CSE 203: Binary Search Tree

Dr. Mohammed Eunus Ali

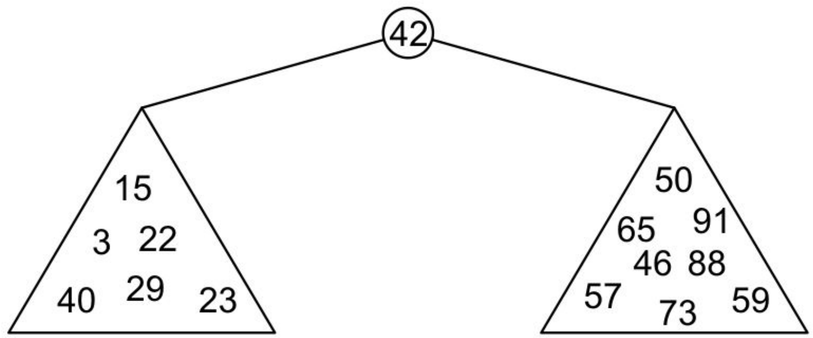
Professor

CSE, BUET

Some Slides are from Douglas Wilhelm Harder, **dwharder@alumni.uwaterloo.ca**

Binary Search Trees

Graphically, we may relationship



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

Binary Search Trees

Notice that we can already use this structure for searching: 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

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

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

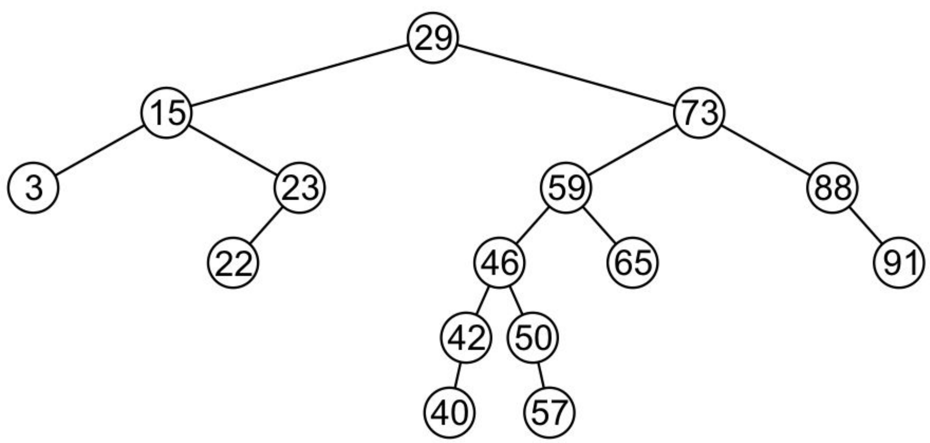
Definition

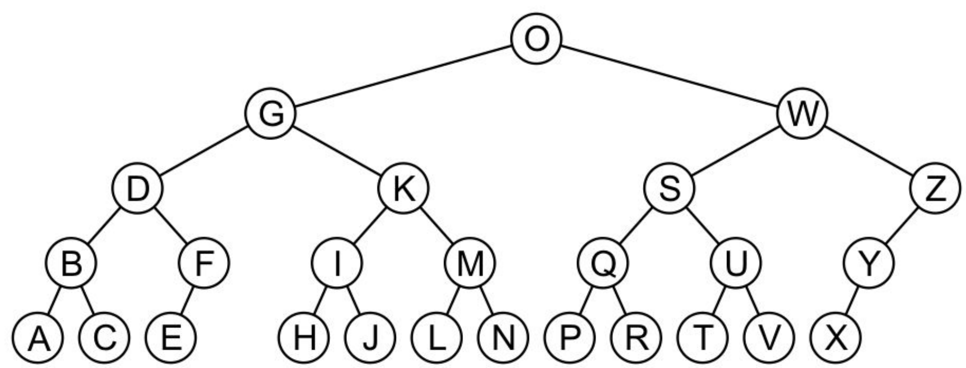
Thus, we define a non-empty binary search tree as a binary tree with the following properties:

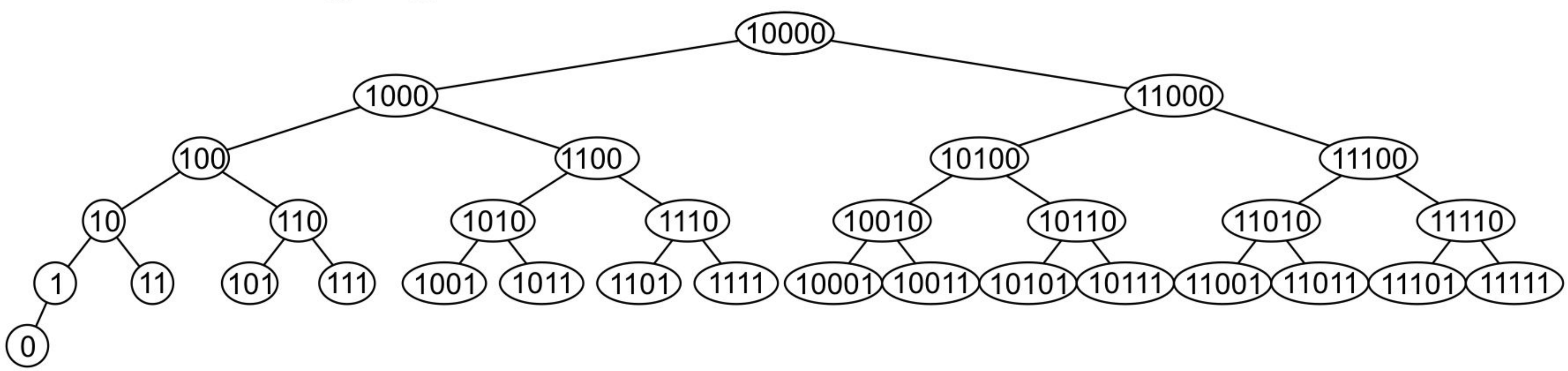
- The left sub-tree (if any) is a binary search tree and all values are less than the root value, and
- The right sub-tree (if any) is a binary search tree and all values are greater than the root value

Examples

Here are other examples of binary search trees:

