COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides from “Algorithms and Data Structures” by Douglas Wilhelm Harder

Binary Search Trees

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`

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

Binary Search - recap

- If we keep the array in sorted order
- We can use binary search to find any item in the array in $\mathbf{O}(\log n)$
- But still insertion takes $\mathbf{O}(n)$ (we should keep the array sorted)

1	3	4	6	9	10	15	18	22	27	28	30	33

Binary Search - recap

- Assume we divide the array in two pieces and keep middle element separate
- When we insert, we only have to shift at most ½ as many elements
  - Time to find $\mathbf{O}(\log n)$
  - Time to insert/remove ½ original time

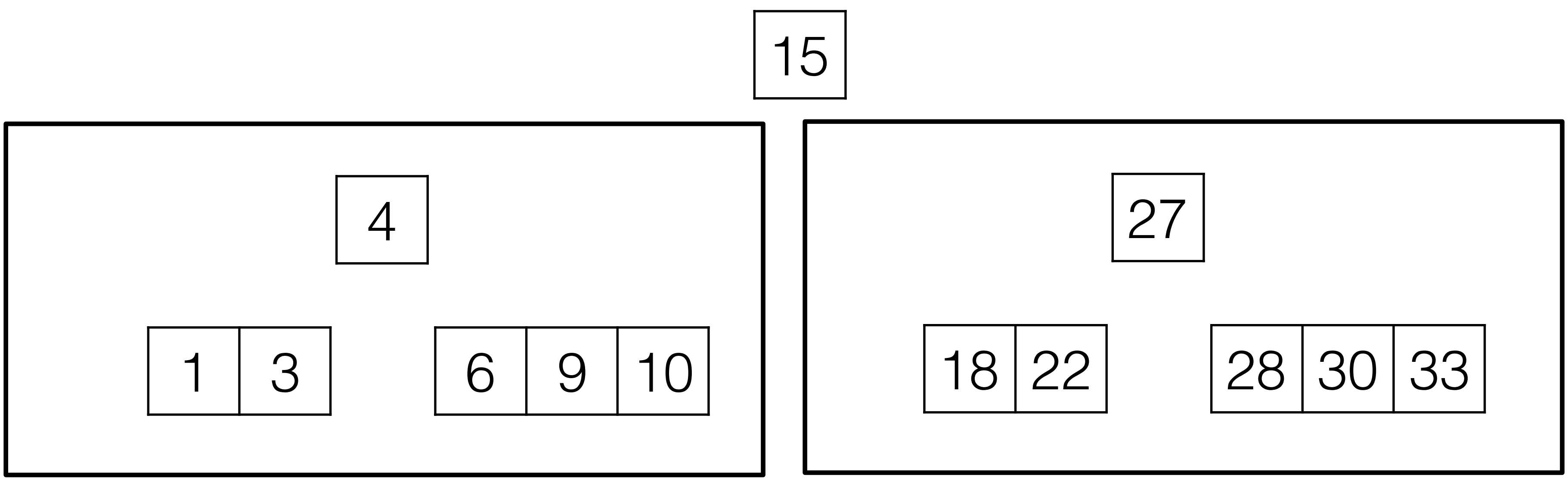
15

1	3	4	6	9	10

18	22	27	28	30	33

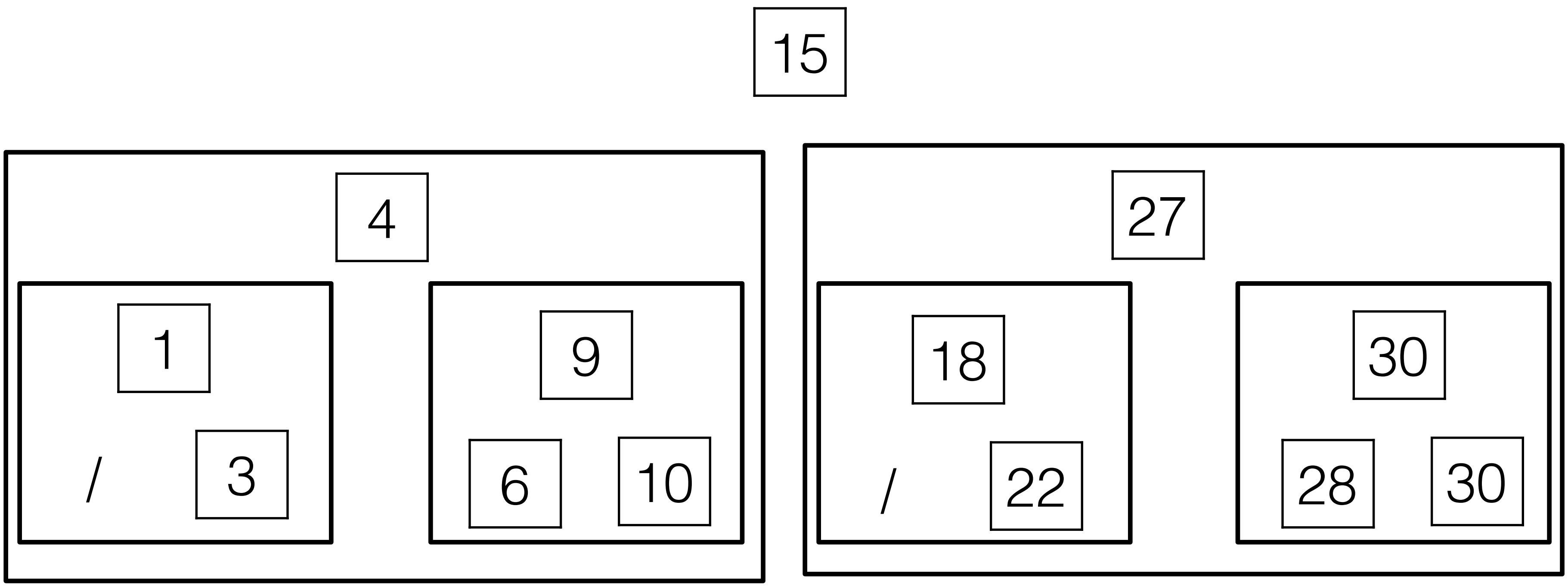
Binary Search - recap

- Time to find $\mathbf{O}(\log n)$
- Time to insert/remove ¼ original time



Binary Search - recap

- and so on …
- Keep splitting and eventually we get rid of array all together.



Binary Search Tree

- We’ve just invented Binary Search Tree!
- $\mathbf{O}(\log n)$ deep: We will see that the runtime for contains, add, remove are all $\mathbf{O}(\log n)$ if we keep the tree balanced!

Recursive Definition

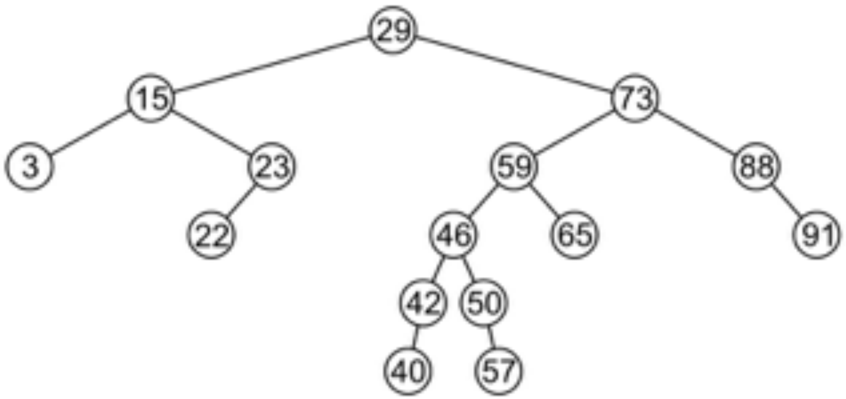
- A binary tree T is a binary search tree if
    - T is empty,

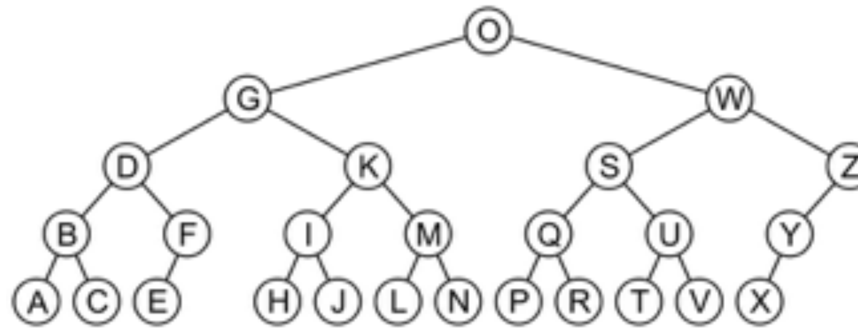
  or

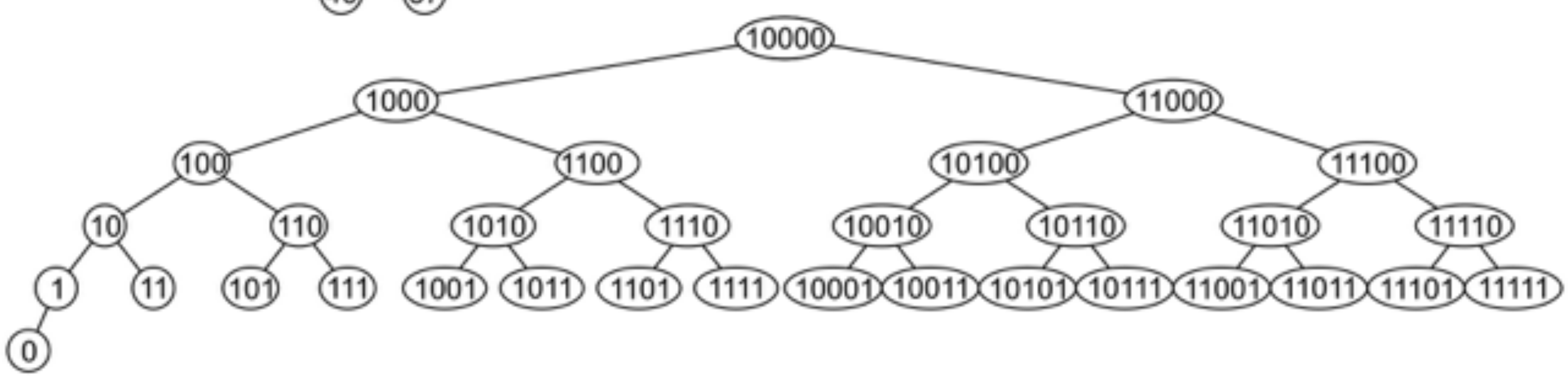
    - T has two subtrees T_L and T_R, where
  - all the values in T_L are less than the root of T,
  - all the values in T_R are greater than the root of T

Examples

Here are other examples of binary search trees:

