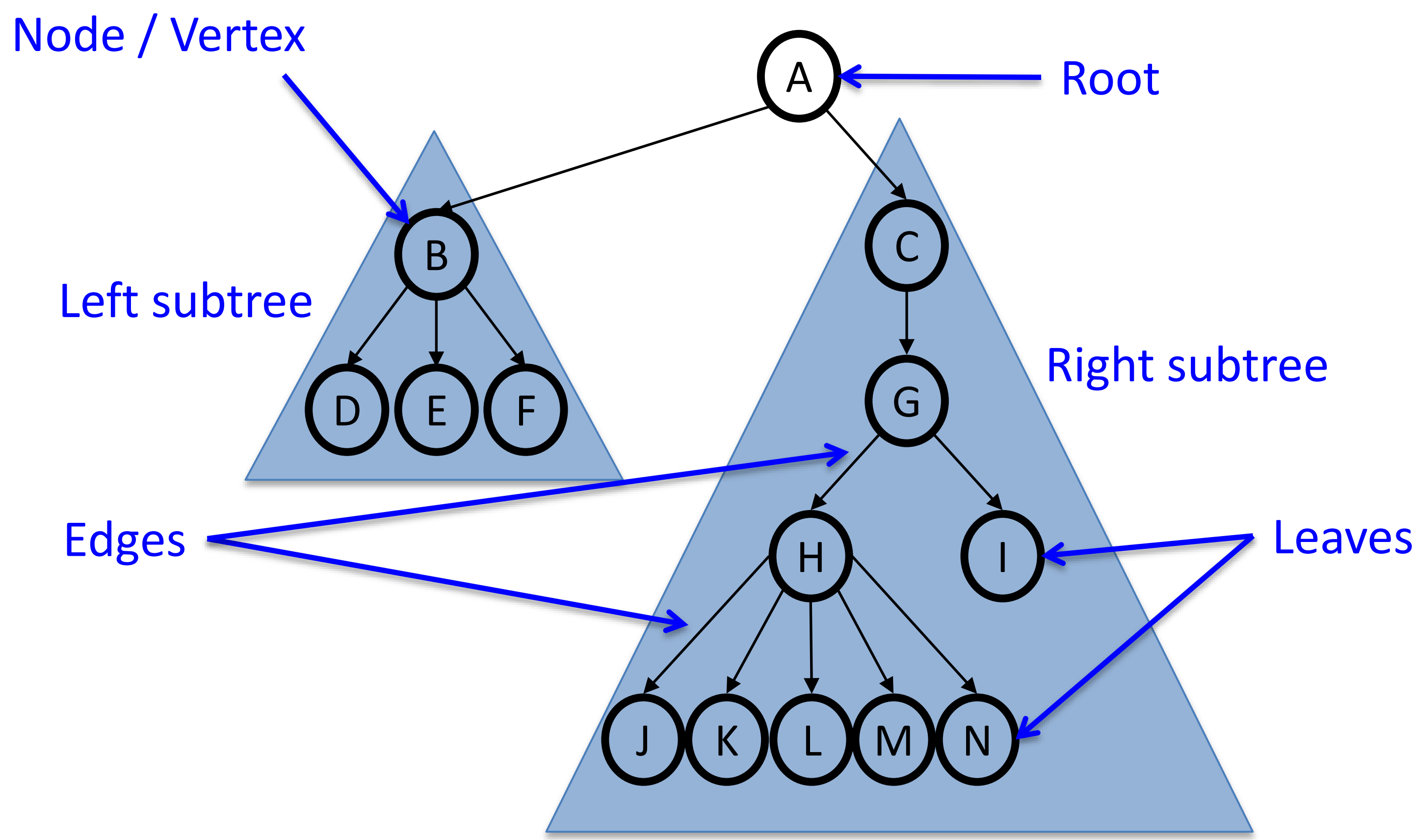
Binary Search Trees

COL 106

Amit Kumar and Shweta Agrawal

Most slides courtesy : Douglas Wilhelm Harder, MMath, Uwaterloo; Linda Shapiro, UW

Reminder: Binary Tree terminology

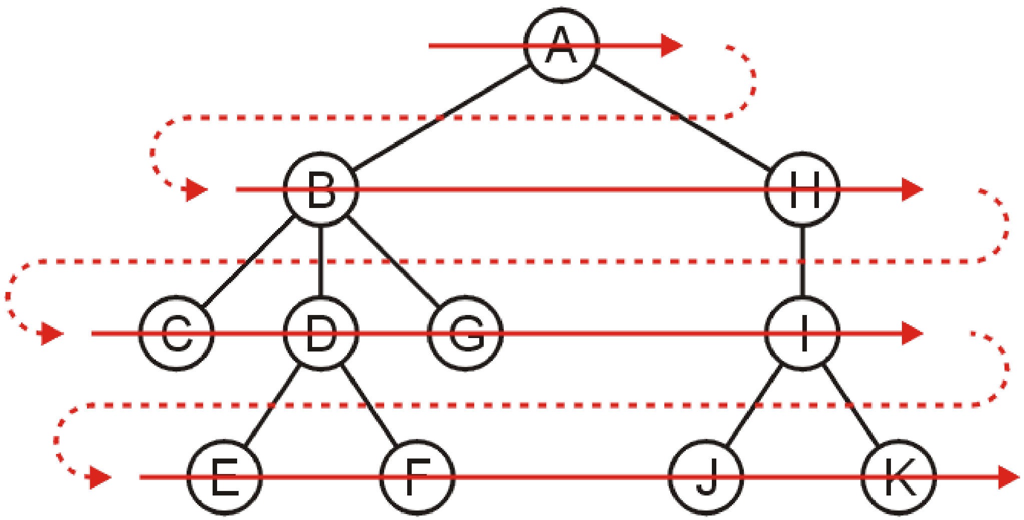


Last time..

- We saw Preorder, Inorder, Postorder traversals
- One more useful traversal…

Breadth first traversal

- The breadth-first traversal visits all nodes at depth k before proceeding onto depth $k + 1$
- Easy to implement using a queue



Order: A B H C D G I E F J K

Breadth-First Traversal

Breadth-first traversals visit all nodes at a given depth

- Memory: max nodes at given depth

- Create a queue and push the root node onto queue
- While the queue is not empty:
  - Push all of its children of the front node onto the queue
  - Pop the front node