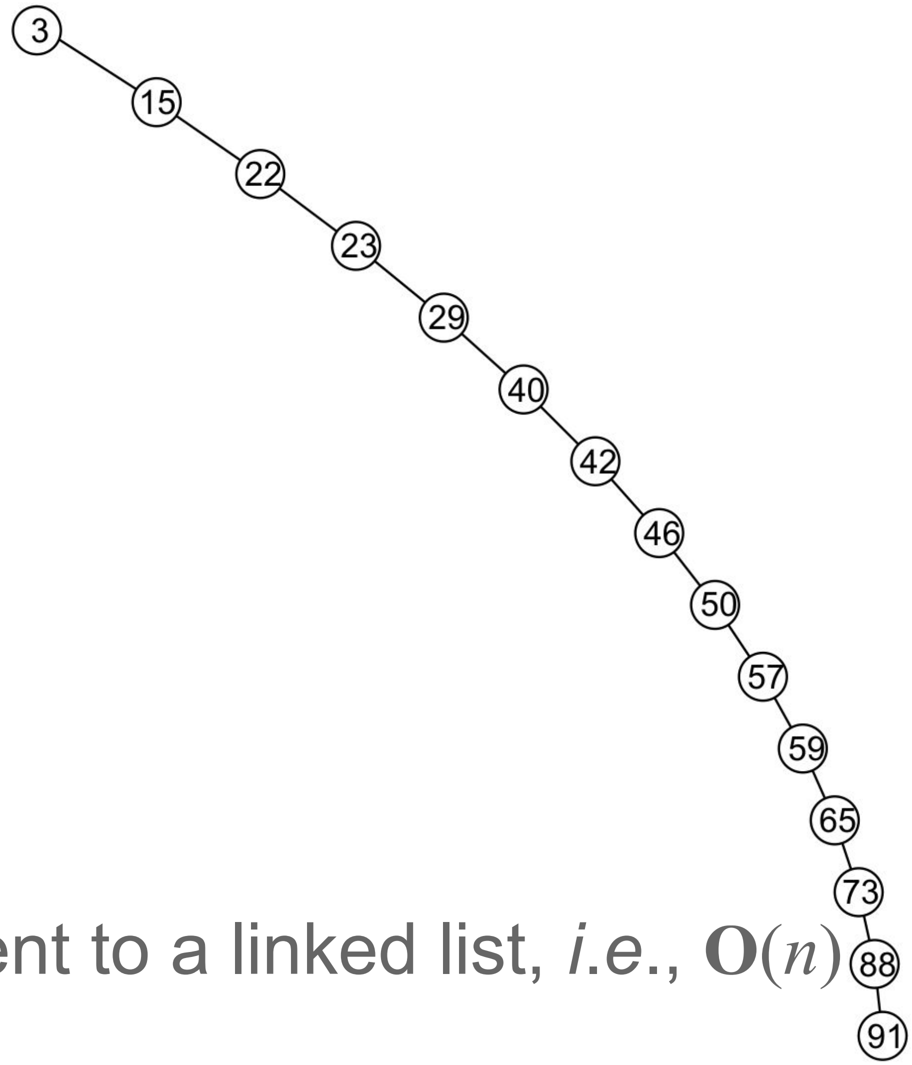






Examples

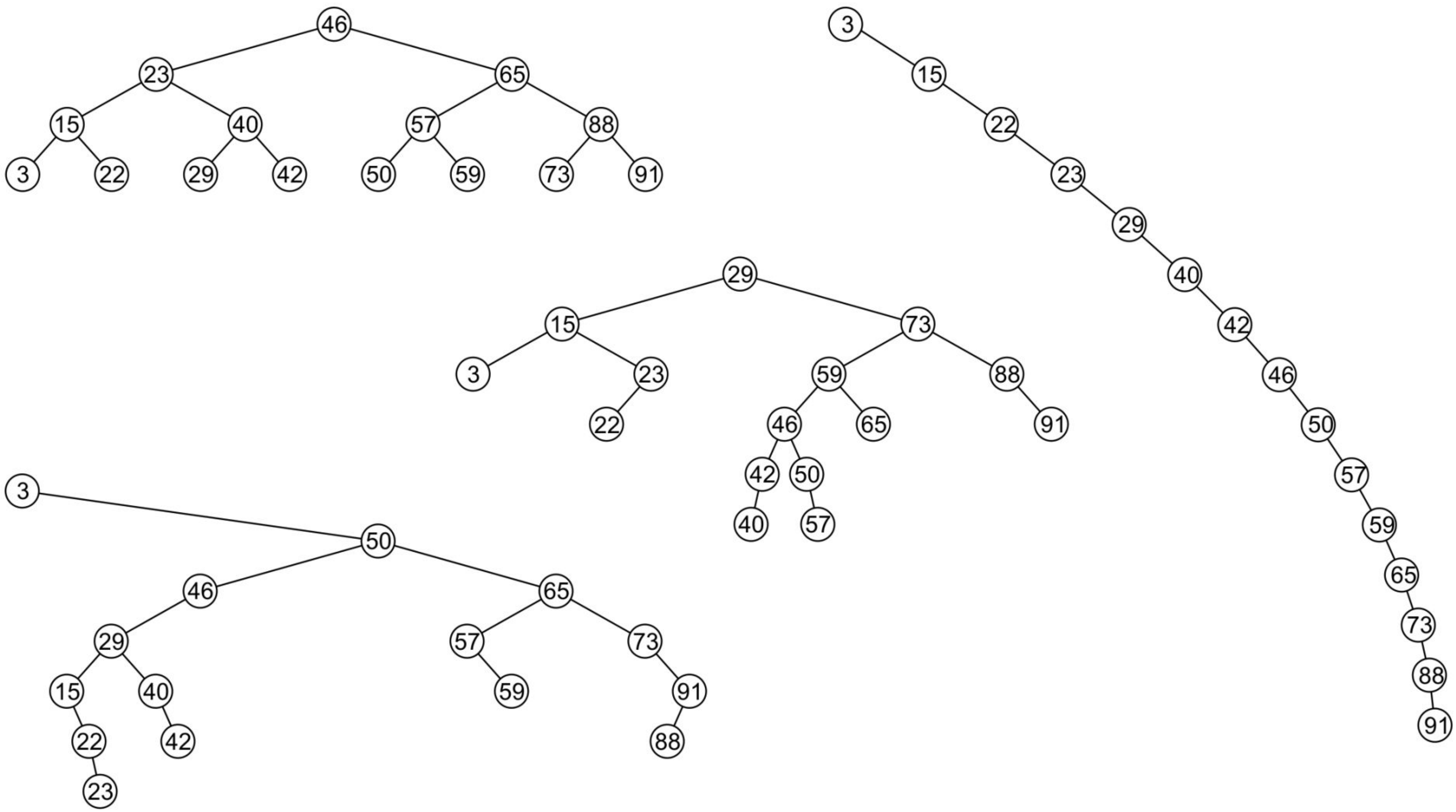
Unfortunately, it is possible to construct *degenerate* binary search trees



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

Examples

All these binary search trees store the same data



Duplicate values

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

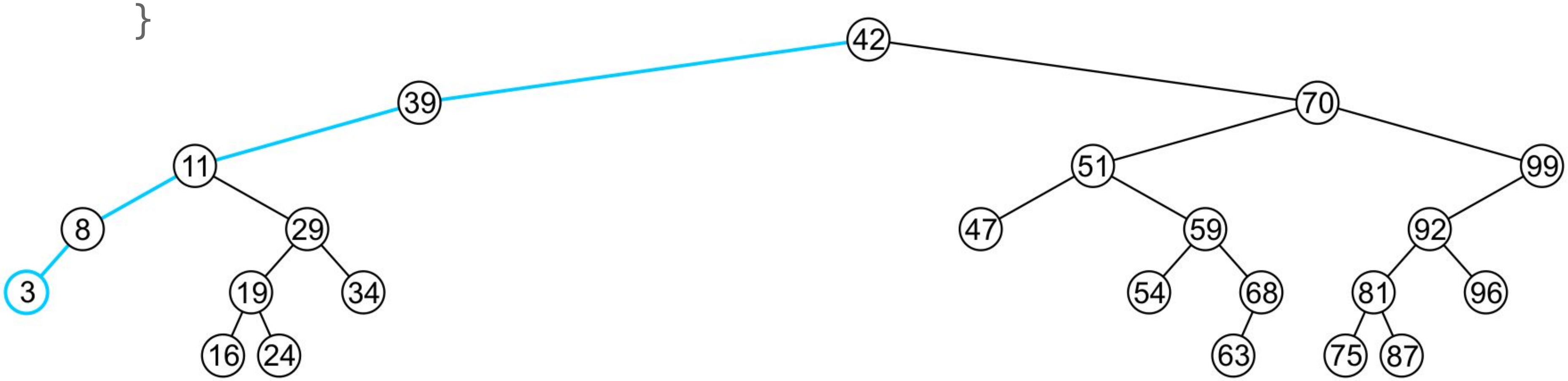
- In reality, it is seldom the case where duplicate values in a container must be stored as separate entities

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

Examples

Finding the Minimum Object

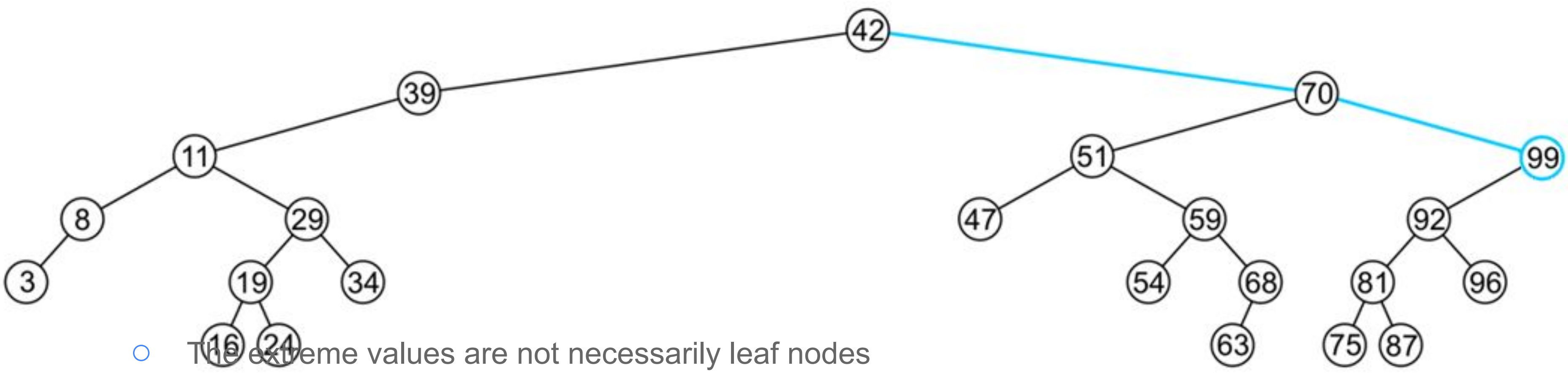
```
template <typename Type>
Type front() const {
    if ( empty() ) {
        throw underflow();
    }

    return (  left()->empty() ) ? value() : left()->front();
}
```



- The run time $\mathbf{O}(h)$

Finding the Maximum Object

```
template <typename Type>
Type Binary_search_node<Type>::back() const {
    if ( empty() ) {
        throw underflow();
    }

    return ( right()->empty() ) ? value() : right()->back();
}
```

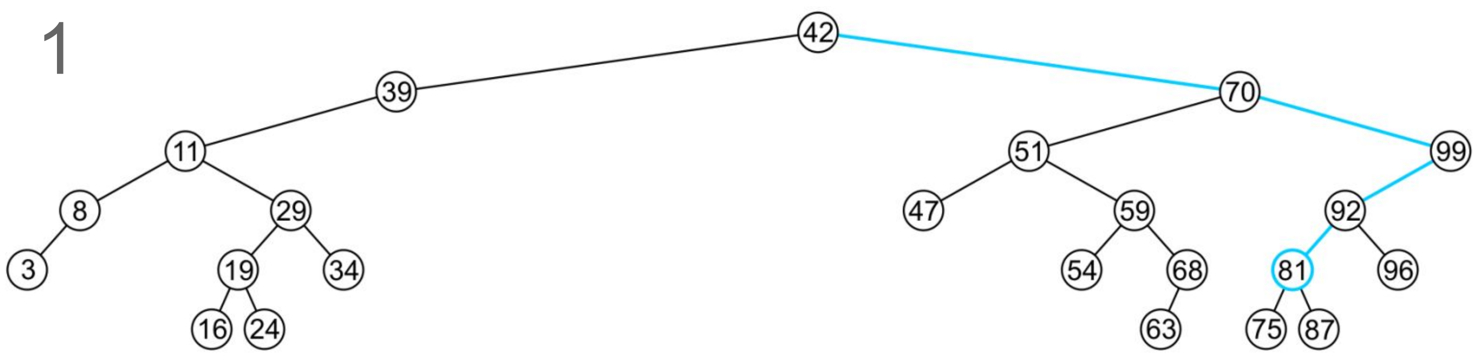


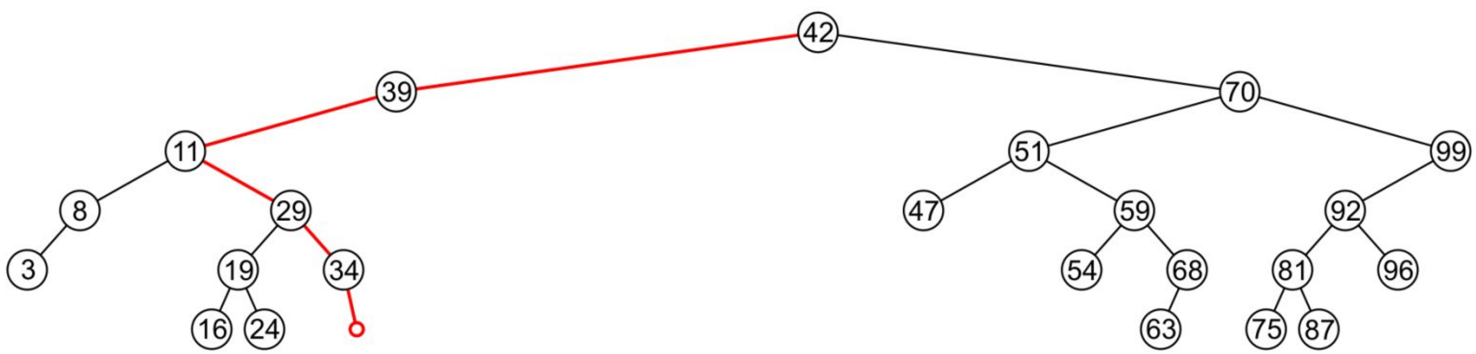
- The extreme values are not necessarily leaf nodes

