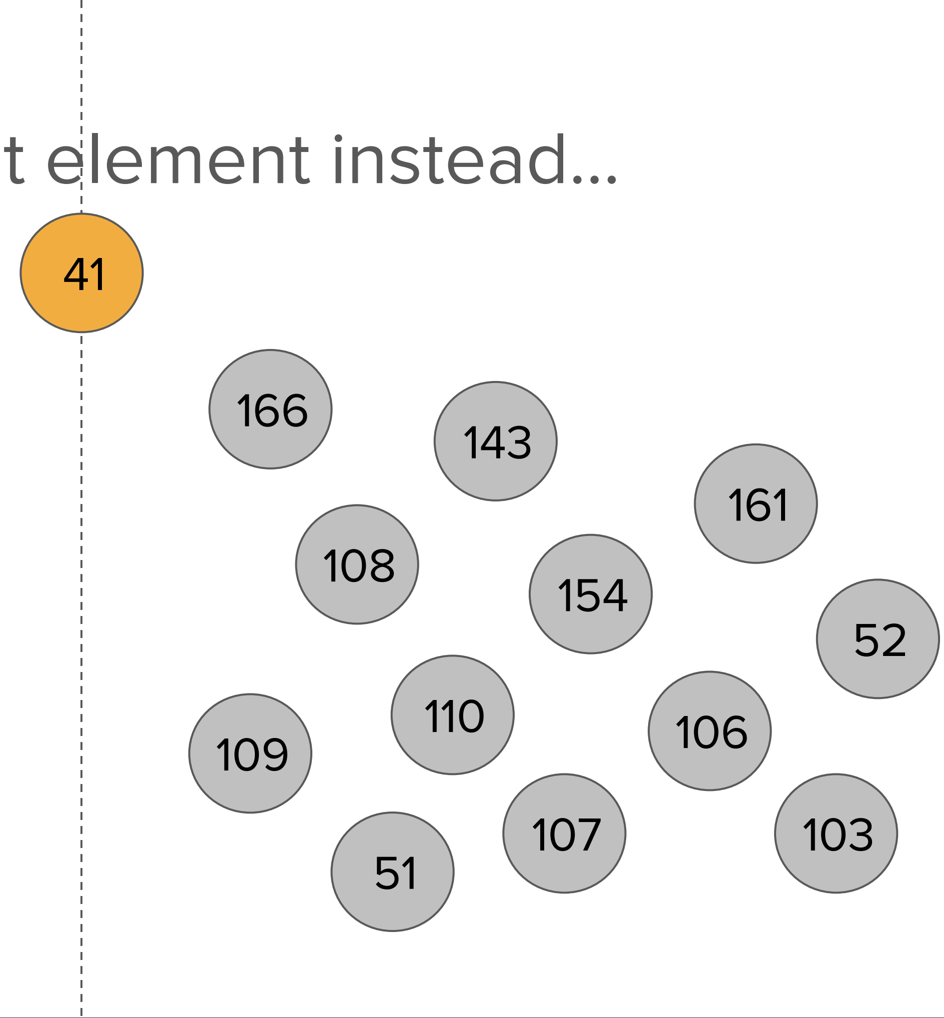
What if we didn’t choose the median?



