ECE430.217 Data Structures

Binary Search Trees

Weiss Book Chapter 4.2/4.3

Byoungyoung Lee
https://compsec.snu.ac.kr
byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

This topic covers binary search trees:

- Abstract Sorted Lists
- Background
- Definition and examples
- Implementation:
  - FindMin, FindMax, insert, erase
  - Previous smaller and next larger objects
  - Finding the k^{th} object

Abstract Sorted Lists

Previously, we discussed Abstract Lists: the objects are linearly ordered by the programmer

We will now discuss the **Abstract Sorted List**:

– The relation is based on an implicit linear ordering

Abstract Sorted Lists

Queries that may be made about data stored in a Sorted List ADT include:

- Finding the smallest and largest values
- Finding the kth largest value
- Find the next larger or previous smaller objects of a given object
- Iterate through objects within an interval $[a, b]$

Limitation: Abstract Sorted Lists with Array or Linked Lists

If we implement an Abstract Sorted List using an array or a linked list, some operations take $\mathbf{O}(n)$

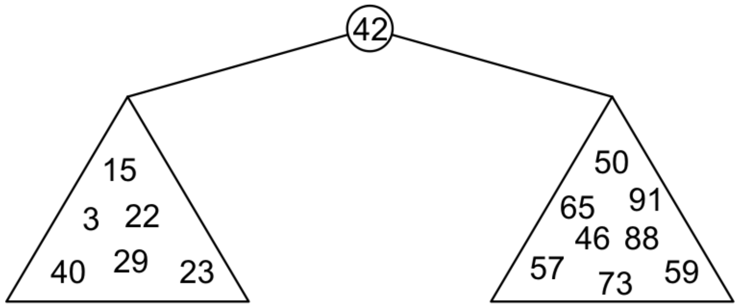
- To perform insertion, we may either traverse or copy, on average, $\mathbf{O}(n)$ objects

Binary Search Trees

Using a binary tree, we can dictate an order on the two children

We will exploit this order:

- All objects in the left sub-tree to be less than the object stored in the root node, and
- All objects in the right sub-tree to be greater than the object in the root object



Recursive definition: Each of the two sub-trees will themselves be binary search trees

Binary Search Trees

We can use this structure for searching

With a linear order, one of the following three must be true:

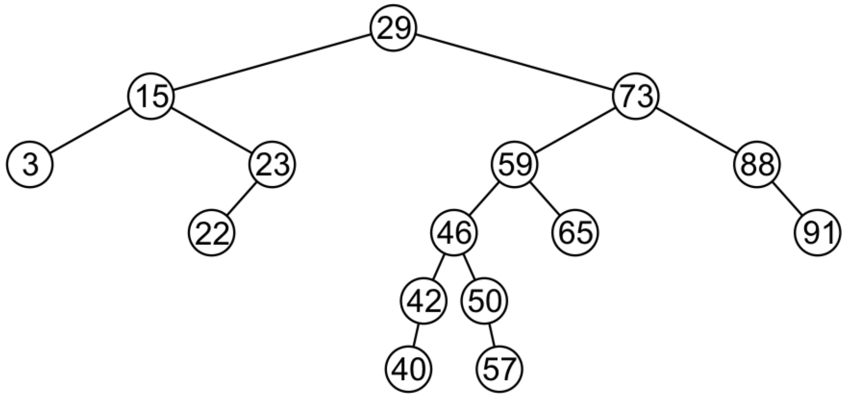
$$a < b \quad a = b \quad a > b$$

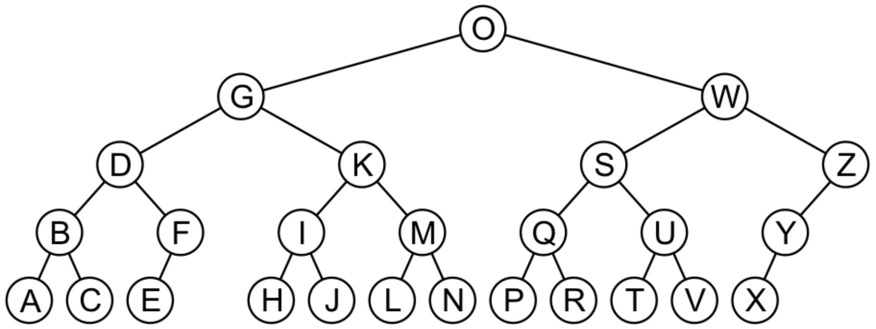
Examine the root node and if we have not found what we are looking for:

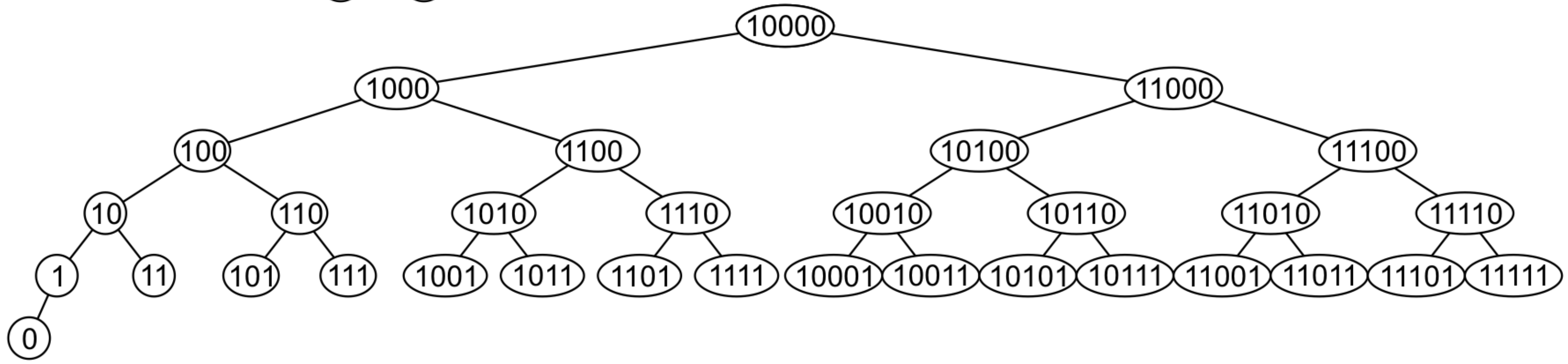
- If the object is less than what is stored in the root node, continue searching in the left sub-tree
- Otherwise, continue searching the right sub-tree

Binary Search Trees: Good Example

Here are other examples of binary search trees:

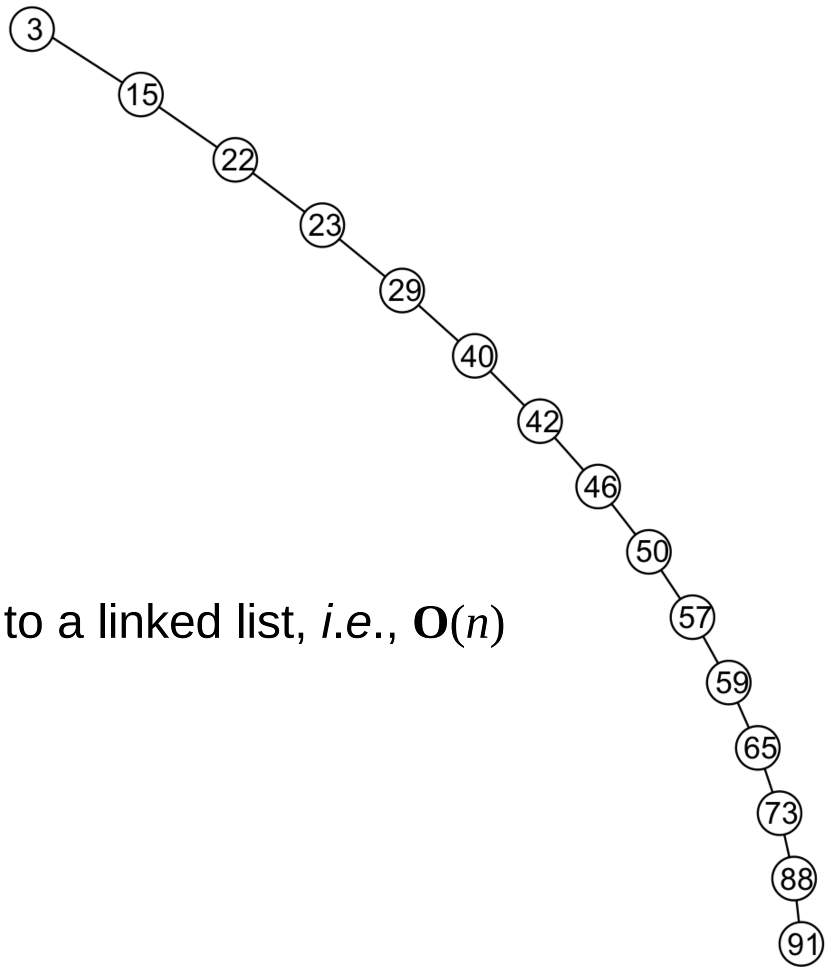






Binary Search Trees: Bad Example

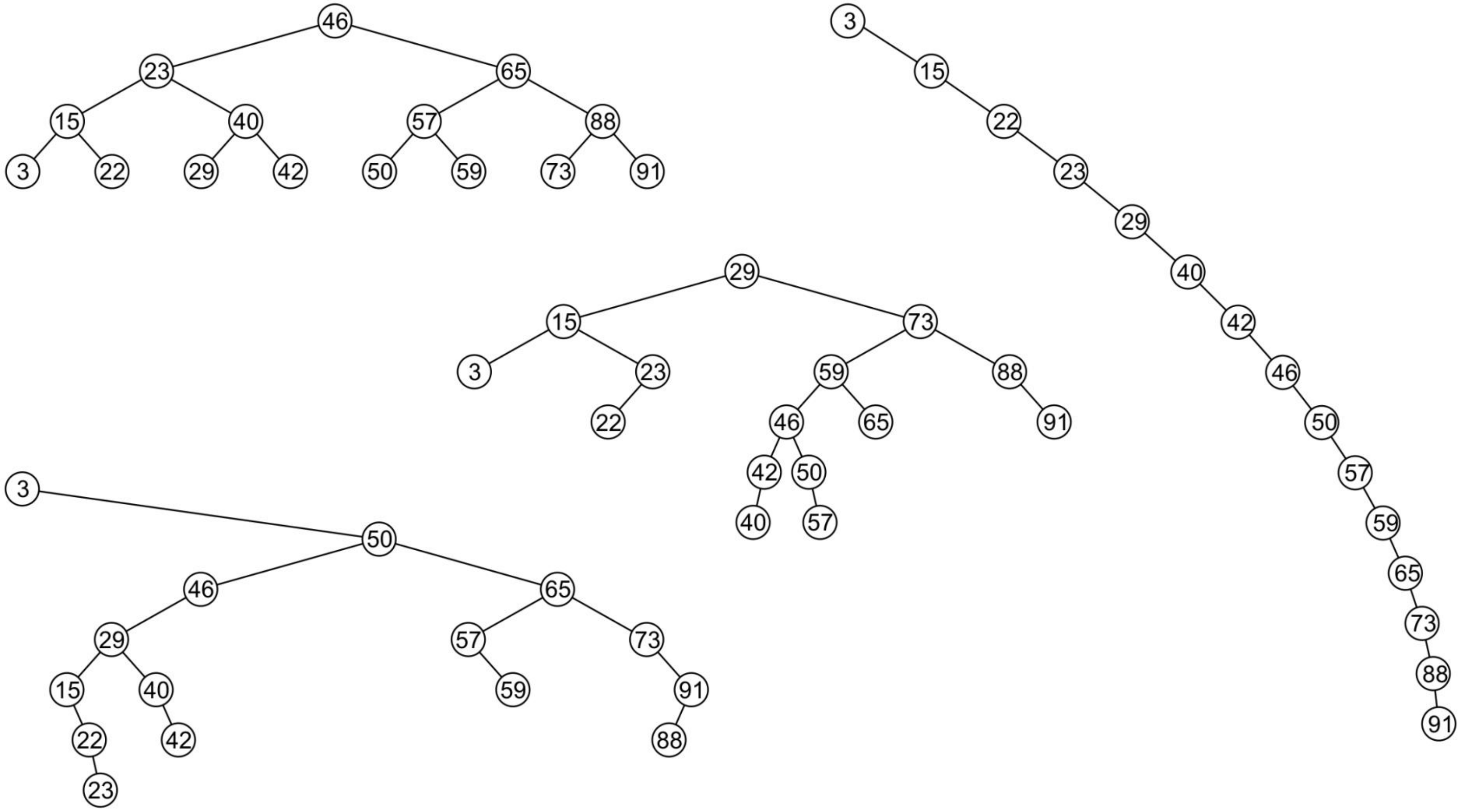
Unfortunately, it is possible to construct *unbalanced* binary search trees