Binary Search Trees

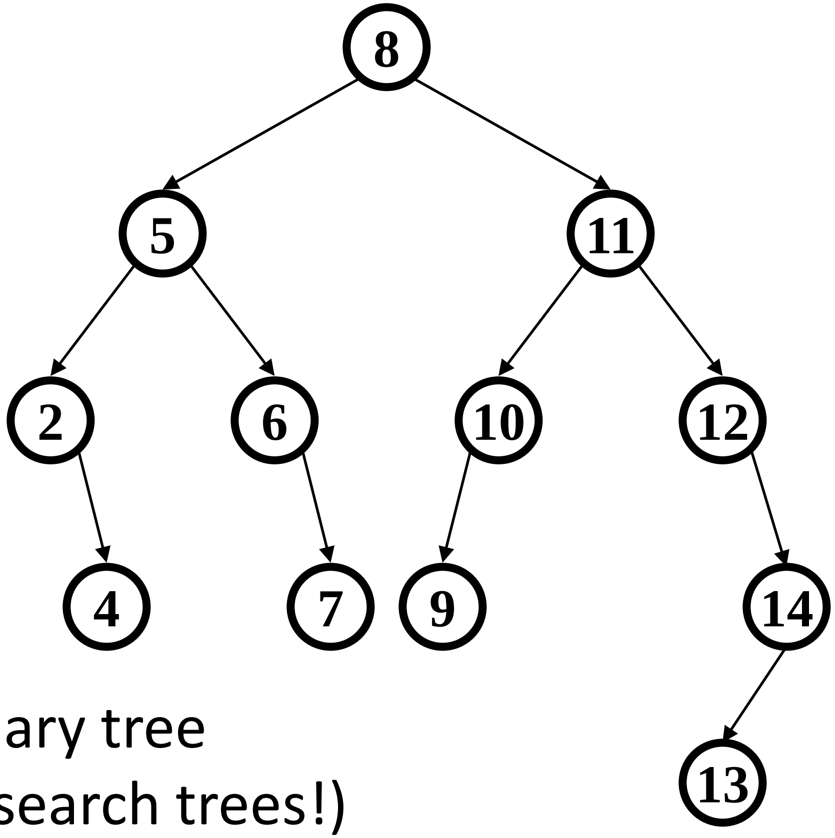
Recall that with a binary tree, we can dictate an order on the two children

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 Tree (BST) Data Structure

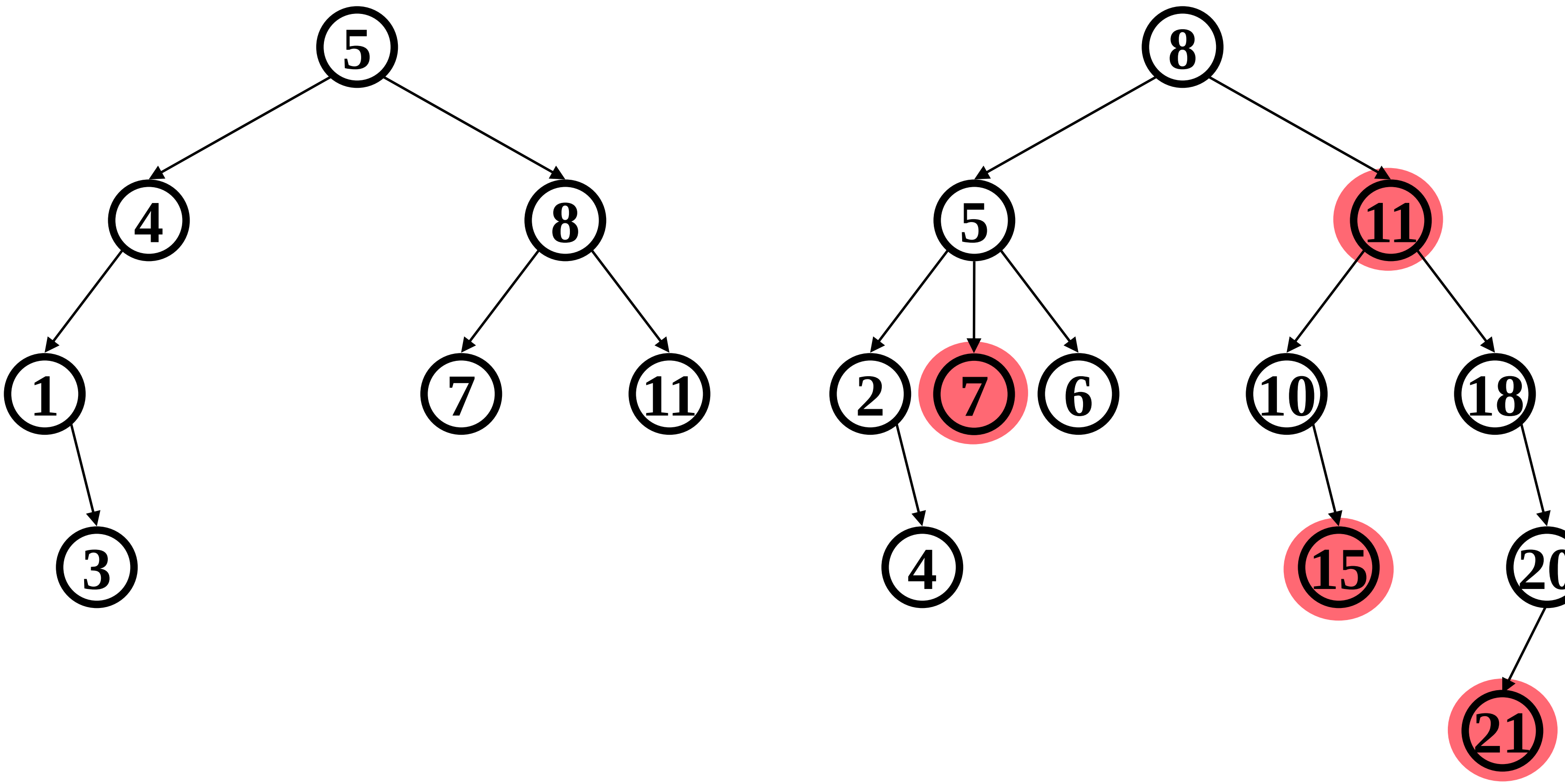
- Structure property (binary tree)
  - Each node has ≤ 2 children
  - Result: keeps operations simple
- Order property
  - All keys in left subtree smaller than node's key
  - All keys in right subtree larger than node's key
  - Result: easy to find any given key



A binary search tree is a type of binary tree
(but not all binary trees are binary search trees!)

