Binary search trees

Outline

This topic covers binary search trees:

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

Abstract Sorted Lists

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

We will now discuss the Abstract Sorted List:

- The relation is based on an implicit linear ordering

Certain operations no longer make sense:

- `push_front` and `push_back` are replaced by a generic `insert`

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

Implementation

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

- As an insertion could occur anywhere in a linked list or array, we must either traverse or copy, on average, $\mathbf{O}(n)$ objects

Background

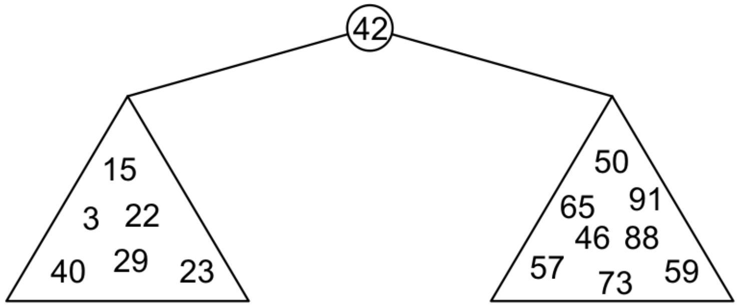
Recall that with a binary tree, we can dictate an order on the two children

We will exploit this order:

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

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:

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

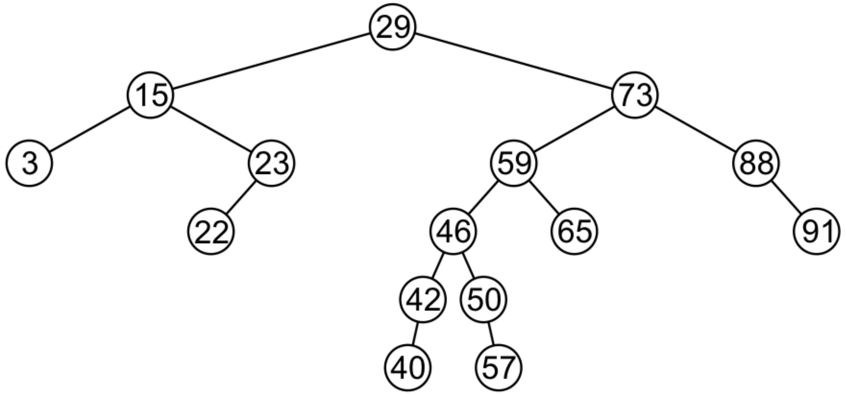
Definition

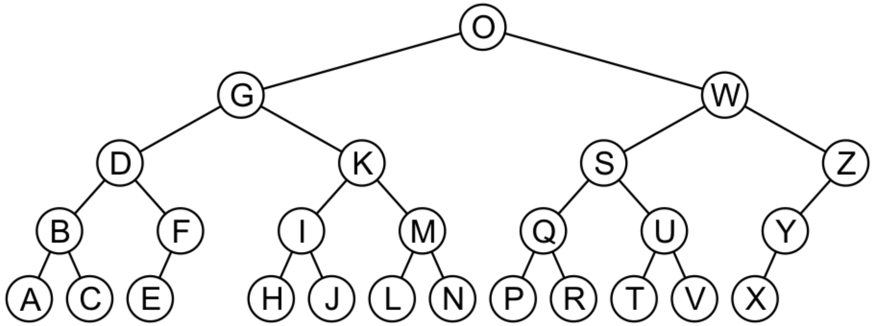
Thus, we define a non-empty binary search tree as a binary tree with the following properties:

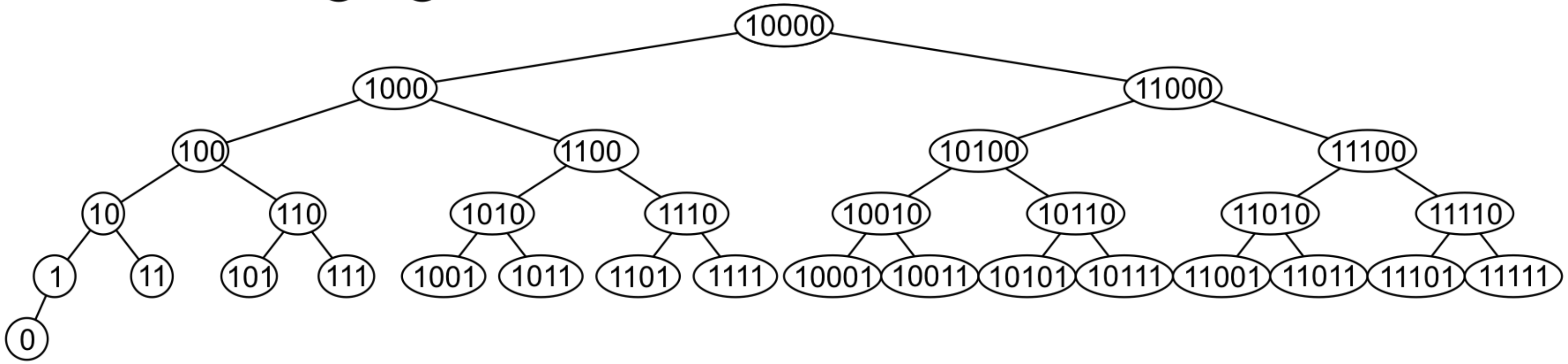
- The left sub-tree (if any) is a binary search tree and all values are less than the root value, and
- The right sub-tree (if any) is a binary search tree and all values are greater than the root value

Examples

Here are other examples of binary search trees:

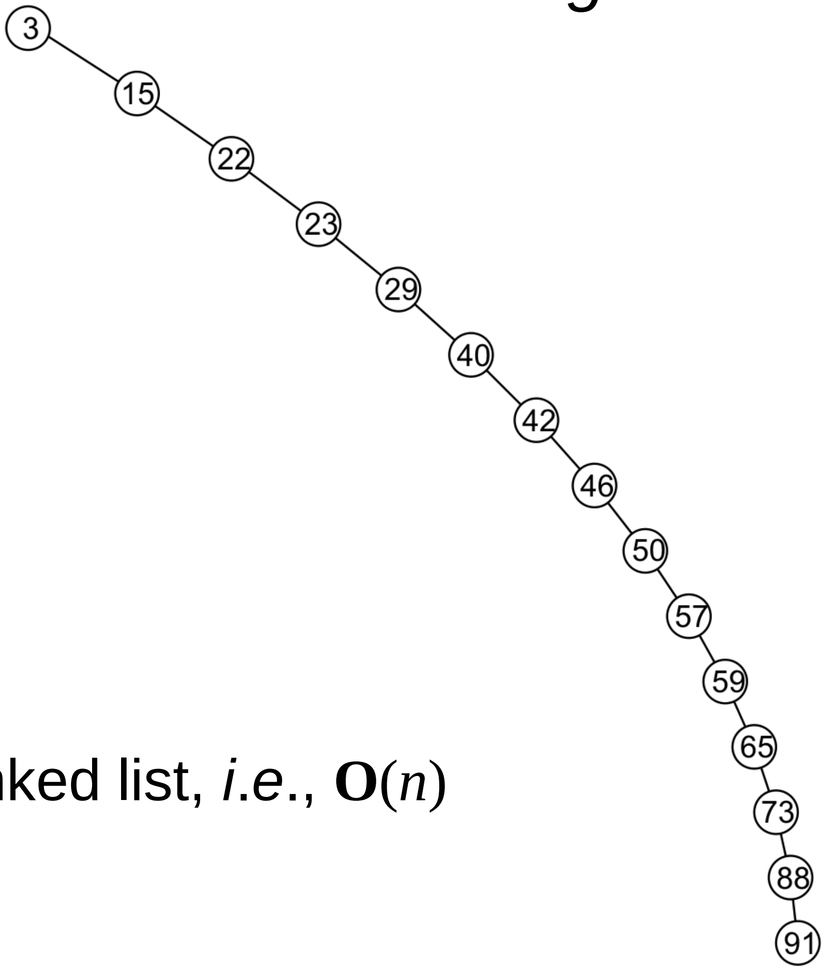






Examples

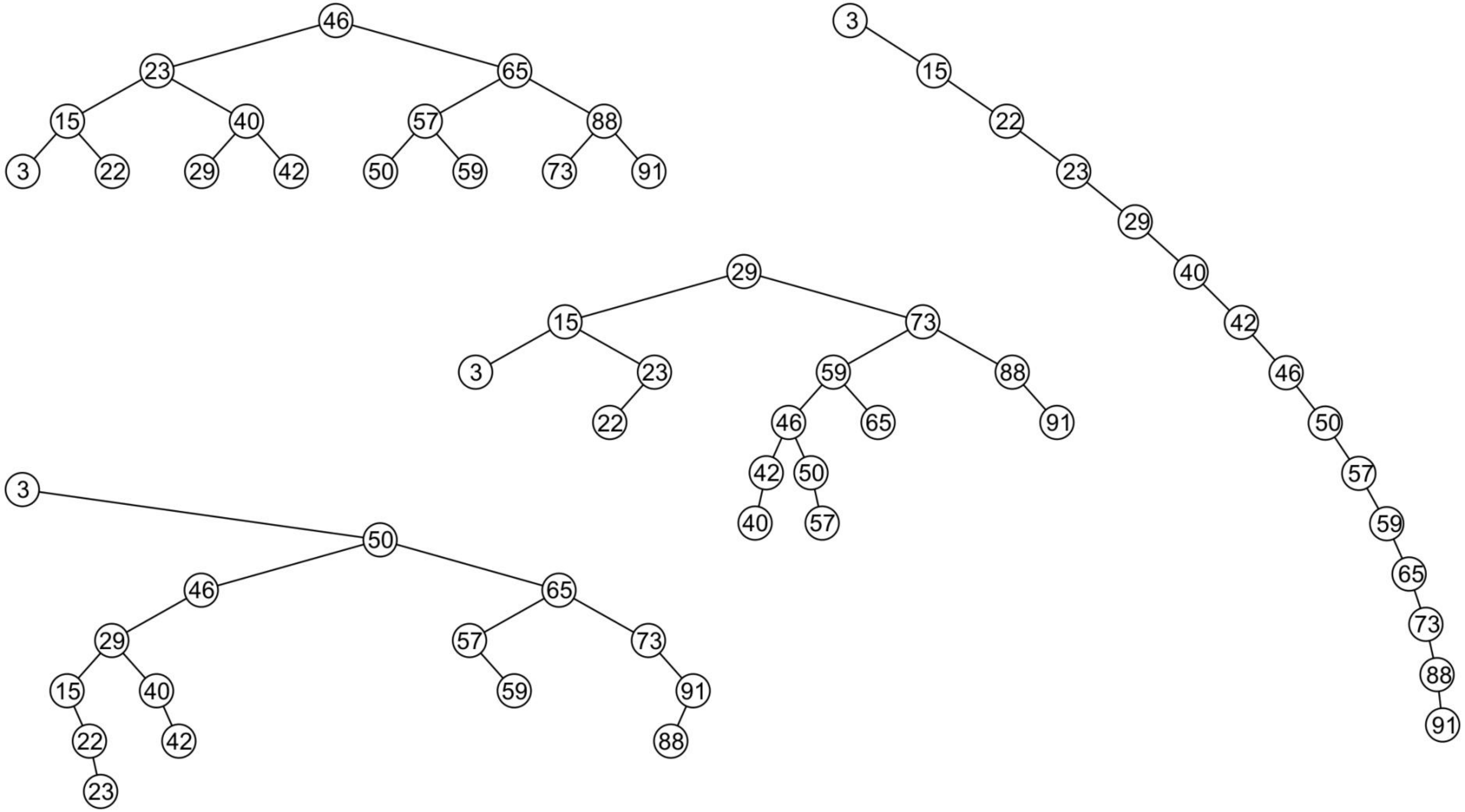
Unfortunately, it is possible to construct *degenerate* binary search trees