– This is equivalent to a linked list, *i.e.*, $\mathbf{O}(n)$

Binary Search Trees: More Examples

All these binary search trees store the same data



Note: No Duplicate Values

We will assume that in any binary tree, we are not storing duplicate values unless otherwise stated

You can always consider duplicate values with modifications to the algorithms we will cover

Implementation: Binary Search Trees

Design with two classes:

1) **BinaryNode**
   - Represent each node in the tree

2) **BinarySearchTree**
   - Represent the tree, which holds the root node (an instance of BinaryNode)

Implementation: class BinaryNode

```
template <typename T>
struct BinaryNode {
    T value;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode<T>(const T &value, BinaryNode<T> *left, BinaryNode<T> *right)
        : value{value}, left{left}, right{right} {}

    BinaryNode<T>(T &&value, BinaryNode<T> *left, BinaryNode<T> *right)
        : value{std::move(value)}, left{left}, right{right} {}
};
```

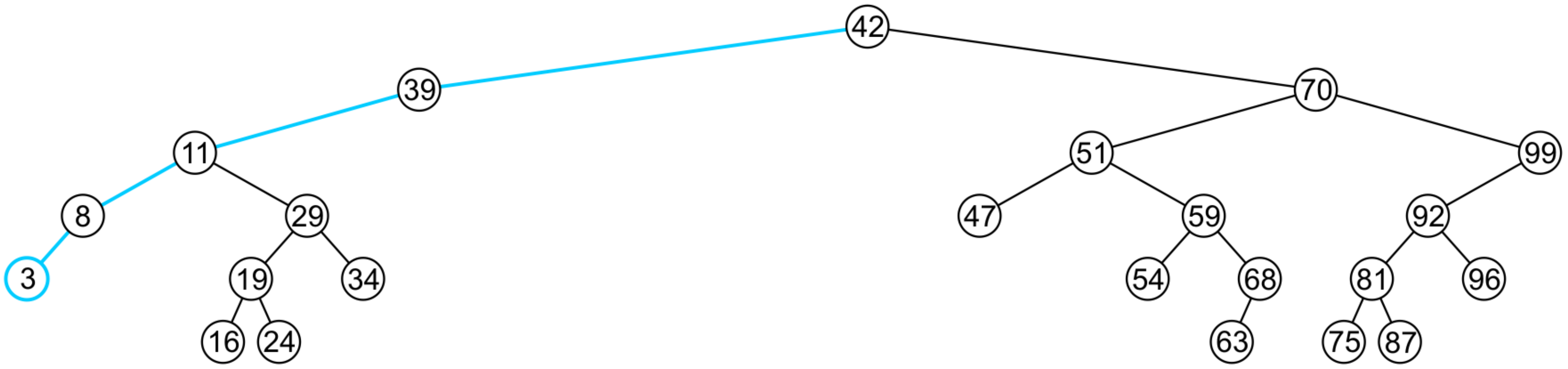
- A value has a template based type <T>
- If <T> is not comparable, you will need to override comparison operators

Implementation: class BinarySearchTree

```
template <typename T>
class BinarySearchTree {
public:
   BinarySearchTree() : root{nullptr} {}

   const T &findMin() const;
   const T &findMax() const;

   bool find(const T &x) const;
   void insert(const T &x);
   void remove(const T &x);
   // something more …

private:
   BinaryNode<T> *root;
   // something more …
};
```

Finding the Minimum Object

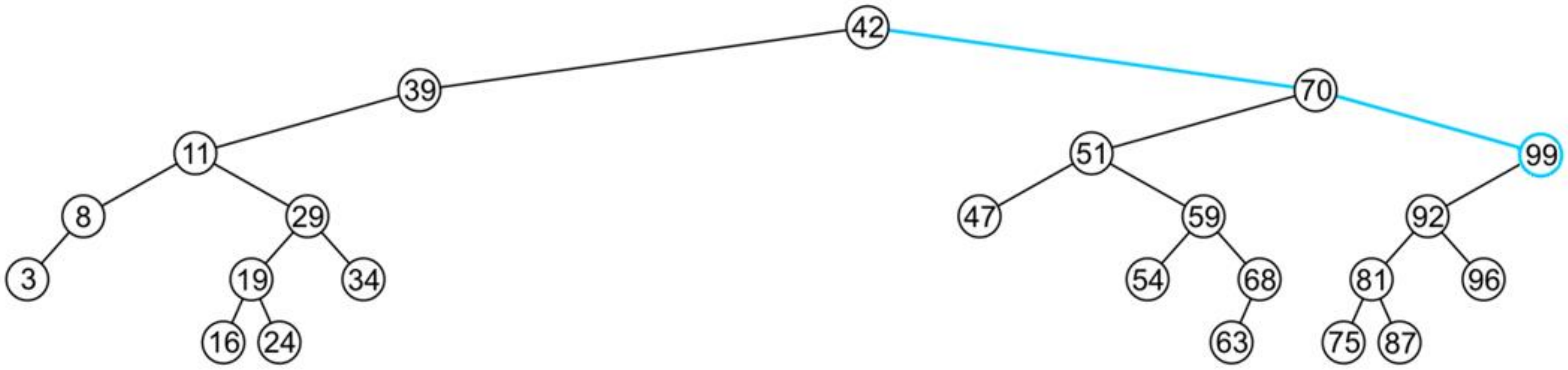
```
const T &findMin() const {
    if (isEmpty())
        throw std::exception{};
    return findMin(root)->value;
}


BinaryNode<T> *findMin(BinaryNode<T> *t) const {
    if (t == nullptr)
        return nullptr;
    if (t->left == nullptr)
        return t;
    return findMin(t->left);
}
```



– **The run time: O(*h*)**

Finding the Maximum Object

```
const T &findMax() const {
    if (isEmpty())
        throw std::exception{};
    return findMax(root)->value;
}

BinaryNode<T> *findMax(BinaryNode<T> *t) const {
    if (t != nullptr)
        while (t->right != nullptr)
            t = t->right;
    return t;
}
```

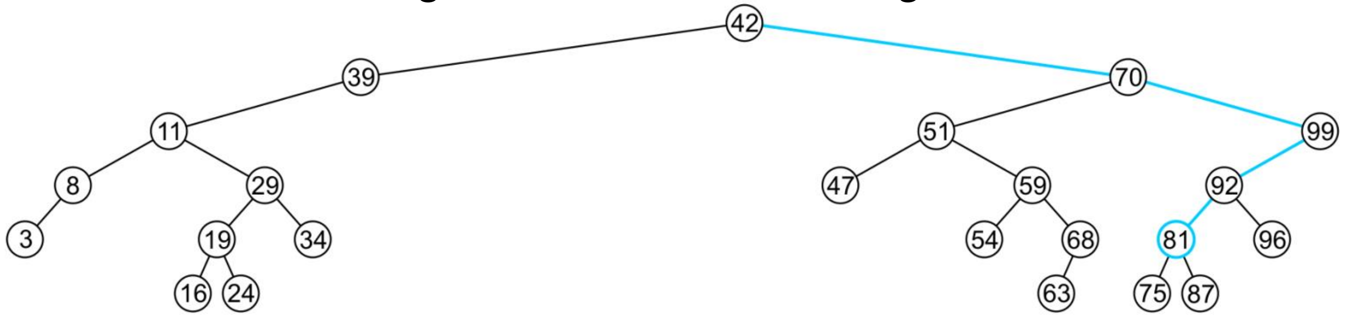


– minimum/maximum values are not necessarily leaf nodes

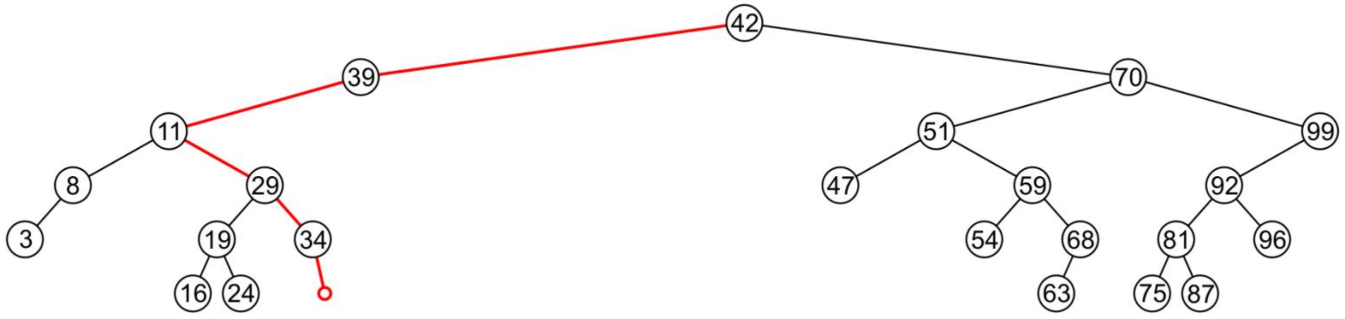
Find

To determine membership, traverse the tree based on the linear relationship:

- If a node containing the value is found, *e*.*g*., 81, return true



- If an empty node is reached, *e*.*g*., 36, return false:



Find

The implementation is similar to `findMin` and findMax:

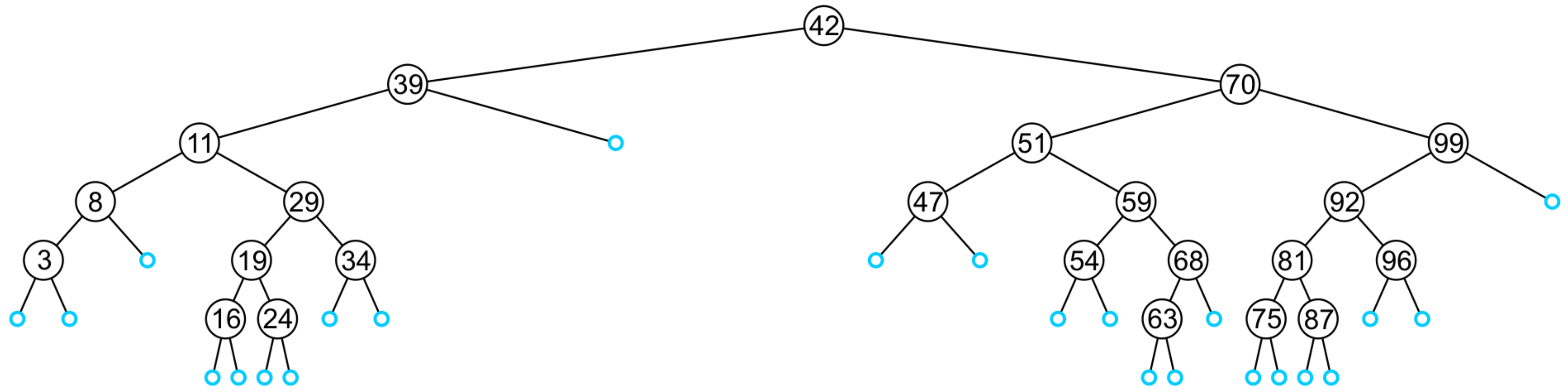
- The run time is **O**(h)

```
bool find(const T &x) const { return contains(x, root); }

bool find(const T &x, BinaryNode<T> *t) const {
    if (t == nullptr)
        return false;
    else if (x < t->value)
        return find(x, t->left);
    else if (t->value < x)
        return find(x, t->right);
    else
        return true; // Match
}
```

Insert

An insertion will be performed at an empty node:

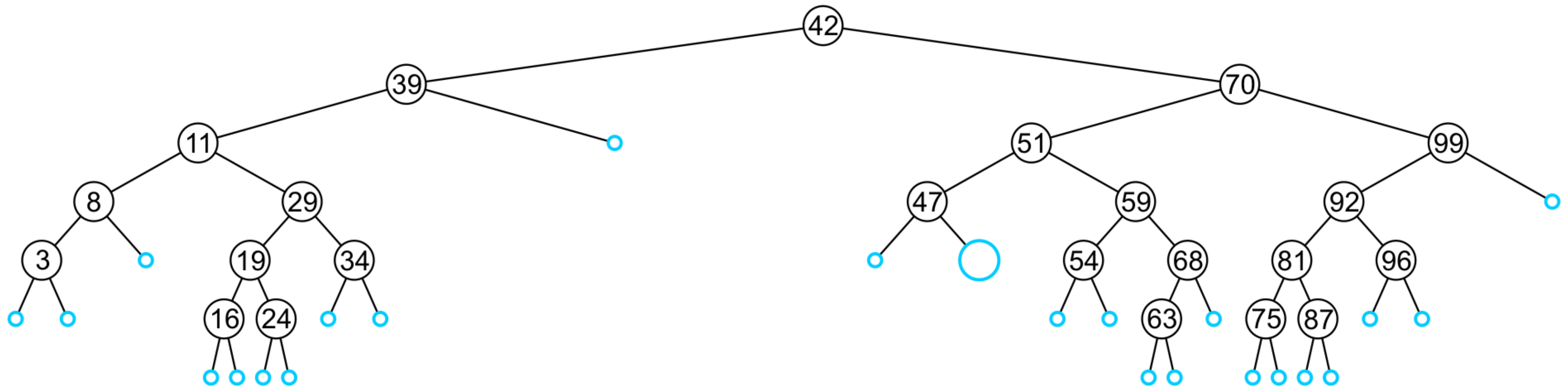
– Any empty node is a possible location for an insertion



The values which may be inserted at any empty node depend on the surrounding nodes

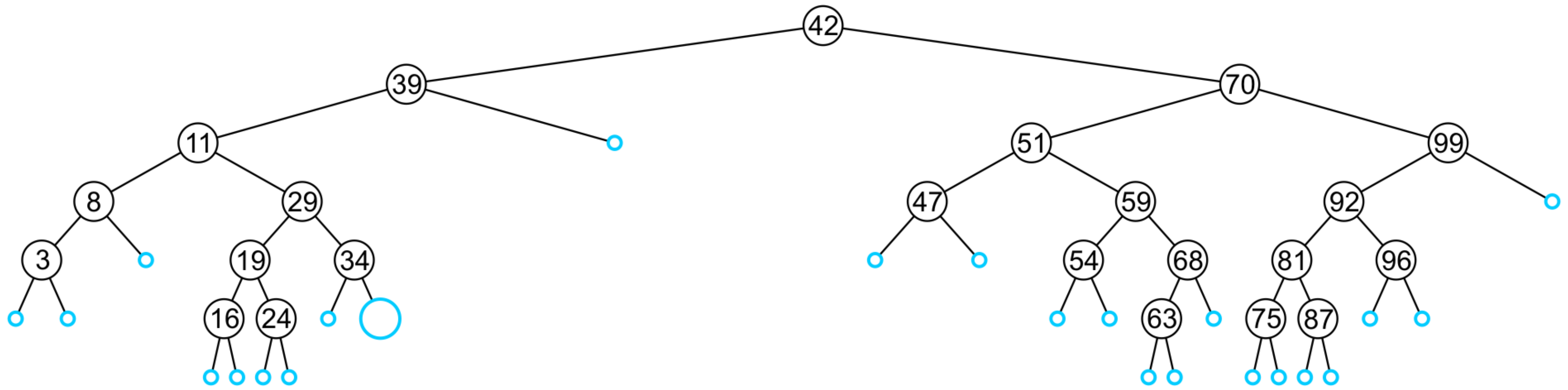
Insert

For example, this node may hold 48, 49, or 50



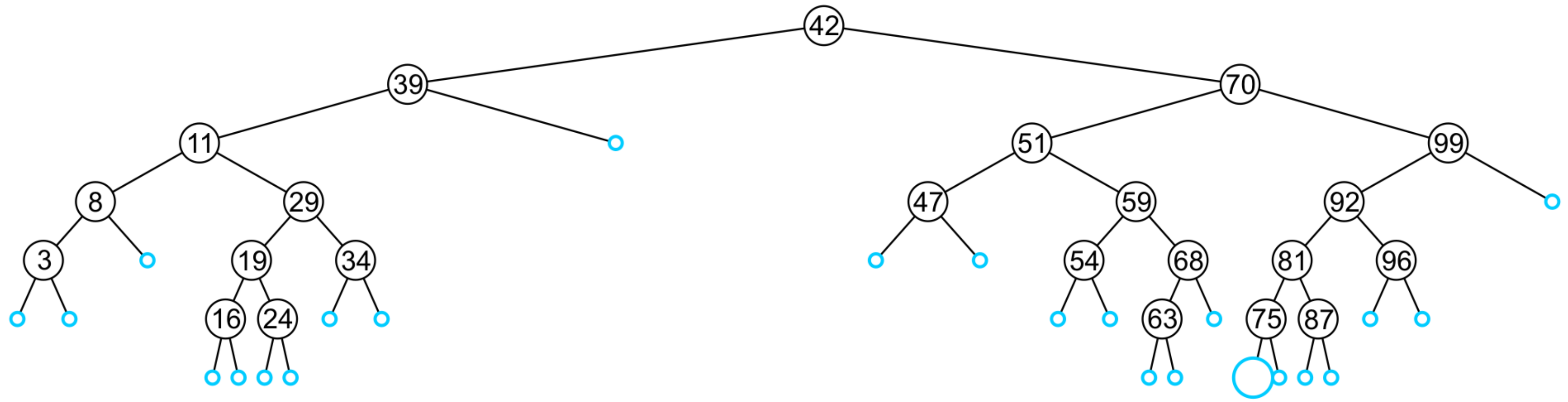
Insert

An insertion at this location must be 35, 36, 37, or 38



Insert

This empty node may hold values from 71 to 74



Insert

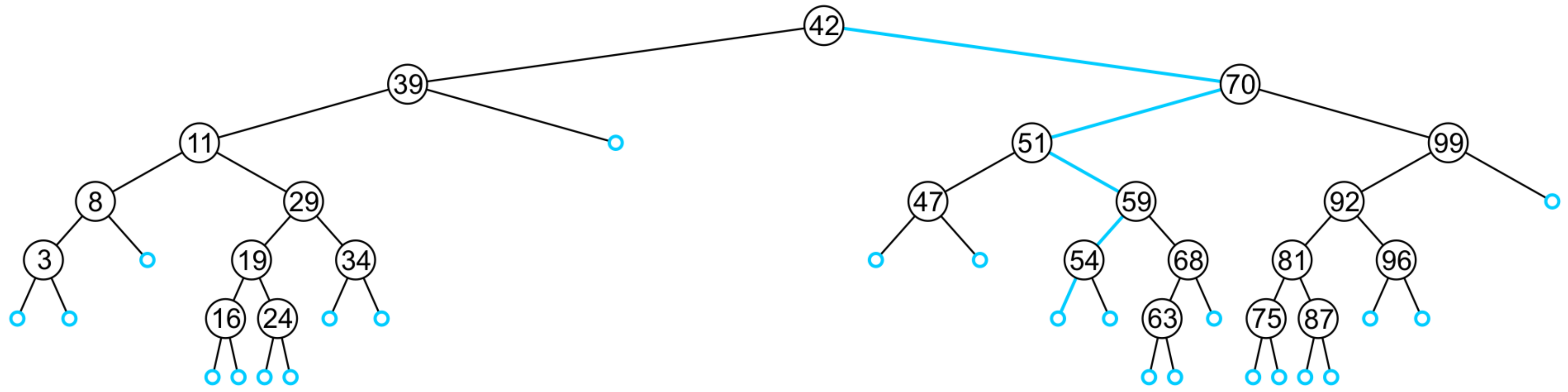
Like find, we will step through the tree

- If we find the object already in the tree, we will return
  - The object is already in the binary search tree (no duplicates)
- Otherwise, we will arrive at an empty node
- The object will be inserted into that location
- The run time is $\mathbf{O}(h)$

Insert 52

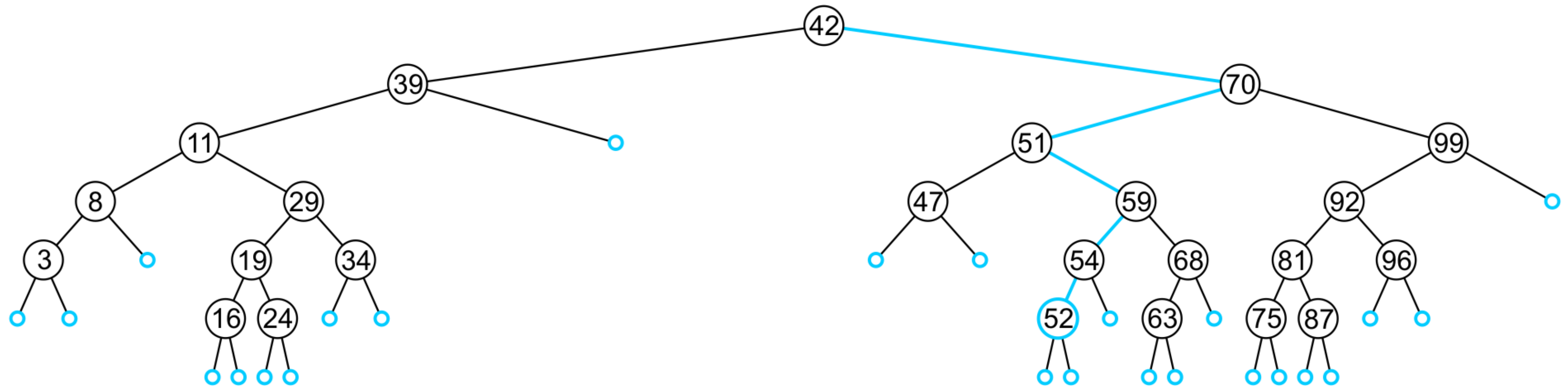
In inserting the value 52, we traverse the tree until we reach an empty node