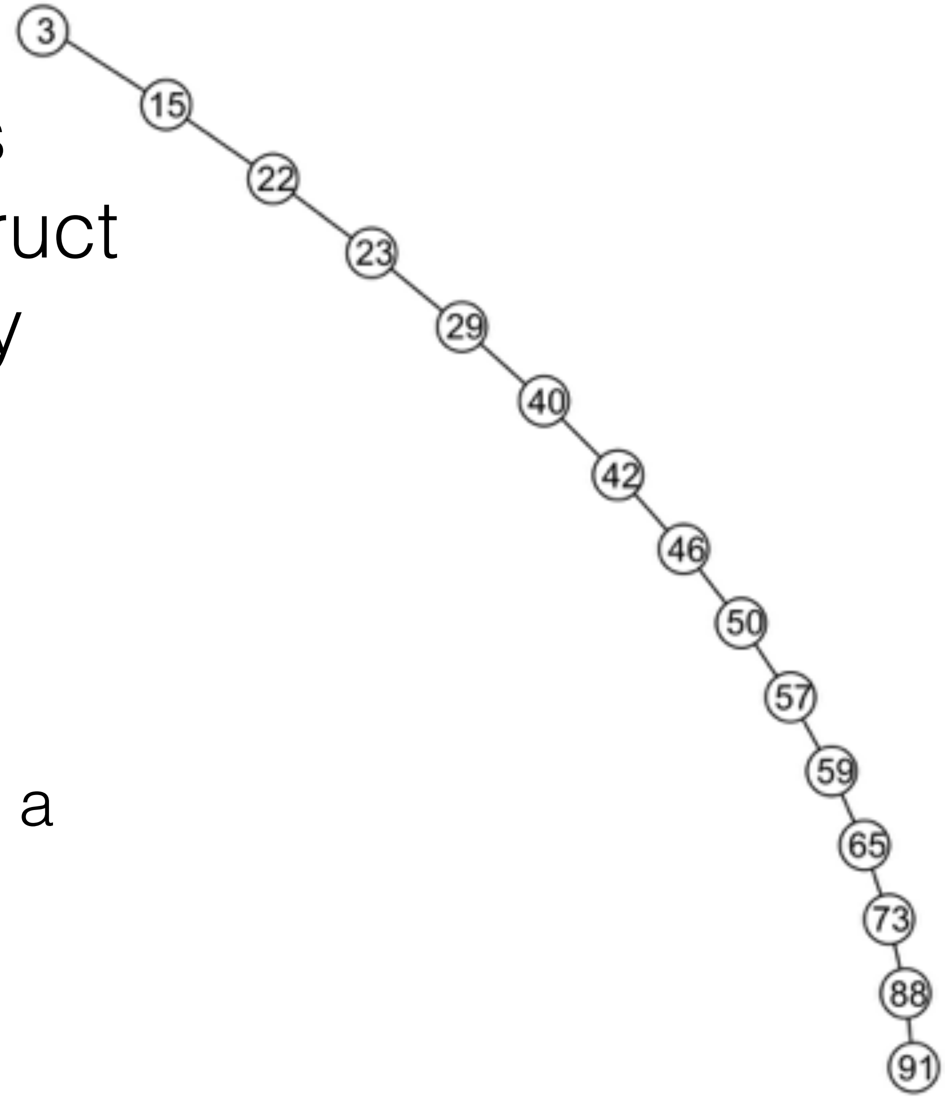






Examples

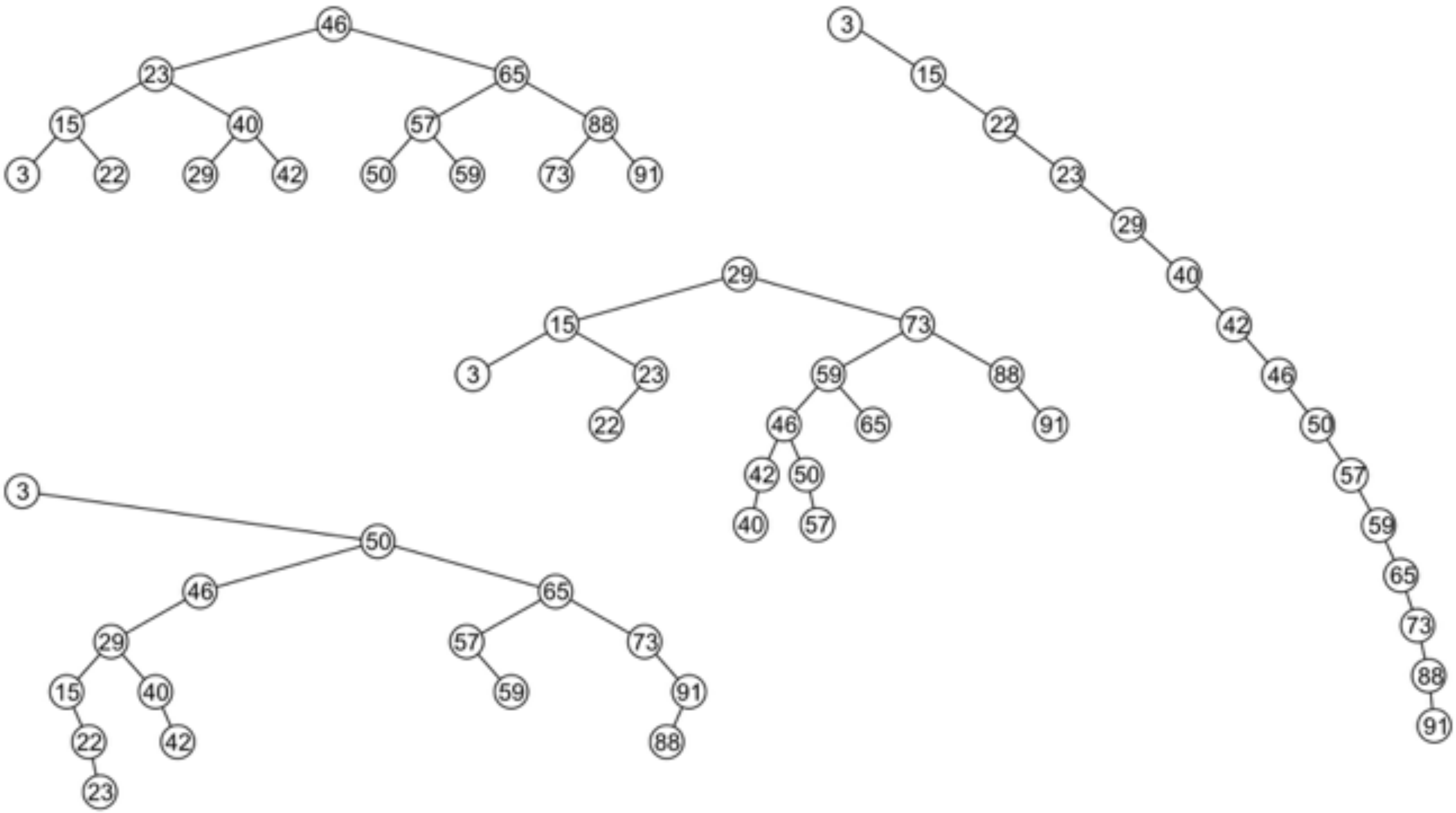
Unfortunately, it is possible to construct *degenerate* binary search trees



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

Examples

All these binary search trees store the same data



Duplicate Elements

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

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

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

Implementation

We will look at an implementation of a binary search tree

- We will have a `BinaryNode` class
- A `BinarySearchTree` class will store a reference to the root node

We will write a generic implementation, however, we will require that the class implements the compareTo() method

Implementation

Any class which uses this binary-search-tree class must implement the compareTo method:

```
public int compareTo(Object obj);
// it compare the given object obj with the current object (this)
```

This method allows us to compare two instances of this class (`obj1.compareTo(obj2)`)

- returns 0 if they are equal
- returns -1 if `obj1` is less than `obj2`
- returns +1 if `obj1` is greater than `obj2`

Binary Node Class

We use the BinaryNode class:

```
class BinaryNode<AnyType>
{
    AnyType             element;
    BinaryNode<AnyType> left;
    BinaryNode<AnyType> right;

    public BinaryNode( ){
        this( null, null, null );
    }

    public BinaryNode( AnyType theElement, BinaryNode<AnyType> lt,
                              BinaryNode<AnyType> rt) {
        element = theElement;
        left    = lt;
        right   = rt;
    }
}
```

Implementation

```java
/** The tree root. */
protected BinaryNode<AnyType> root;

/** public methods. */
public BinarySearchTree() // --> constructor (create an empty tree)
void insert( x )       // --> Insert x
void remove( x )       // --> Remove x
void removeMin( )      // --> Remove minimum item
AnyType find( x )      // --> Return item that matches x
AnyType findMin( )     // --> Return smallest item
AnyType findMax( )     // --> Return largest item
boolean isEmpty( )     // --> Return true if empty; else false
void clear( )          // --> Remove all items
```

Note: AnyType is comparable! (i.e. class AnyType must implement the compareTo method)

Constructor

The constructor creates an empty binary search tree (simply set root to null)

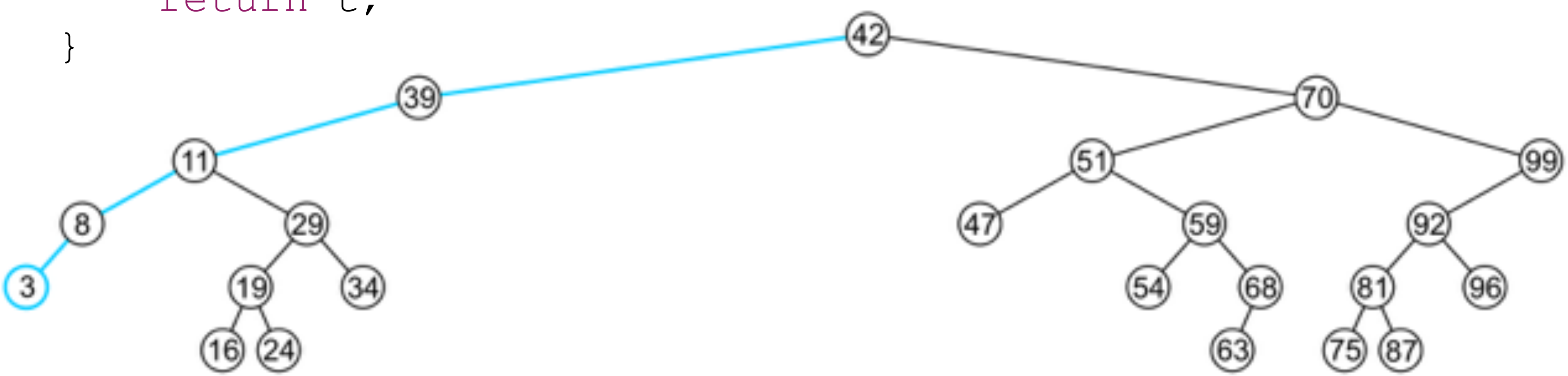
```
public BinarySearchTree( )
{
        root = null;
}
```

Finding the Minimum Object

```
public AnyType findMin( ){
    return elementAt( findMin( root ) );
}

protected BinaryNode<AnyType> findMin(BinaryNode<AnyType> t) {
    if( t != null )
        while( t.left != null )
            t = t.left;

    return t;
}
```



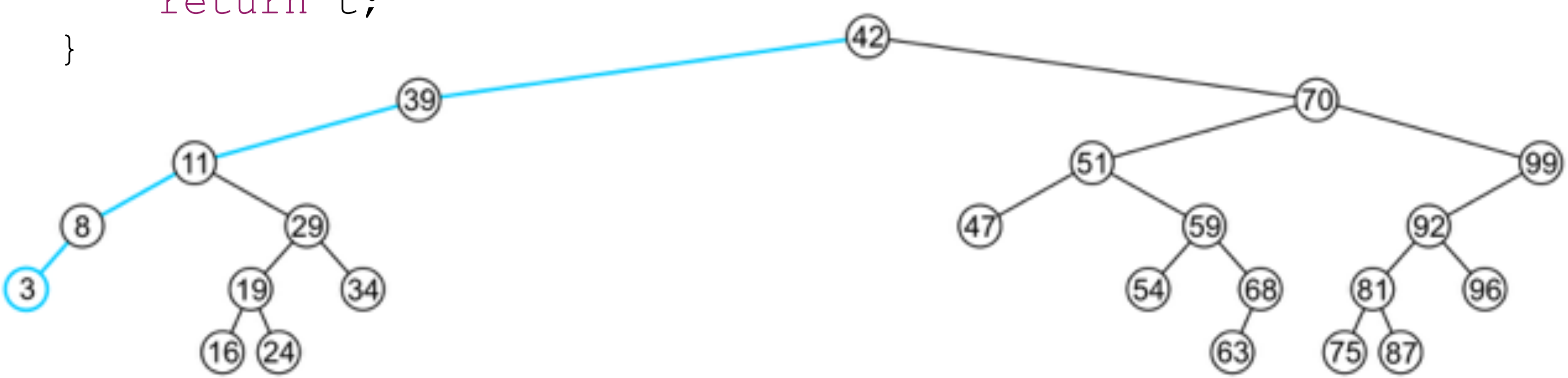
Finding the Minimum Object

```
public AnyType findMin( ){
    return elementAt( findMin( root ) );
}

protected BinaryNode<AnyType> findMin(BinaryNode<AnyType> t) {
    if( t != null )
        while( t.left != null )
            t = t.left;

    return t;
}
```

run time O(*h*)