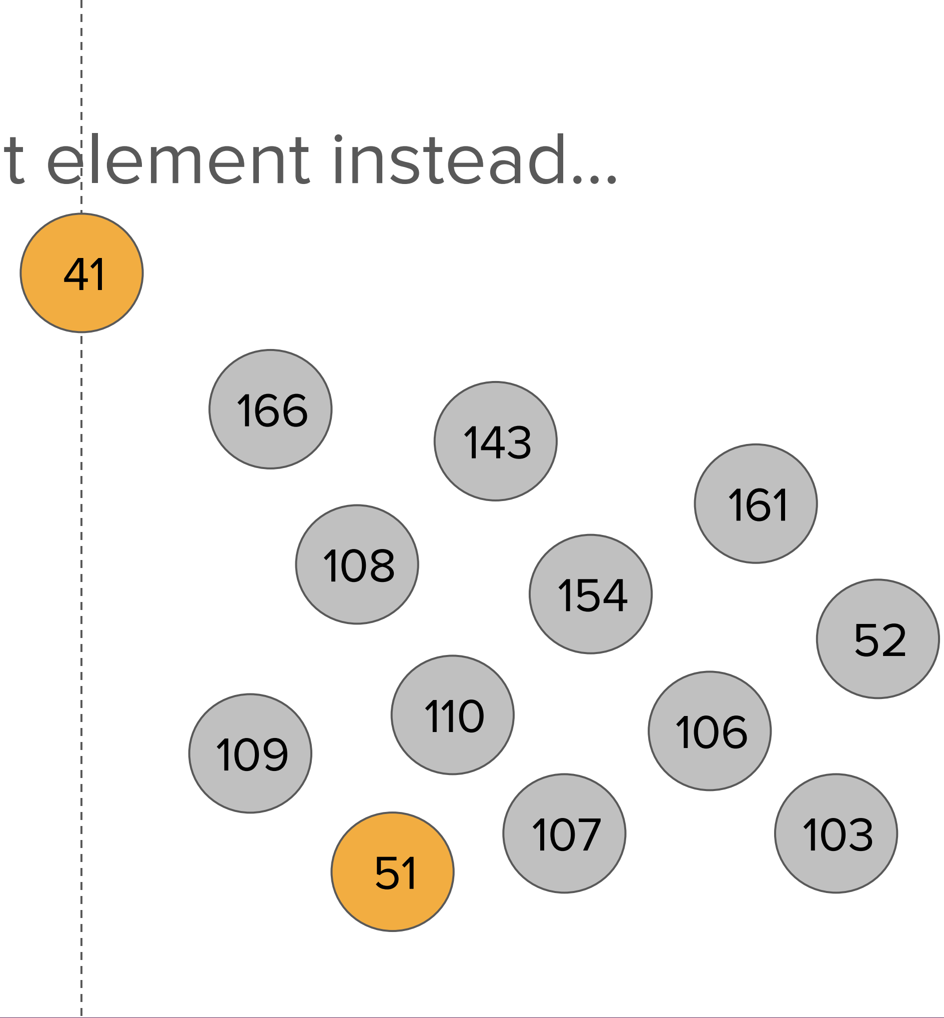
Let's choose the smallest element instead...



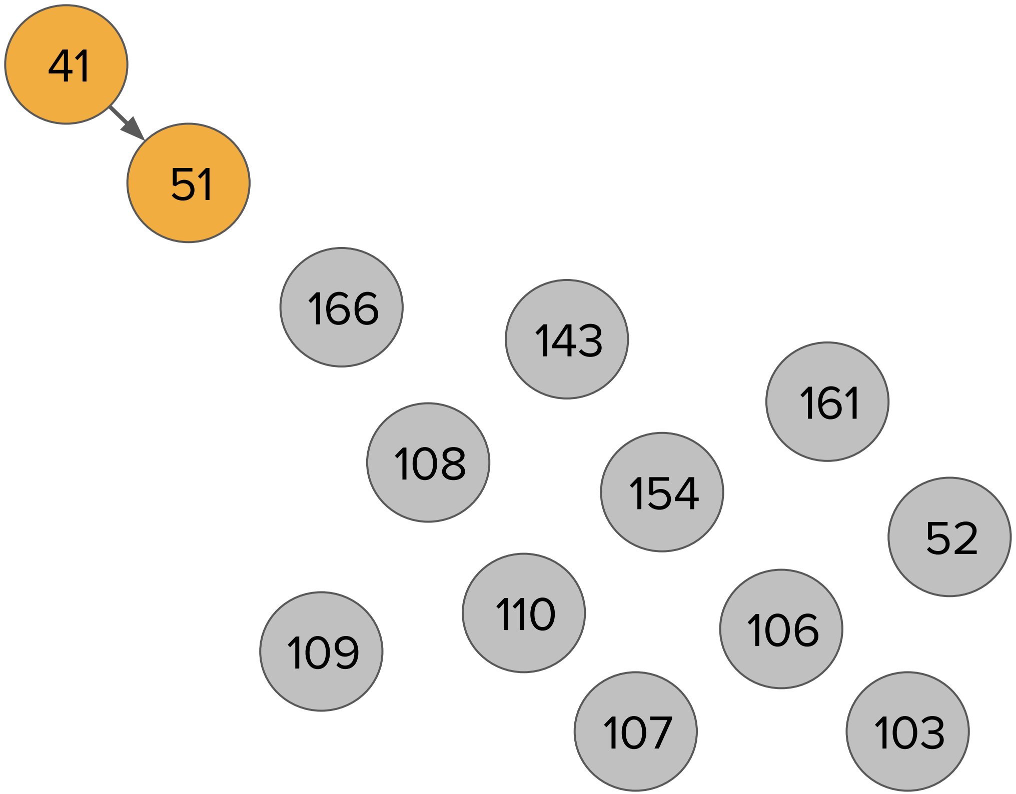
Let's choose the smallest element instead...