- The left sub-tree of 54 is an empty node



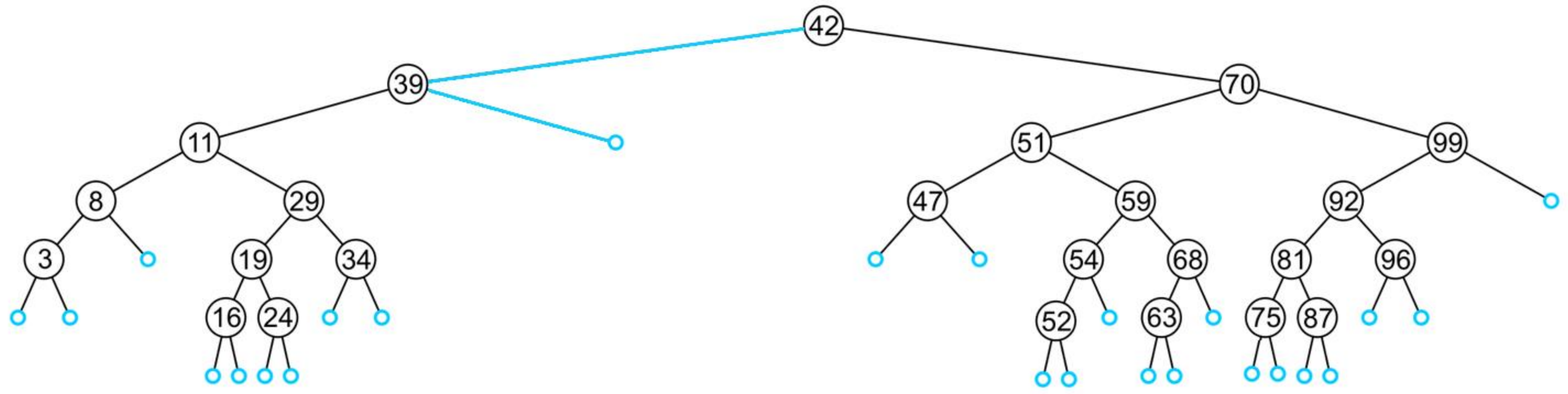
Insert 52

A new leaf node is created and assigned to the member variable `left`



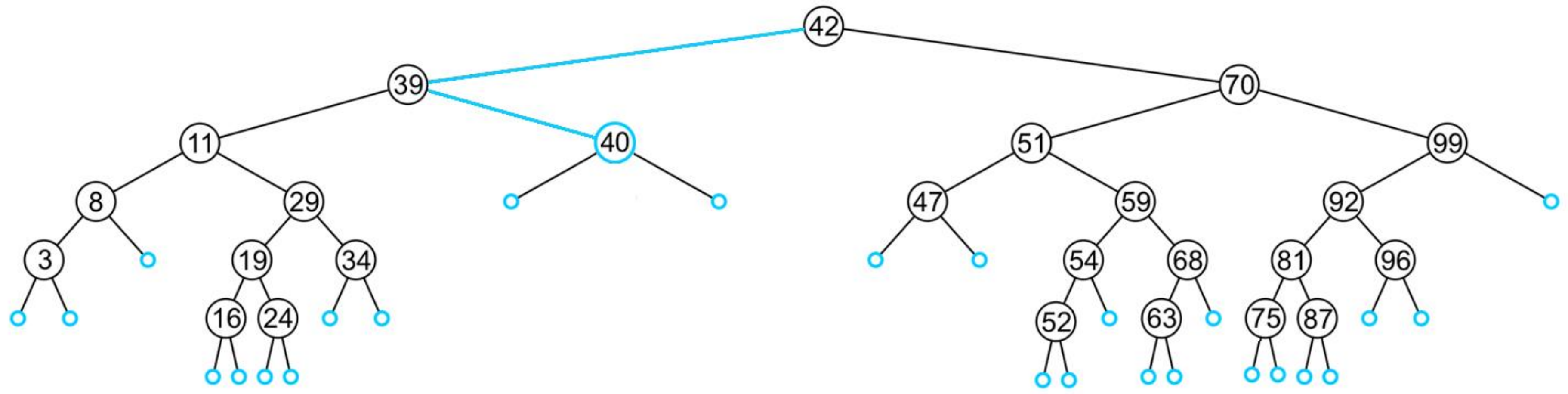
Insert 40

In inserting 40, we determine the right sub-tree of 39 is an empty node



Insert 40

A new leaf node storing 40 is created and assigned to the member variable `right`



Insert

```
    void insert(const T &x) { insert(x, root); }

    void insert(const T &x, BinaryNode<T> *&t) {
        if (t == nullptr)
            t = new BinaryNode<T>{x, nullptr, nullptr};
        else if (x < t->value)
            insert(x, t->left);
        else if (t->value < x)
            insert(x, t->right);
        else
            ; // Duplicate; do nothing
    }
```

Insert

Blackboard example:

- In the given order, insert these objects into an initially empty binary search tree:

  31 45 36 14 52 42 6 21 73 47 26 37 33 8

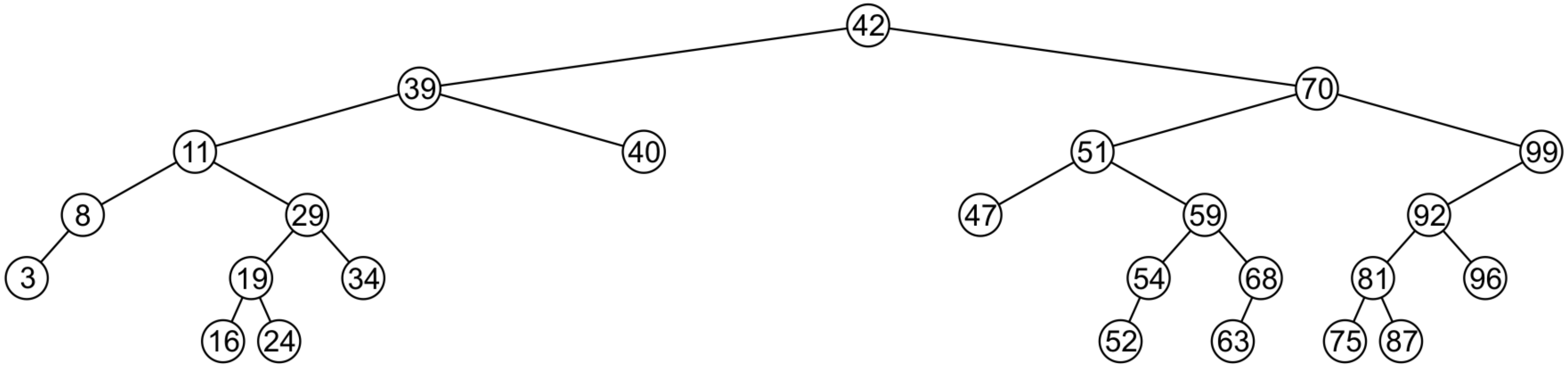
- What values could be placed:
  - To the left of 21?
  - To the right of 26?
  - To the left of 47?
- How would we determine if 40 is in this binary search tree?
- Which values could be inserted to increase the height of the tree?

Erase

A node being erased is not always going to be a leaf node

There are three possible scenarios:

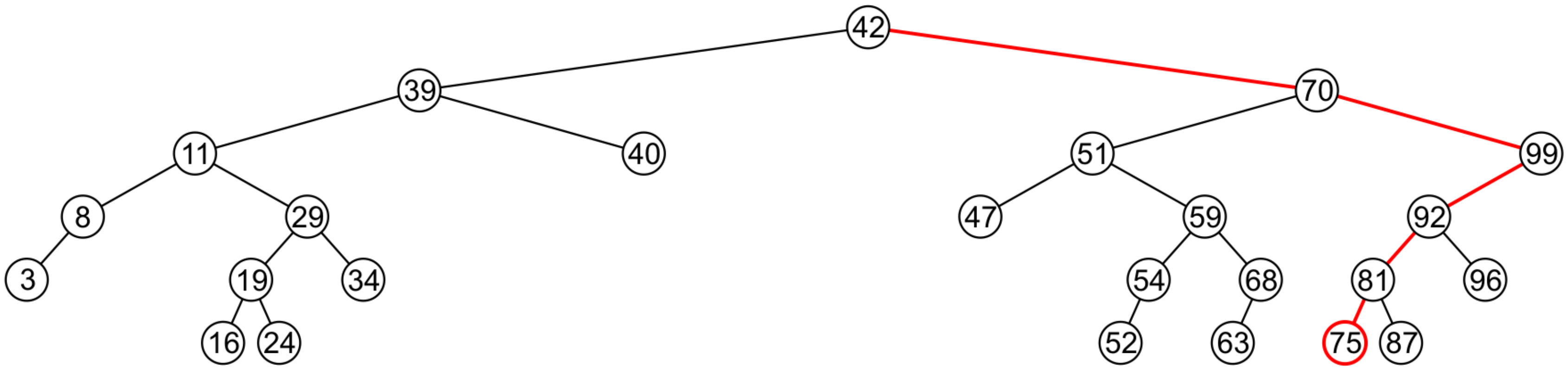
- Case #1: The node is a leaf node,
- Case #2: It has exactly one child, or
- Case #3: It has two children (it is a full node)



Erase: Case #1 - Leaf Node

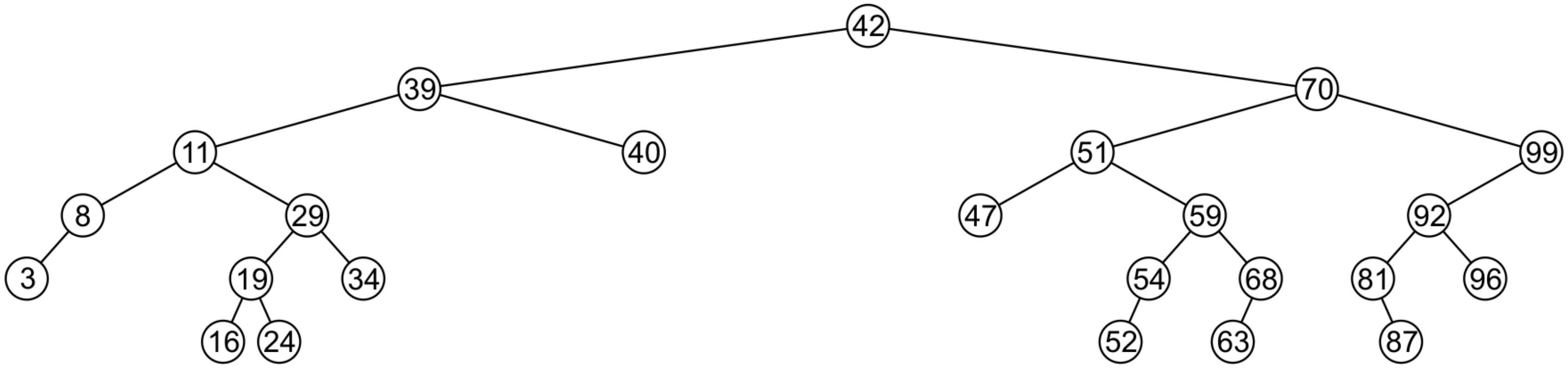
A leaf node must be removed and the appropriate member variable of the parent is set to `nullptr`