- 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

Implementation

We will look at an implementation of a binary search tree in the same spirit as we did with our `Single_list` class

- We will have a `Binary_search_nodes` class
- A `Binary_search_tree` class will store a pointer to the root

We will use templates, however, we will require that the class overrides the comparison operators

Implementation

Any class which uses this binary-search-tree class must therefore implement:

```
bool operator<=( Type const &, Type const & );
bool operator< ( Type const &, Type const & );
bool operator==( Type const &, Type const & );
```

That is, we are allowed to compare two instances of this class

- Examples: `int` and `double`

Implementation

```
#include "Binary_node.h"

template <typename Type>
class Binary_search_tree;

template <typename Type>
class Binary_search_node:public Binary_node<Type> {
    using Binary_node<Type>::node_value;
    using Binary_node<Type>::left_tree;
    using Binary_node<Type>::right_tree;

    public:
        Binary_search_node( Type const & );

        Binary_search_node *left() const;
        Binary_search_node *right() const;
```

Implementation

```
        Type front() const;
        Type back() const;
        bool find( Type const & ) const;

        void clear();
        bool insert( Type const & );
        bool erase( Type const &, Binary_search_node *& );

        friend class Binary_search_tree<Type>;
};
```

Constructor

The constructor simply calls the constructor of the base class

- Recall that it sets both `left_tree` and `right_tree` to `nullptr`
- It assumes that this is a new leaf node

```
template <typename Type>
Binary_search_node<Type>::Binary_search_node( Type const &obj ):
Binary_node<Type>( obj ) {
     // Just calls the constructor of the base class
}
```

Standard Accessors

Because it is a derived class, it already inherits the function:

```
Type value() const;
```

Because the base class returns a pointer to a `Binary_node`, we must recast them as `Binary_search_node`:

```
template <typename Type>
Binary_search_node<Type> *Binary_search_node<Type>::left() const {
        return reinterpret_cast<Binary_search_node *>( Binary_node<Type>::left() );
}

template <typename Type>
Binary_search_node<Type> *Binary_search_node<Type>::right() const {
        return reinterpret_cast<Binary_search_node *>( Binary_node<Type>::right() );
}
```

Inherited Member Functions

The member functions

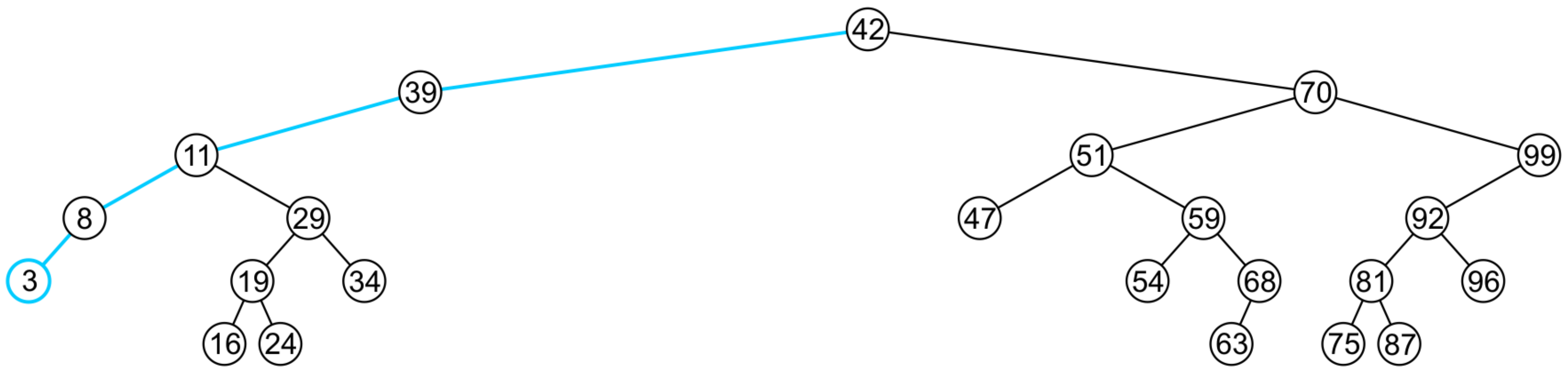
```
bool empty() const
bool is_leaf() const
int size() const
int height() const
```

are inherited from the bas class `Binary_node`

Finding the Minimum Object

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

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

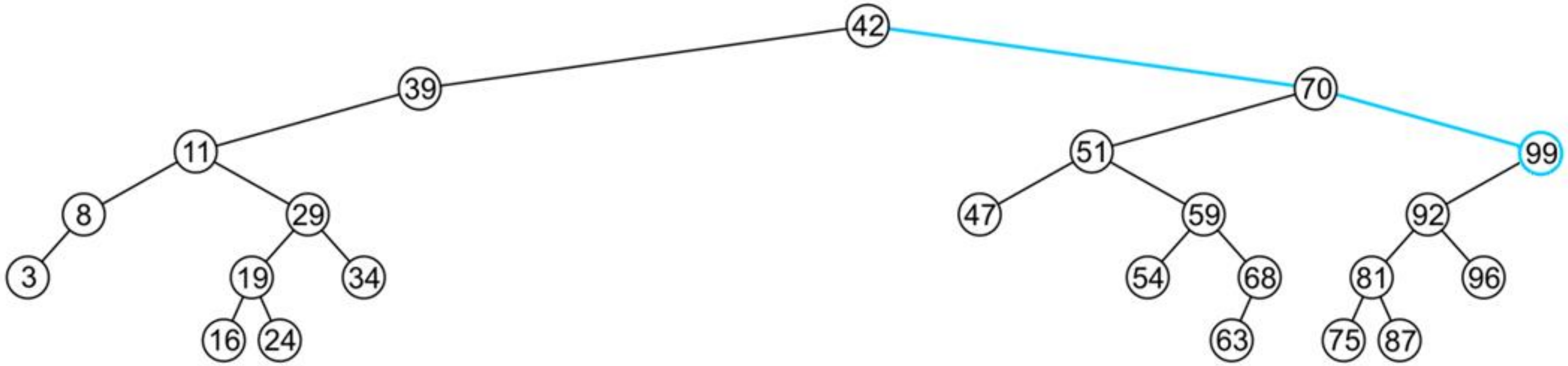


- The run time $\mathbf{O}(h)$

Finding the Maximum Object

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

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

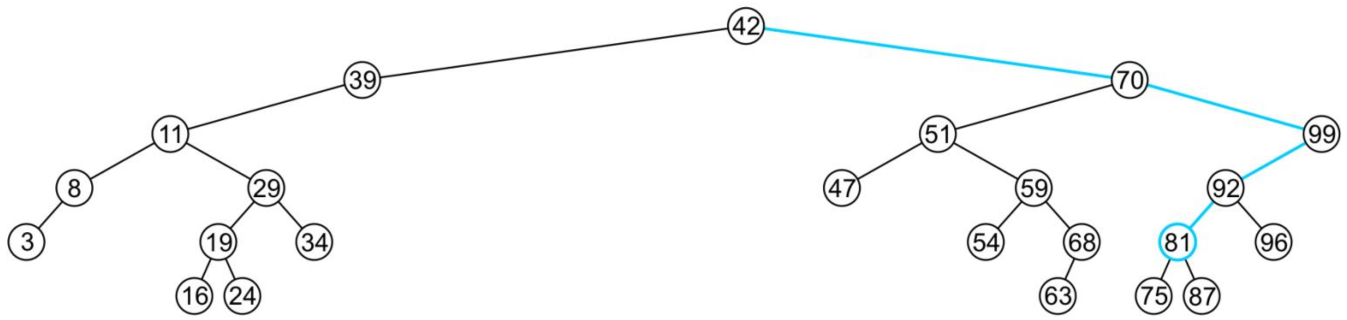


- The extreme values are not necessarily leaf nodes

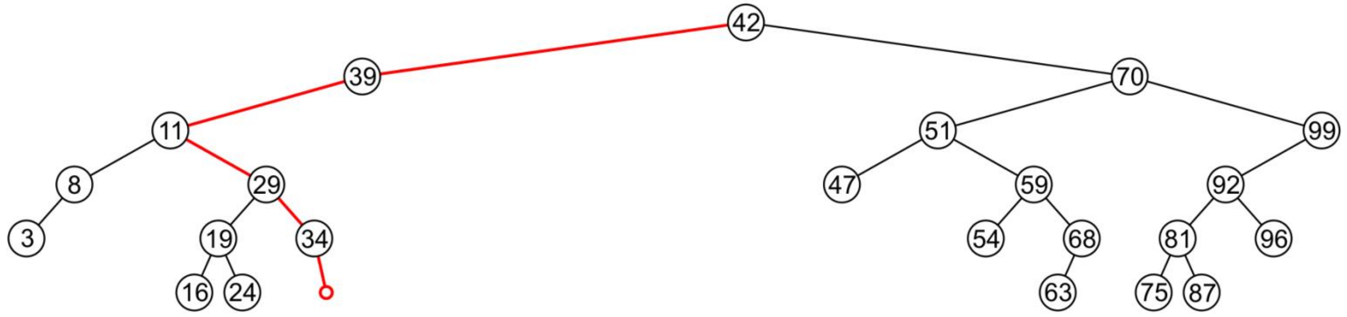
Find

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 Binary_search_node<Type>::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

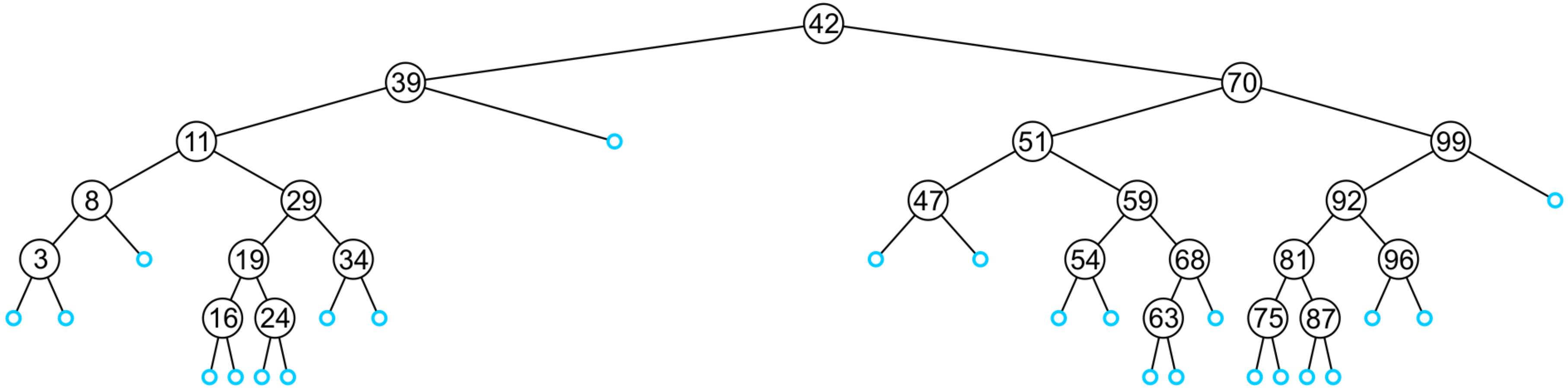
Recall that a Sorted List is implicitly ordered