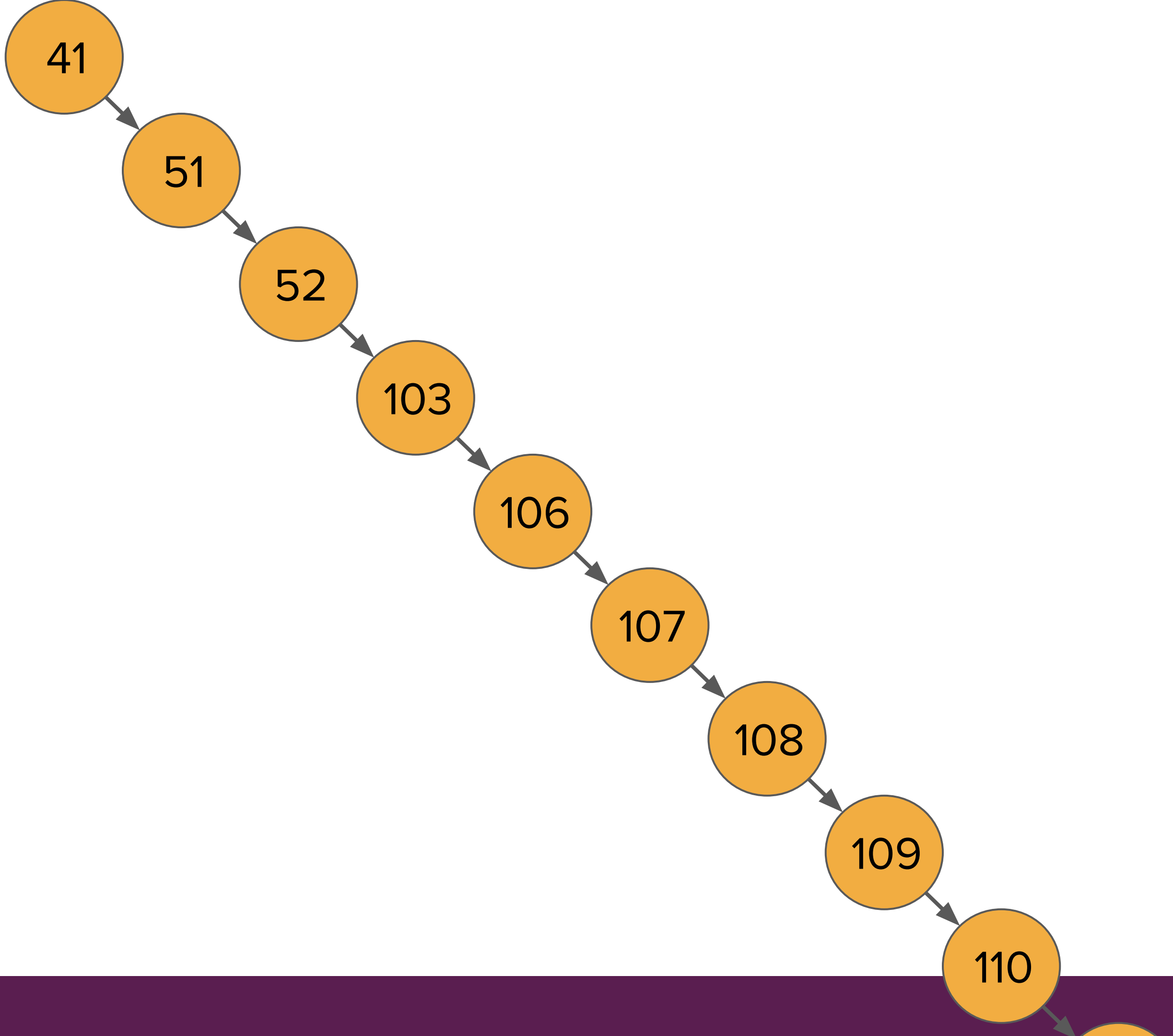


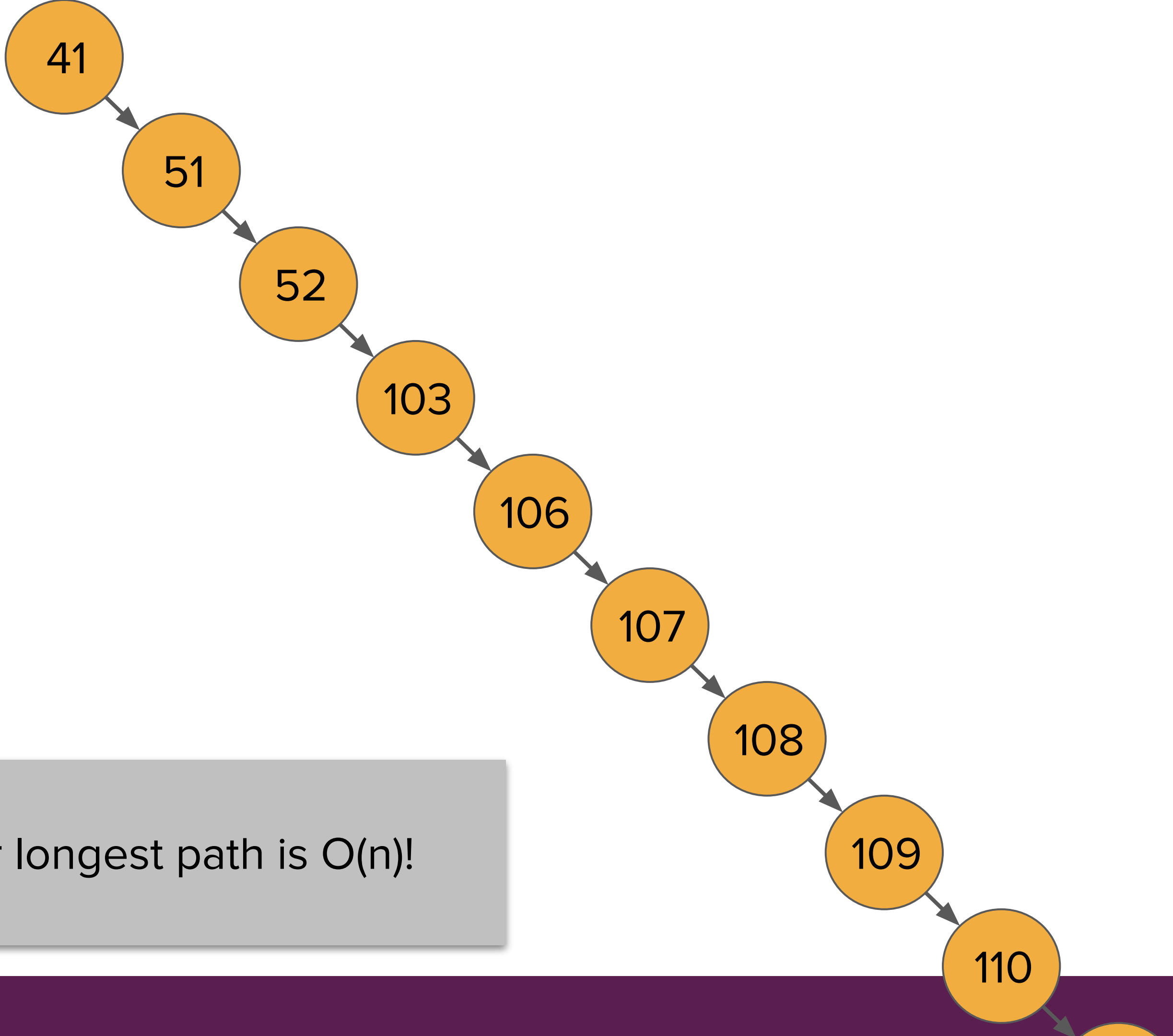
Let's choose the smallest element instead…