- Consider removing 75



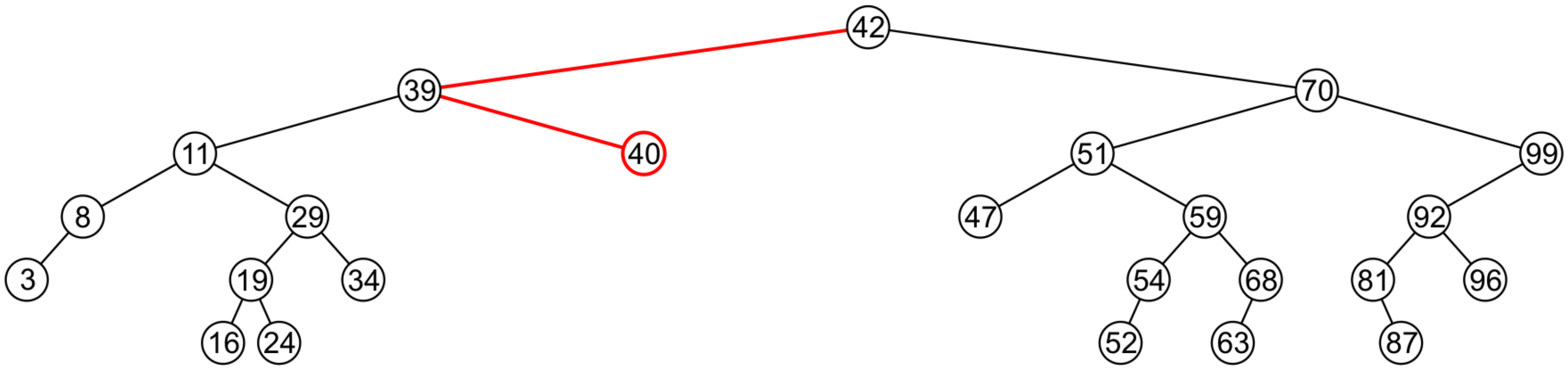
Erase: Case #1 - Leaf Node

The node is deleted and `left` of 81 is set to `nullptr`



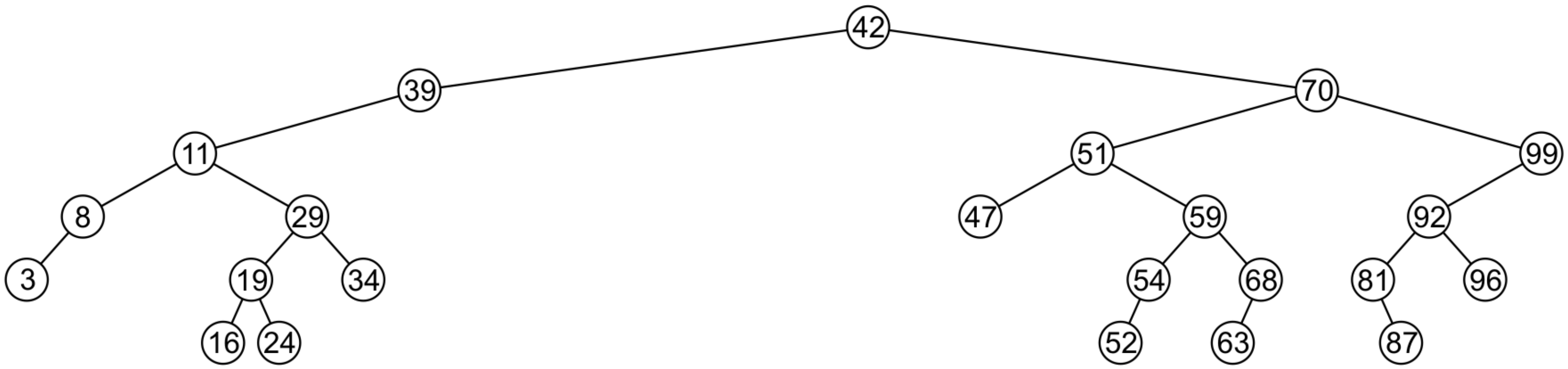
Erase: Case #1 - Leaf Node

Erasing the node containing 40 is similar



Erase: Case #1 - Leaf Node

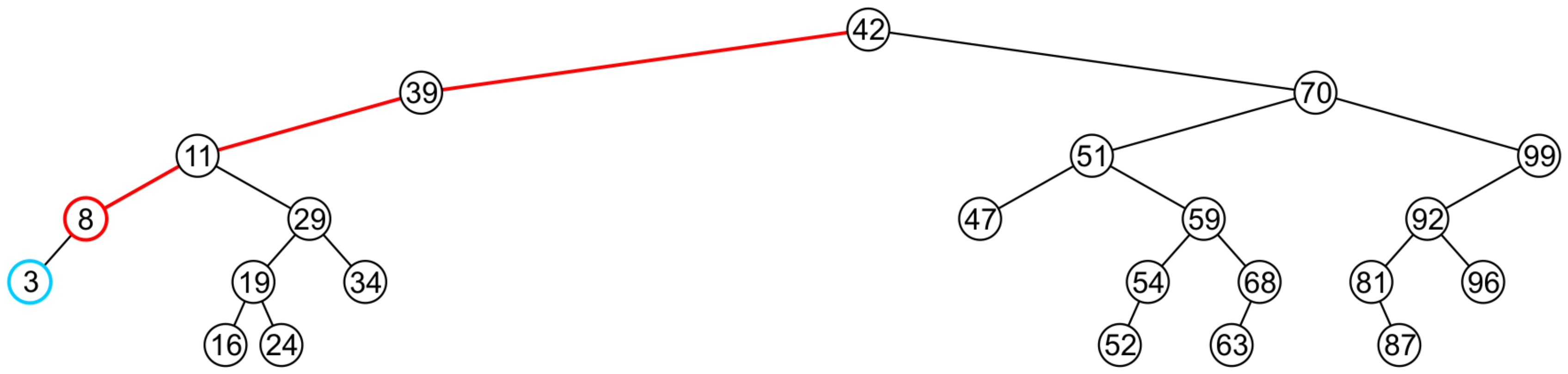
The node is deleted and `right` of 39 is set to `nullptr`



Erase: Case #2 - Non-leaf node w/ one child

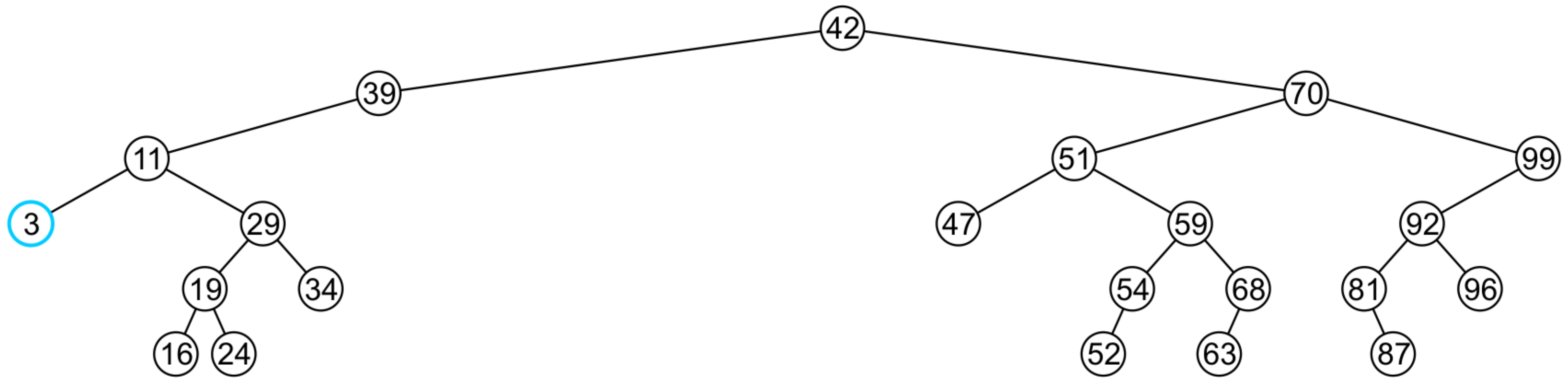
If a node has only one child, we can simply promote the sub-tree associated with the child

- Consider removing 8 which has one left child



Erase: Case #2 - Non-leaf node w/ one child

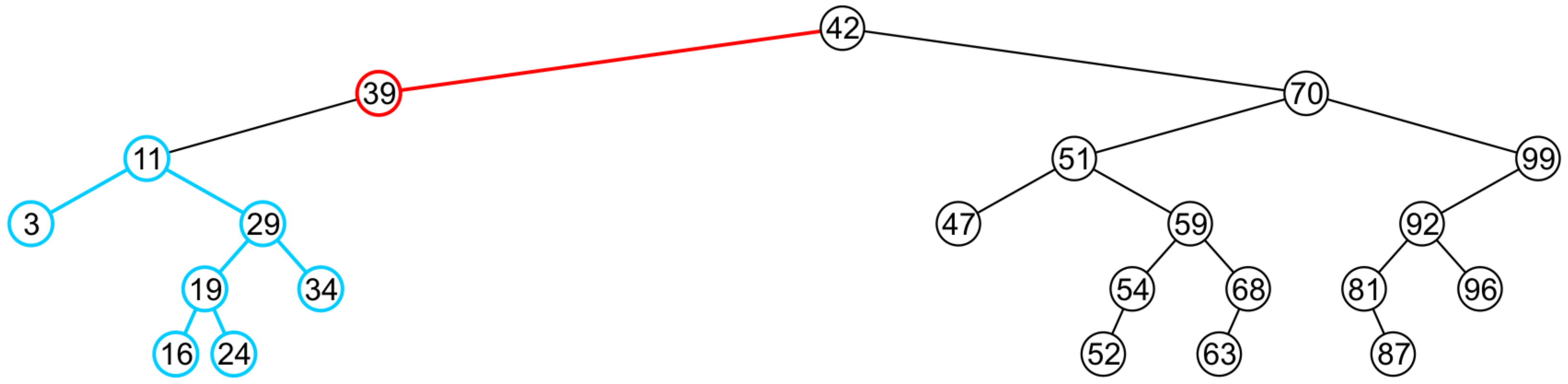
The node 8 is deleted and the `left` of 11 is updated to point to 3



Erase: Case #2 - Non-leaf node w/ one child

There is no difference in promoting a single node or a sub-tree

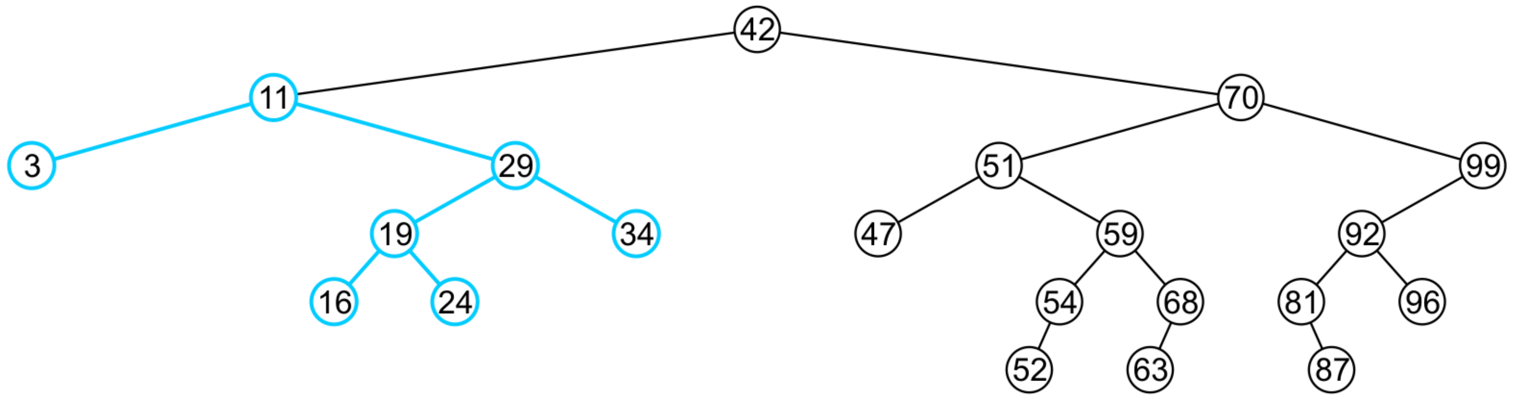
- To remove 39, it has a single child 11