- It does not make sense to have member functions such as `push_front` and `push_back`
- Insertion will be performed by a single `insert` member function which places the object into the correct location

Insert

An insertion will be performed at a leaf node:

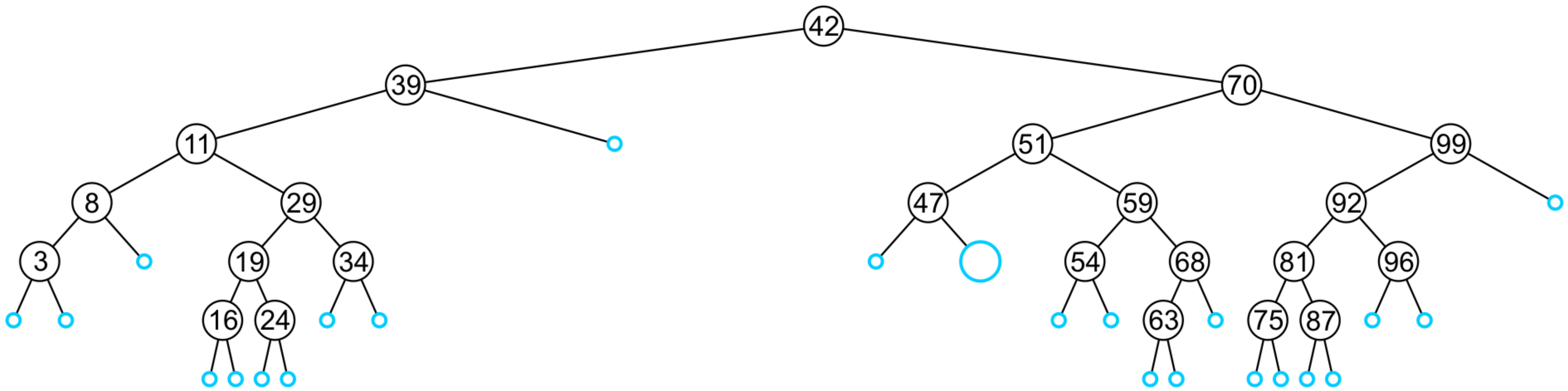
- Any empty node is a possible location for an insertion



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

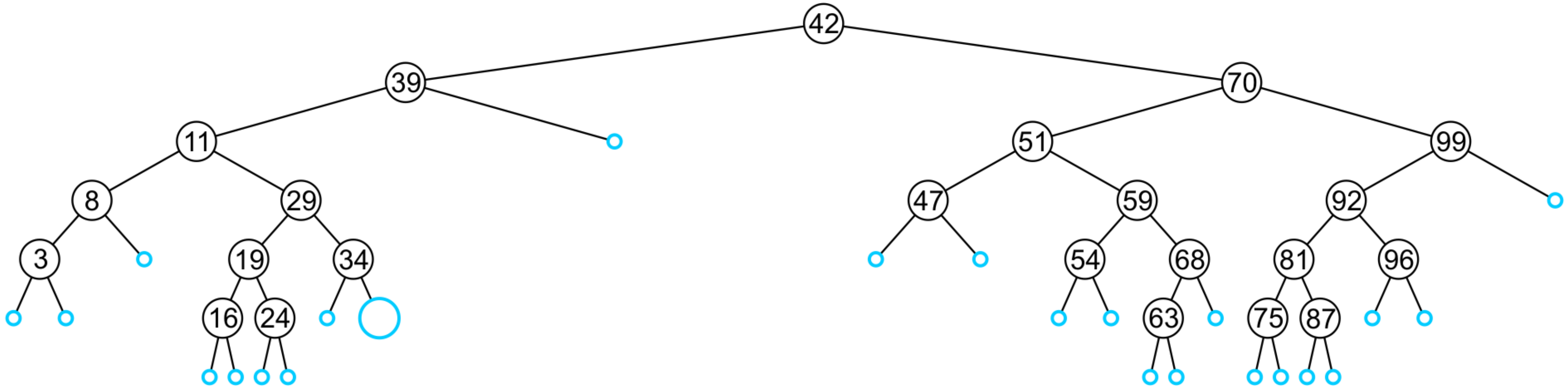
Insert

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



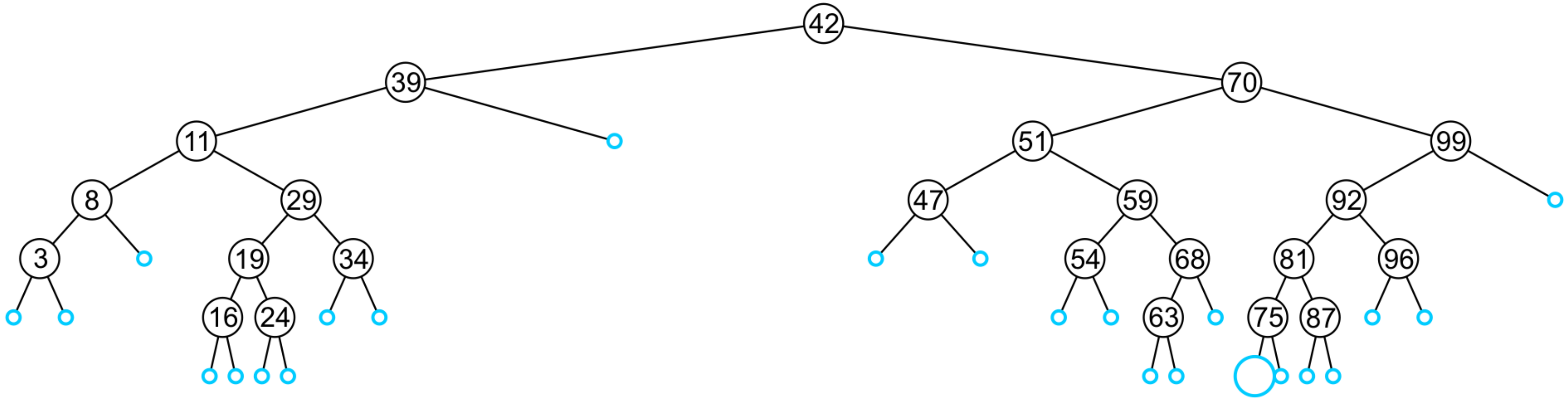
Insert

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



Insert

This empty node may hold values from 71 to 74



Insert

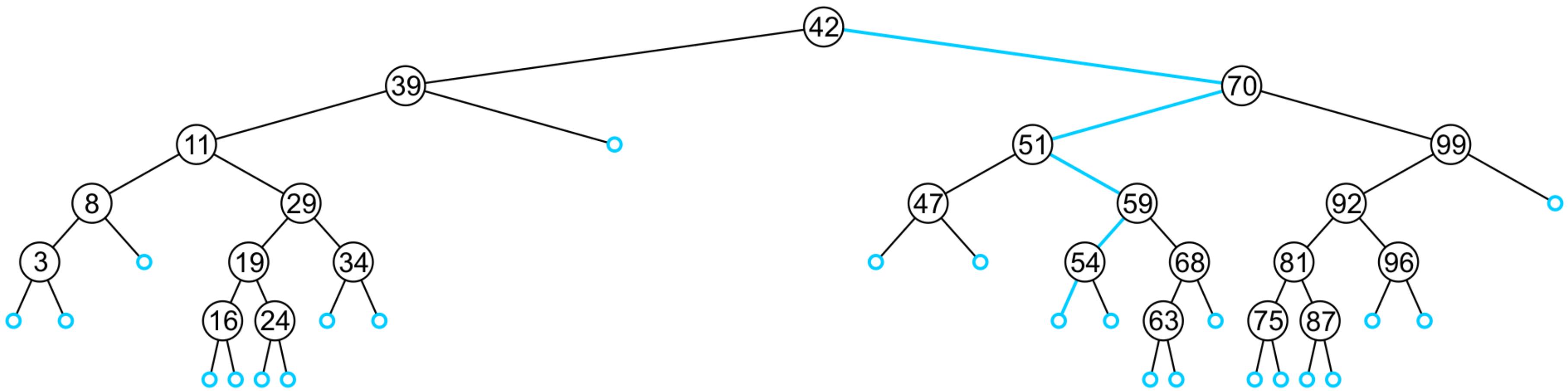
Like find, we will step through the tree

- 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

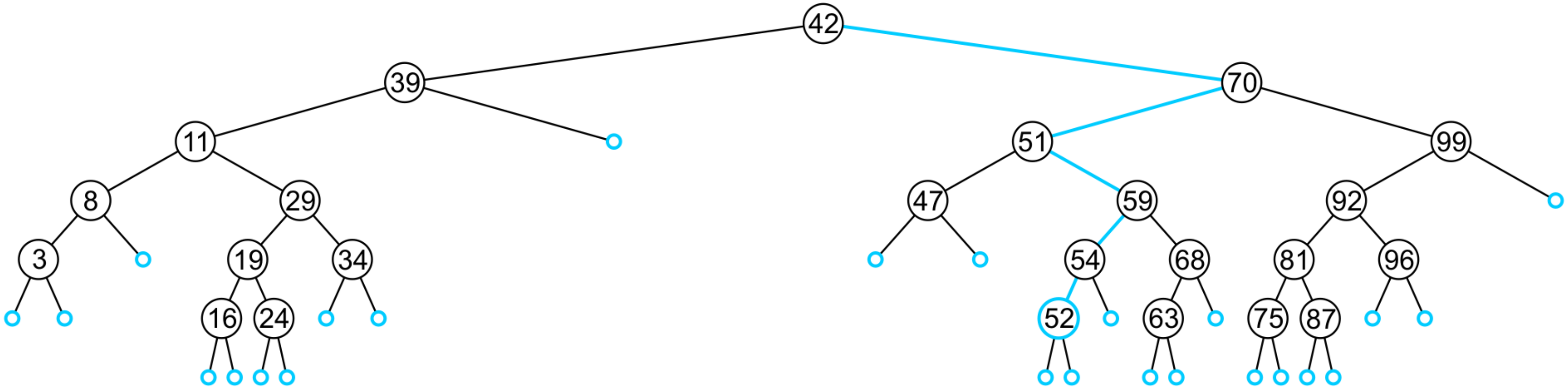
In inserting the value 52, we traverse the tree until we reach an empty node