Let’s choose the smallest element instead…




41
51
52
103
106
107
108
109
110



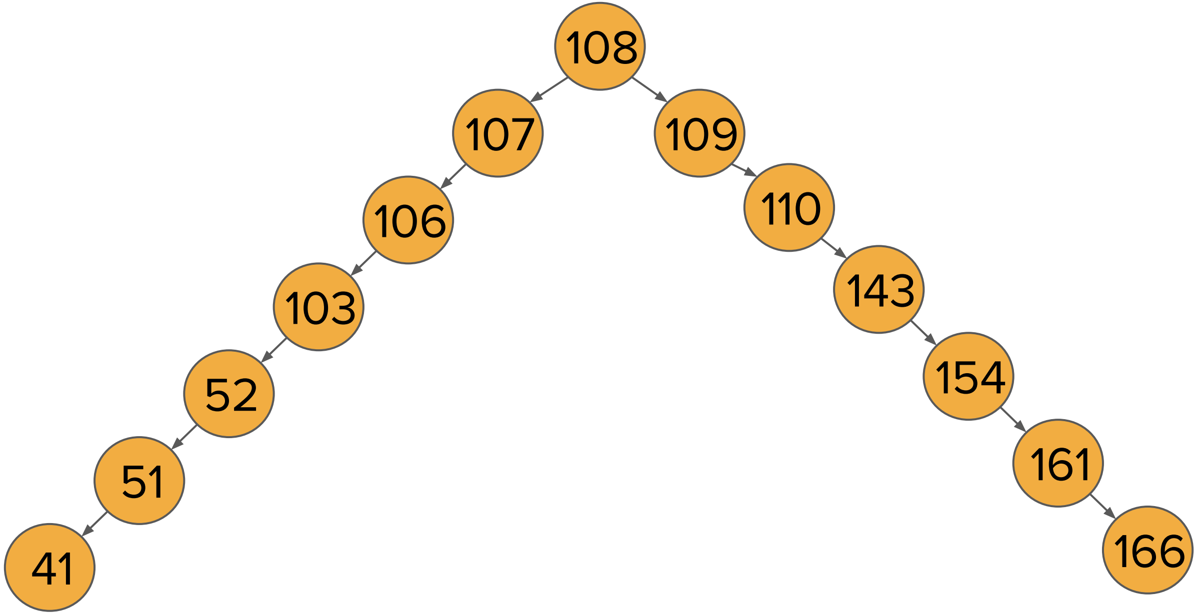
Now our longest path is O(n)!

Takeaways

- There are multiple valid BSTs for the same set of data.

Takeaways

- There are multiple valid BSTs for the same set of data.
  - Another example with the previous dataset:



Takeaways

- There are multiple valid BSTs for the same set of data.
- How you construct the tree/the order in which you add the elements to the tree matters!

Takeaways

- There are multiple valid BSTs for the same set of data.

- How you construct the tree/the order in which you add the elements to the tree matters!

- A binary search tree is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree (i.e. left/right subtrees of a given node don't differ in height by more than 1).
    - Lookup, insertion, and deletion with balanced BSTs all operate in `O(log n)` runtime.

Takeaways

- There are multiple valid BSTs for the same set of data.
- How you construct the tree/the order in which you add the elements to the tree matters!
- A binary search tree is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree (i.e. left/right subtrees don't differ in height by more than 1).
  - Lookup, insertion, and deletion with balanced BSTs all operate in `O(log n)` runtime.
  - **Theorem**: If you start with an empty tree and add in random values, then with high probability the tree is balanced. ➜ take CS161 to learn why!