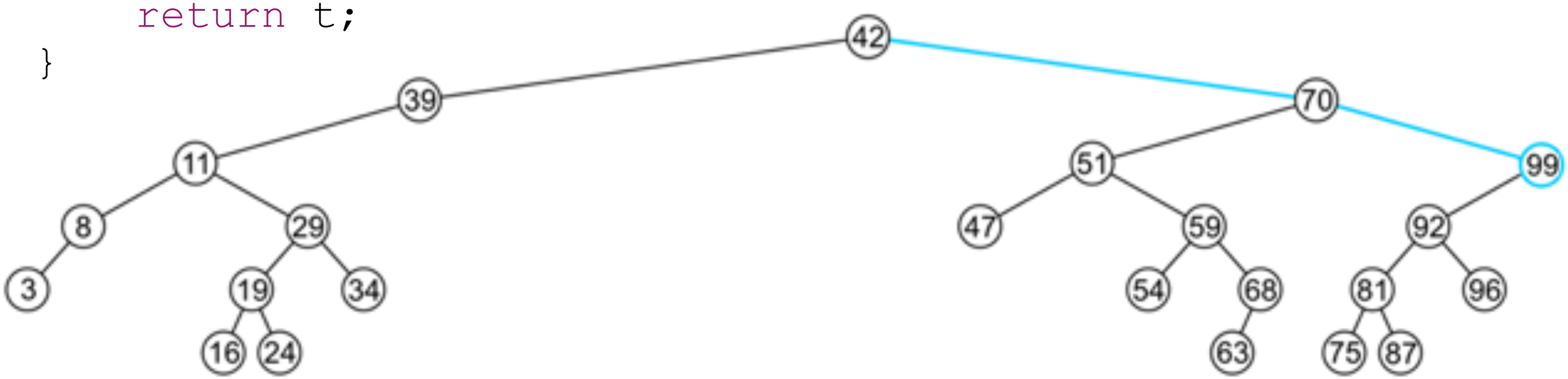
Finding the Maximum Object

```
public AnyType findMax( ){
    return elementAt( findMax( root ) );
}

protected BinaryNode<AnyType> findMax(BinaryNode<AnyType> t) {
    if( t != null )
        while( t.right != null )
            t = t.right;

    return t;
}
```

run time O(*h*)

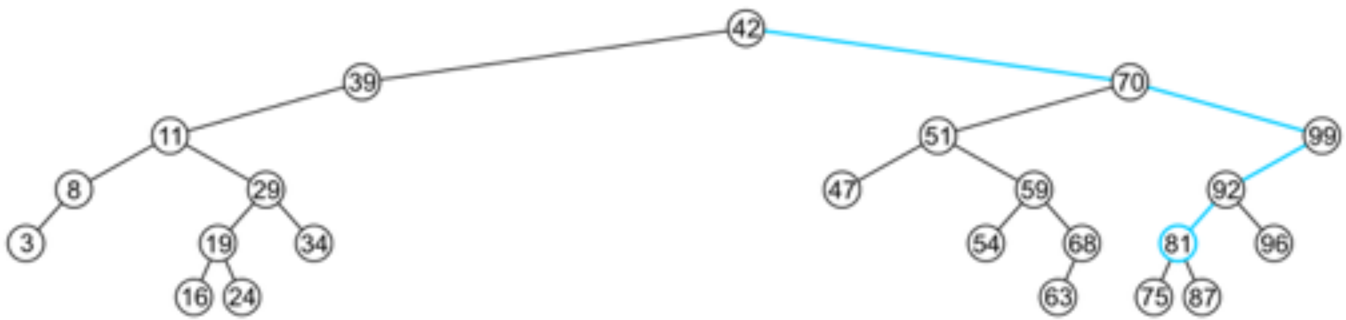


The extreme values are not necessarily leaf nodes

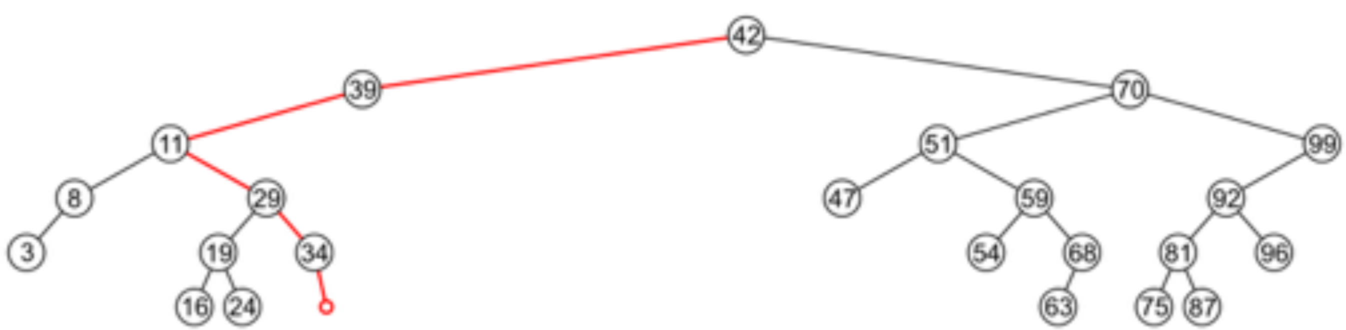
Find

To determine whether given value is in the BST, we traverse the tree to find the value

- 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

Starting at the root, we determine wether the value we are looking for:

- is in the root
- might be in the root's left subtree
- might be in the root's right subtree

We can write a recursive method or non-recursive method

Be careful, in recursive case, there are actually two base cases:

- The tree is empty; return false.
- The value is in the root node; return true.

Find

```
public AnyType find( AnyType x ){
    return elementAt( find( x, root ) );
}

protected BinaryNode<AnyType> find(Anytype x, BinaryNode<AnyType> t){
    while( t != null ){
        if( x.compareTo( t.element ) < 0 )
           t = t.left;
        else if( x.compareTo( t.element ) > 0 )
           t = t.right;
        else
           return t;    // x is matched to t.element
    }

    return null;          // Not found
}
```

run time O(h)

non-recursive implementation

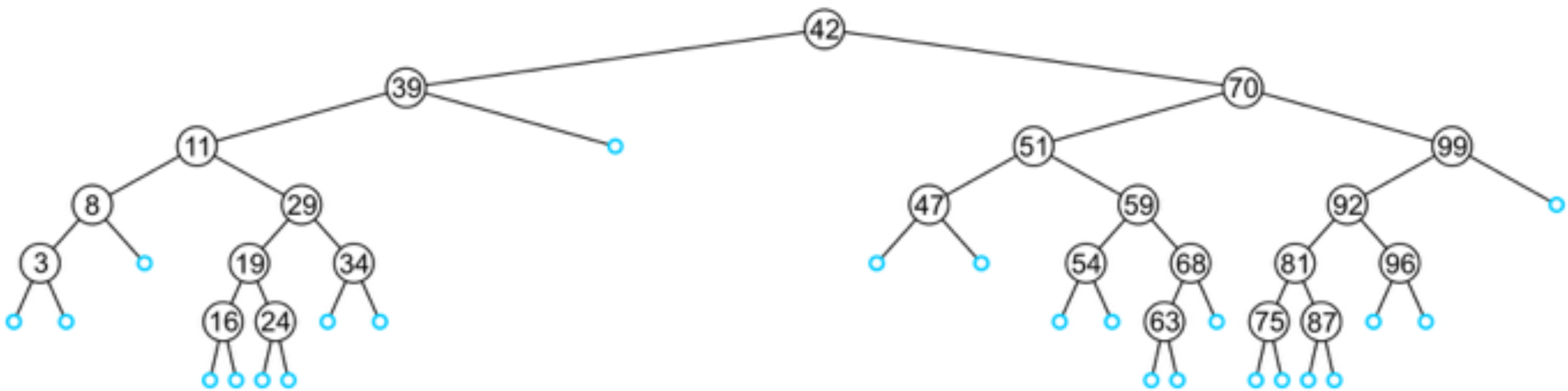
Insert

- Recall that a Sorted List is implicitly ordered
  - It does not make sense to have member functions such as `addFront` and `addEnd`
  - Insertion will be performed by a single `insert` member function which places the object into the correct location
    - It is the task of method `insert` to find the correct location for the given object

Insert

An insertion will be performed at a leaf node:

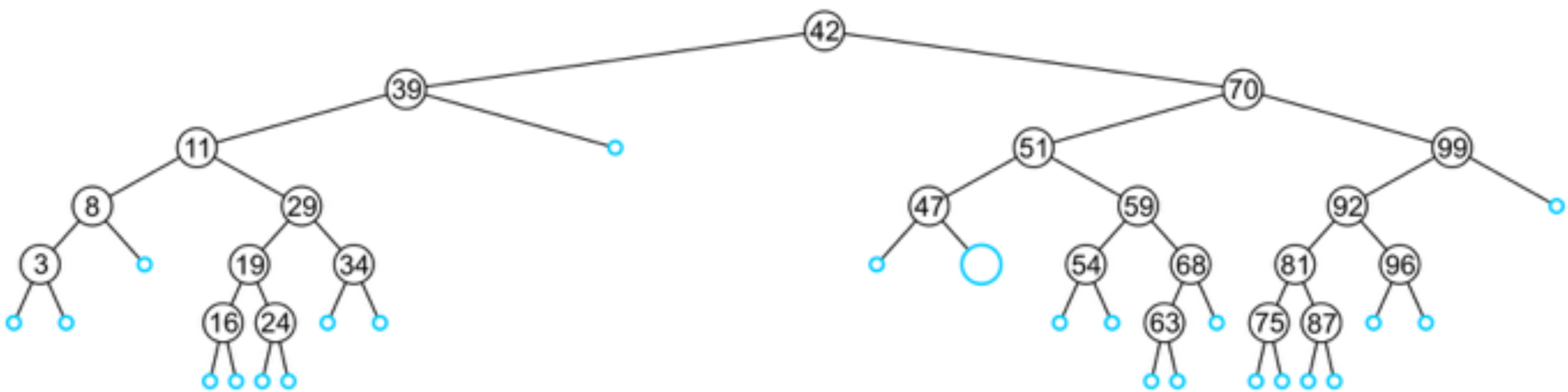
– Any empty node is a possible location for an insertion