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



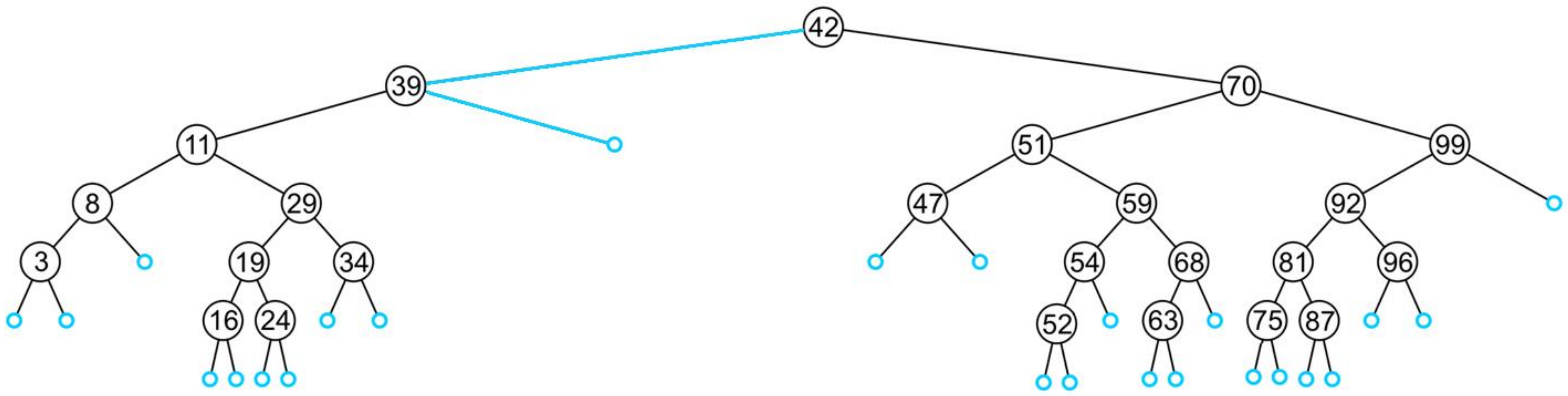
Insert

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



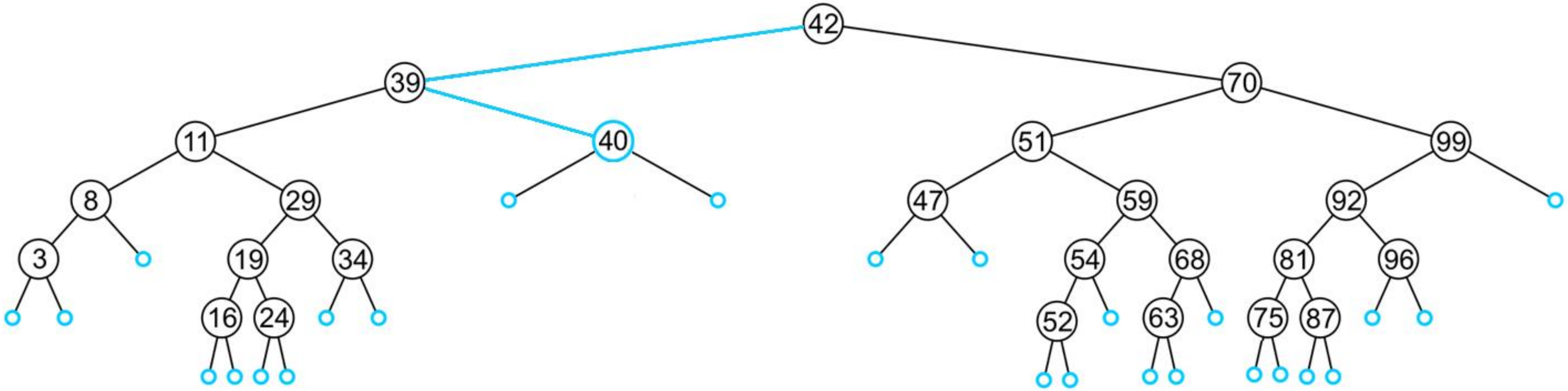
Insert

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

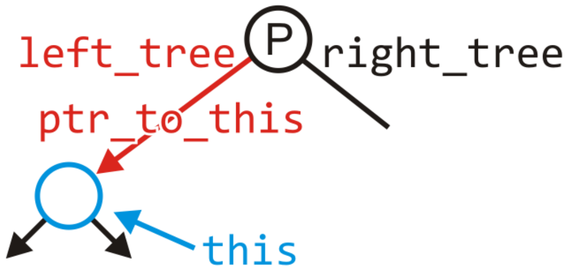


Insert

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



Insert




```
template <typename Type>
bool Binary_search_node<Type>::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

Insert

Blackboard example:

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

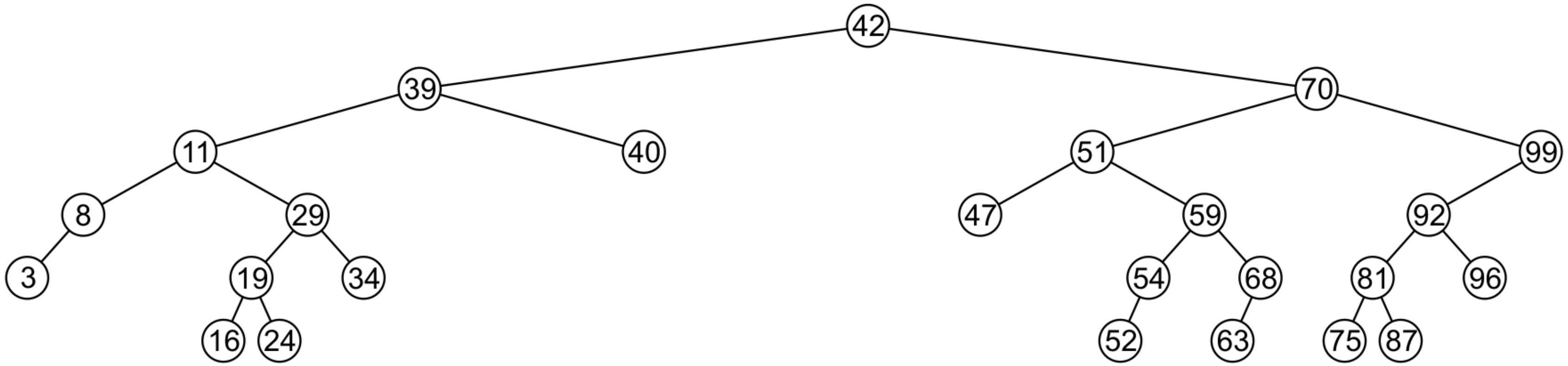
  31 45 36 14 52 42 6 21 73 47 26 37 33 8
- What values could be placed:
  - To the left of 21?
  - To the right of 26?
  - To the left of 47?
- How would we determine if 40 is in this binary search tree?
- Which values could be inserted to increase the height of the tree?

Erase

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

There are three possible scenarios:

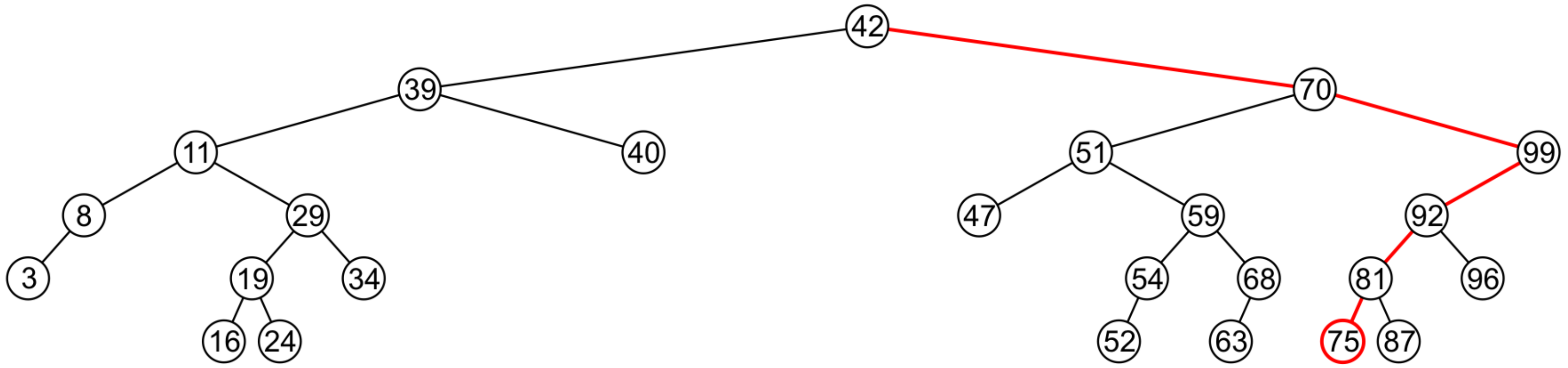
- The node is a leaf node,
- It has exactly one child, or