Find

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

- If a node containing the value is found, *e.g.*, 81, return 1





- If an empty node is reached, *e.g.*, 36, the object is not in the tree:

Find

The implementation is similar to `front` and `back`:

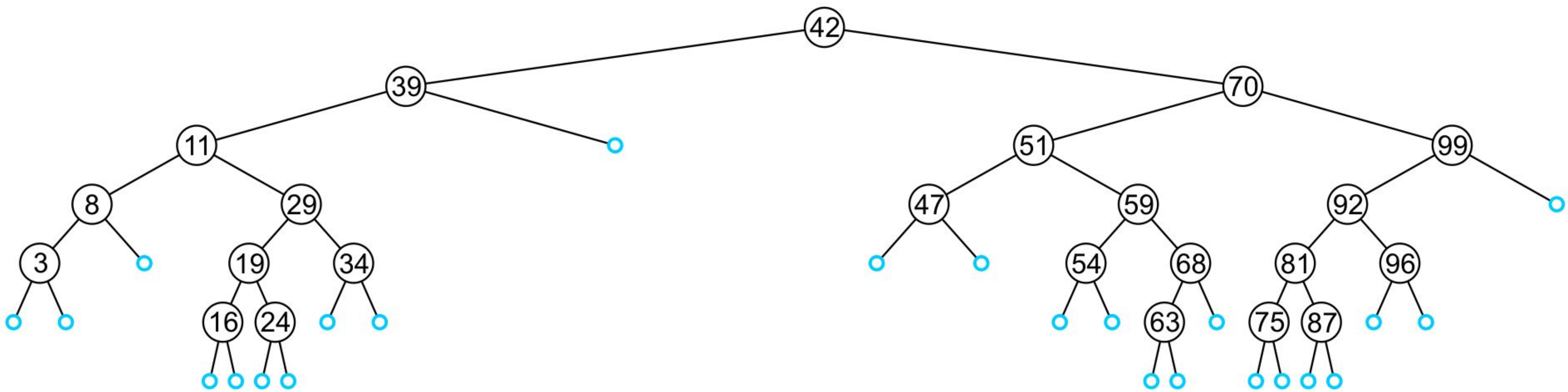
```
template <typename Type>
bool find( Type const &obj ) const {
    if ( empty() ) {
        return false;
    } else if ( value() == obj ) {
        return true;
    }

    return ( obj < value() ) ?
        left()->find( obj ) : right()->find( obj );
}
```

- The run time is $\mathbf{O}(h)$

Insert

An insertion will be performed at a leaf node:

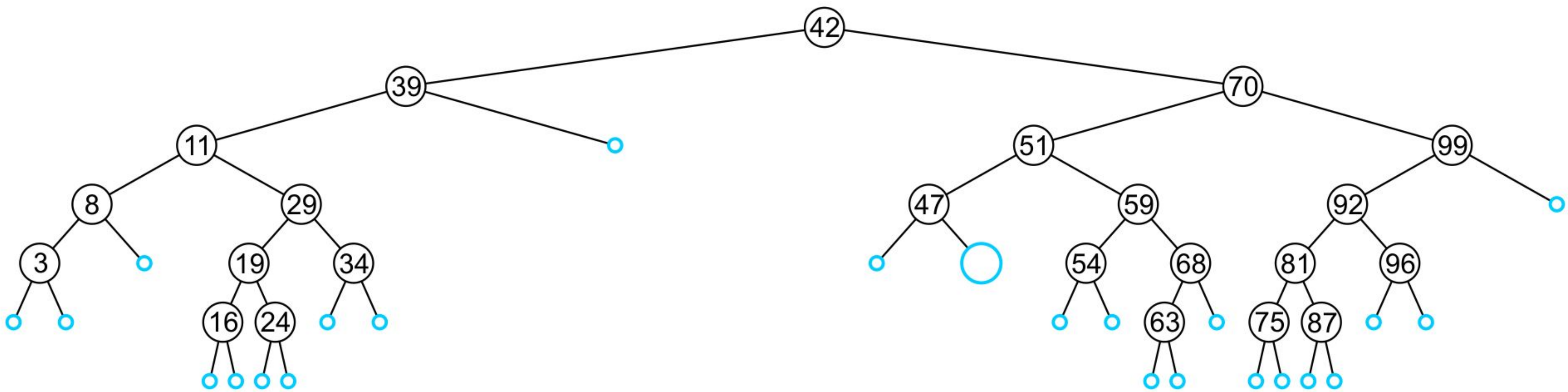
- Any empty node is a possible location for an insertion



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

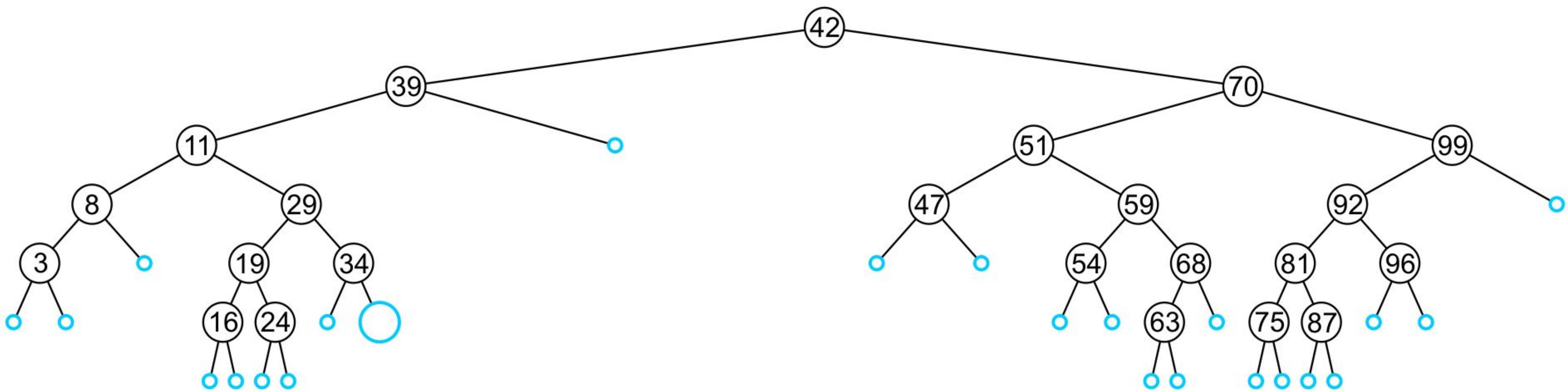
Insert

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



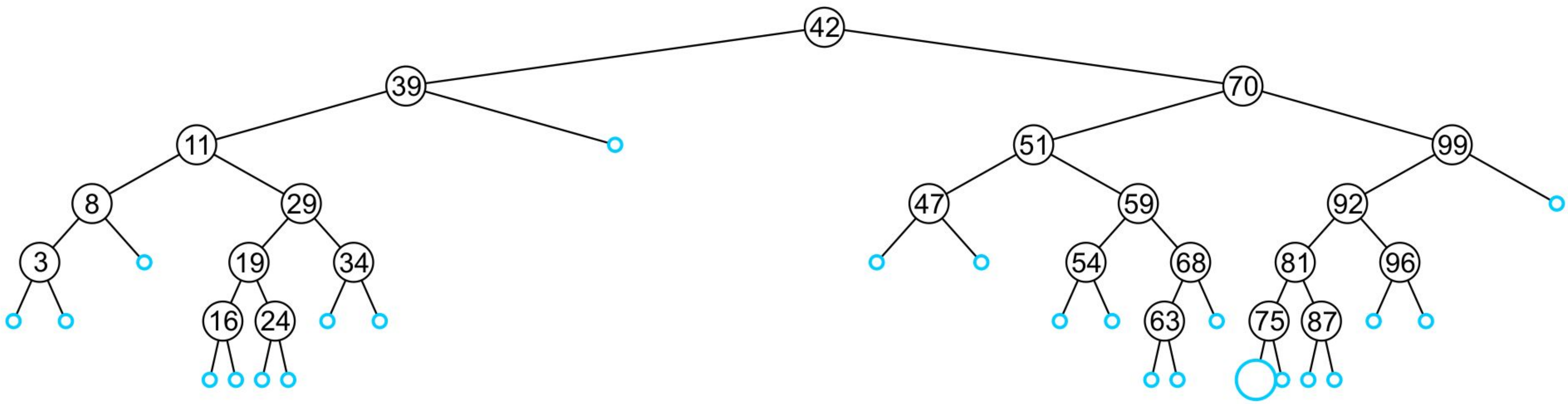
Insert

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



Insert

This empty node may hold values from 71 to 74



Insert

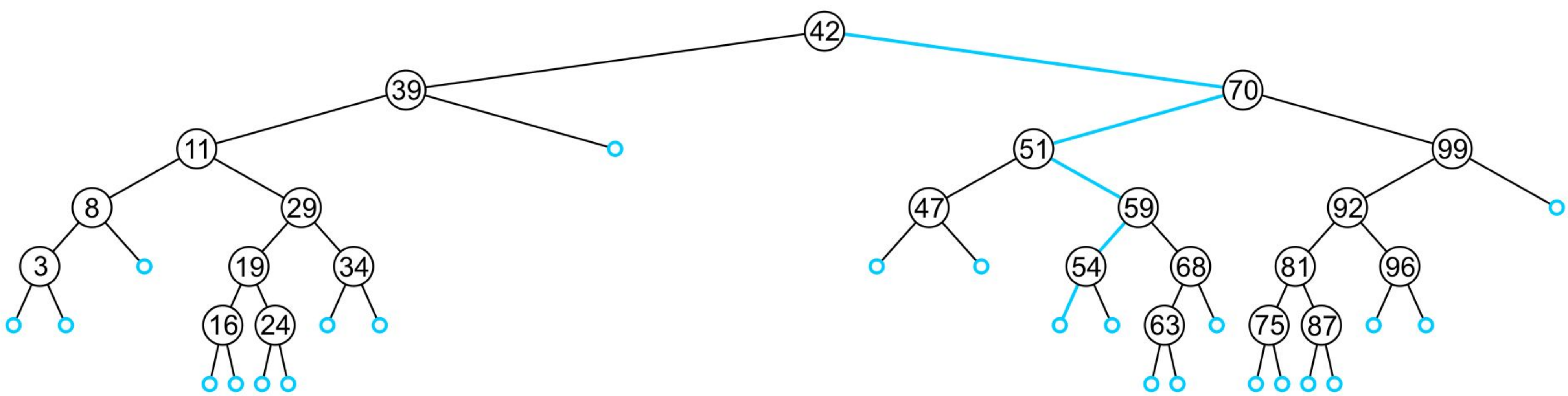
Like find, we will step through the tree