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

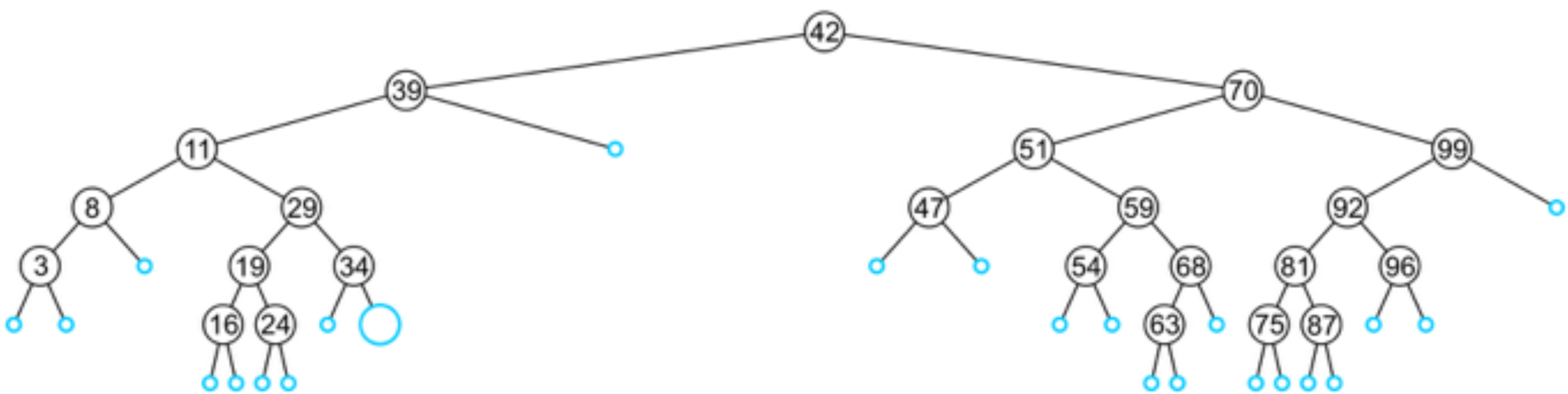
Insert

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



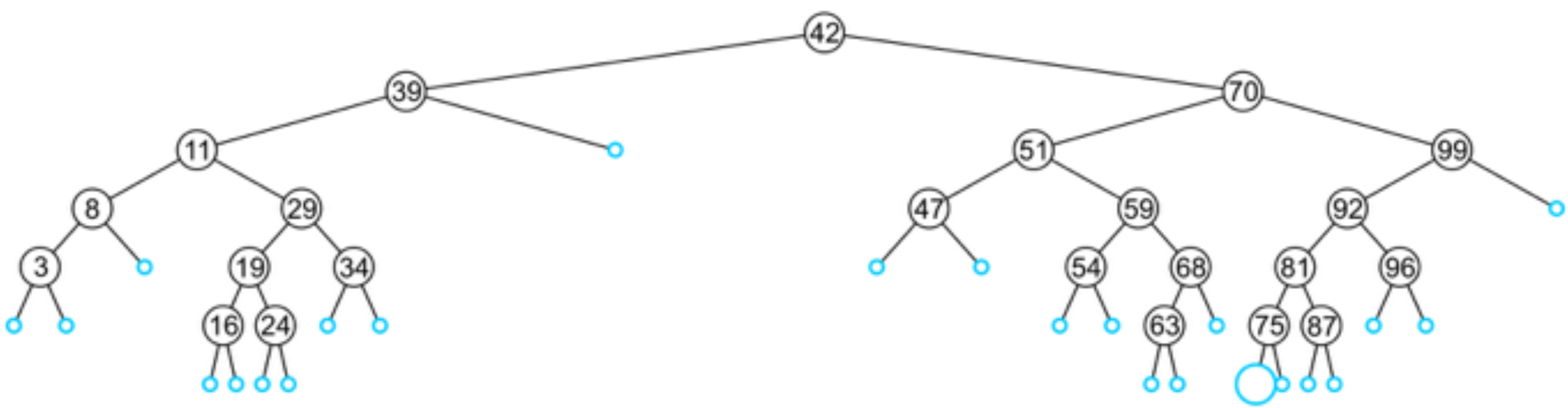
Insert

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



Insert

This empty node may hold values from 71 to 74



Insert

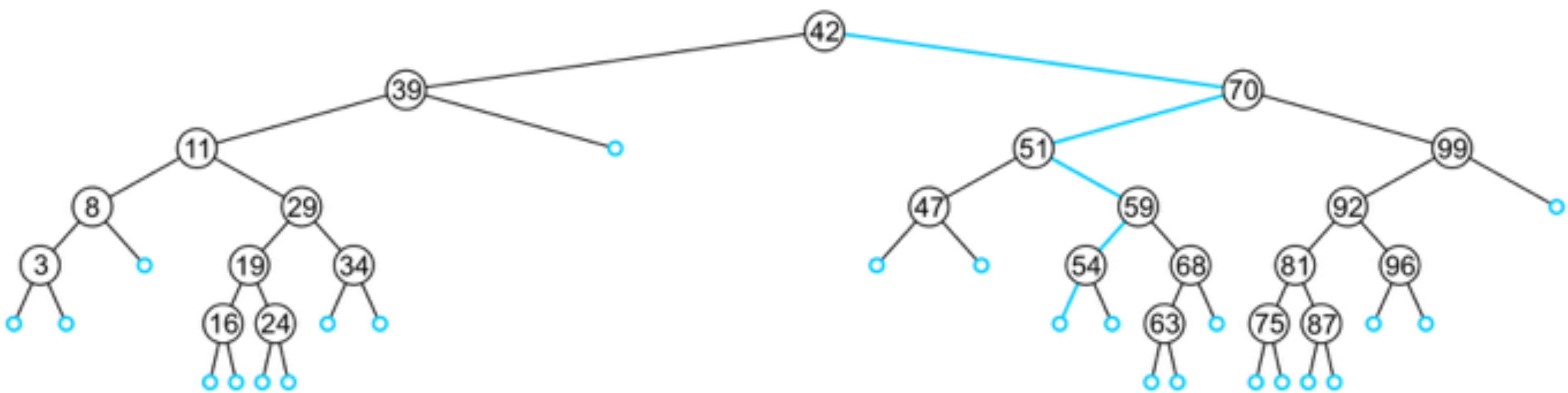
Like find, we will step through the tree

- If we find the object already in the tree, we will return
  - The object is already in the binary search tree (no duplicates) and we do not need to insert it again.
- Otherwise, we will arrive at an empty node
- The object will be inserted into that location
- The run time is $\mathbf{O}(h)$

Insert

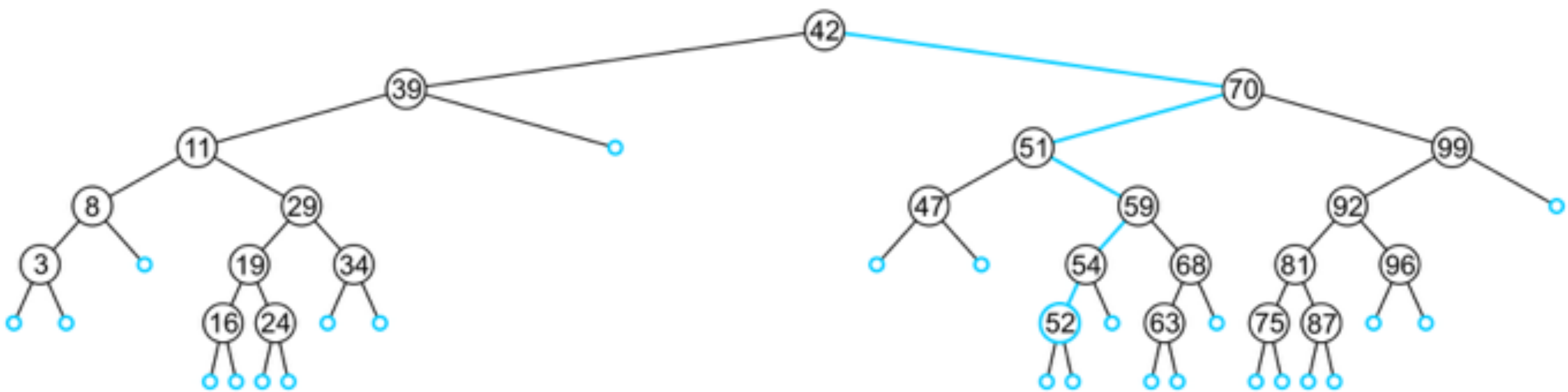
In inserting the value 52, we traverse the tree until we reach an empty node

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



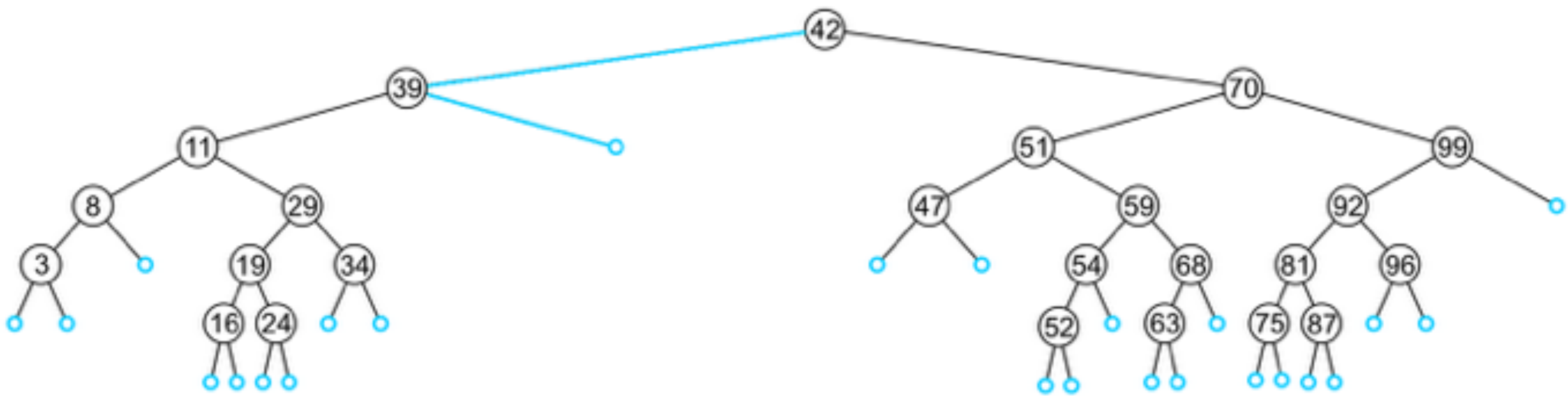
Insert

A new binary node (leaf) is created and assigned to the `left` child of the node 54



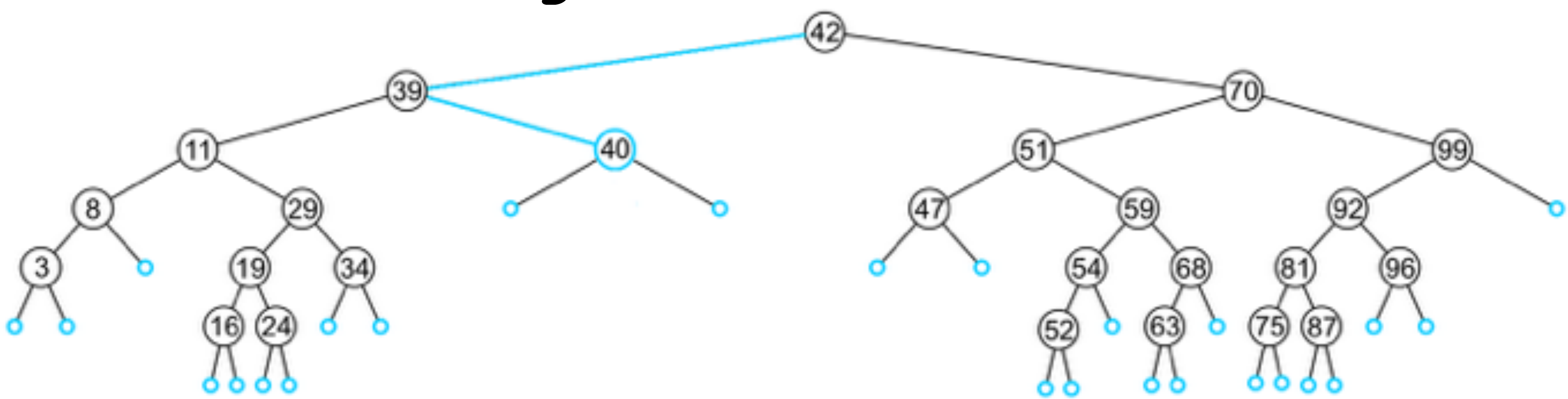
Insert

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



Insert

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



Insert

```
public AnyType insert( AnyType x ){
    return root = insert( x, root );
}

protected BinaryNode<AnyType> insert(Anytype x, BinaryNode<AnyType> t)
{
    if( t == null )
        t = new BinaryNode<AnyType>( x );
    else if( x.compareTo( t.element ) < 0 )
        t.left = insert( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = insert( x, t.right );
    else
        // Duplicate
        throw new DuplicateItemException( x.toString( ) );

    return t;
}
```

recursive implementation

Insert

example:

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

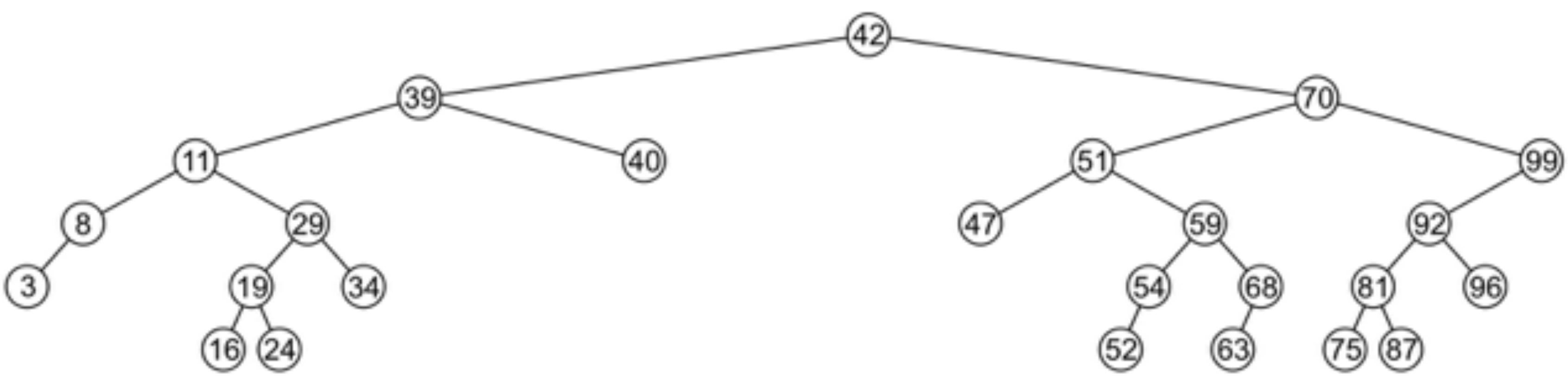
  31 45 36 14 52 42 6 21 73 47 26 37 33 8
- What values could be placed:
  - To the left of 21?
  - To the right of 26?
  - To the left of 47?
- How would we determine if 40 is in this binary search tree?
- Which values could be inserted to increase the height of the tree?

Remove

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

There are three possible scenarios:

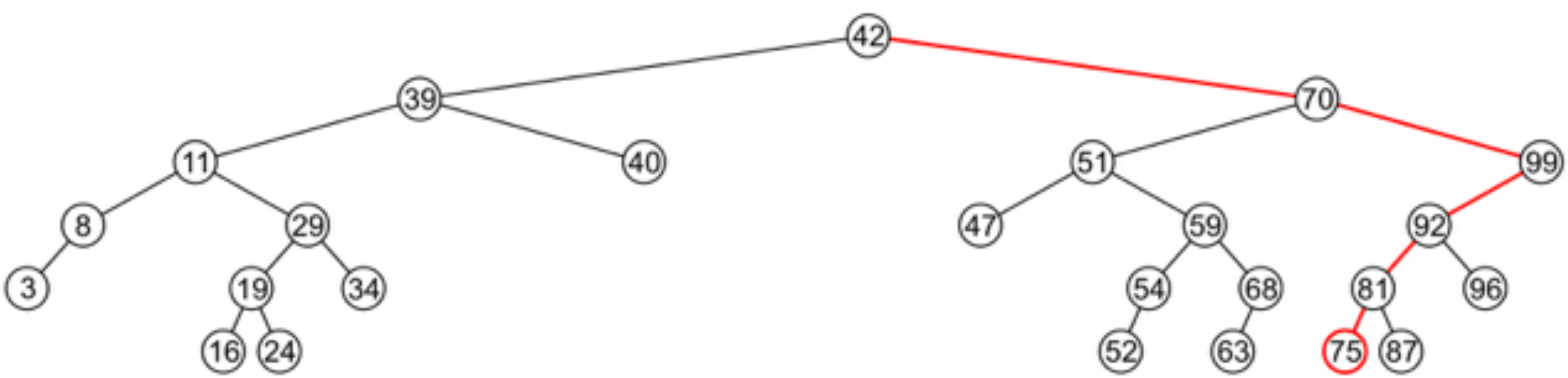
- The node is a leaf node,
- It has exactly one child, or
- It has two children (it is a full node)



Remove

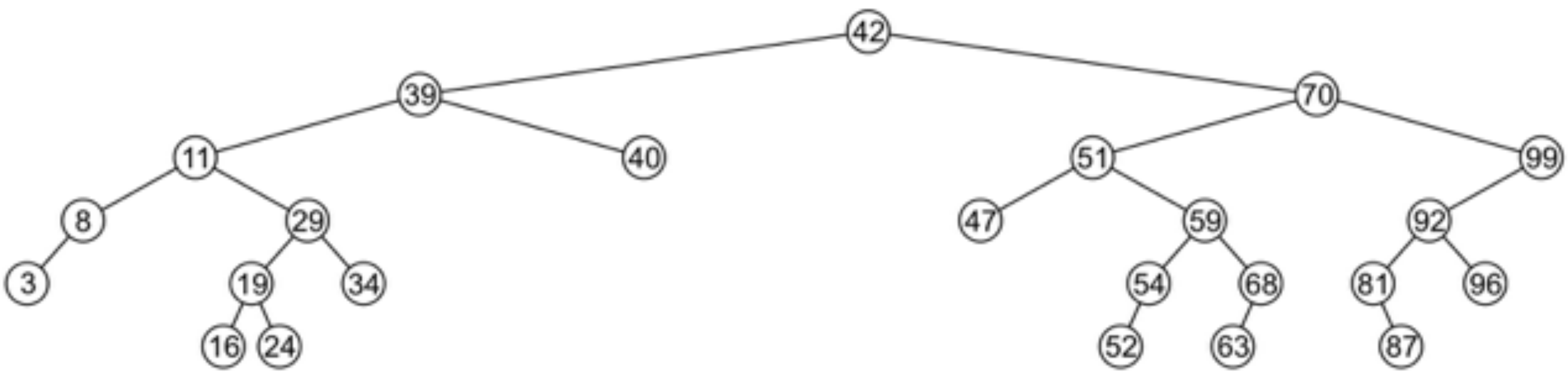
A leaf node simply must be removed and the appropriate field of the parent (`right` or `left` child) is set to `null`

- Consider removing 75



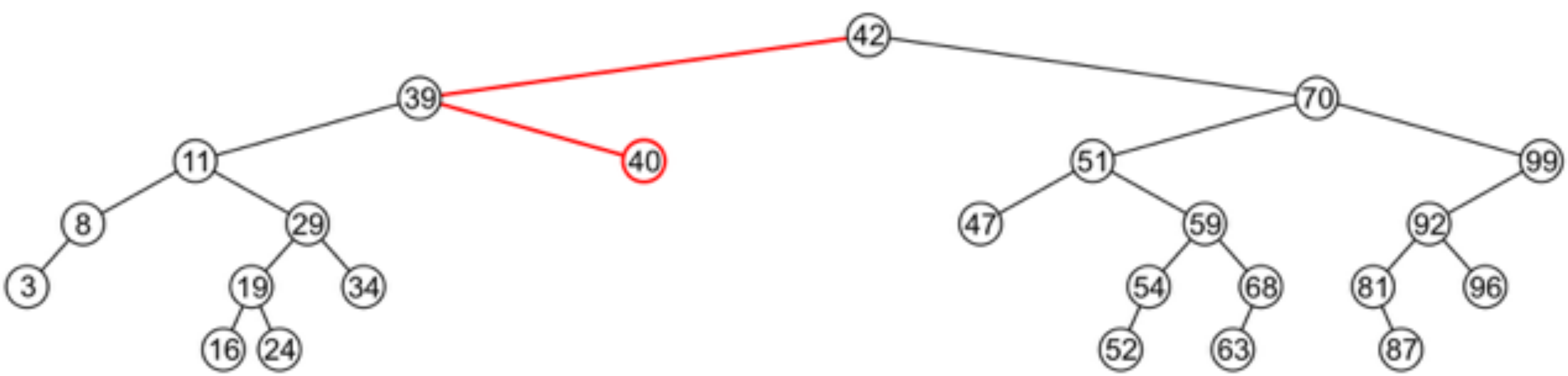
Remove

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



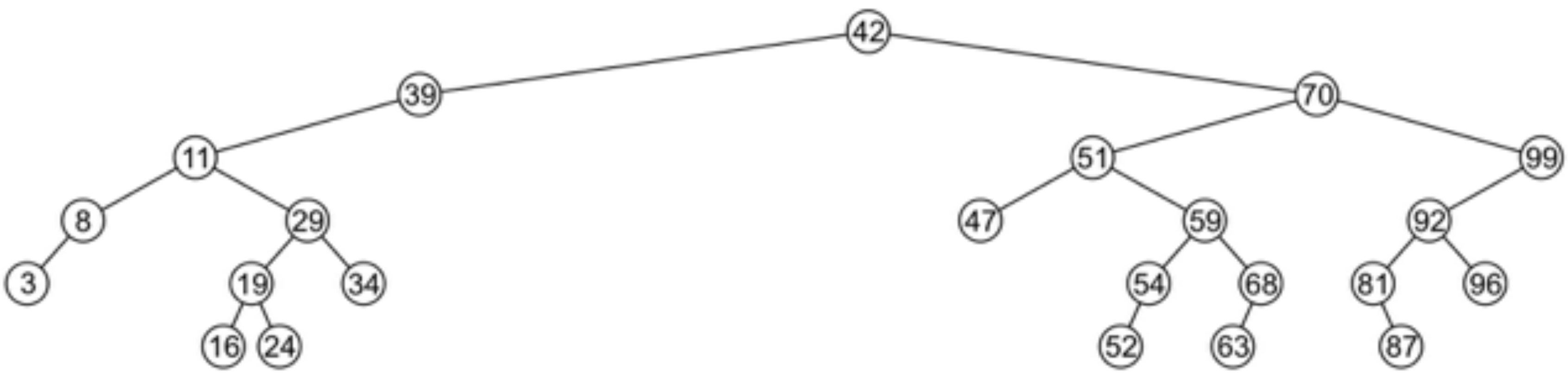
Remove

Removing the node containing 40 is similar



Remove

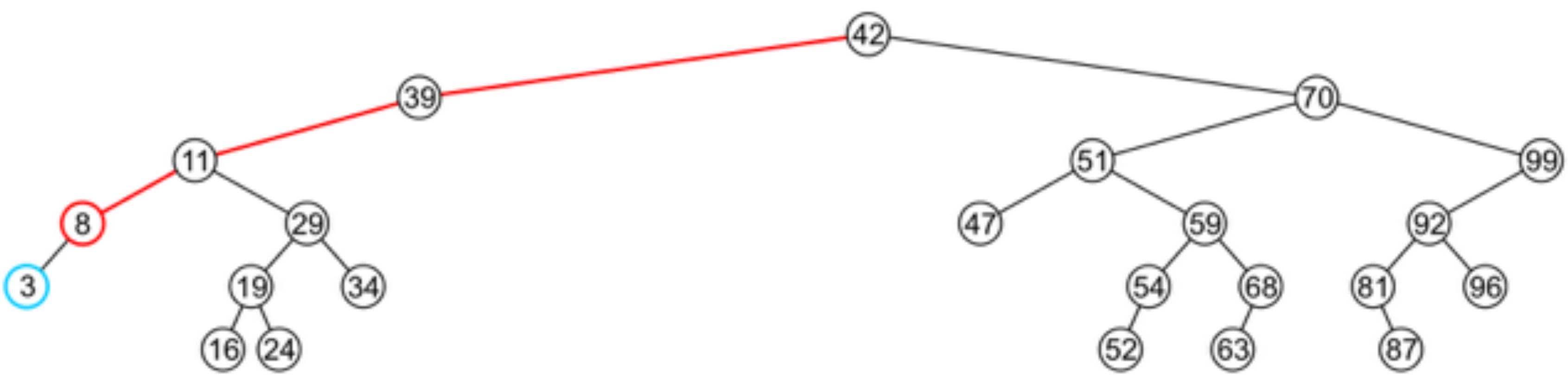
The node is deleted and `right` child of 39 is set to `null`