- It has two children (it is a full node)



Erase

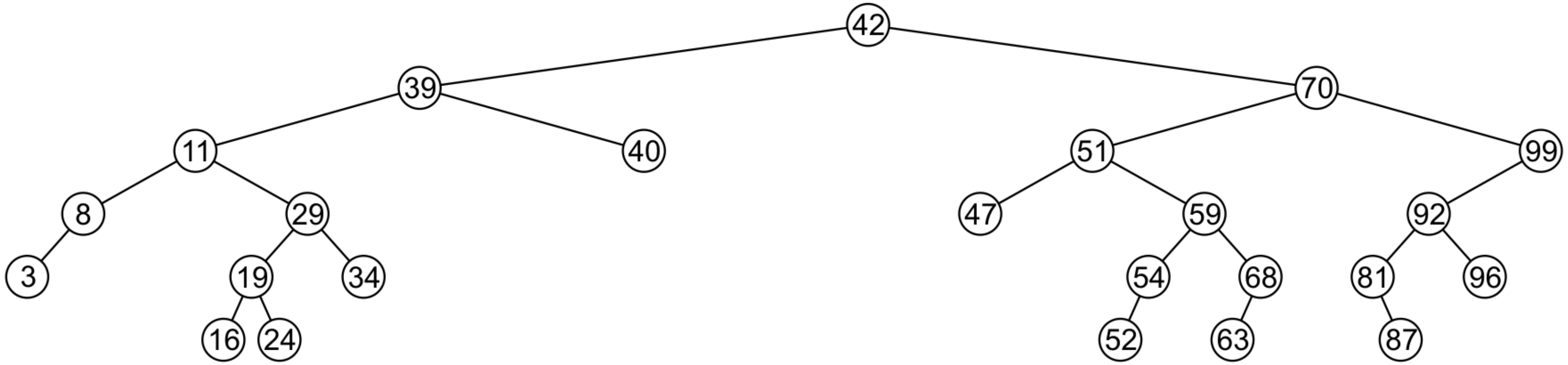
A leaf node simply must be removed and the appropriate member variable of the parent is set to `nullptr`

- Consider removing 75



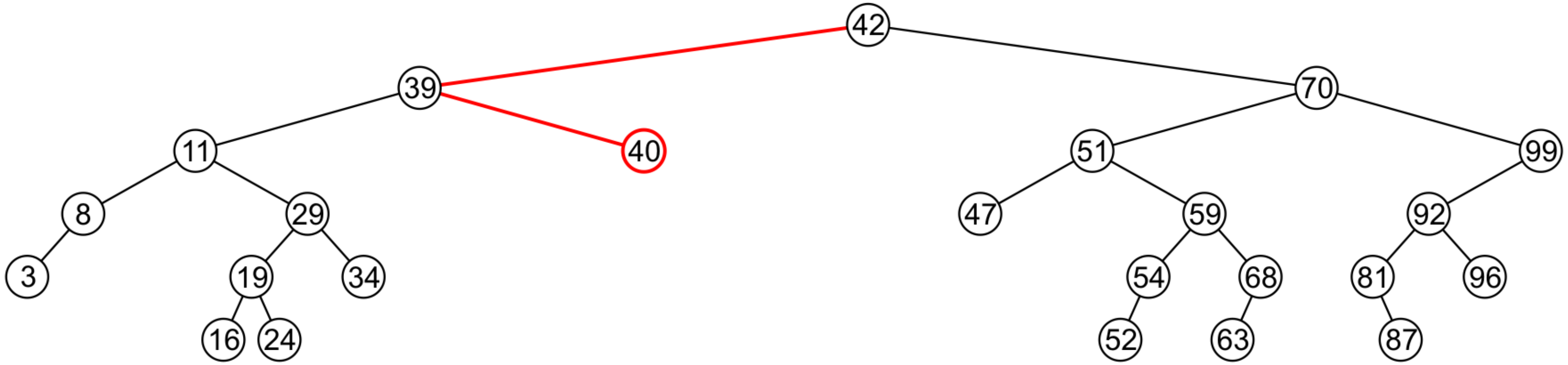
Erase

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



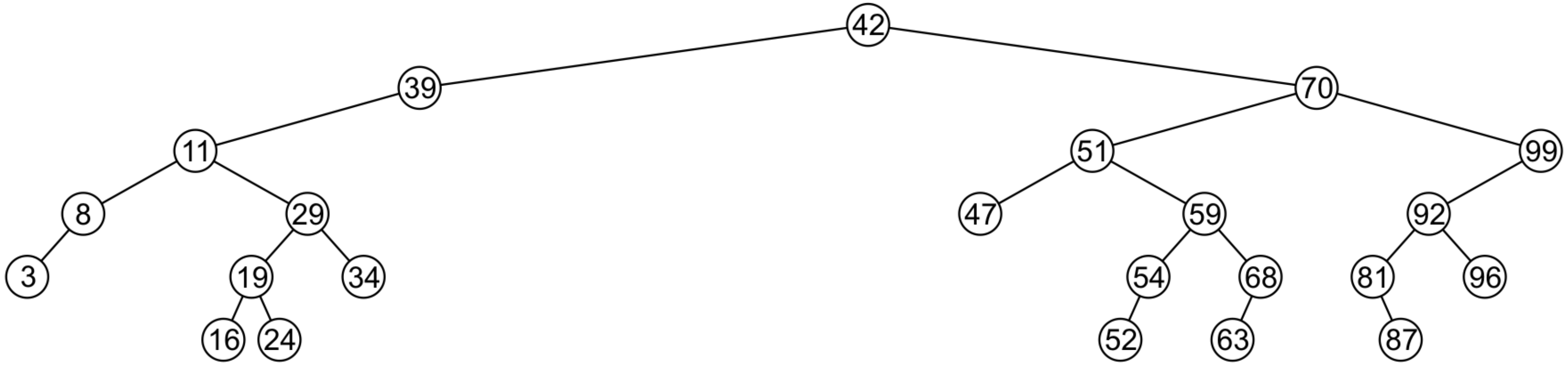
Erase

Erasing the node containing 40 is similar



Erase

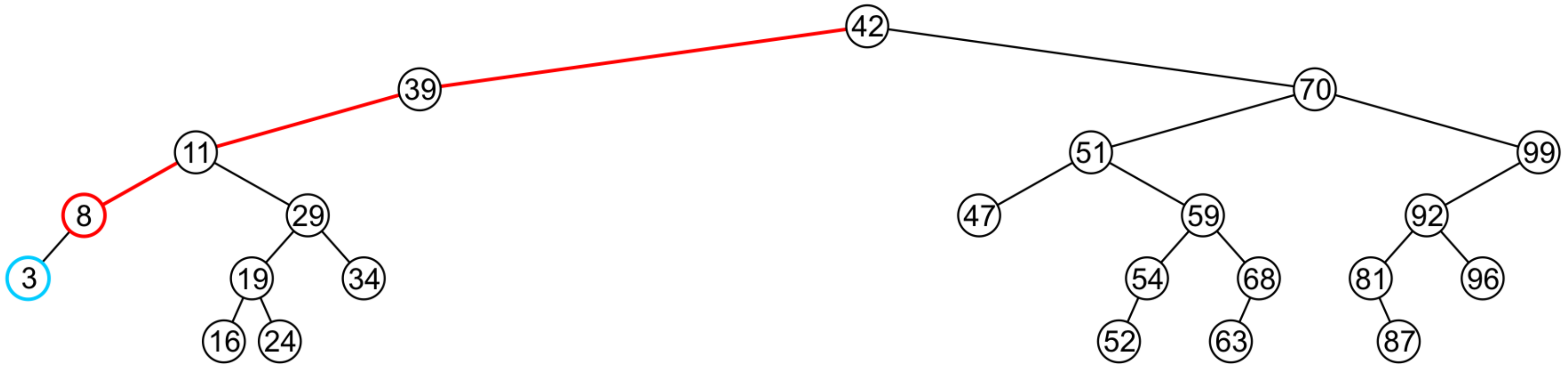
The node is deleted and `right_tree` of 39 is set to `nullptr`



Erase

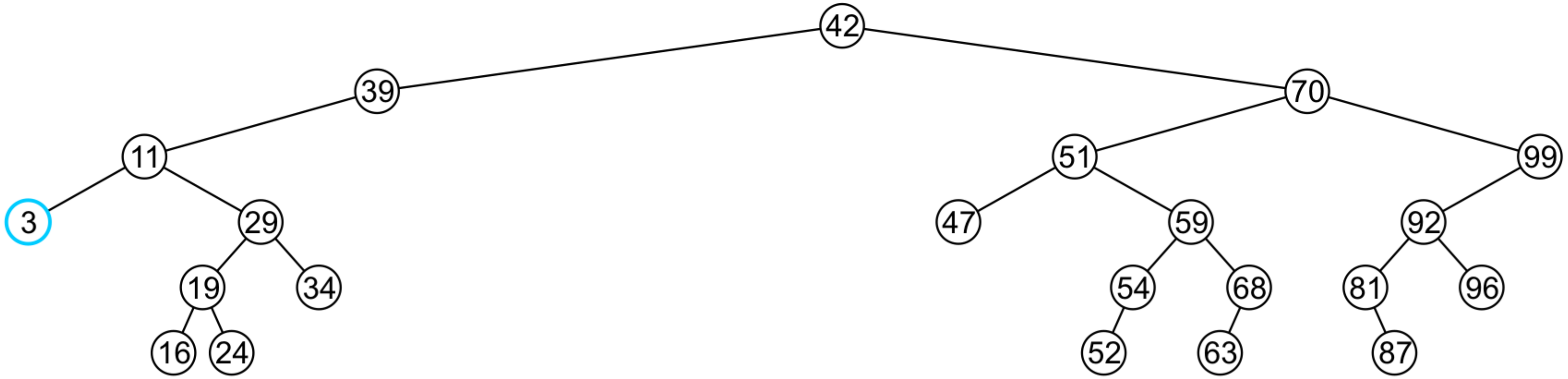
If a node has only one child, we can simply promote the sub-tree associated with the child

- Consider removing 8 which has one left child



Erase

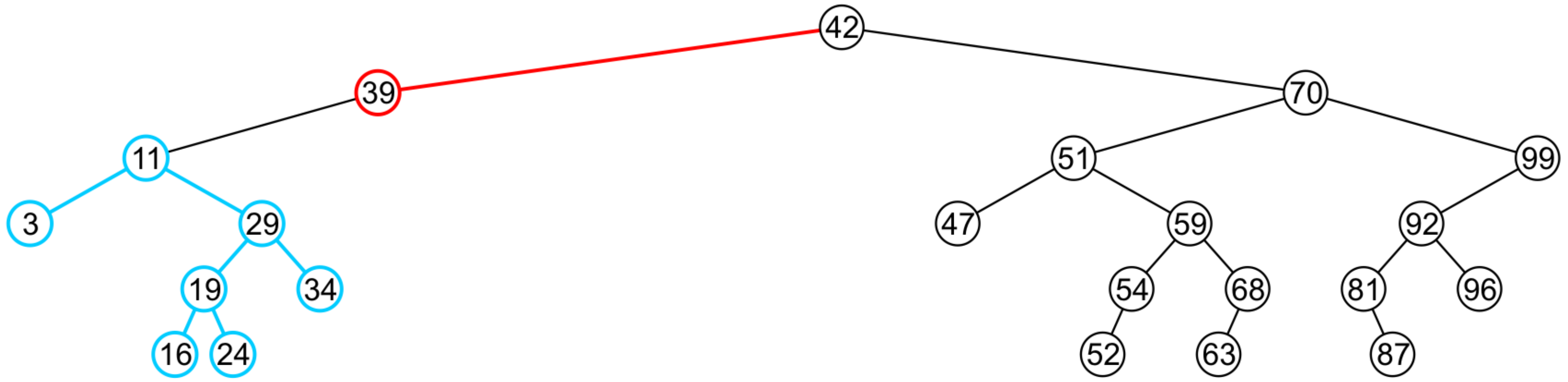
The node 8 is deleted and the `left_tree` of 11 is updated to point to 3



Erase

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

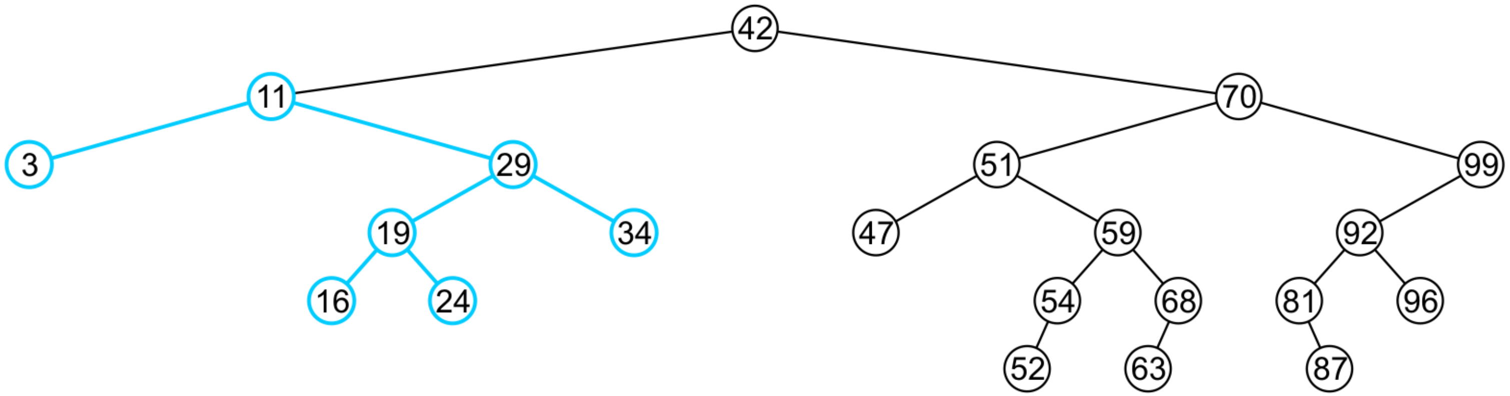
- To remove 39, it has a single child 11



Erase

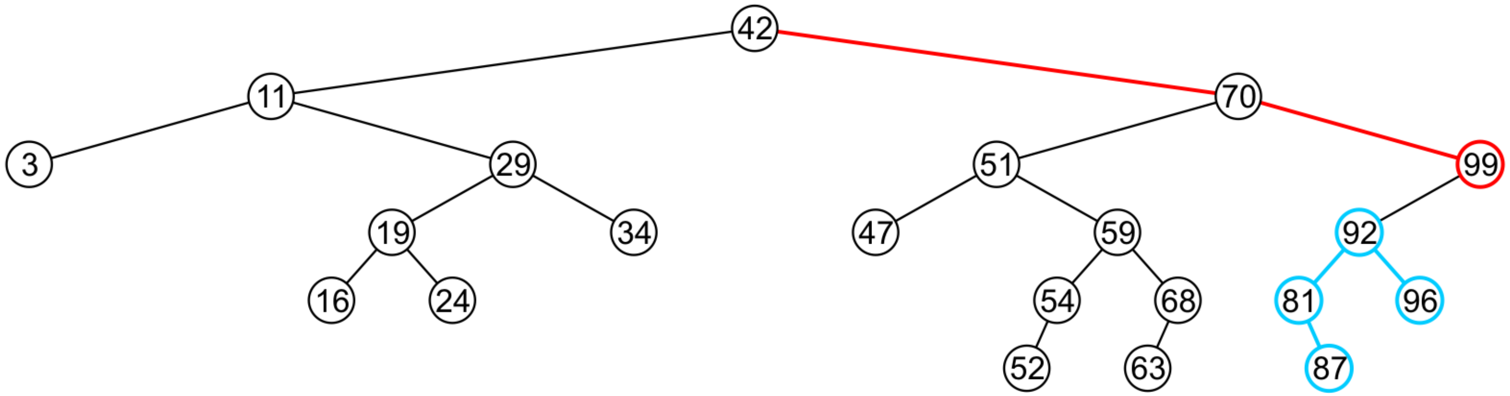