- 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

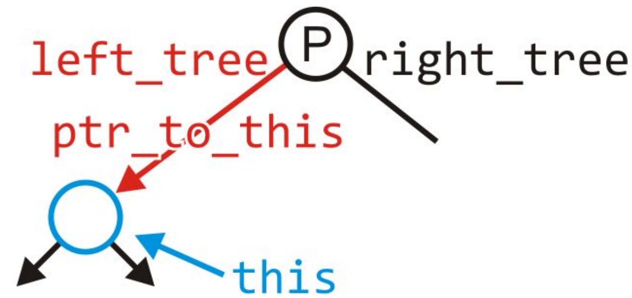
In inserting the value 52, we traverse the tree until we reach an empty node

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



6.1.4.4

Insert




```
template <typename Type>
bool insert( Type const &obj,
                                Binary_search_node *&ptr_to_this  ) {
    if ( empty() ) {
        ptr_to_this = new Binary_search_node<Type>( obj );
        return true;
    } else if ( obj < value() ) {
        return left()->insert( obj, left_tree );
    } else if ( obj > value() ) {
        return right()->insert( obj, right_tree );
    } else {
        return false;
    }
}
```

Insert

It is assumed that if neither of the conditions:

```
obj < value()
obj > value()
```

then `obj == value()` and therefore we do nothing

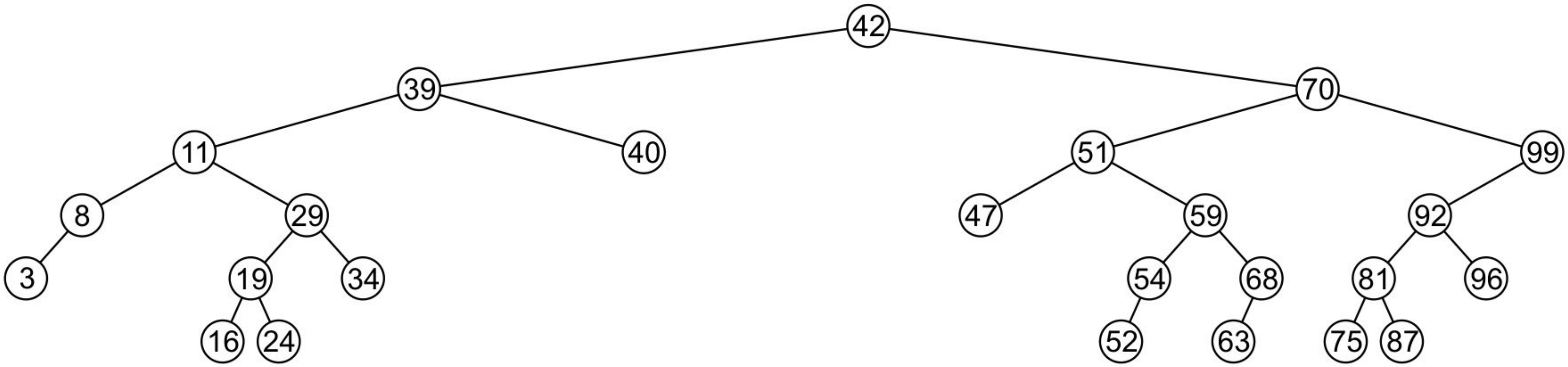
- The object is already in the binary search tree

Erase

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

There are three possible scenarios:

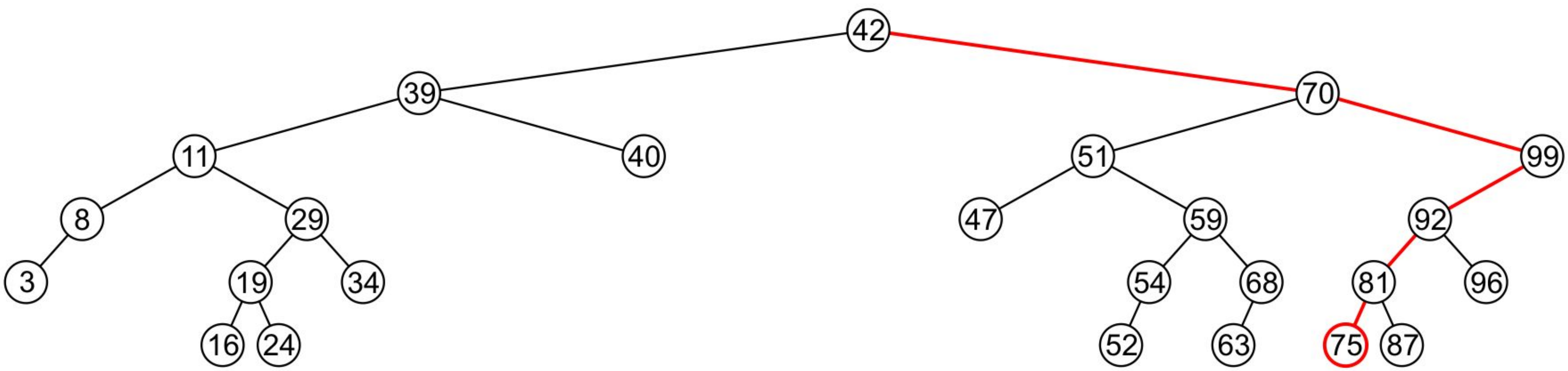
- The node is a leaf node,
- It has exactly one child, or



Erase

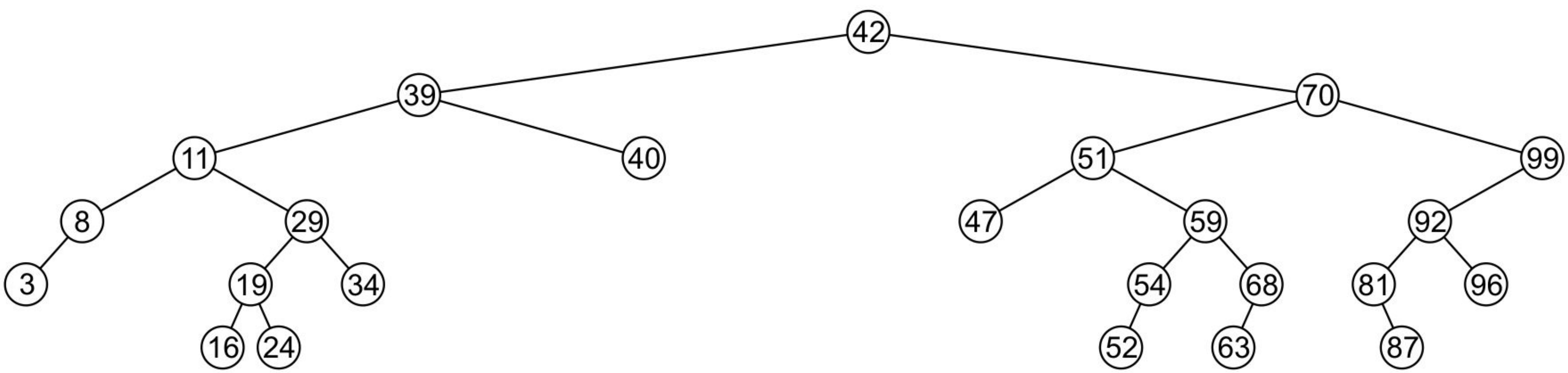
A leaf node simply must be removed and the appropriate member variable of the parent is set to `nullptr`