Takeaways

- There are multiple valid BSTs for the same set of data.

- How you construct the tree/the order in which you add the elements to the tree matters!

- A binary search tree is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree (i.e. left/right subtrees don't differ in height by more than 1).
    - Lookup, insertion, and deletion with balanced BSTs all operate in `O(log n)` runtime.
    - **Theorem**: If you start with an empty tree and add in random values, then with high probability the tree is balanced. ➜ take CS161 to learn why!
    - A **self-balancing** BST reshapes itself on insertions and deletions to stay balanced (how to do this is beyond the scope of this class).

Announcements

Announcements

- Trip's Group OH will be on Friday, 8/5 from 10am-12pm in Huang 019.

- Final project write-up due **THIS** Sunday, August 7. **No grace period.**

- Assignment 5 is due today at 11:59pm (with 24-hour grace period).
- Assignment 4 revisions are due this Friday at 11:59pm.
- Assignment 6 comes out Wednesday!
  - Due to the end of quarter timeline, there will be **no revisions on Assignments 6**.

Implementing Sets with BSTs

We’re going to implement a Set using a BST!

- Our Set will only store strings as its data type.

We’re going to implement a Set using a BST!

- Our Set will only store strings as its data type.

```
struct TreeNode {
   std::string data;
   TreeNode* left;
   TreeNode* right;

   // default constructor does not initialize
   TreeNode() {}
   // 3-arg constructor sets fields from arguments
   TreeNode(std::string d, TreeNode* l, TreeNode* r) {
       data = d;
       left = l;
       right = r;
   }
};
```

We’re going to implement a Set using a BST!

- Our Set will only store strings as its data type.

- We have a header file that will include a public interface already defined.

OurSet Public Interface

```
class OurSet {
public:
     OurSet();    // constructor
    ~OurSet();    // destructor

    bool contains(string value);
    void add(string value);
    void remove(string value);
    void clear();
    int size();
    bool isEmpty();
    void printSetContents();

private:
     /* To be defined soon! */
};
```

We’re going to implement a Set using a BST!

- Our Set will only store strings as its data type
- We have a header file that will include a public interface already defined.
- As we write the Set methods, think about how their runtimes would change for a balanced vs. an unbalanced BST.
  - Note: Actual sets are self-balancing, but we won’t go into the details of how to implement that!

How do we design `OurSet`?

We must answer the following three questions:

1. Member functions: *What public interface should* **`OurSet`** *support? What functions might a client want to call?*

2. Member variables: *What private information will we need to store in order to keep track of the data stored in* **`OurSet`**?

3. Constructor: *How are the member variables initialized when a new instance of* **`OurSet`** *is created?*

OurSet Public Interface

```
class OurSet {
public:
    OurSet();    // constructor
   ~OurSet();    // destructor

   bool contains(string value);
   void add(string value);
   void remove(string value);
   void clear();
   int size();
   bool isEmpty();
   void printSetContents();

private:
    /* To be defined soon! */
};
```

Let’s code it!

(constructor, destructor, `clear()`, etc.)