Remove

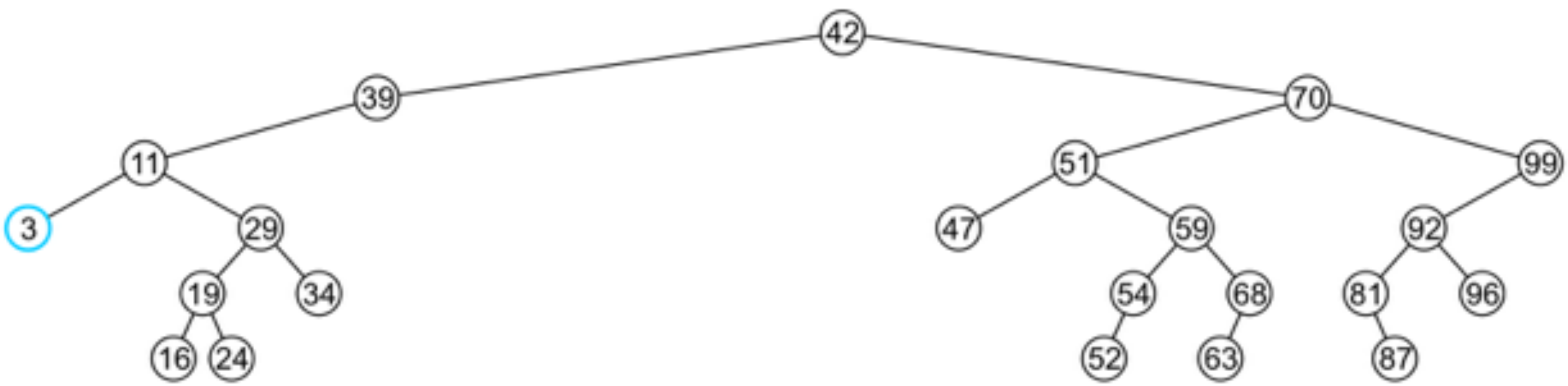
If a node has only one child, we can simply promote the sub-tree associated with the child

- Consider removing 8 which has one left child



Remove

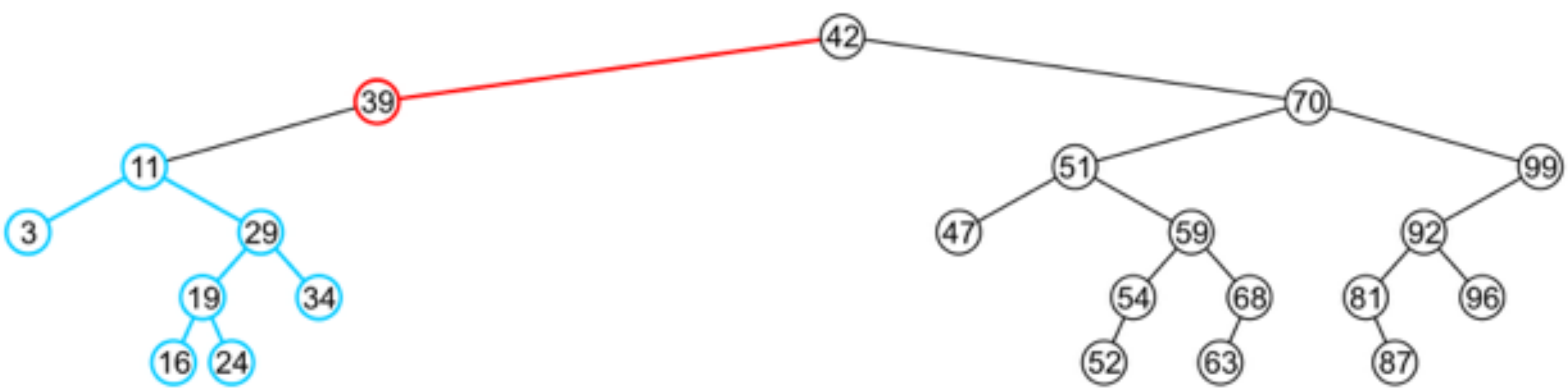
The node 8 is deleted and the `left` child of 11 is updated to point to 3



Remove

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

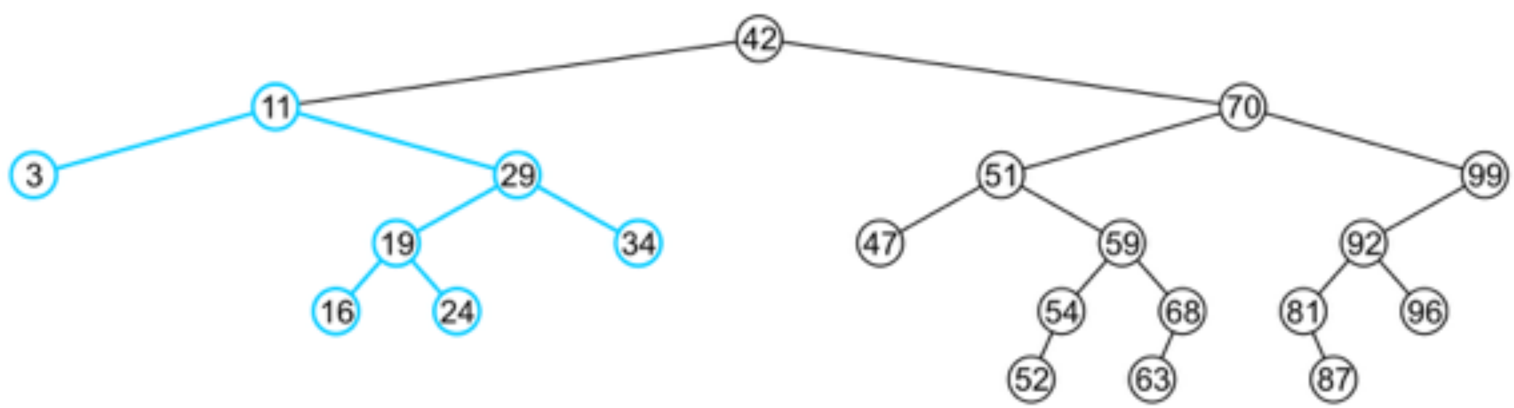
– To remove 39, it has a single child 11



Remove

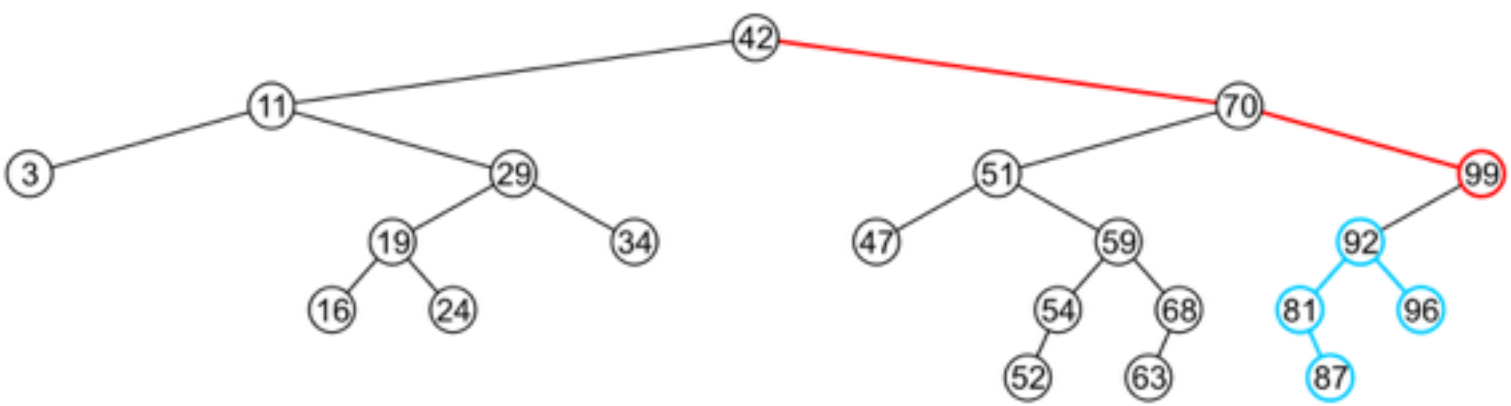
The node containing 39 is deleted and `left` child of 42 is updated to point to the node 11