Erase: Case #2 - Non-leaf node w/ one child

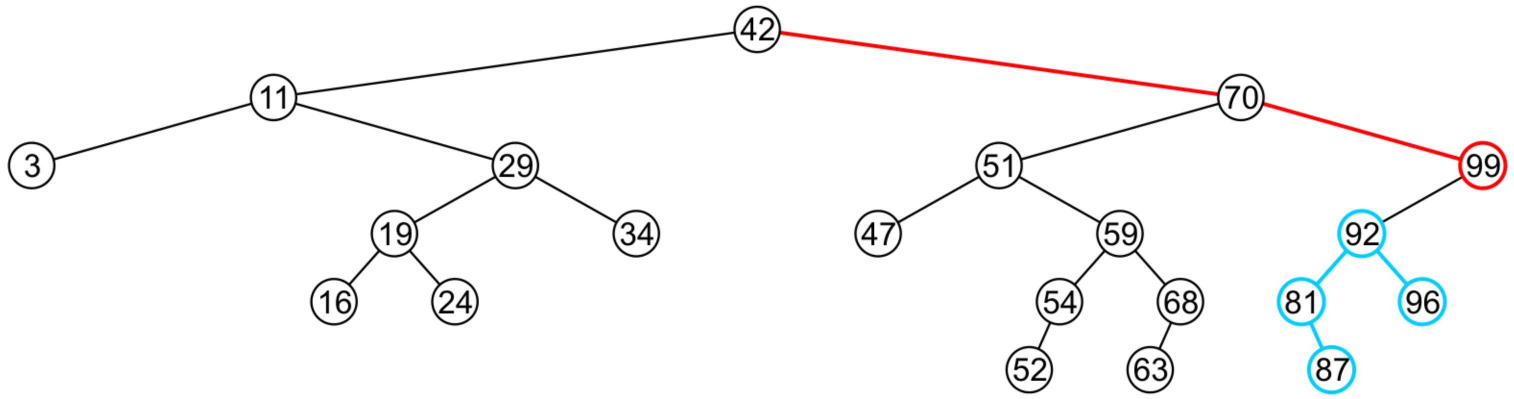
The node containing 39 is deleted and `left` of 42 is updated to point to 11

- Notice that order is still maintained



Erase: Case #2 - Non-leaf node w/ one child

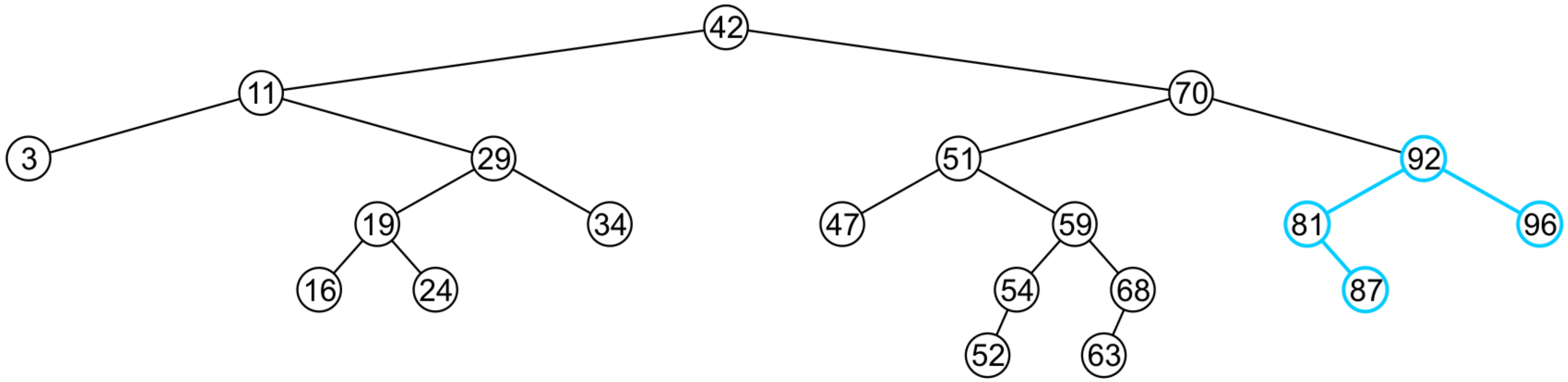
Consider erasing the node containing 99



Erase: Case #2 - Non-leaf node w/ one child

The node is deleted and the left sub-tree is promoted:

- The member variable `right` of 70 is set to point to 92
- Again, the order of the tree is maintained

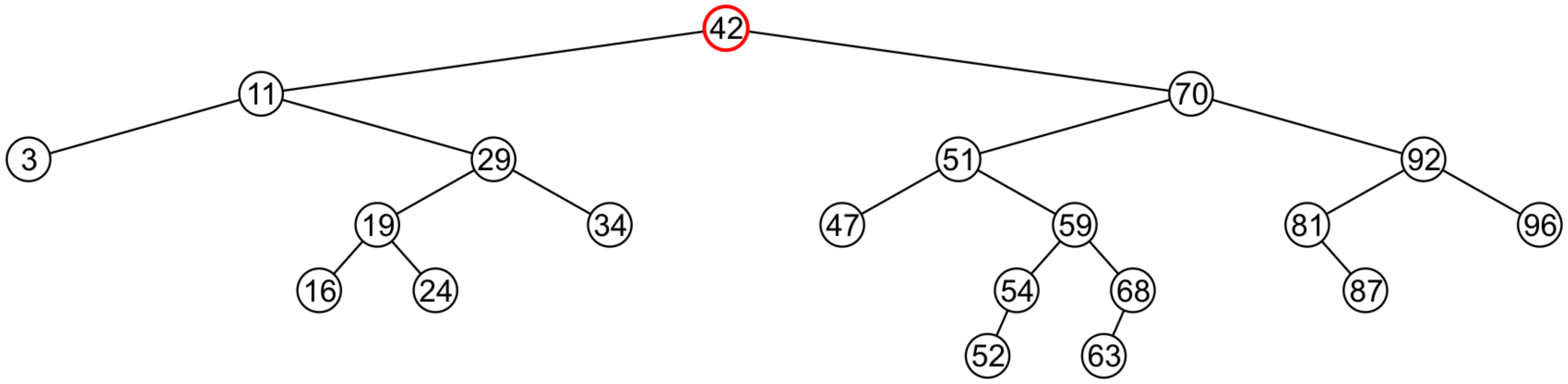


Erase: Case #3 - Full node

To delete a full node (*e.g.*, 42),

we will perform two operations:

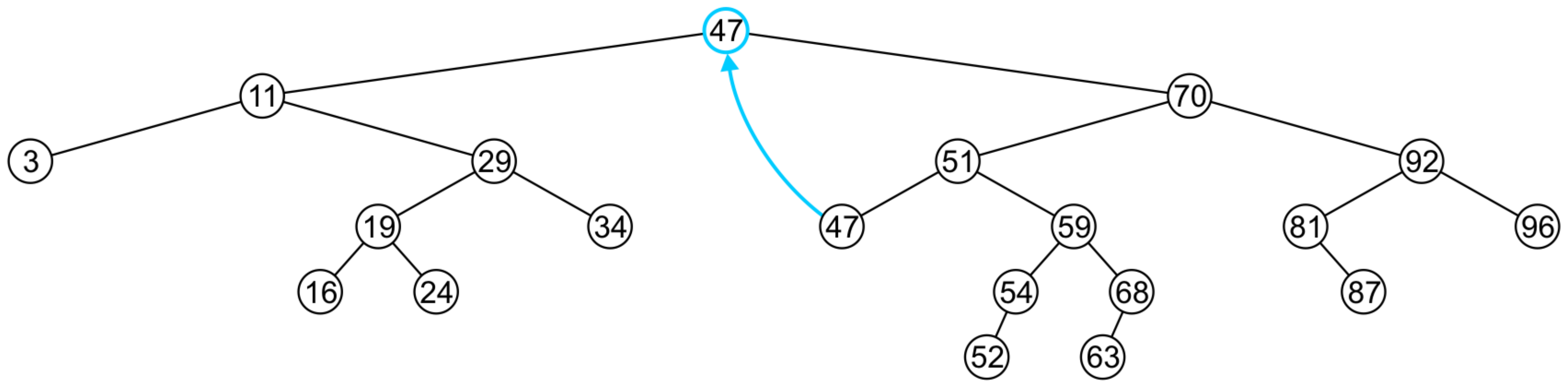
- Replace 42 with the minimum object in the right sub-tree
- Erase that object from the right sub-tree



Erase: Case #3 - Full node

In this case, we replace 42 with 47

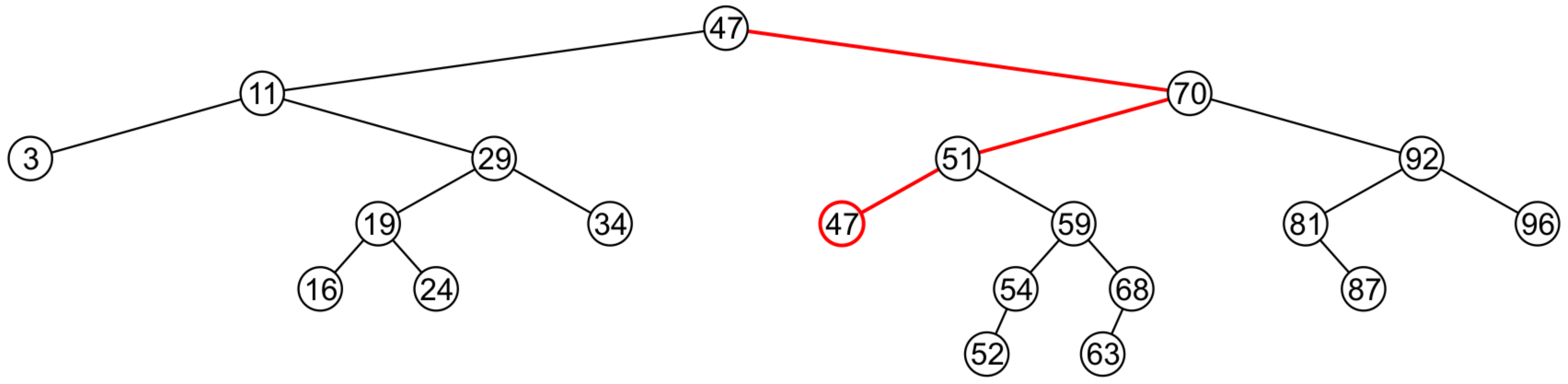
- We temporarily have two copies of 47 in the tree