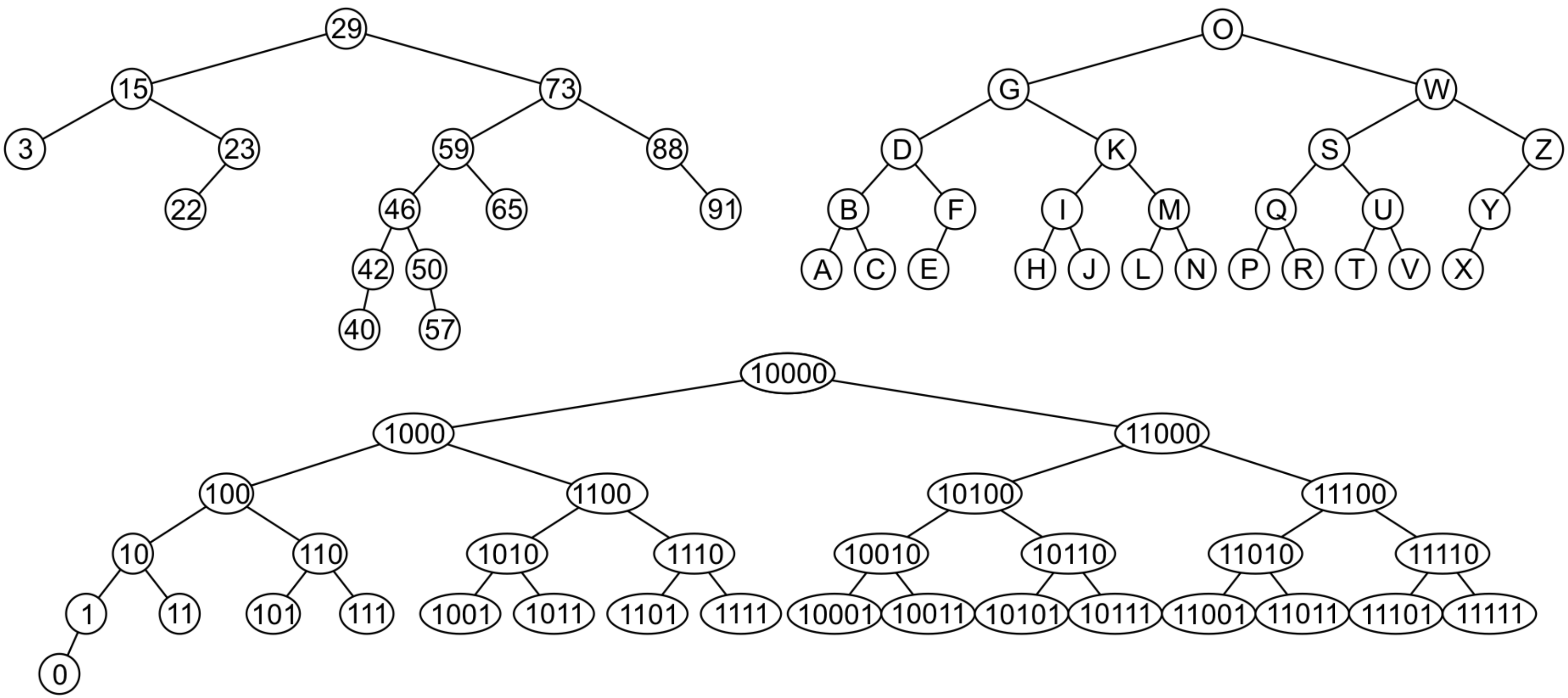
Are these BSTs?

Activity! What nodes violate the BST properties?



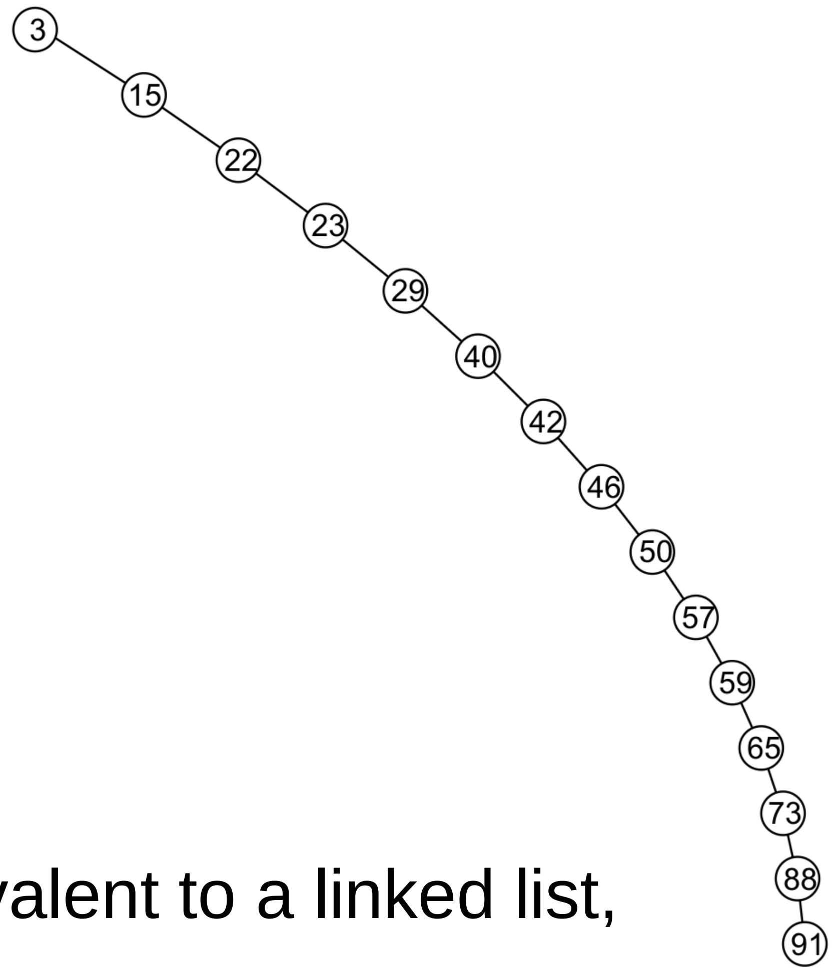
Examples

Here are other examples of binary search trees:



Examples

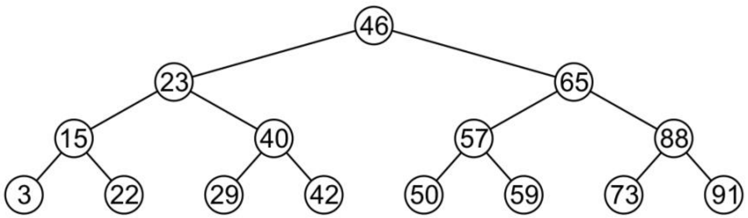
Unfortunately, it is possible to construct *degenerate* binary search trees

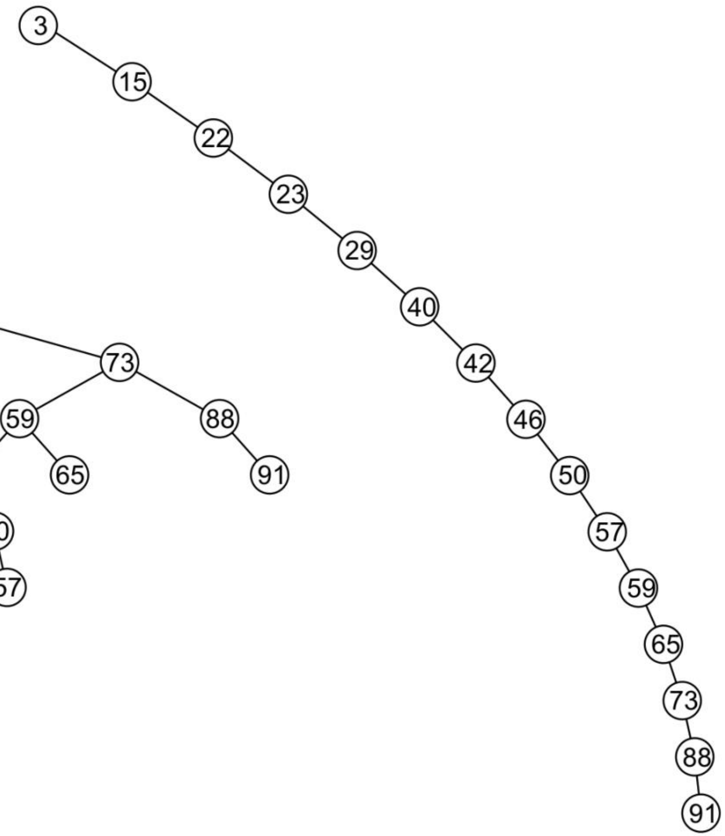


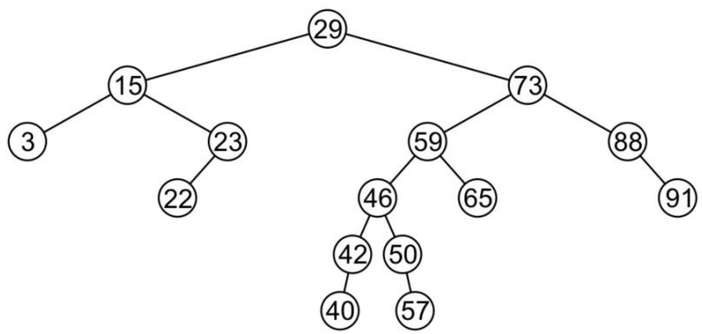
– This is equivalent to a linked list, *i.e.*, $\mathbf{O}(n)$