– Notice that order is still maintained



Remove

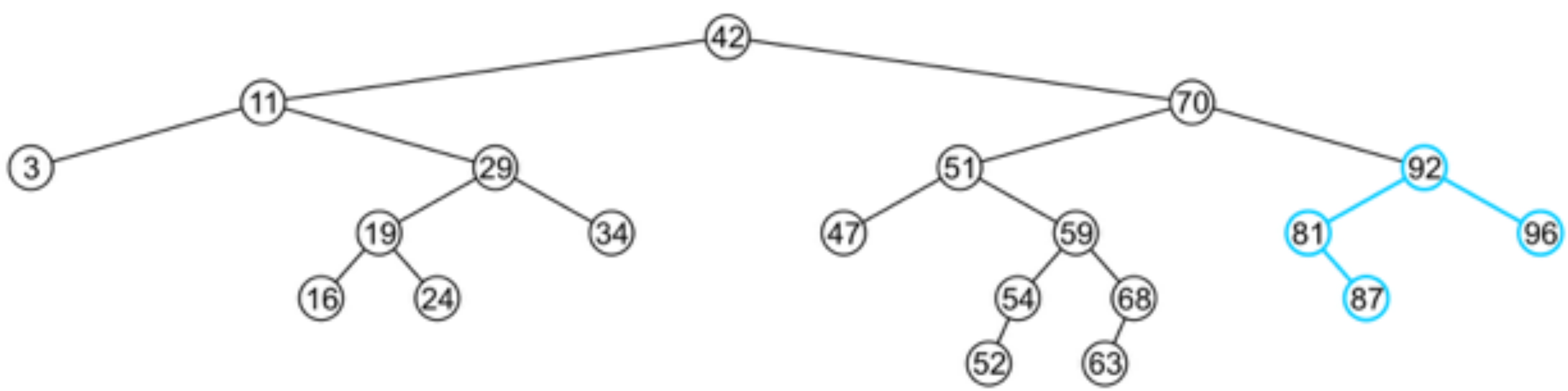
Consider erasing the node containing 99



Remove

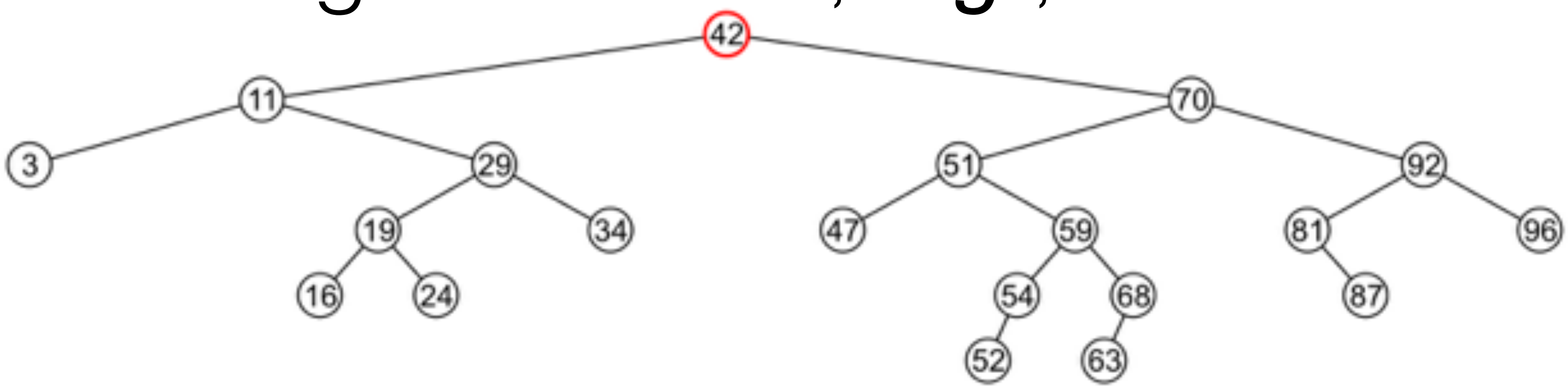
The node is deleted and the left sub-tree is promoted:

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



Remove

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



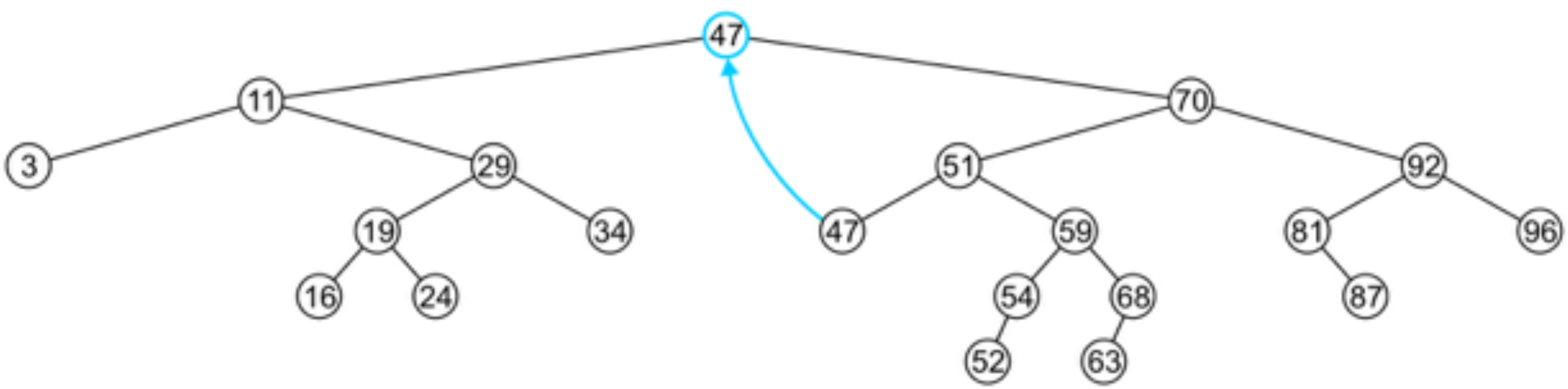
We will perform two operations:

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

Remove

In this case, we replace 42 with 47

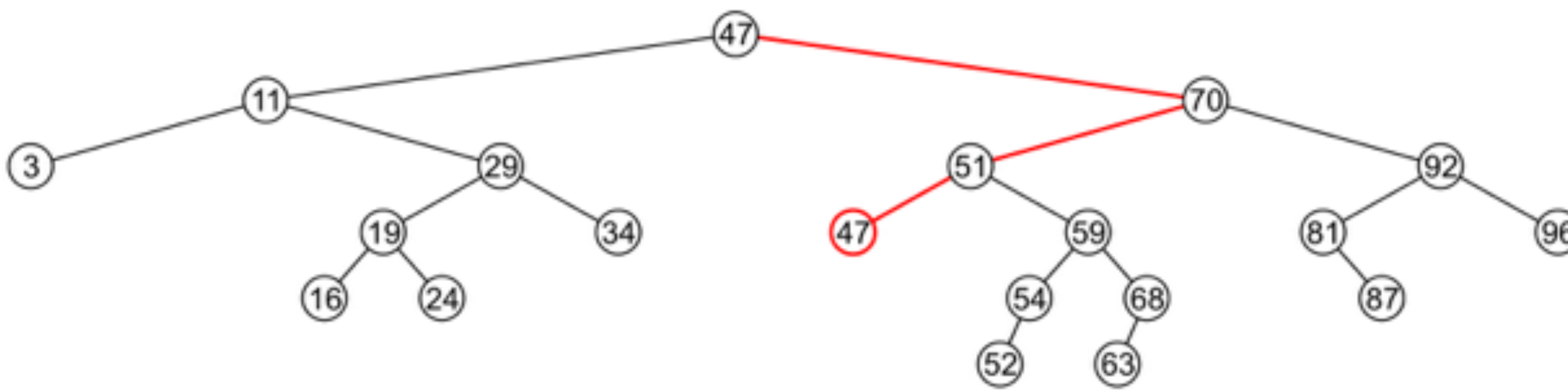
- We temporarily have two copies of 47 in the tree



Remove

We now recursively erase 47 from the right sub-tree

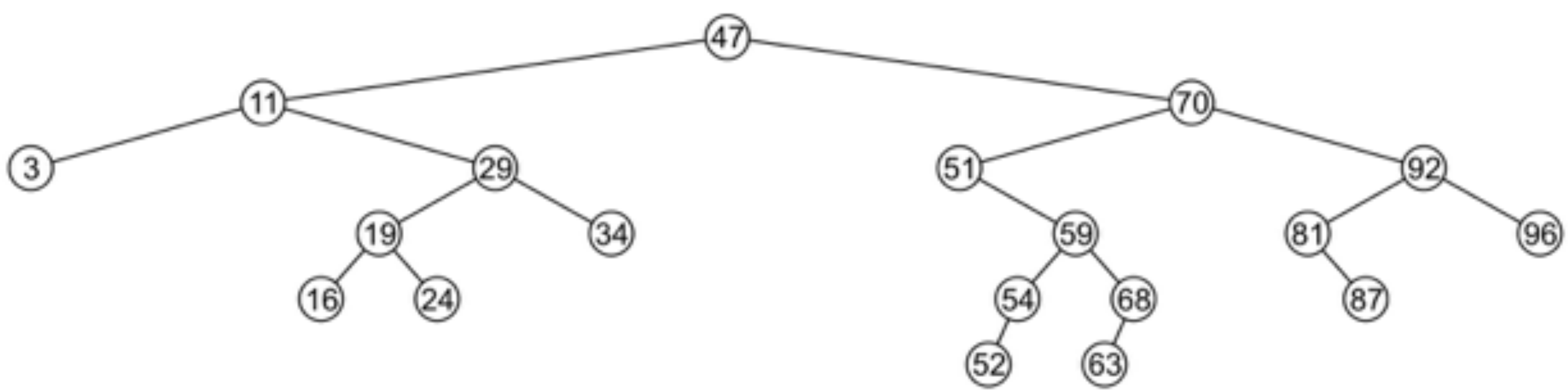
– We note that 47 is a leaf node in the right sub-tree



Remove

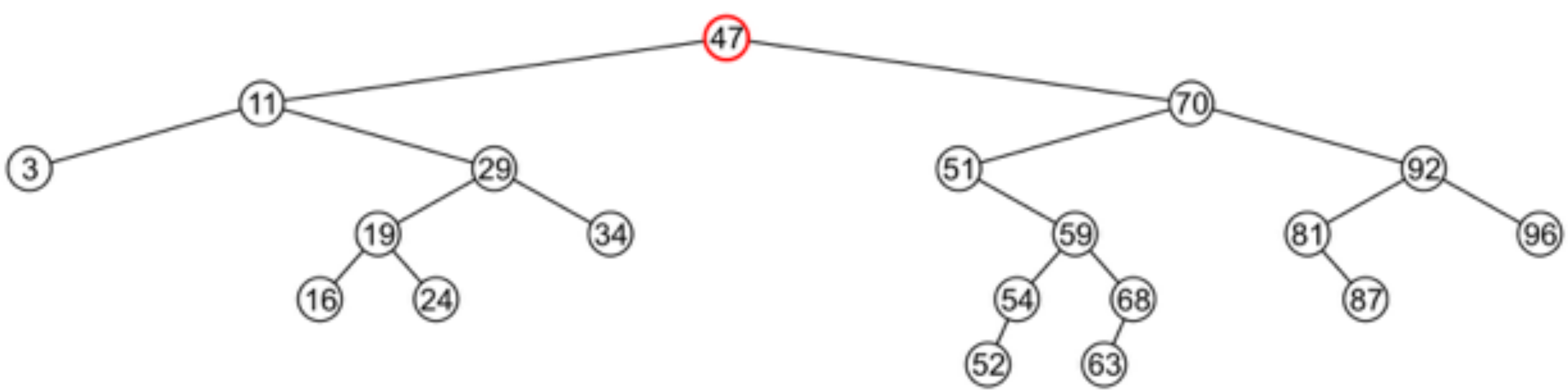
- Leaf nodes are simply removed (`left` child of 51 is set to `null`)
  - Notice that the tree is still sorted:

    47 was the smallest object in the right sub-tree



Remove

- Suppose we want to erase the root 47 again:
  - We must copy the minimum of the right sub-tree
  - Alternatively we could promote the maximum object in the left sub-tree and achieve correct results