Erase: Case #3 - Full node

We now recursively erase 47 from the right sub-tree

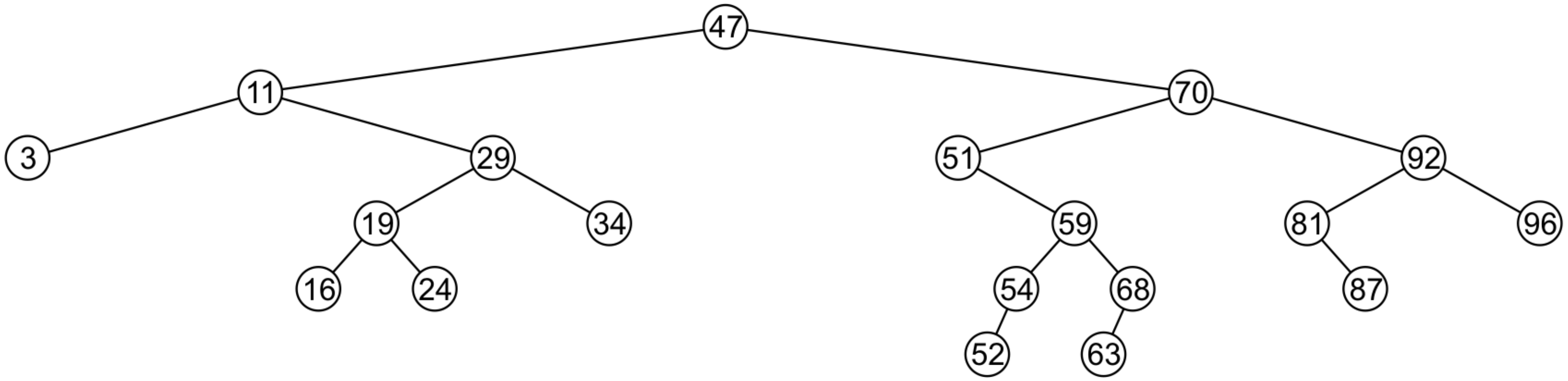
- We note that 47 is a leaf node in the right sub-tree



Erase: Case #3 - Full node

Leaf nodes are simply removed and `left` of 51 is set to `nullptr`

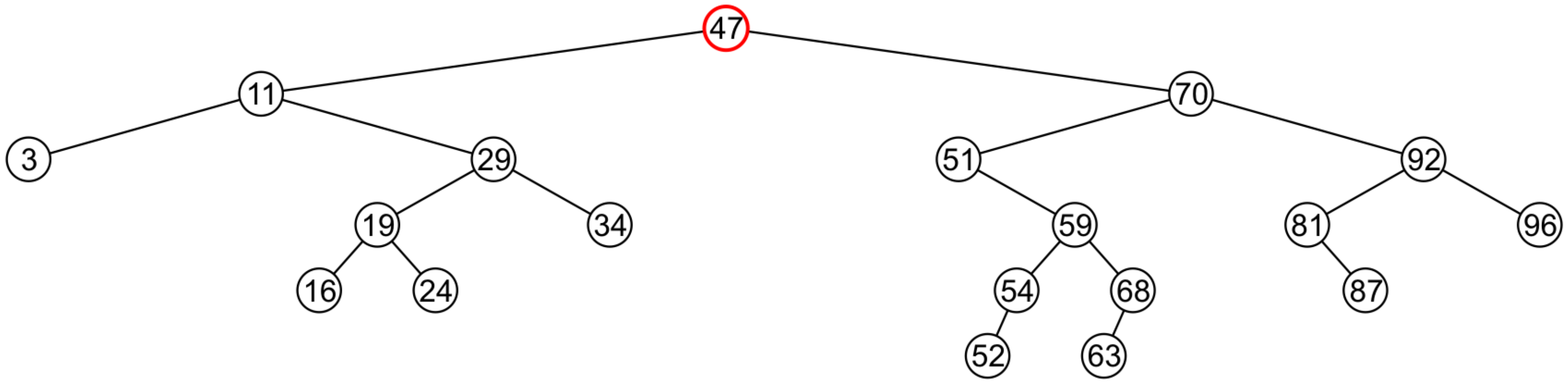
– Notice that the tree is still sorted



Erase: Case #3 - Full node

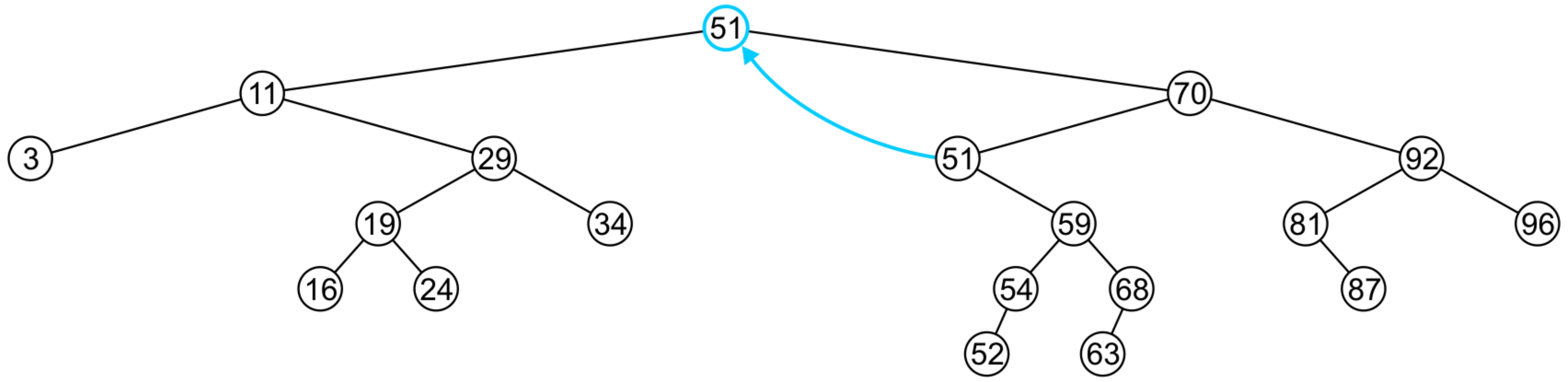
Suppose we want to erase the root 47 again:

- We must copy the minimum of the right sub-tree
- We could promote the maximum object in the left sub-tree and achieve similar results