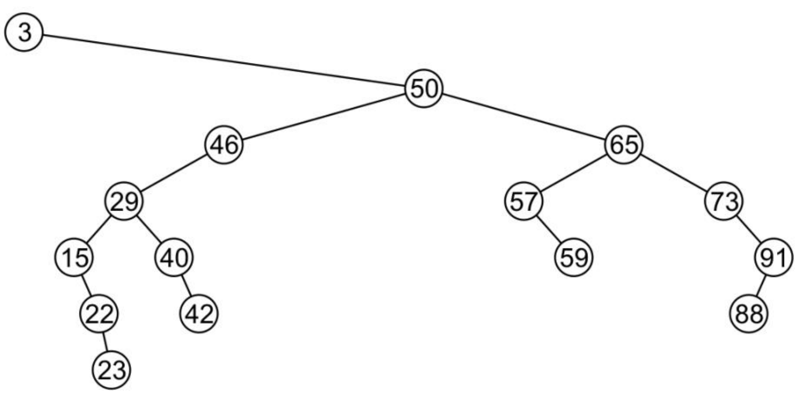
Examples

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

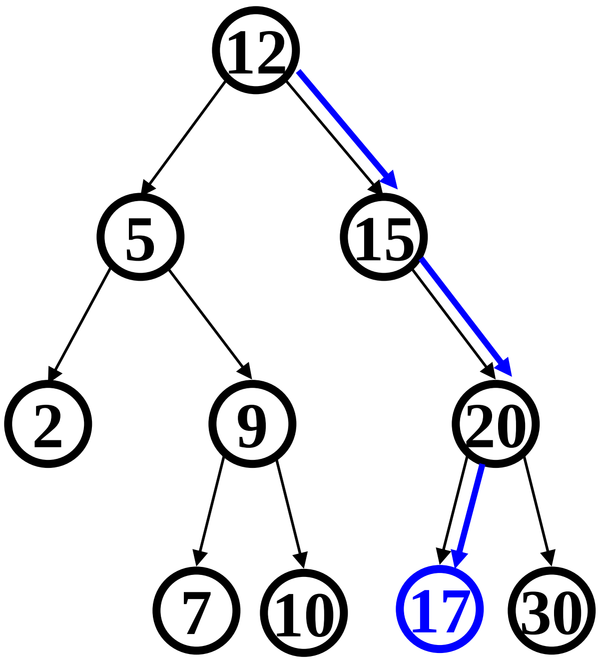
Any class which uses this binary-search-tree class must therefore implement:

```
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`

Find in BST, (Tail) Recursive

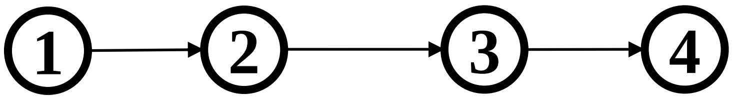



```
Data find(Key key, Node root){
  if(root == null)
    return null;
  if(root.key == key)
    return root.data;
  if(key < root.key)
    return find(key,root.left);
  if(key > root.key)
    return find(key,root.right);
}
```

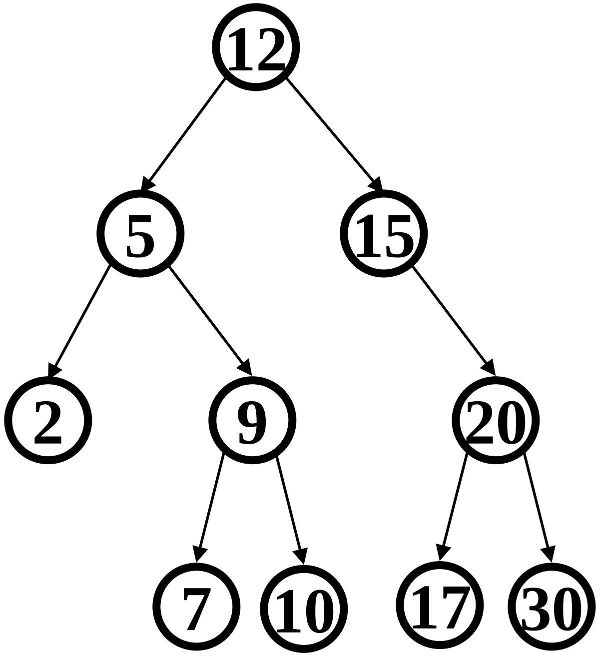
What is the time complexity? O(h)

Worst case running time is O(n).

- Happens if the tree is very lopsided (e.g. list)



Find in BST, Iterative



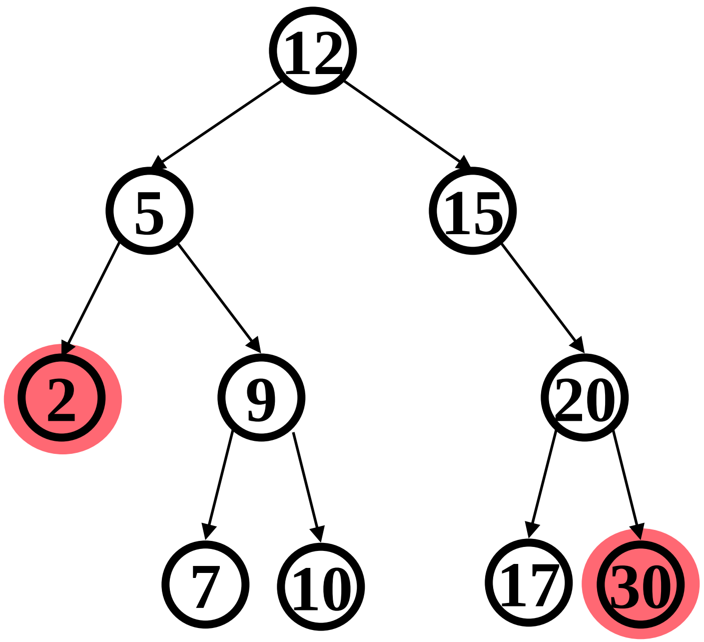