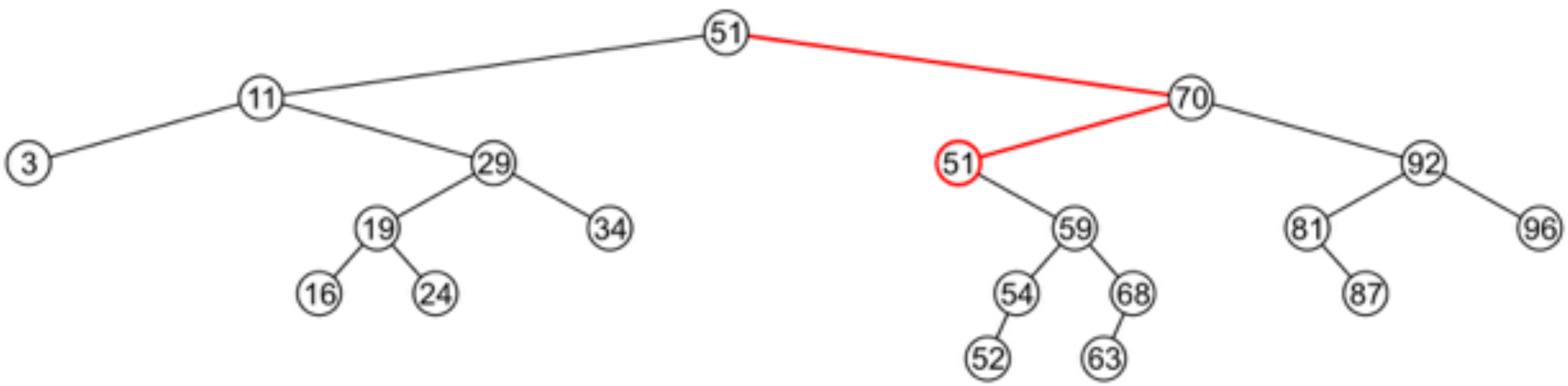


Remove

We copy 51 from the right sub-tree

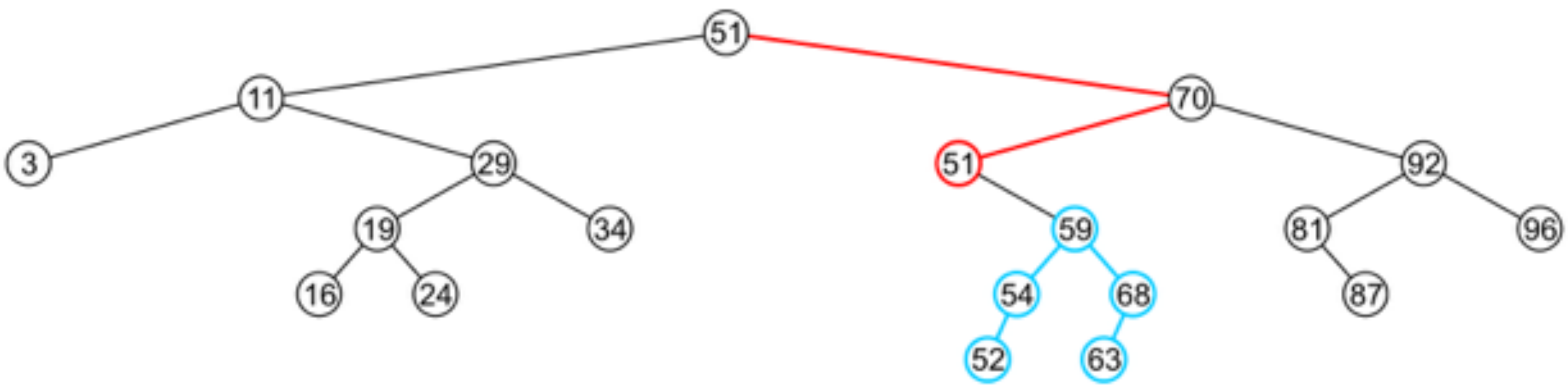
Remove

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



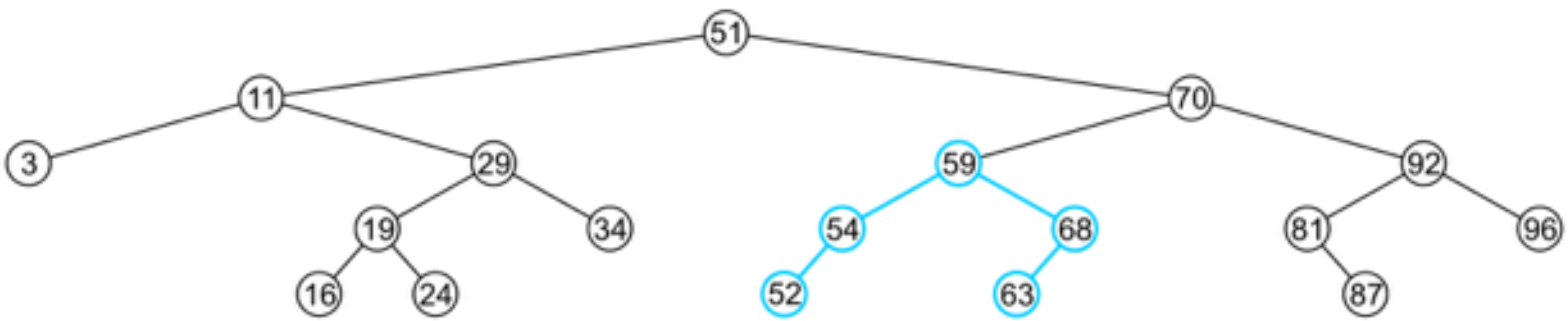
Remove

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



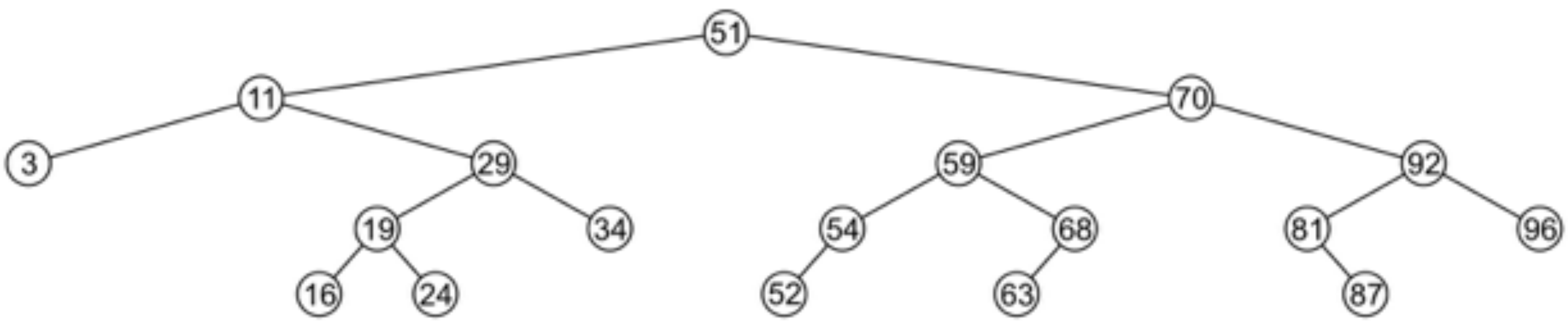
Remove

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



Remove

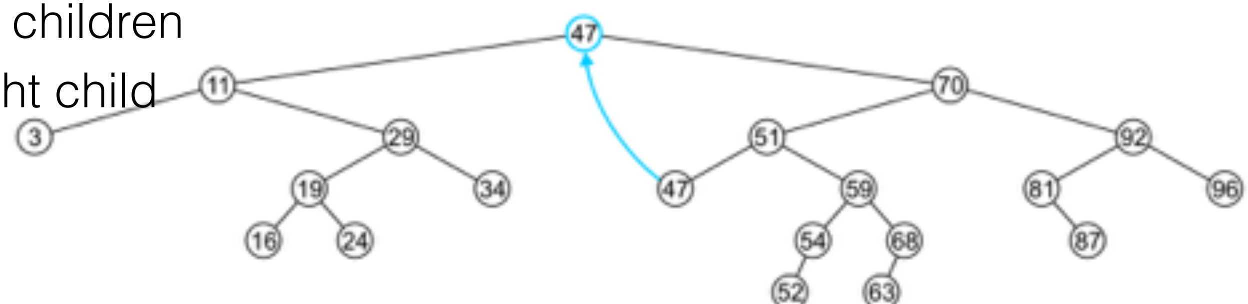
Note that after several removals, the remaining tree is still a BST (correctly sorted)



Remove

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

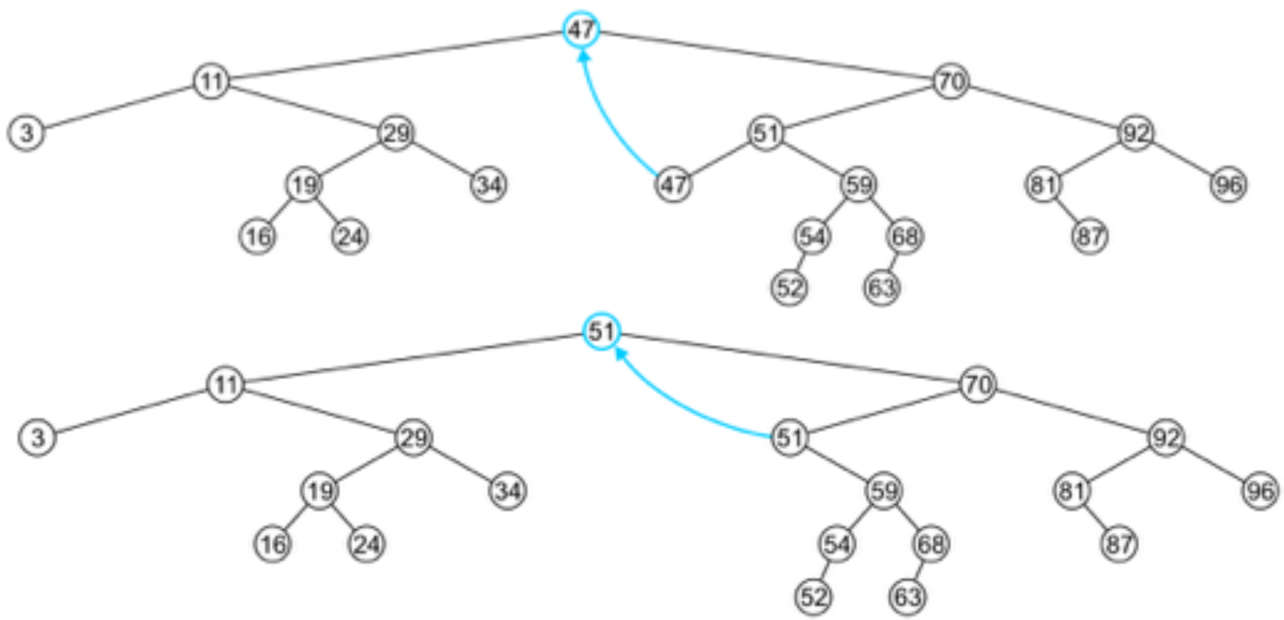
- A node with no children
- A node with right child



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

Remove

- Recall that we promoted the minimum element in the right sub-tree
  - If that node had a left sub-tree, that sub-tree would contain a smaller value! Always the node which we choose to promote has at most one child.



Remove

```
public void remove( AnyType x ){
    return root = remove( x, root );
}

protected BinaryNode<AnyType> remove(Anytype x, BinaryNode<AnyType> t){
    if( t == null )
        throw new ItemNotFoundException( x.toString( ) );
    else if( x.compareTo( t.element ) < 0 )
        t.left = remove( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = remove( x, t.right );
    else if ( t.left != null && t.right != null ){ // x is the root of current tree!
        // and the root (t) has two children
        t.element = findMin(t.right).element;
        t.right = removeMin( t.right );
    }
    else // x is the root of current tree!
        // and the root (t) has at most one child
        t = (t.left != null) ? t.left : t.right;

    return t;
}
```

removeMin()

```
public void removeMin( ){
    return root = removeMin( root );
}

protected BinaryNode<AnyType> removeMin(BinaryNode<AnyType> t){
    if( t == null )
        throw new ItemNotFoundException( );
    else if ( t.left != null ){
        t.left = removeMin( t.left );
        return t;
    }
    else
        return t.right;
}
```

removeMax()

```
public void removeMax( ){
    return root = remove( root );
}

protected BinaryNode<AnyType> removeMax(BinaryNode<AnyType> t){
    if( t == null )
        throw new ItemNotFoundException( );
    else if ( t.right != null ){
        t.right = removeMax( t.right );
        return t;
    }
    else
        return t.left;
}
```

Remove

Example:

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

Other Methods

- We can add more basic methods
  - `clear()`
  - `isEmpty()`
  - `size()`
  - `height()`
  - `count()`
- The implementation would similar to corresponding methods in BinaryTree class.

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: $\mathrm{O}(\log(n))$
- Our goal will be to find tree structures where we can maintain a height of $\Theta(\log(n))$

We will need to fix this issue and to make sure all the operations run in $\mathrm{O}(\log(n))$