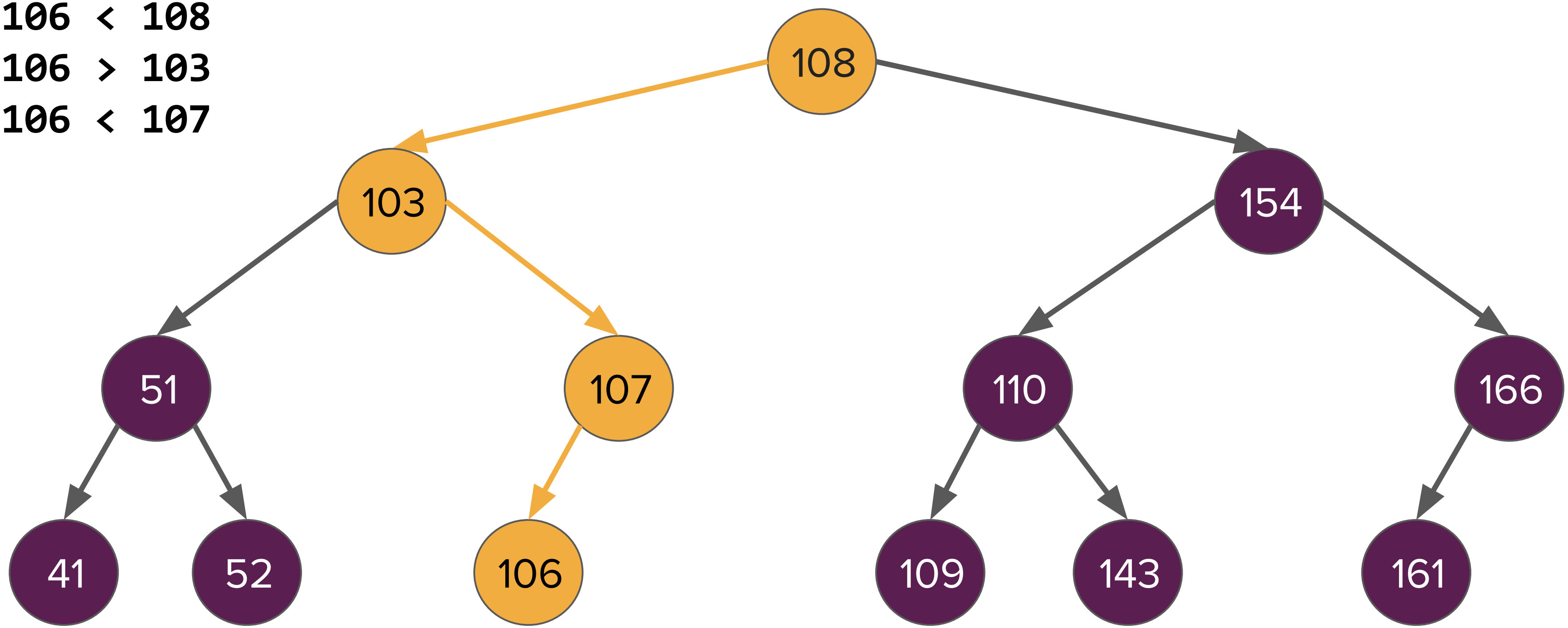
OurSet Public Interface

```
class OurSet {
public:
     OurSet();    // constructor
    ~OurSet();    // destructor

    bool contains(string value);
    void add(string value);
    void remove(string value);
    void clear();
    int size();
    bool isEmpty();
    void printSetContents();

private:
     /* ... */
};
```

```
106 < 108
106 > 103
106 < 107
```



We found **106** so we're done!

Attendance ticket:

https://tinyurl.com/setContains

Please don’t send this link to students who are not here. It’s on your honor!

Let's code it!

(`contains()`, `add()`)

`OurSet` summary

- Our tree utility functions (**`inorderPrint`**, **`freeTree`**) showed up as private member functions/helpers!
    - In-order traversal prints our elements in the correctly sorted order!

OurSet summary

- Our tree utility functions (**inorderPrint**, **freeTree**) showed up as private member functions/helpers!
    - In-order traversal prints our elements in the correctly sorted order!

- Using a BST allowed us to take advantage of recursion to traverse our data and get an O(log n) runtime for our methods.