```
Data find(Key key, Node root){
 while(root != null
       && root.key != key) {
  if(key < root.key)
    root = root.left;
  else(key > root.key)
    root = root.right;
 }
 if(root == null)
    return null;
 return root.data;
}
```

Worst case running time is O(n).

- Happens if the tree is very lopsided (e.g. list)

Bonus: Other BST "Finding" Operations

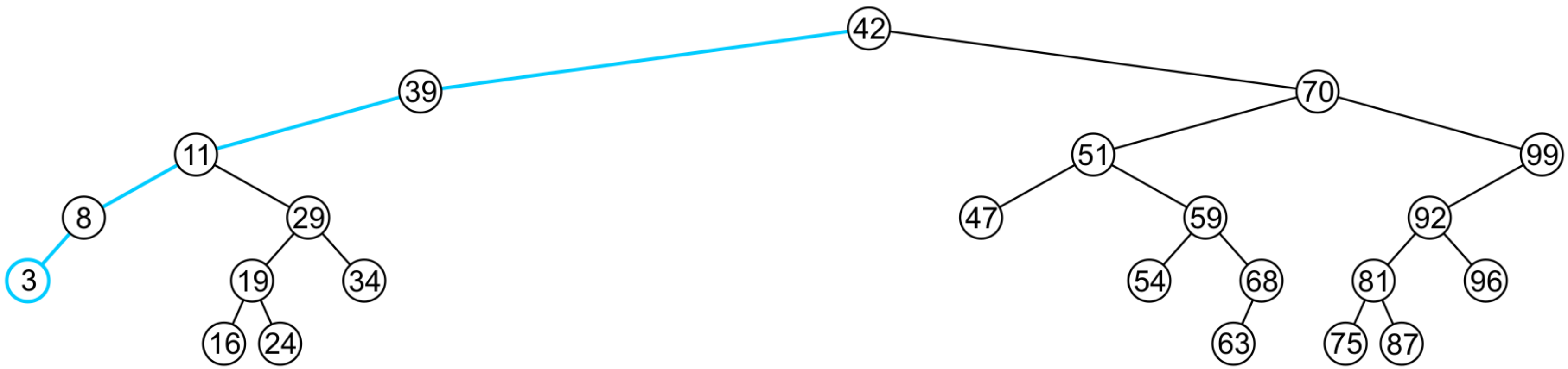
- **`FindMin`**: Find *minimum* node
  - Left-most node
- **`FindMax`**: Find *maximum* node
  - Right-most node



How would we implement?

Finding the Minimum Object

The minimum object may be found recursively



– The run time $\mathbf{O}(h)$