The node containing 39 is deleted and `left_node` of 42 is updated to point to 11

- Notice that order is still maintained



Erase

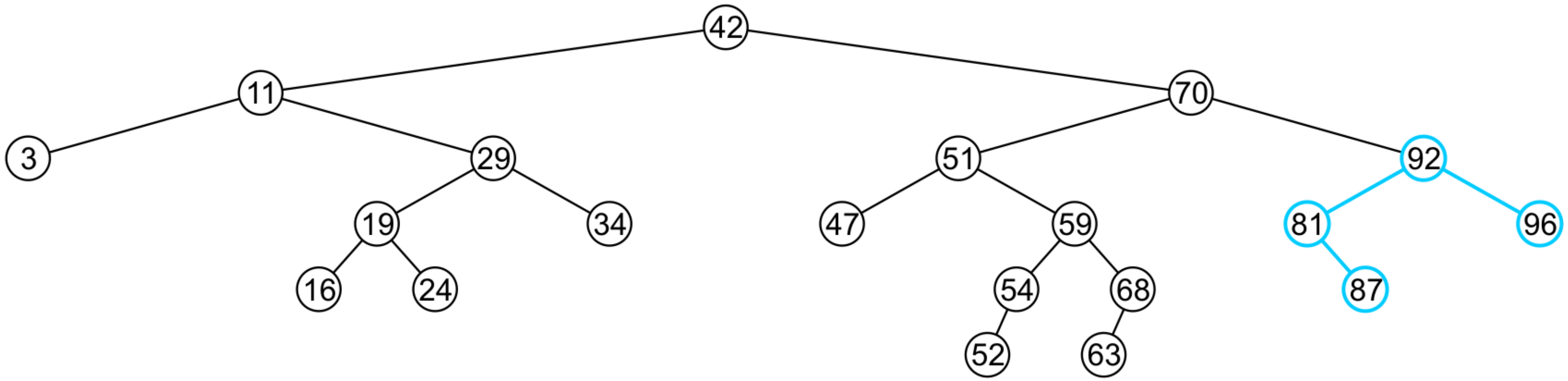
Consider erasing the node containing 99



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

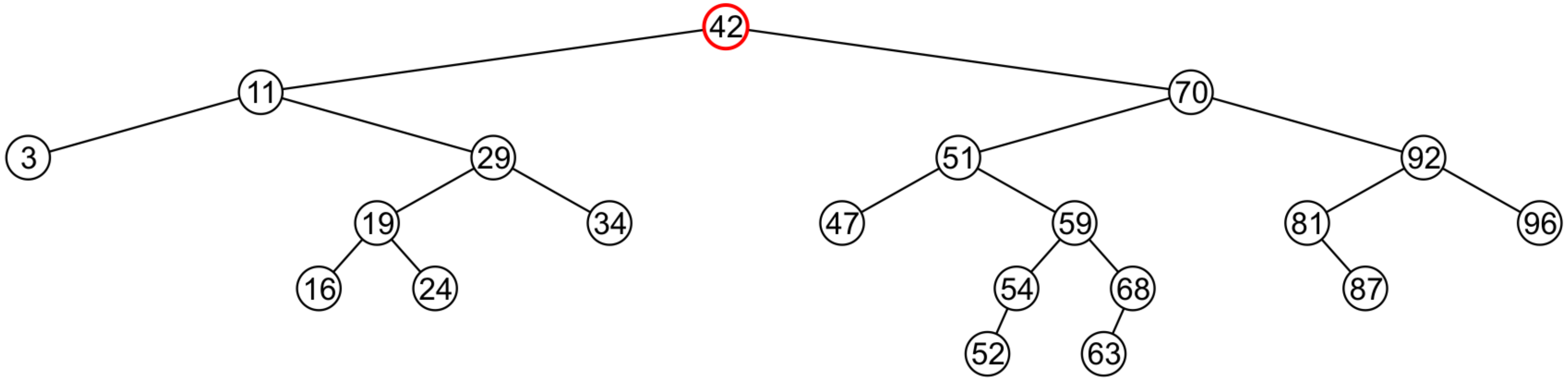


Erase

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

We will perform two operations:

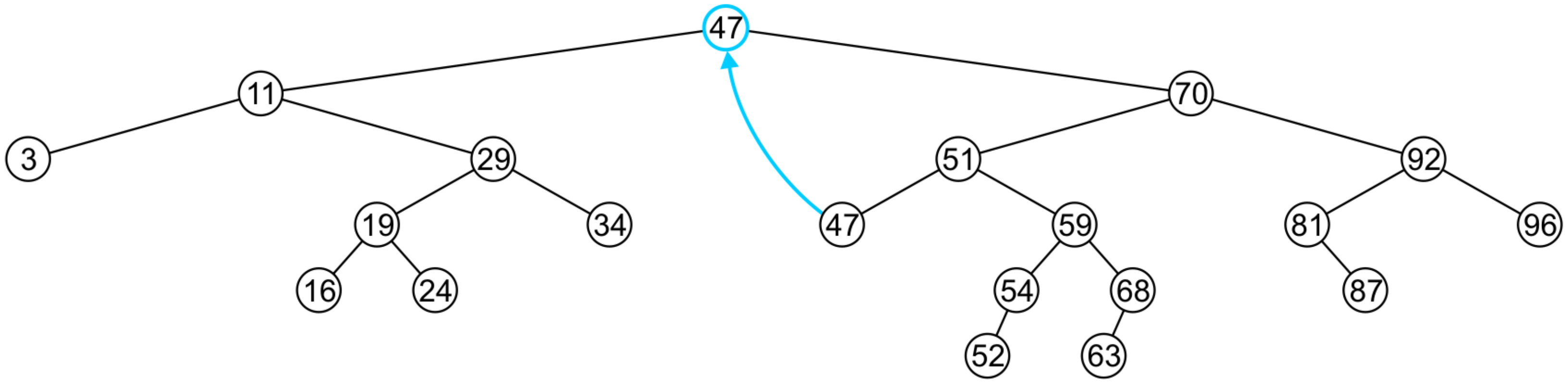
- Replace 42 with the minimum object in the right sub-tree
- Erase that object from the right sub-tree



Erase

In this case, we replace 42 with 47

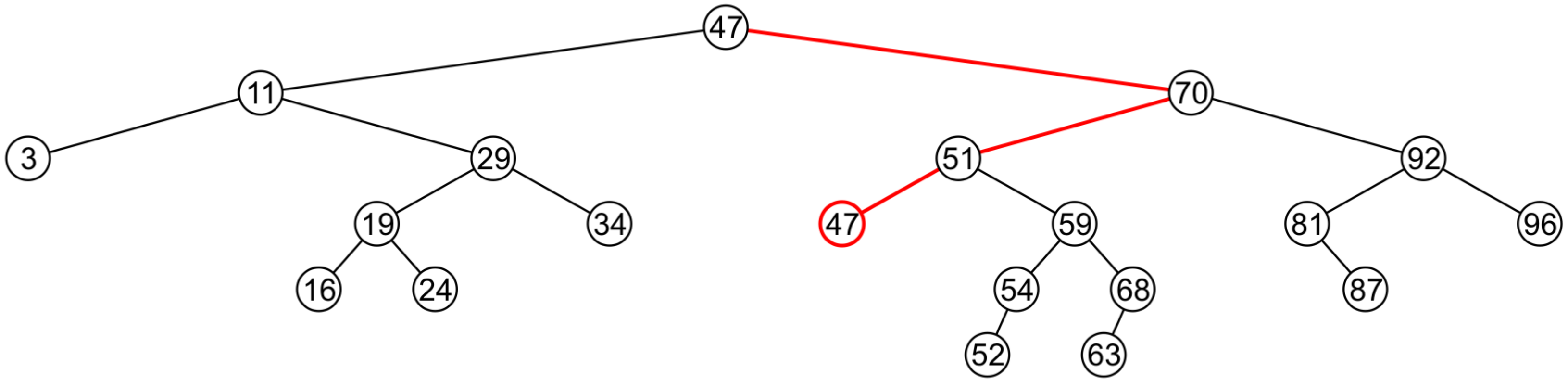
- We temporarily have two copies of 47 in the tree



Erase

We now recursively erase 47 from the right sub-tree

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

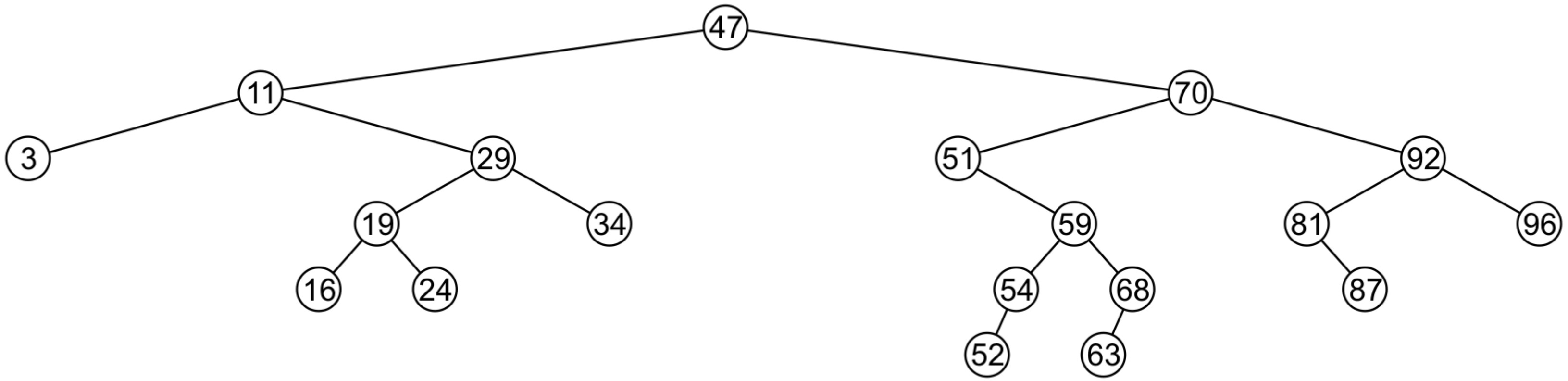


Erase

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

- Notice that the tree is still sorted:

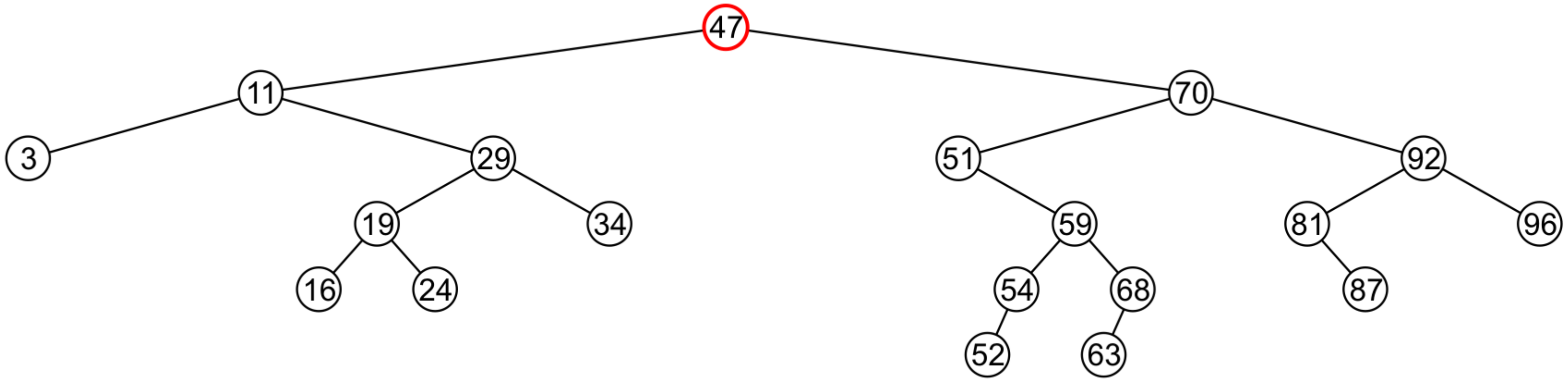
  47 was the least object in the right sub-tree



Erase

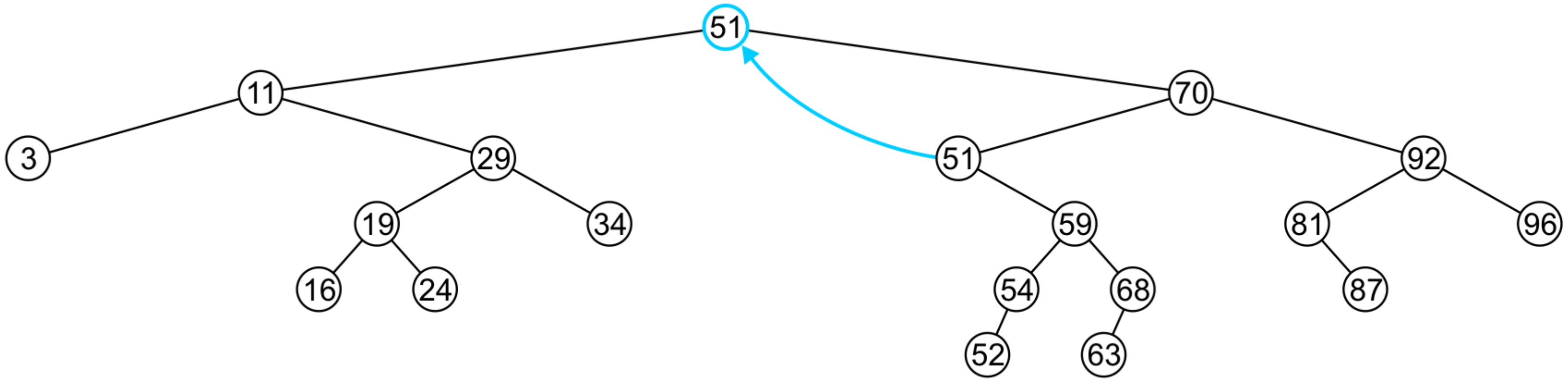