- Consider removing 75



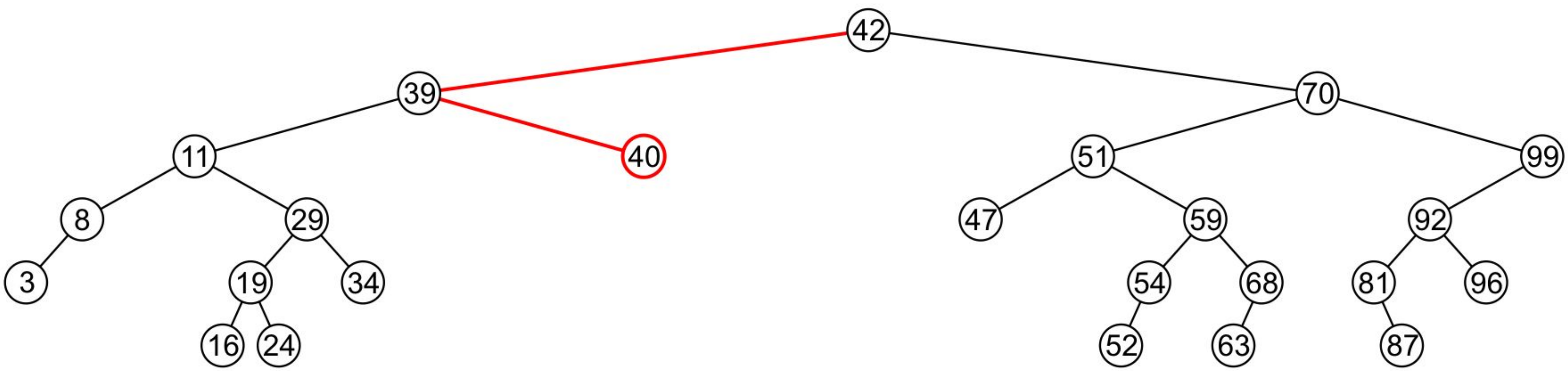
Erase

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



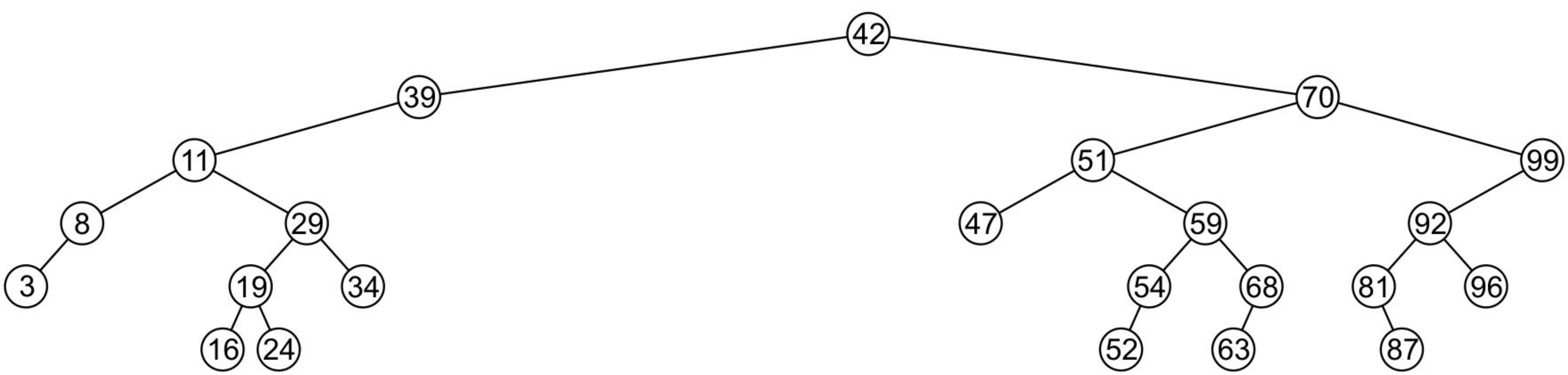
Erase

Erasing the node containing 40 is similar



Erase

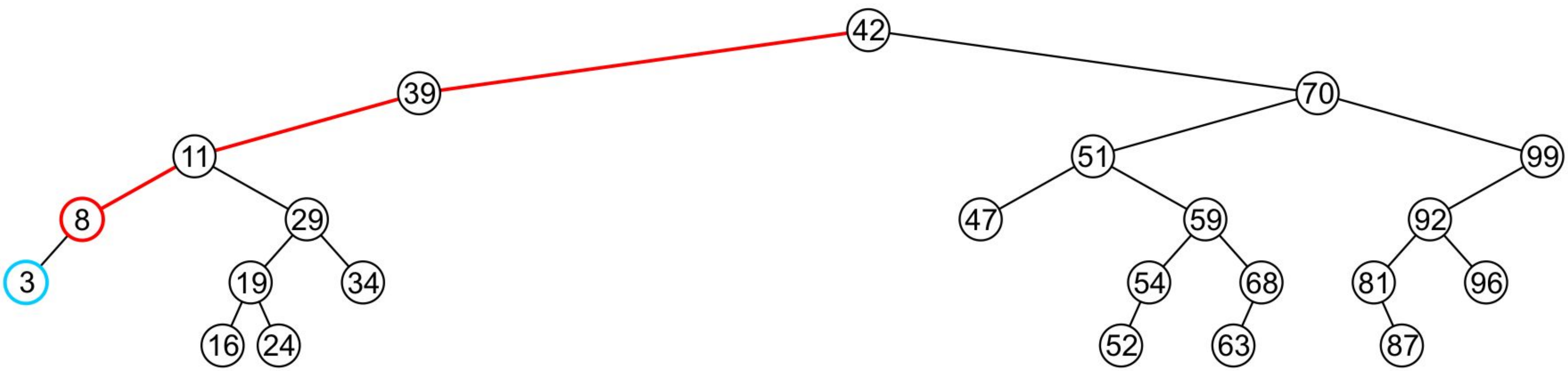
The node is deleted and `right_tree` of 39 is set to `nullptr`



Erase

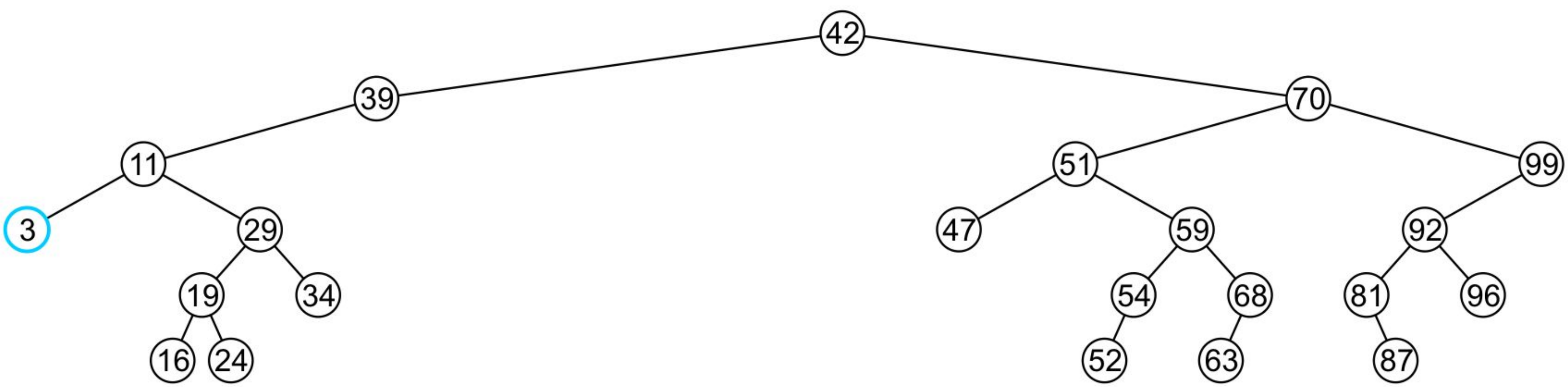
If a node has only one child, we can simply promote the sub-tree associated with the child

- Consider removing 8 which has one left child



Erase

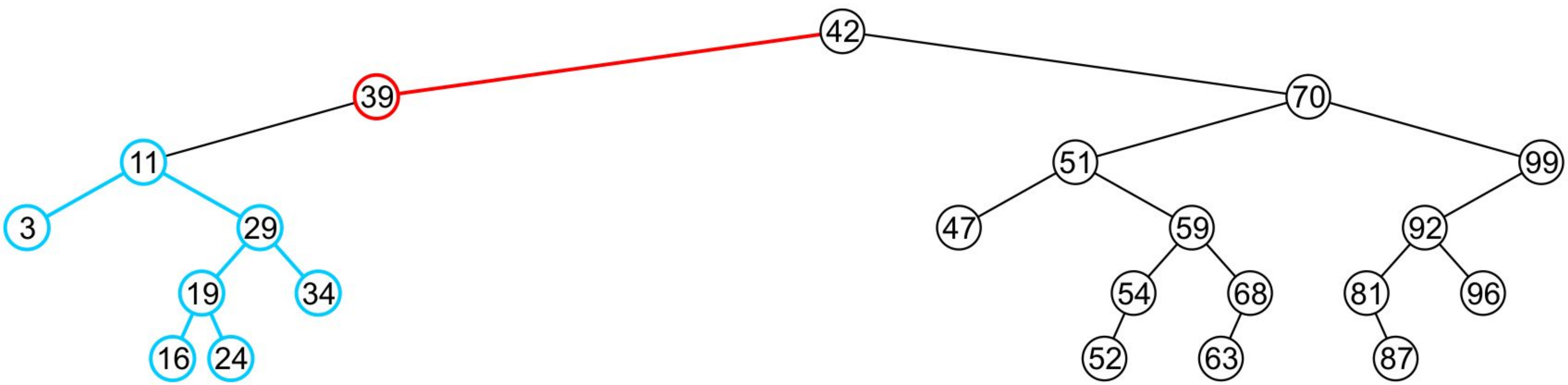
The node 8 is deleted and the `left_tree` of 11 is updated to point to 3



Erase

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

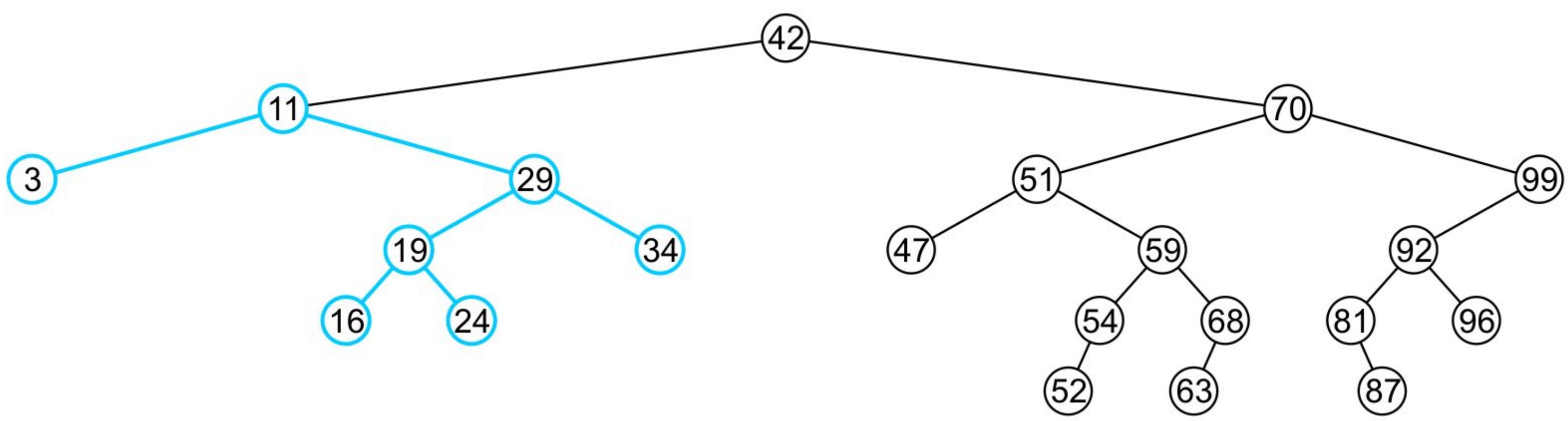
- To remove 39, it has a single child 11