`OurSet` summary

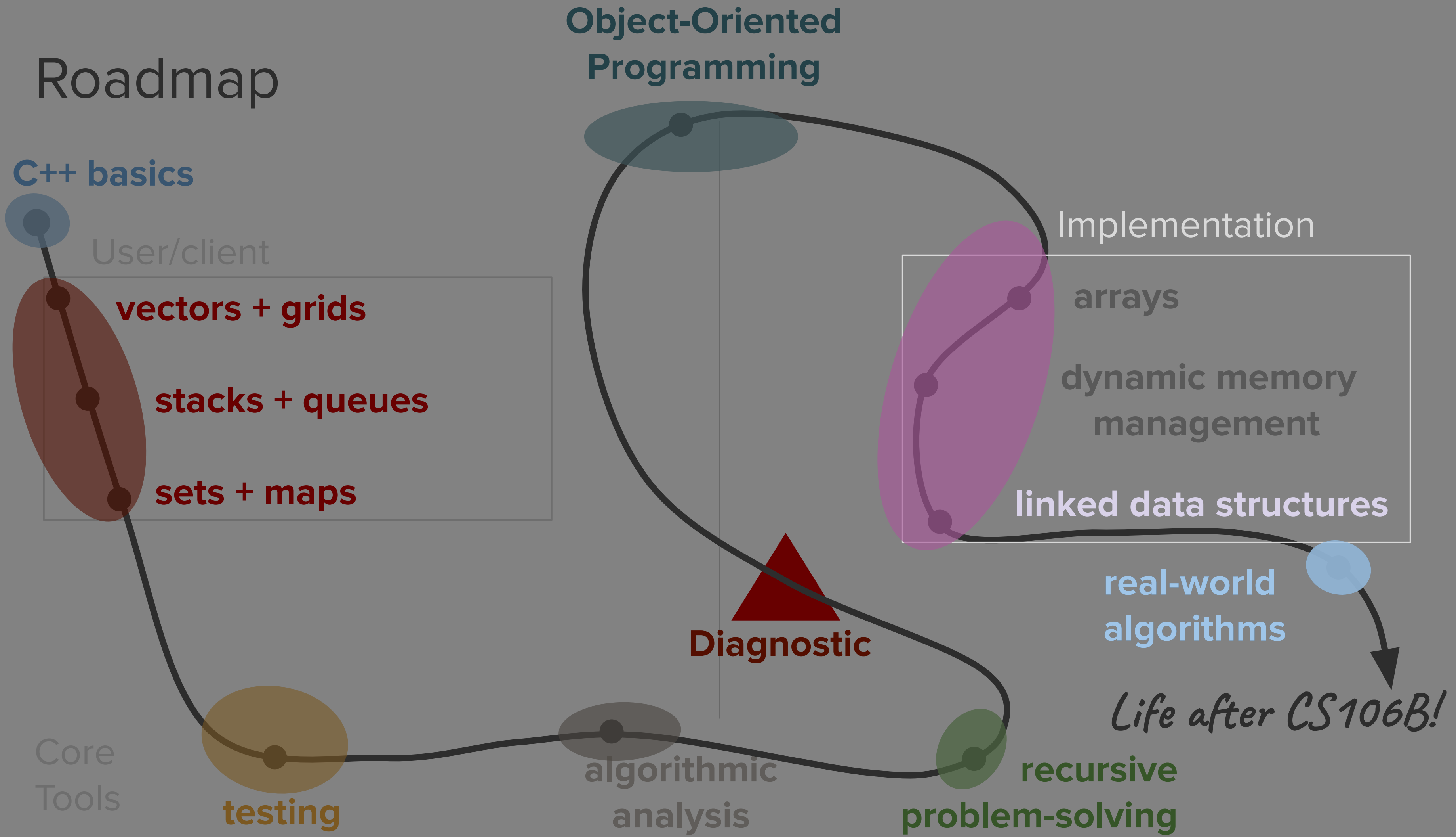
- Our tree utility functions (**`inorderPrint`**, **`freeTree`**) showed up as private member functions/helpers!
  - In-order traversal prints our elements in the correctly sorted order!

- Using a BST allowed us to take advantage of recursion to traverse our data and get an O(log n) runtime for our methods.

- Rewiring trees can be complicated!
  - Make sure to consider when nodes need to be passed by reference.
  - Check out the remove method after class if you're interested in seeing an example of tree rewiring (you won't be required to do anything this complex with tree rewiring).

What's next?

Roadmap



Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Levels of abstraction

What is the interface for the user?

How is our data organized?
(binary heaps, BSTs, **Huffman trees**)

What stores our data?
(arrays, linked lists, **trees**)

How is data represented electronically?
(RAM)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Computer Hardware

Huffman coding