Suppose we want to erase the root 47 again:

- 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



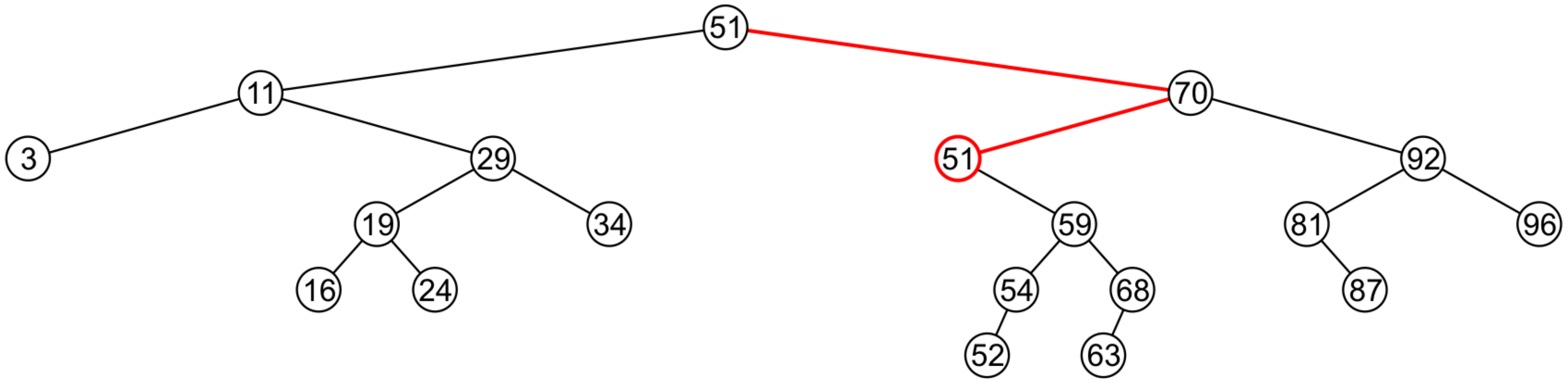
Erase

We copy 51 from the right sub-tree



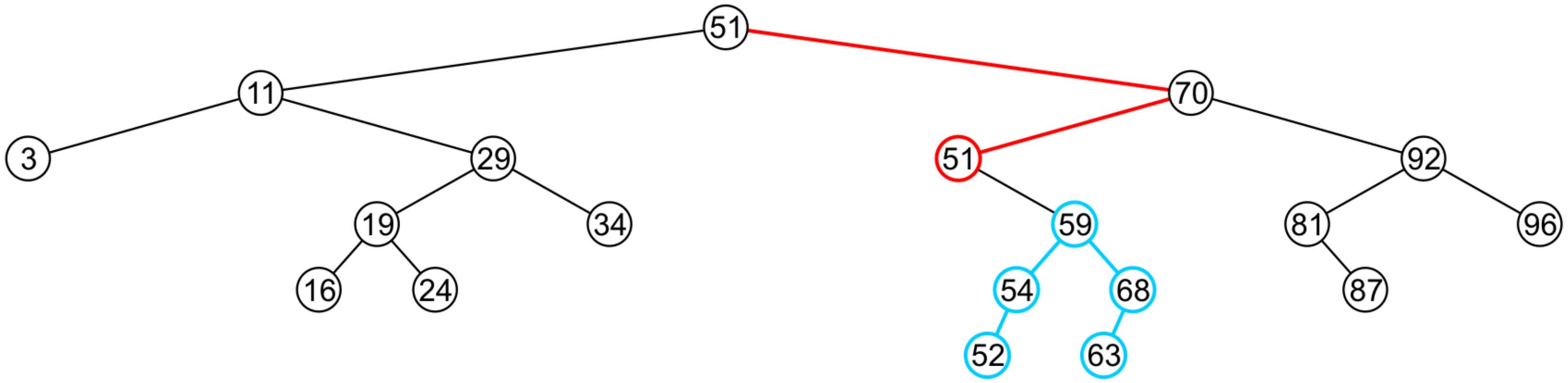
Erase

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



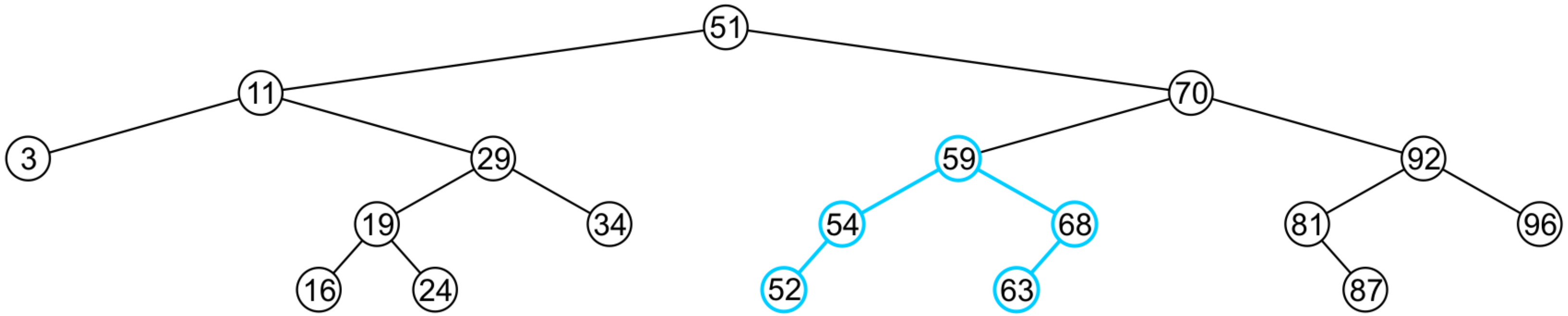
Erase

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



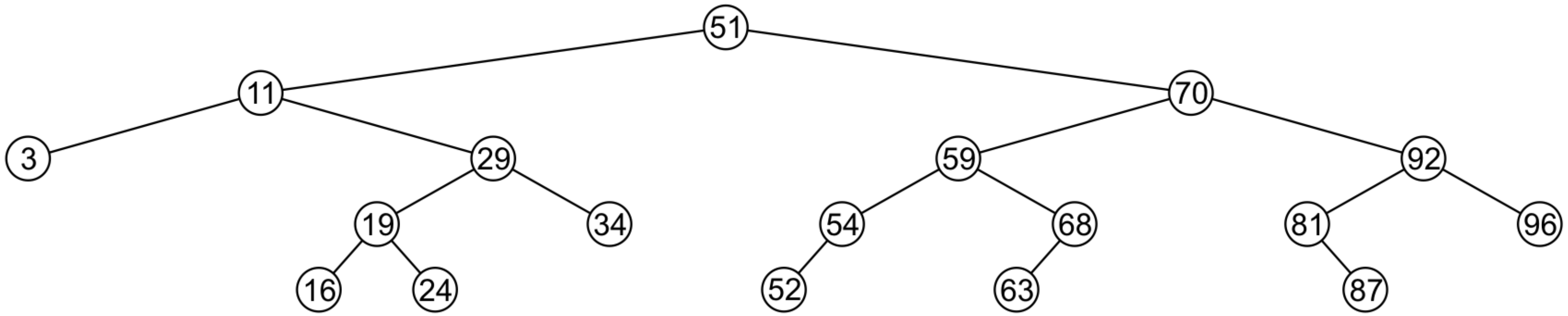
Erase

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

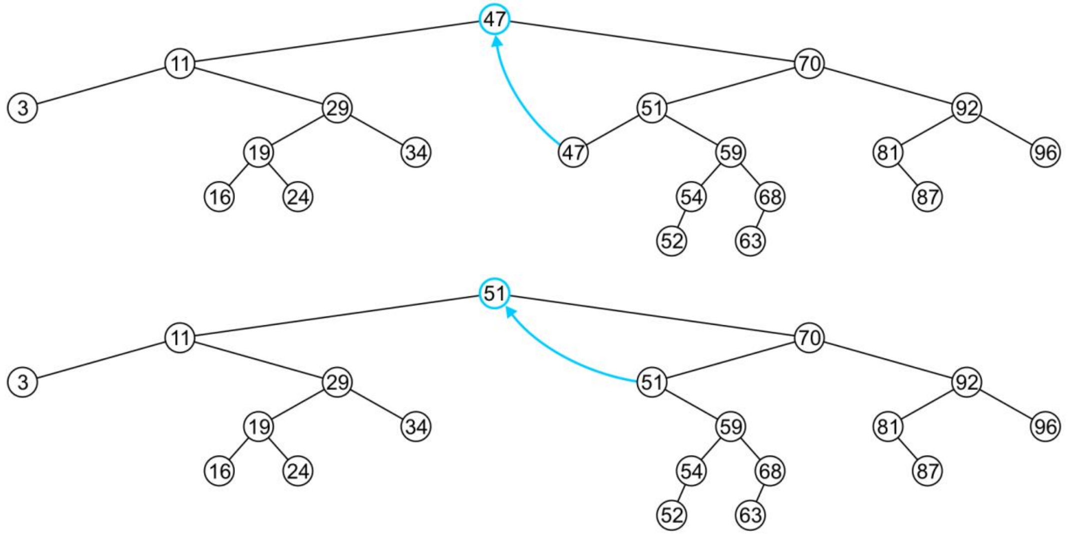


Erase

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

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

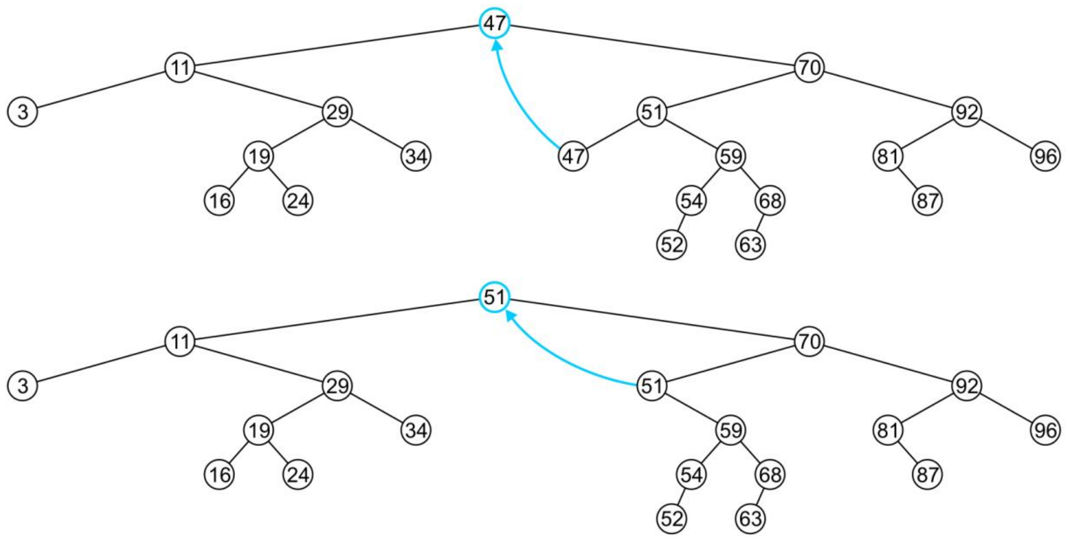
Is it possible, in removing a full node, to promote a child with two children?



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 contain a smaller value



Erase

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

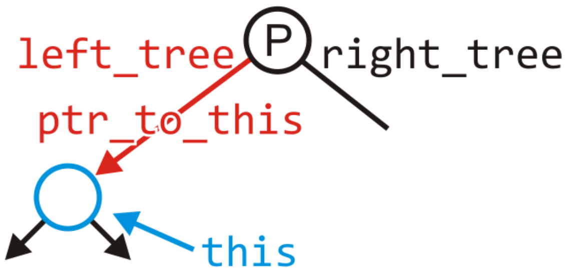
- To do this, we will pass that member variable by reference

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

Erase