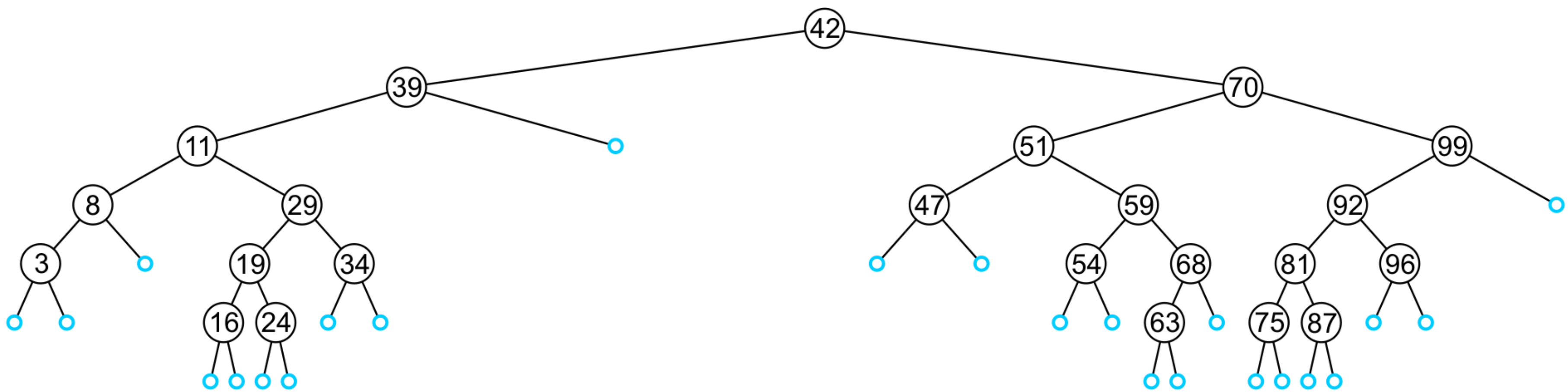
```
int findMin(Node root){
 if(root == null)
   return null;
 if(root.left == null)
   return root.data;
 return findMin(root.left);
}
```

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

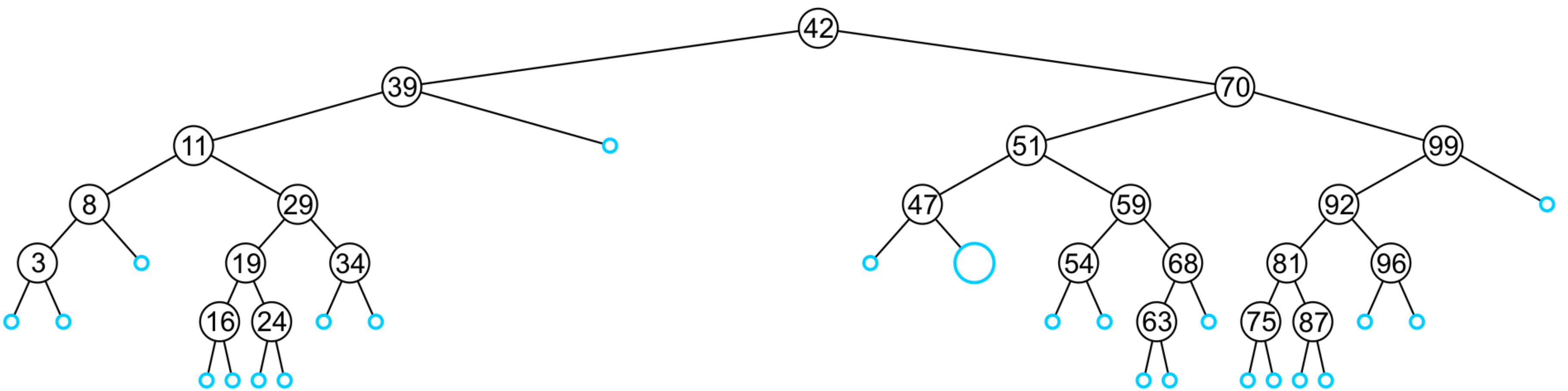
An insertion will be performed at a leaf node:
- Any empty node is a possible location for an insertion



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

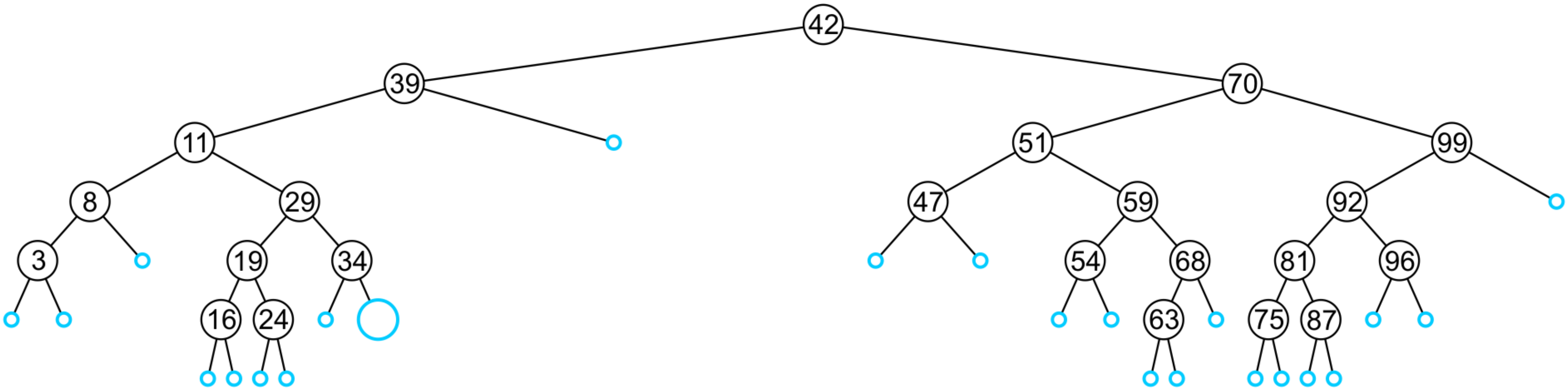
Insert

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



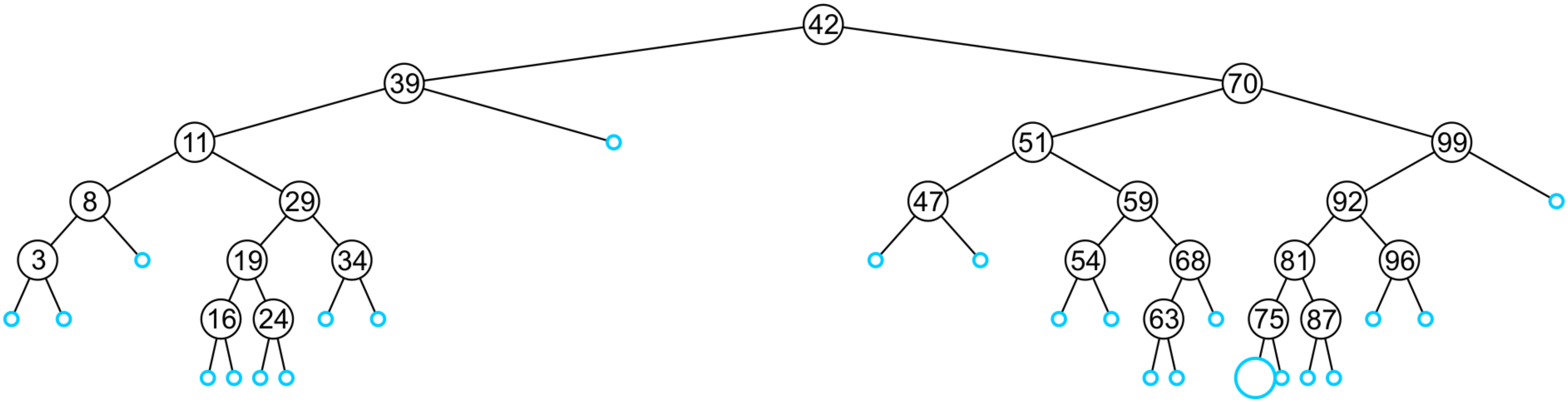
Insert

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