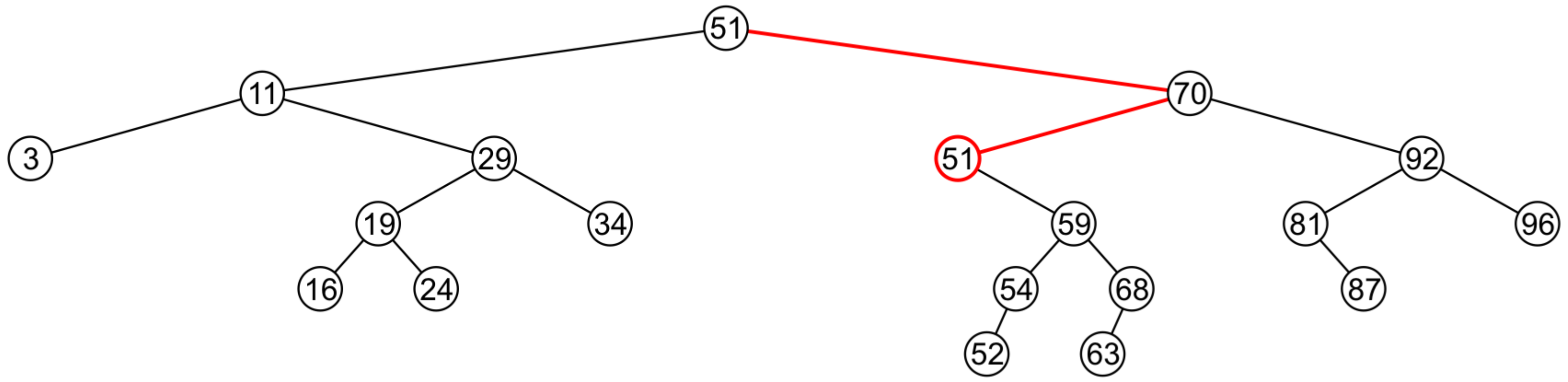
Erase: Case #3 - Full node

We copy 51 from the right sub-tree



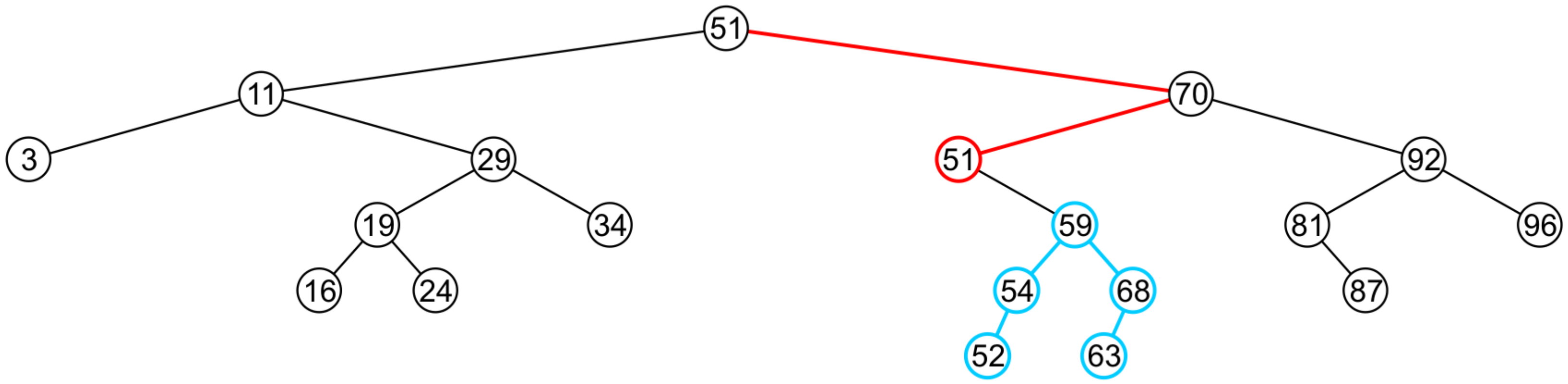
Erase: Case #3 - Full node

We must proceed by delete 51 from the right sub-tree



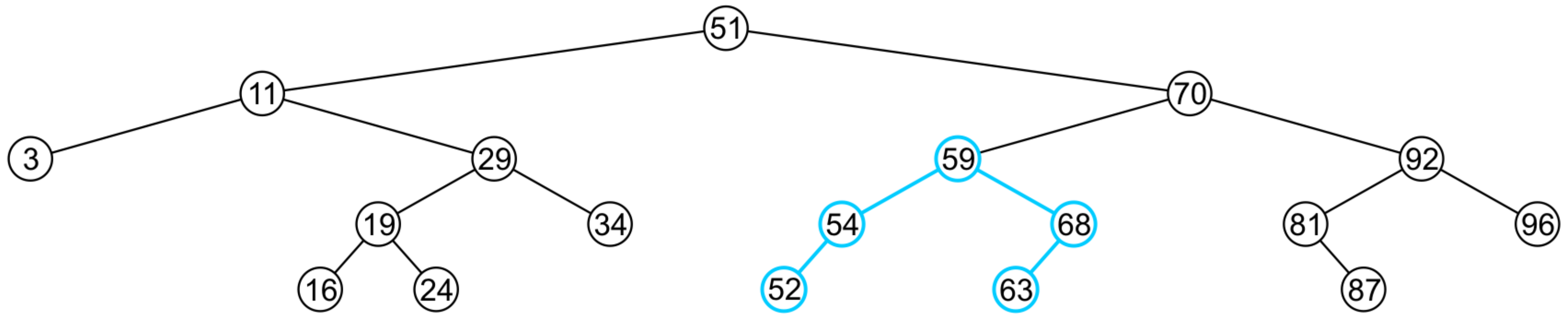
Erase: Case #3 - Full node

In this case, the node storing 51 has just a single child



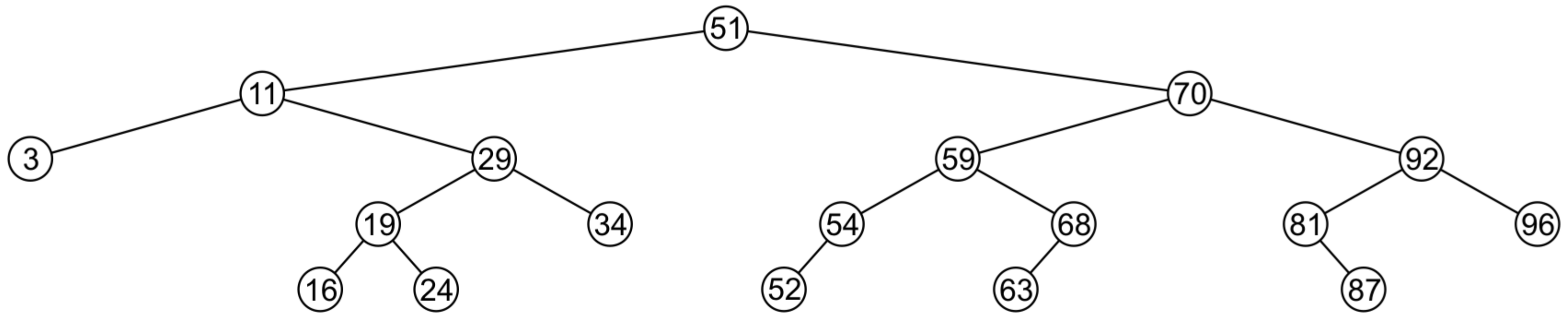
Erase: Case #3 - Full node

We delete the node containing 51 and assign the member variable `left` of 70 to point to 59



Erase: Case #3 - Full node

Note that after seven removals, the remaining tree is still correctly sorted



Erase: Case #3 - Full node

```
void remove(const T &x) { remove(x, root); }

void remove(const T &x, BinaryNode<T> *&t) {
    if (t == nullptr)
        return; // Item not found; do nothing
    if (x < t->value)
        remove(x, t->left);
    else if (t->value < x)
        remove(x, t->right);
    else if (t->left != nullptr && t->right != nullptr) { // two children
            t->value = findMin(t->right)->value;
            remove(t->value, t->right);
    }
    else { // single child
        BinaryNode<T> *oldNode = t;
        t = (t->left != nullptr) ? t->left : t->right;
        delete oldNode;
    }
}
```

Erase: Case #3 - Full node

Blackboard example:

- In the binary search tree generated previously:
    - Erase 47
    - Erase 21
    - Erase 45
    - Erase 31
    - Erase 36

Other Relation-based Operations

We will quickly consider two other relation-based queries that are very quick to calculate with an array of sorted objects:

- Finding the previous and next values, and
- Finding the k^{th} value

Previous and Next Objects

All the operations up to now have been operations which work on any container: count, insert, *etc*.

- If these are the only relevant operations, use a *hash table*

Operations specific to linearly ordered data include:

- Find the next larger and previous smaller objects of a given object which may or may not be in the container
- Find the kth value of the container
- Iterate through those objects that fall on an interval $[a, b]$

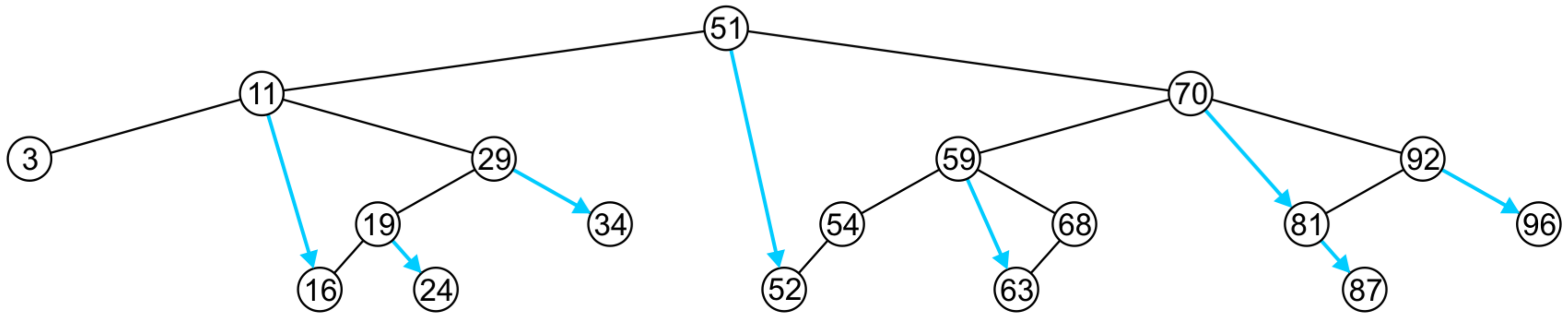
We will focus on finding the next largest object

- The others will follow

Previous and Next Objects

To find the next largest object:

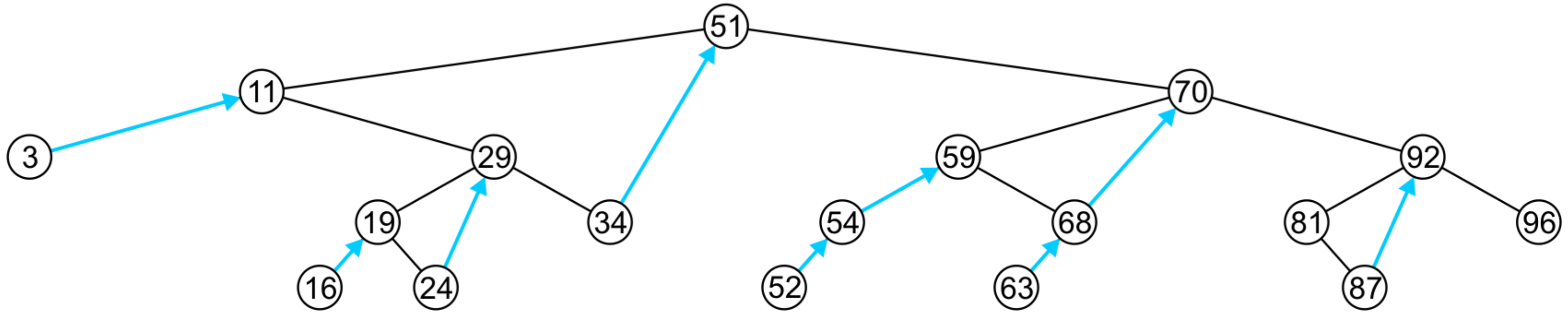
– If the node has a right sub-tree, the minimum object in that sub-tree is the next-largest object



Previous and Next Objects

If, however, there is no right sub-tree:

- It is the next largest object (if any) that exists in the path from the root to the node



Previous and Next Objects

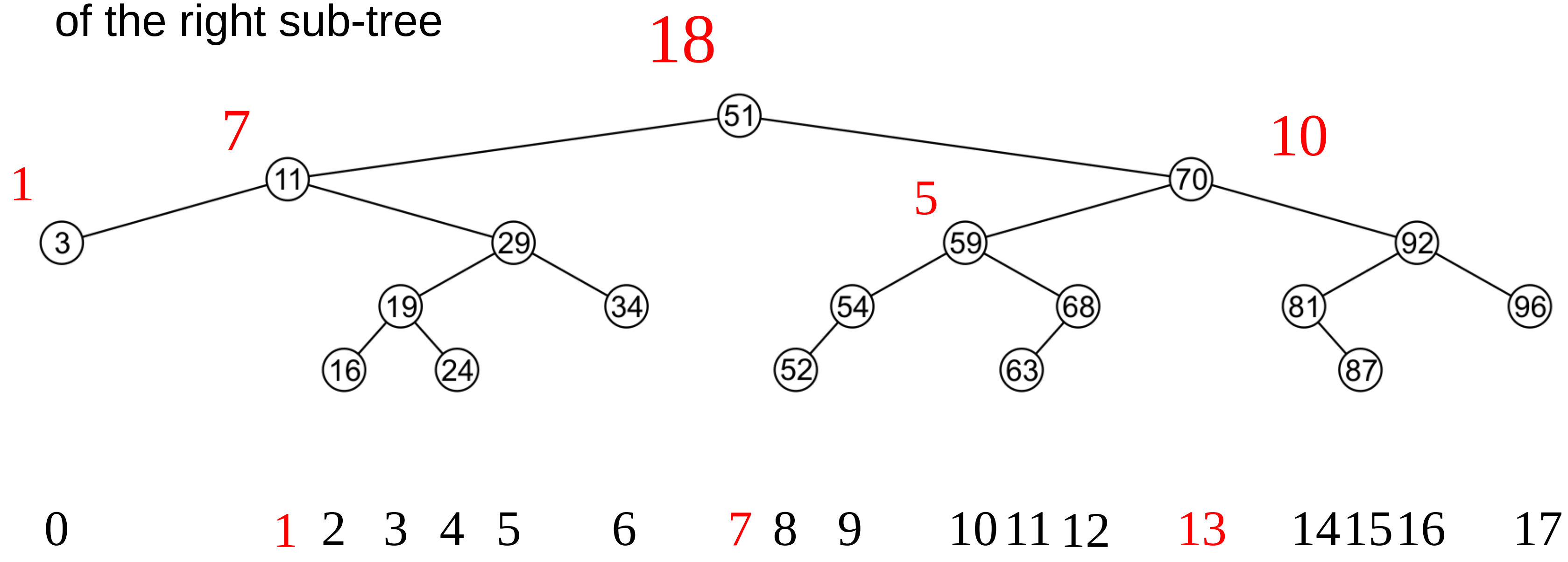
More generally: what is the next largest value of an arbitrary object?

- This can be found with a single search from the root node to one of the leaves—an $\mathrm{O}(h)$ operation
- This function returns the object if it did not find something greater than it

Finding the kth Object

Another operation on sorted lists may be finding the kth largest object

- Recall that k goes from 0 to $n – 1$
- If the left-sub-tree has $\ell = k$ values, return the current node,
- If the left sub-tree has $\ell > k$ values, return the kth value of the left sub-tree,
- Otherwise, the left sub-tree has $\ell < k$ values, so return the $(k - \ell - 1)$th value of the right sub-tree



Run Time: **O(*h*)**

Almost all relevant operations on a binary search tree are $\mathrm{O}(h)$

- If the tree is *close* to a linked list, the run times is $\mathrm{O}(n)$
  - Insert 1, 2, 3, 4, 5, 6, 7, ..., n into a empty binary search tree
- The best we can do is if the tree is perfect: $\mathrm{O}(\ln(n))$
- Our goal will be to find tree structures where we can maintain a height of $\Theta(\ln(n))$

We will look at

- AVL trees
- B+ trees

both of which ensure that the height remains $\Theta(\ln(n))$

Summary

In this topic, we covered binary search trees

- Described Abstract Sorted Lists
- Problems using arrays and linked lists
- Definition a binary search tree
- Looked at the implementation of:
  - Empty, size, height, count
  - FindMin, FindMax, insert, erase
  - Previous smaller and next larger objects