Erase

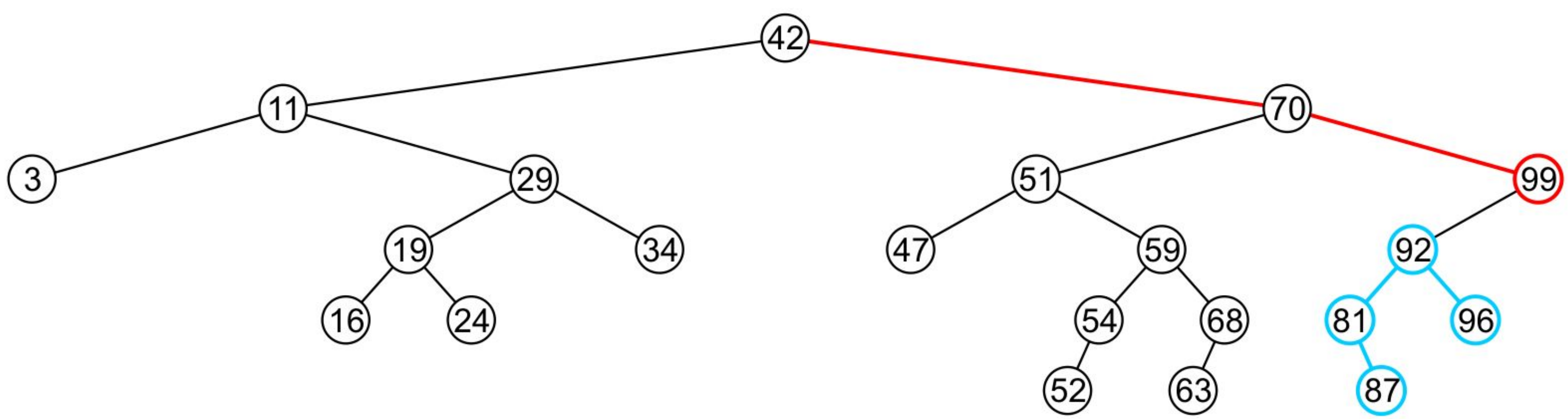
The node containing 39 is deleted and `left_node` of 42 is updated to point to 11

- Notice that order is still maintained



Erase

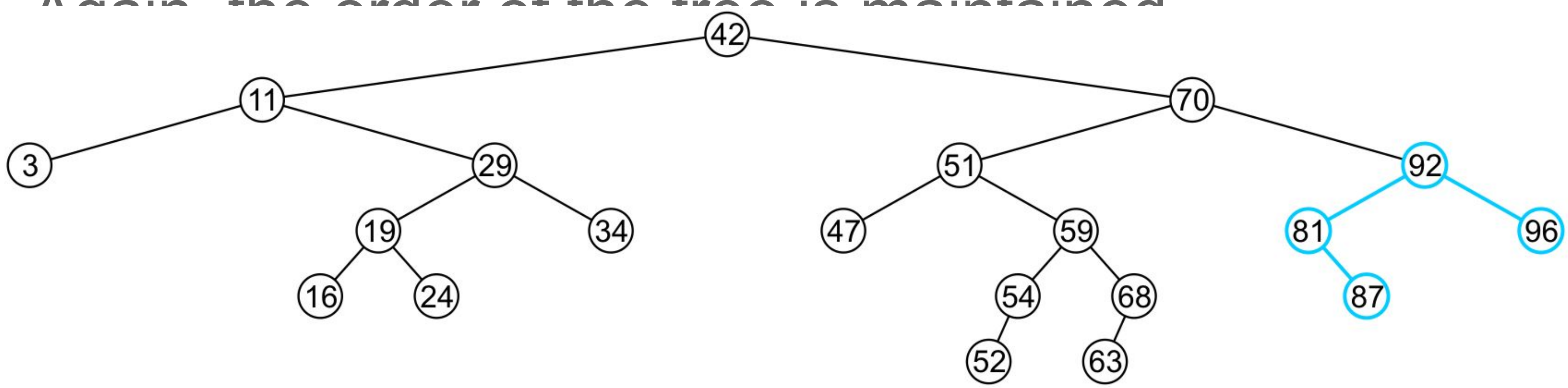
Consider erasing the node containing 99



Erase

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

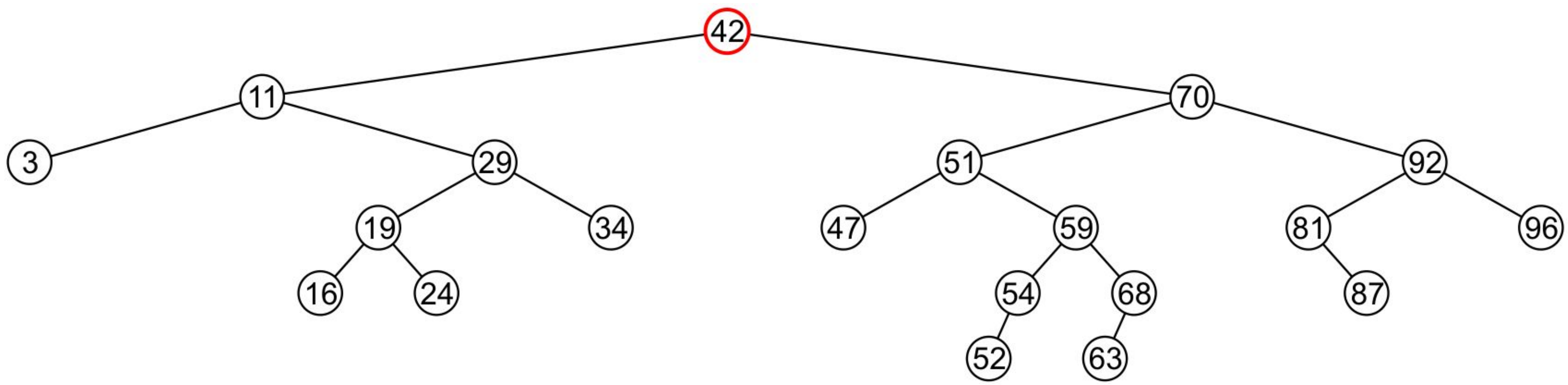
- The member variable `right_tree` of 70 is set to point to 92
- Again, the order of the tree is maintained



Erase

Finally, we will consider the problem of erasing a full node, *e.g.*, 42

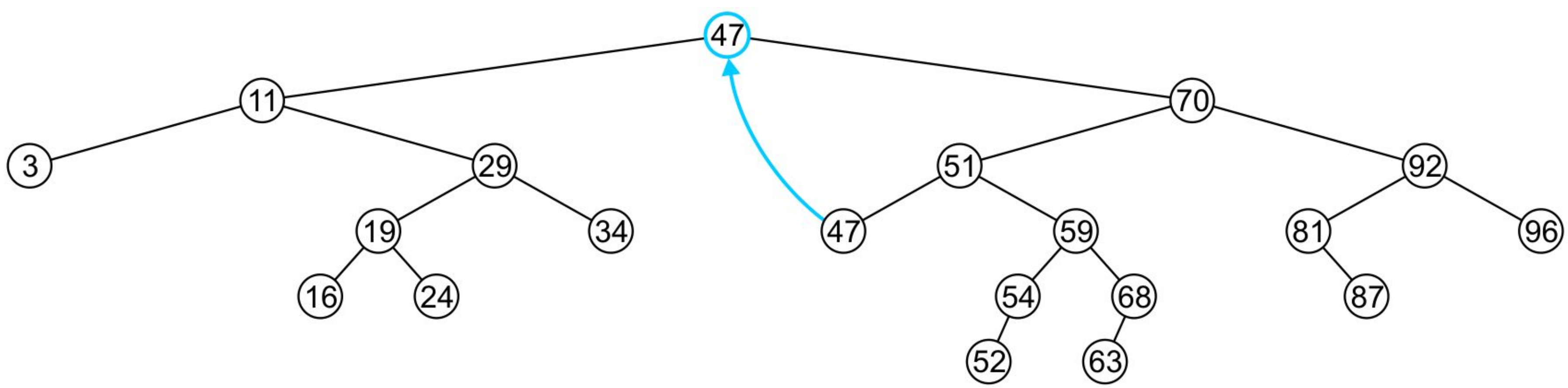
We will perform two operations:



Erase

In this case, we replace 42 with 47

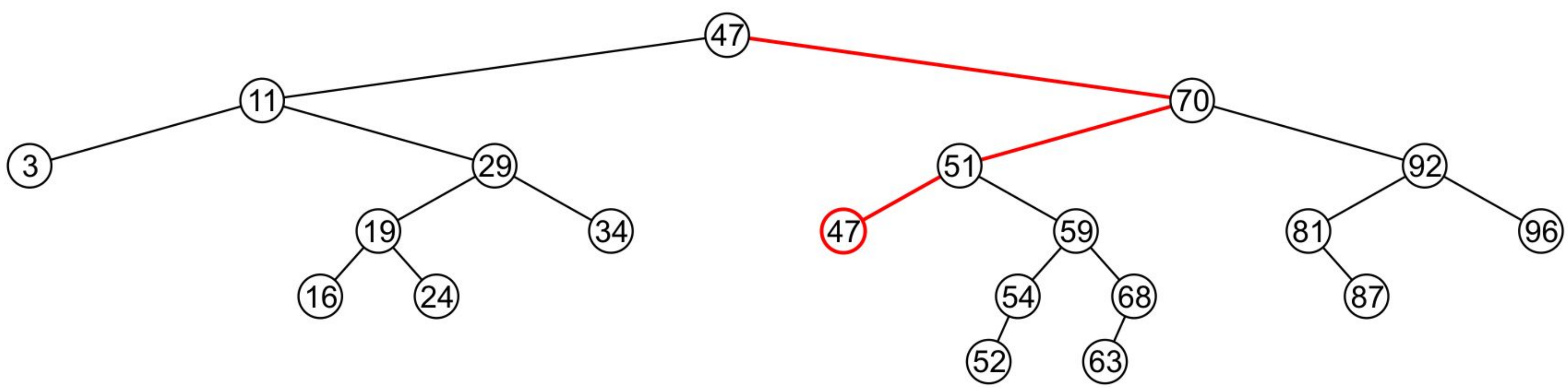
- We temporarily have two copies of 47 in the tree



Erase

We now recursively erase 47 from the right sub-tree

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

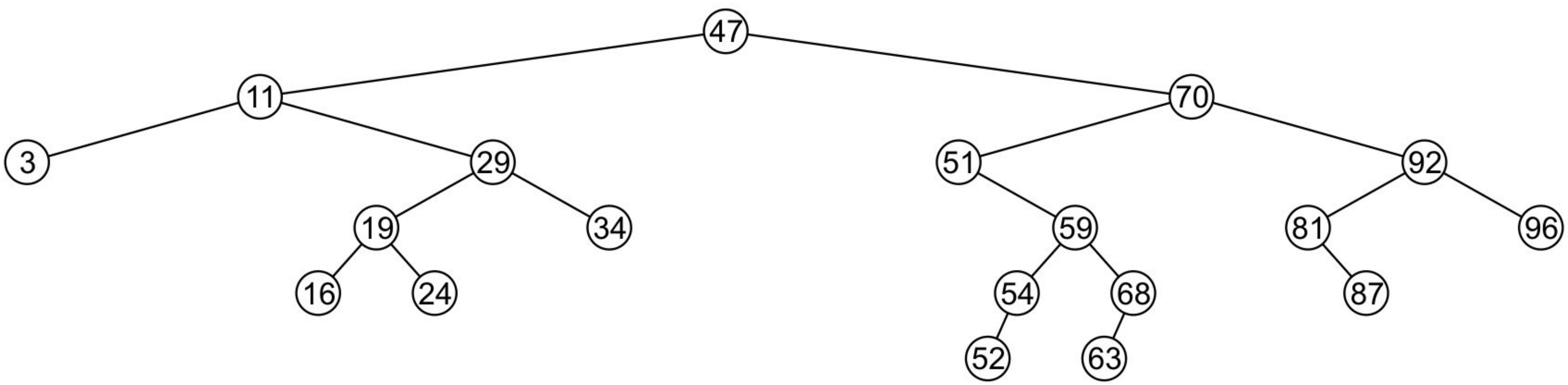


Erase

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

- Notice that the tree is still sorted:

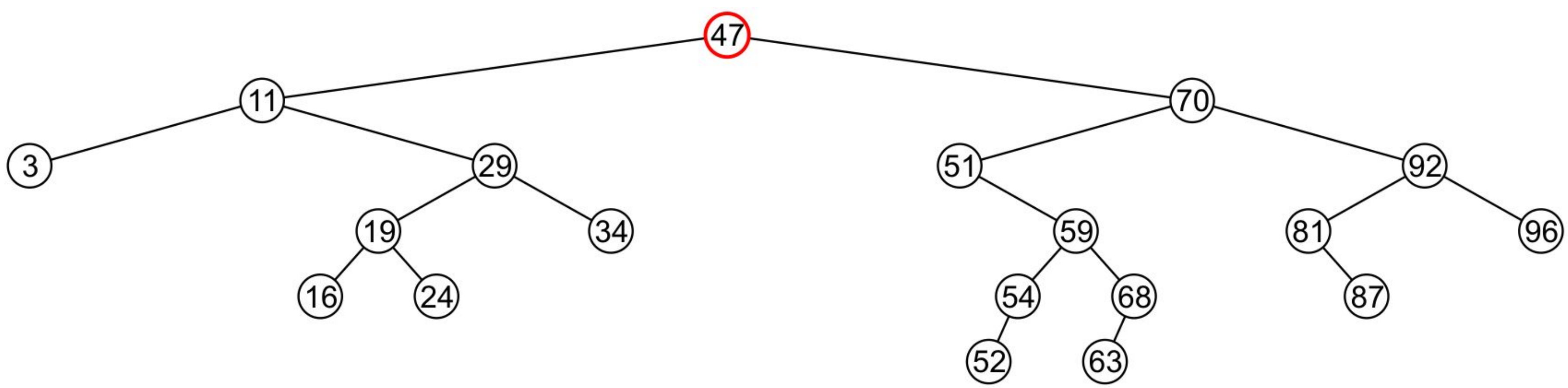
    47 was the least object in the right sub-tree



Erase

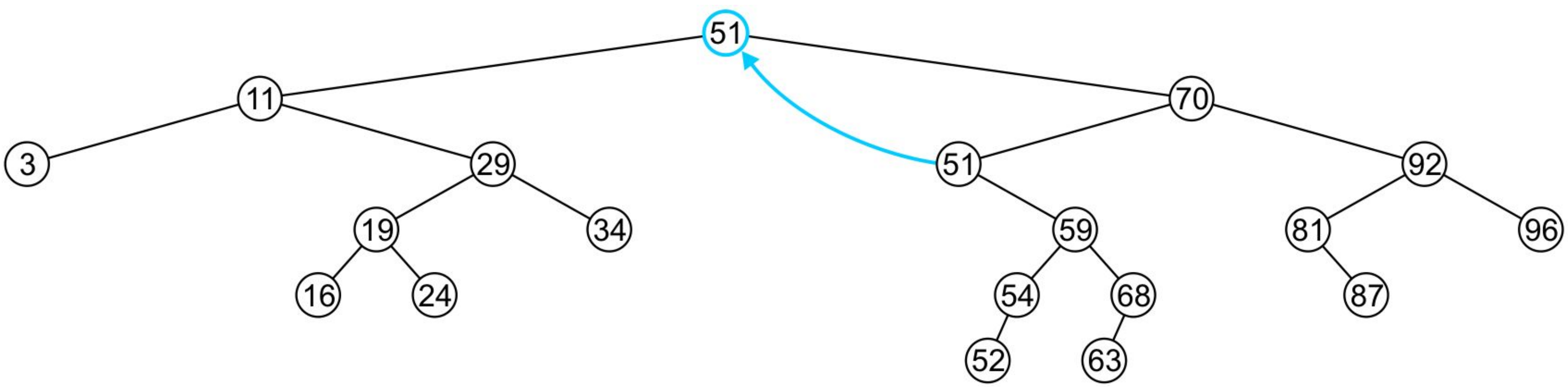
Suppose we want to erase the root 47 again:

- 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



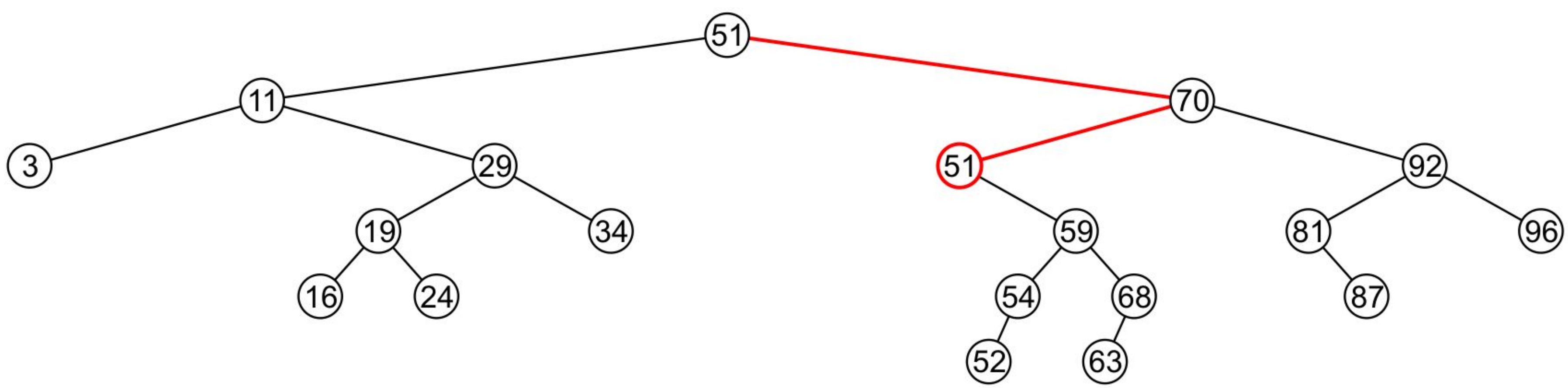
Erase

We copy 51 from the right sub-tree



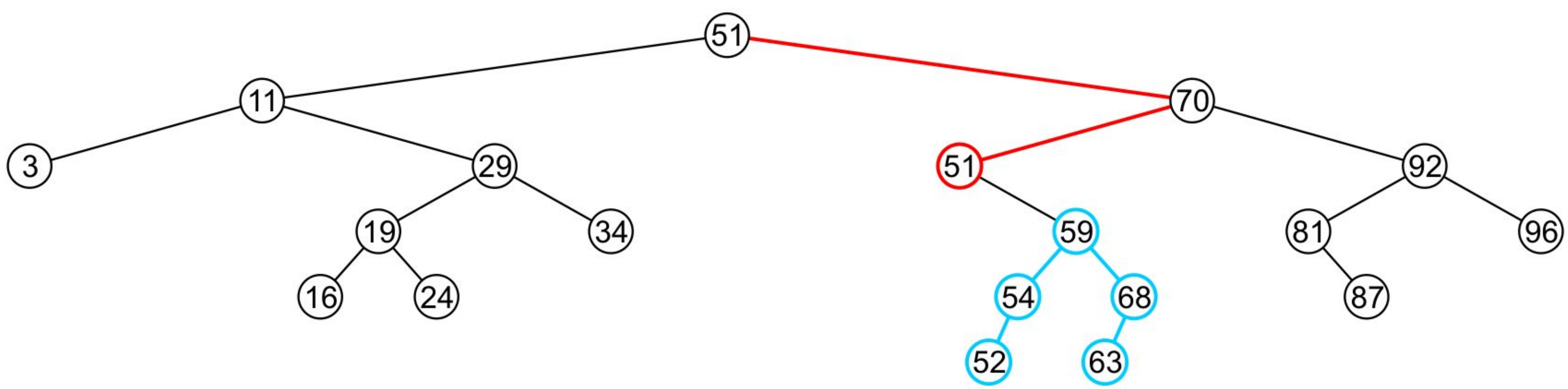
Erase

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



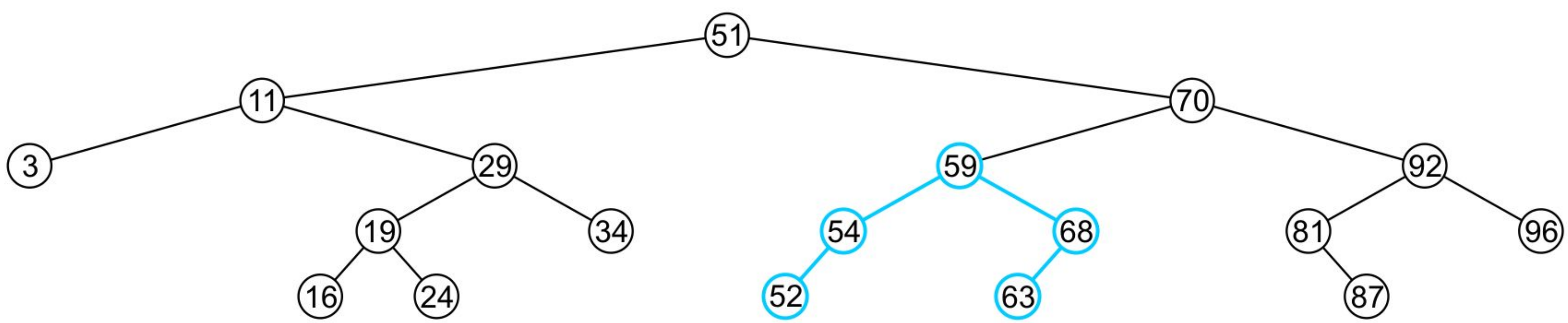
Erase

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



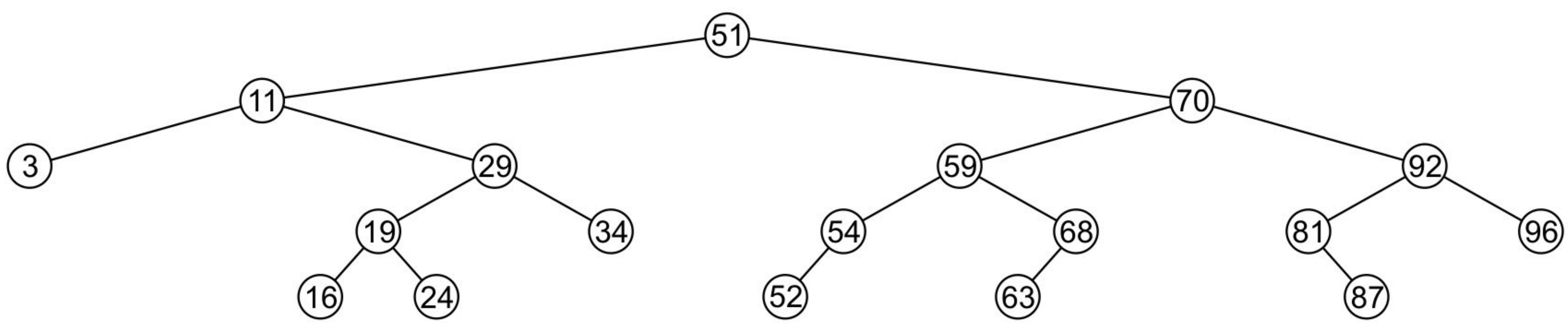
Erase

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



Erase

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

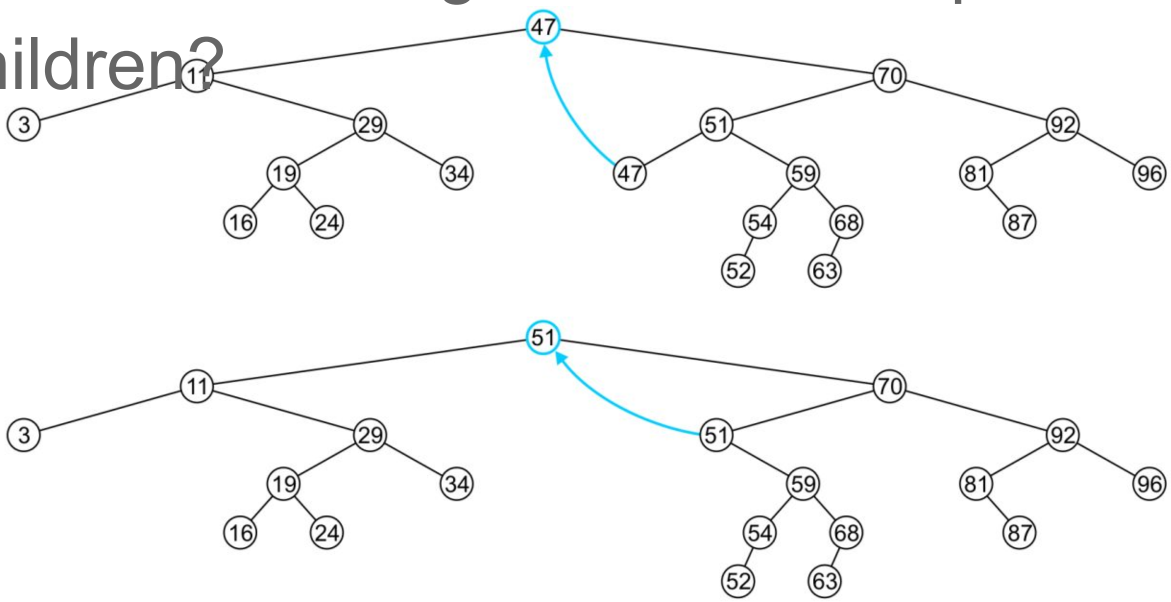


Erase

In the two examples of removing a full node, we promoted:

- A node with no children
- A node with right child

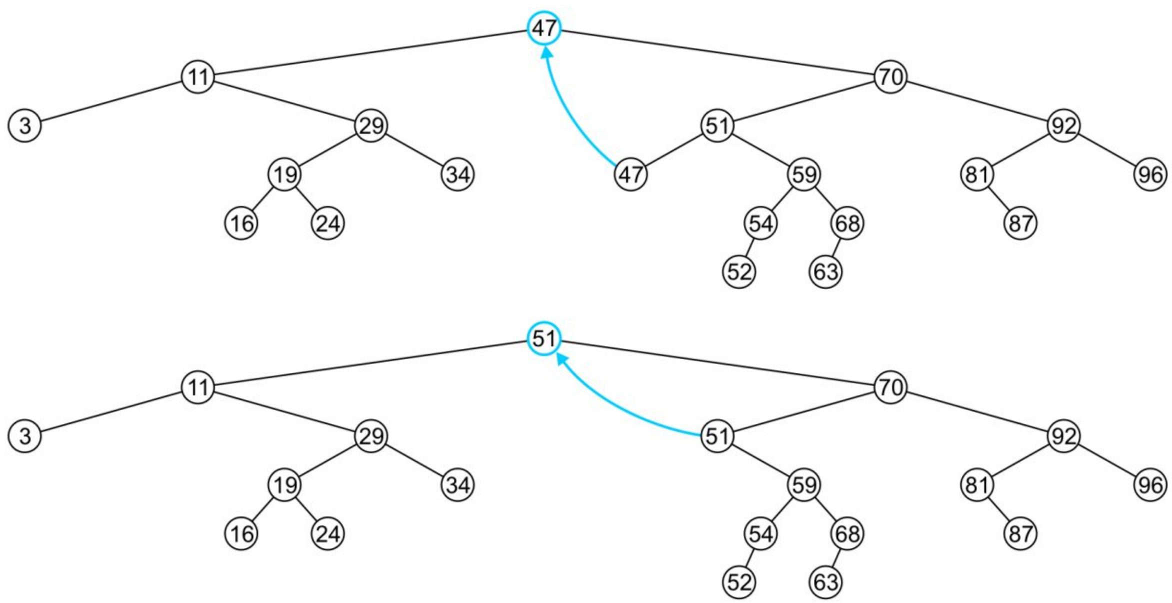
Is it possible, in removing a full node, to promote a child with two children?



Erase

Recall that we promoted the minimum value in the right sub-tree

- If that node had a left sub-tree, that sub-tree would cor



6.1.4.5

Erase

In order to properly remove a node, we will have to change the member variable pointing to the node

- To do this, we will pass that member variable by reference

Additionally: We will return 1 if the object is removed and 1 if the object was not found

Erase

```
template <typename Type>
rase( Type const &obj, Binary_search_node *&ptr_to_this ) {
    if ( empty() ) {
        return false;
    } else if ( obj == value() ) {
        if ( is_leaf() ) {                                     // leaf node
            ptr_to_this = nullptr;
            delete this;
        } else if ( !left()->empty() && !right()->empty() ) {   // full node
            node_value = right()->front();
            right()->erase( value(), right_tree );
        } else {                                               // only one child
            ptr_to_this = ( !left()->empty() ) ? left() : right();
            delete this;
        }

        return true;
    } else if ( obj < value() ) {
        return left()->erase( obj,  left_tree );
    } else {
        return right()->erase( obj, right_tree );
    }
```