Insert

This empty node may hold values from 71 to 74



Insert

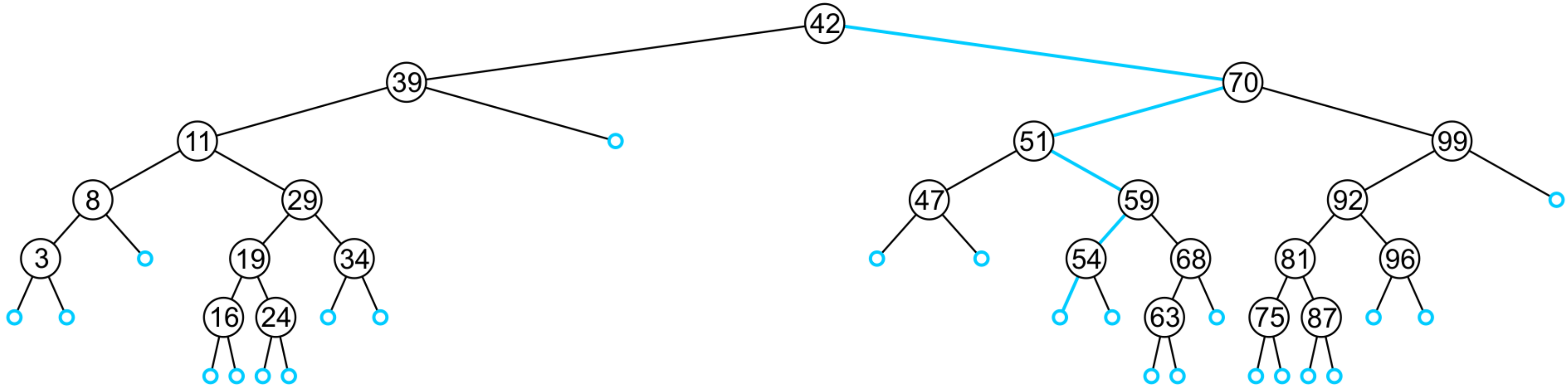
Like find, we will step through the tree

- 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

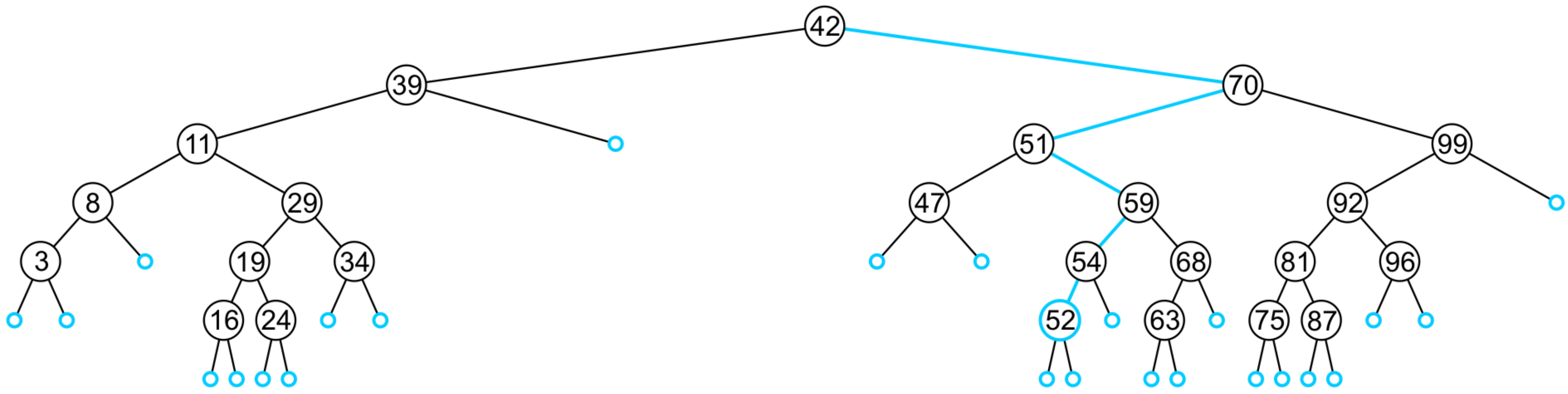
In inserting the value 52, we traverse the tree until we reach an empty node

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



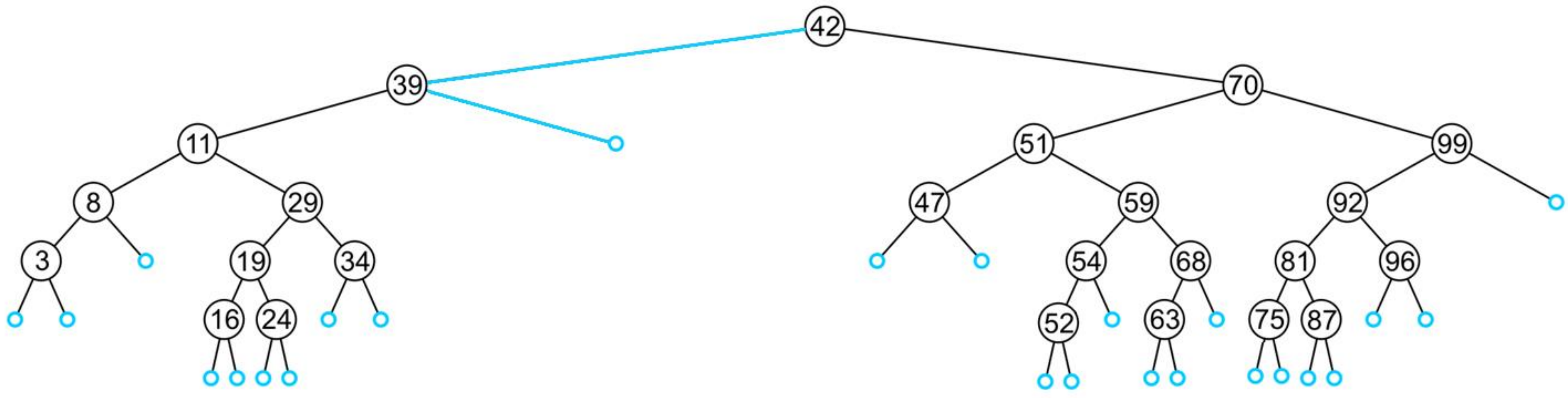
Insert

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



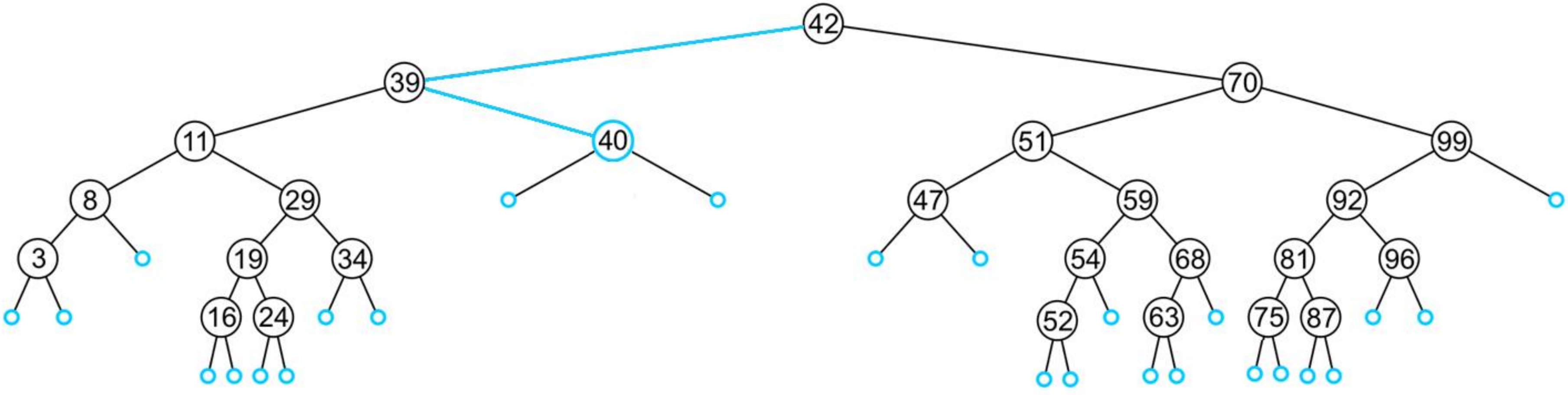
Insert

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



Insert