```
template <typename Type>
bool Binary_search_node<Type>::erase( 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 );
    }
}
```



Erase

Blackboard example:

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

Binary Search Tree

We have defined binary search nodes

- Similar to the `Single_node` in Project 1

We must now introduce a container which stores the root

- A `Binary_search_tree` class

Most operations will be simply passed to the root node

Implementation

```
template <typename Type>
class Binary_search_tree {
    private:
        Binary_search_node<Type> *root_node;
        Binary_search_node<Type> *root() const;
    public:
        Binary_search_tree();
        ~Binary_search_tree();

        bool empty() const;
        int size() const;
        int height() const;
        Type front() const;
        Type back() const;
        int count( Type const &obj ) const;

        void clear();
        bool insert( Type const &obj );
        bool erase( Type const &obj );
};
```

Constructor, Destructor, and Clear

```
template <typename Type>
Binary_search_tree<Type>::Binary_search_tree():
root_node( nullptr ) {
    // does nothing
}

template <typename Type>
Binary_search_tree<Type>::~Binary_search_tree() {
    clear();
}

template <typename Type>
void Binary_search_tree<Type>::clear() {
    root()->clear( root_node );
}
```

Constructor, Destructor, and Clear

```
template <typename Type>
Binary_search_tree<Type> *Binary_search_tree<Type>::root() const {
    return tree_root;
}

template <typename Type>
bool Binary_search_tree<Type>::empty() const {
    return root()->empty();
}

template <typename Type>
int Binary_search_tree<Type>::size() const {
     return root()->size();
}
```

Empty, Size, Height and Count

```
template <typename Type>
int Binary_search_tree<Type>::height() const {
     return root()->height();
}

template <typename Type>
bool Binary_search_tree<Type>::find( Type const &obj ) const {
     return root()->find( obj );
}
```

Front and Back

```
// If root() is nullptr, 'front' will throw an underflow exception
template <typename Type>
Type Binary_search_tree<Type>::front() const {
    return root()->front();
}

// If root() is nullptr, 'back' will throw an underflow exception
template <typename Type>
Type Binary_search_tree<Type>::back() const {
    return root()->back();
}
```

Insert and Erase

```
template <typename Type>
bool Binary_search_tree<Type>::insert( Type const &obj ) {
    return root()->insert( obj, root_node );
}

template <typename Type>
bool Binary_search_tree<Type>::erase( Type const &obj ) {
    return root()->erase( obj, root_node );
}
```

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 k^{th} value of the container
- Iterate through those objects that fall on an interval $[a, b]$

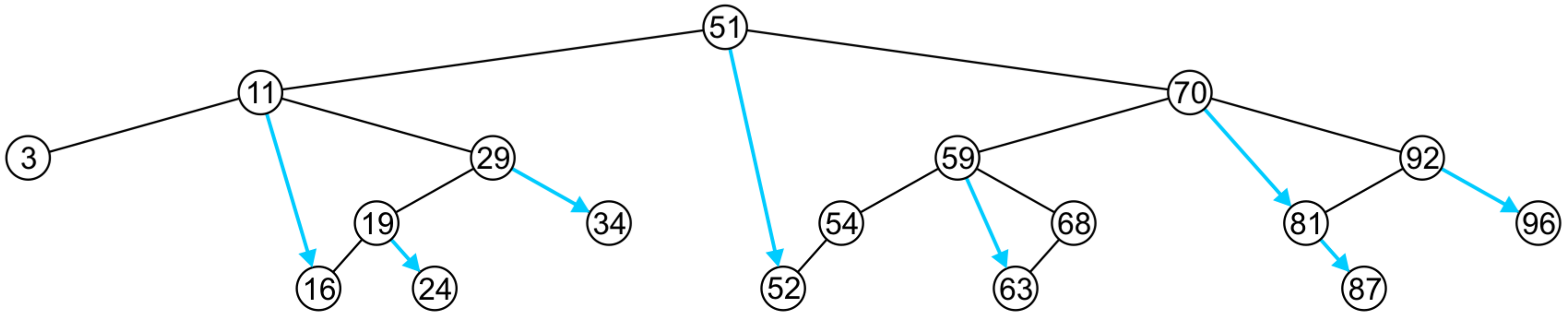
We will focus on finding the next largest object

- The others will follow

Previous and Next Objects

To find the next largest object:

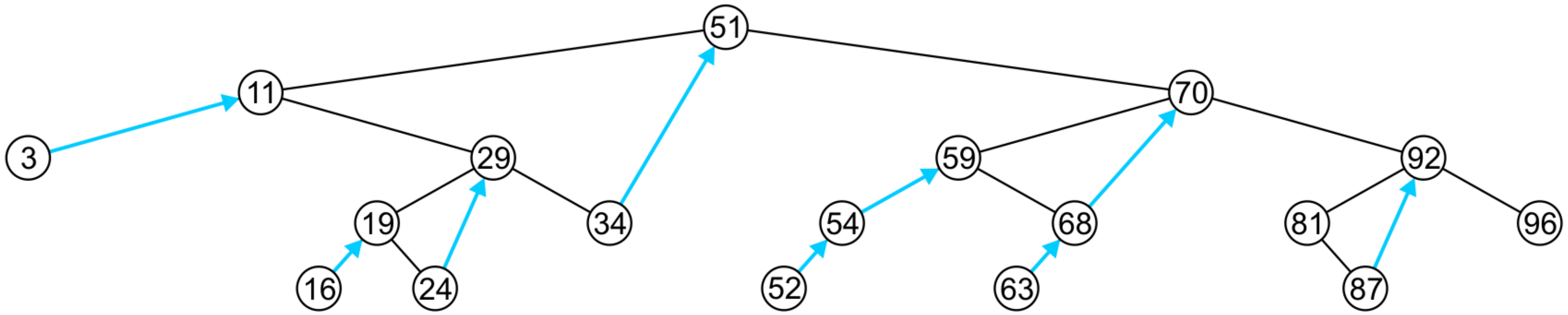
- If the node has a right sub-tree, the minimum object in that sub-tree is the next-largest object



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

```
template <typename Type>
Type Binary_search_node<Type>::next( Type const &obj ) const {
    if ( empty() ) {
        return obj;
    } else if ( value() == obj  ) {
        return ( right()->empty() ) ? obj : right()->front();
    } else if ( value() > obj ) {
        Type tmp = left()->next( obj );

        return ( tmp == obj ) ? value() : tmp;
    } else {
        return right()->next( obj );
    }
}
```

Run Time: $\mathrm{O}(h)$

Almost all of the 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 $\Theta(\ln(n))$

Others exist, too

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
    - Front, back, insert, erase
    - Previous smaller and next larger objects