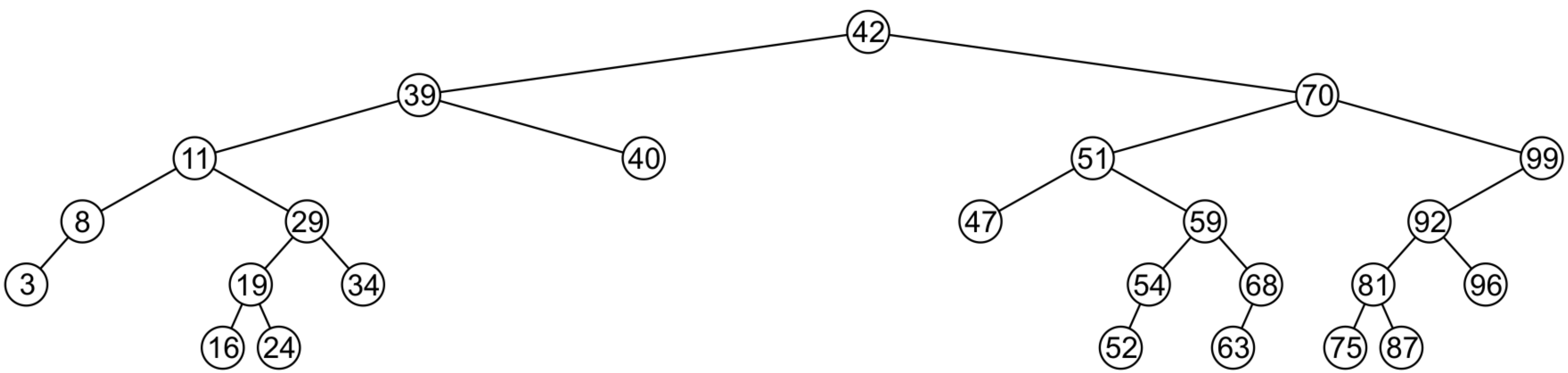
A new leaf node storing 40 is created and assigned to the member variable `right_tree`



Erase

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

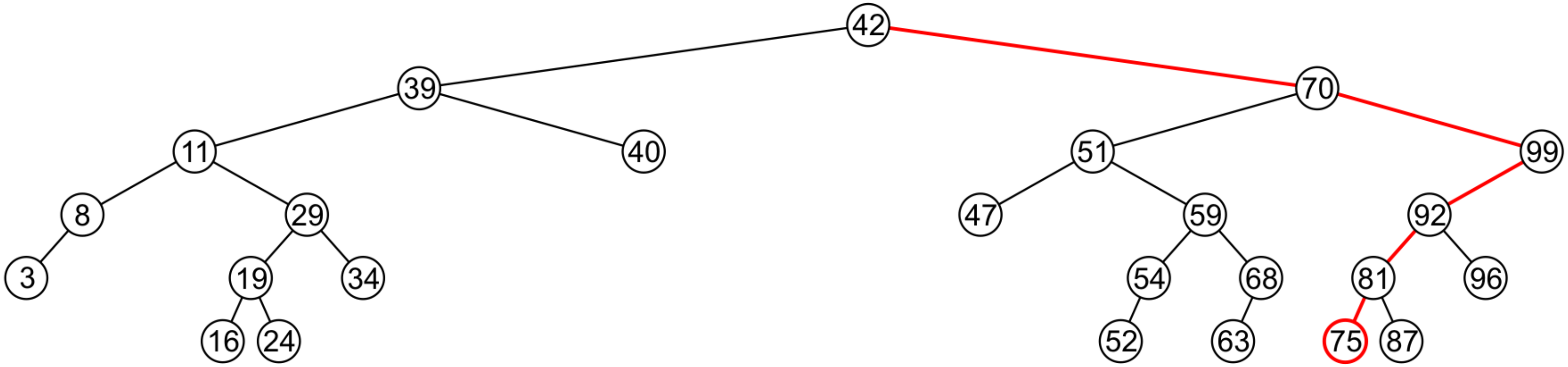
There are three possible scenarios:



Erase

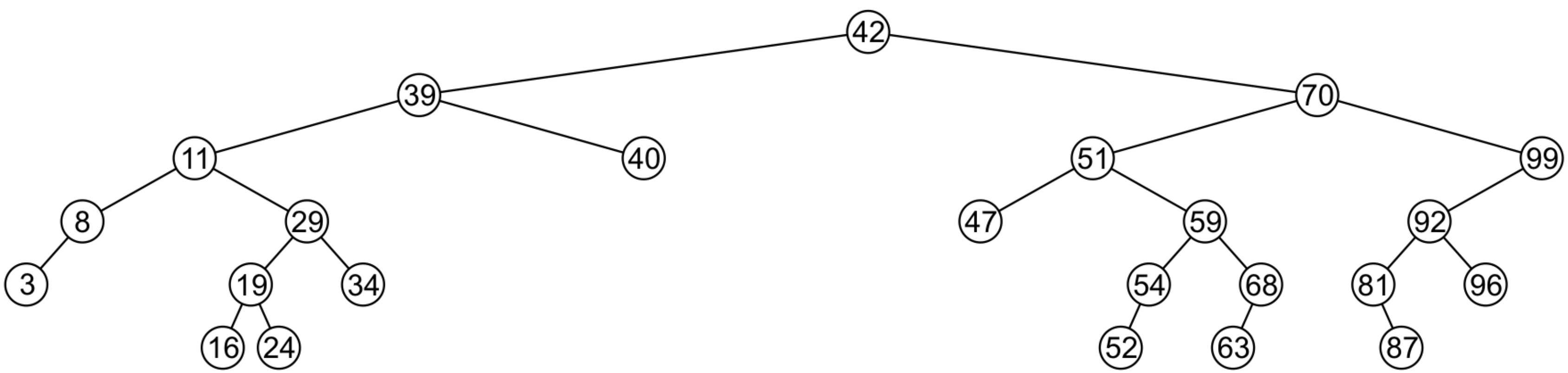
A leaf node simply must be removed and the appropriate member variable of the parent is set to `nullptr`

– Consider removing 75



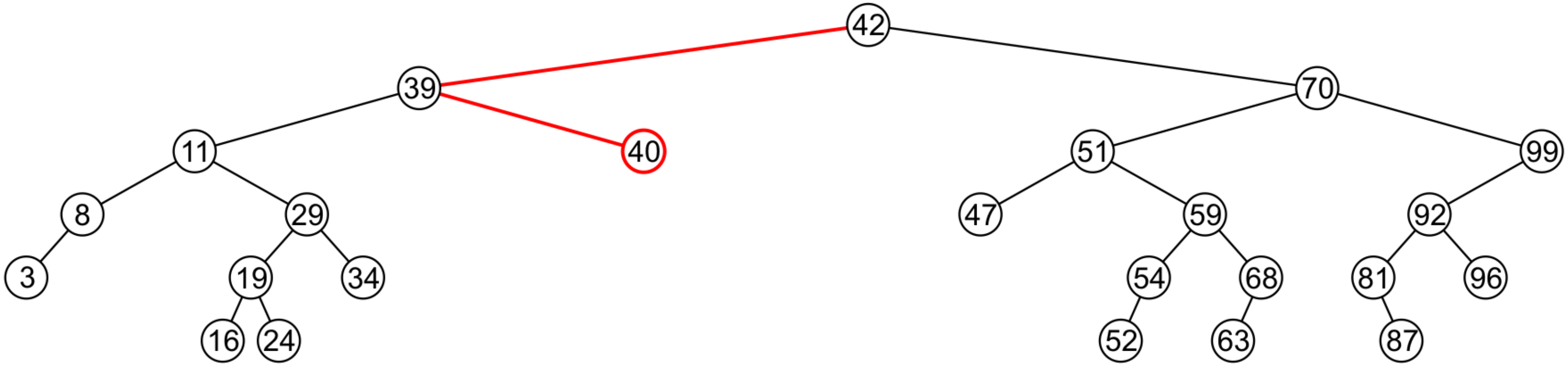
Erase

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



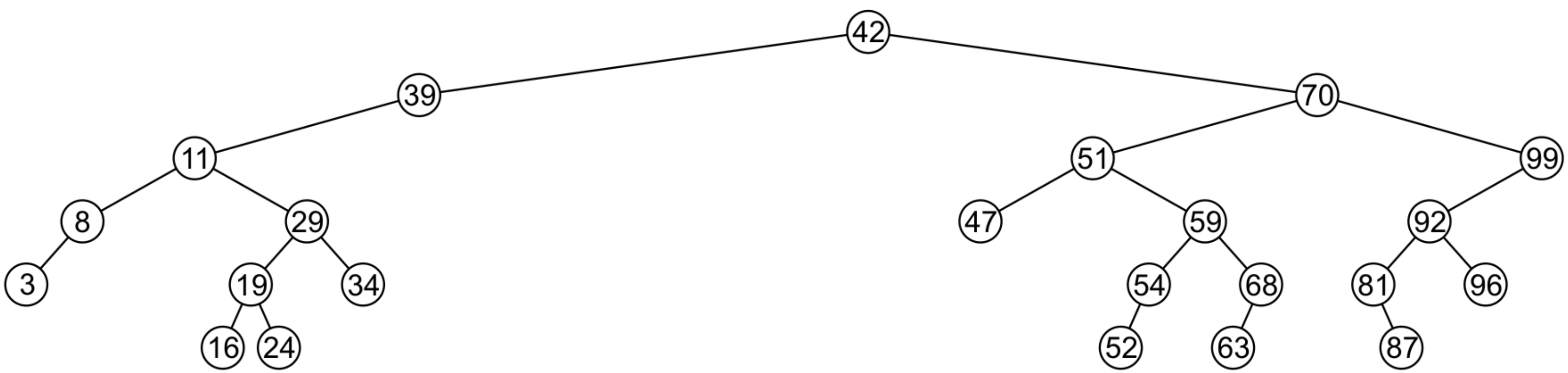
Erase

Erasing the node containing 40 is similar



Erase

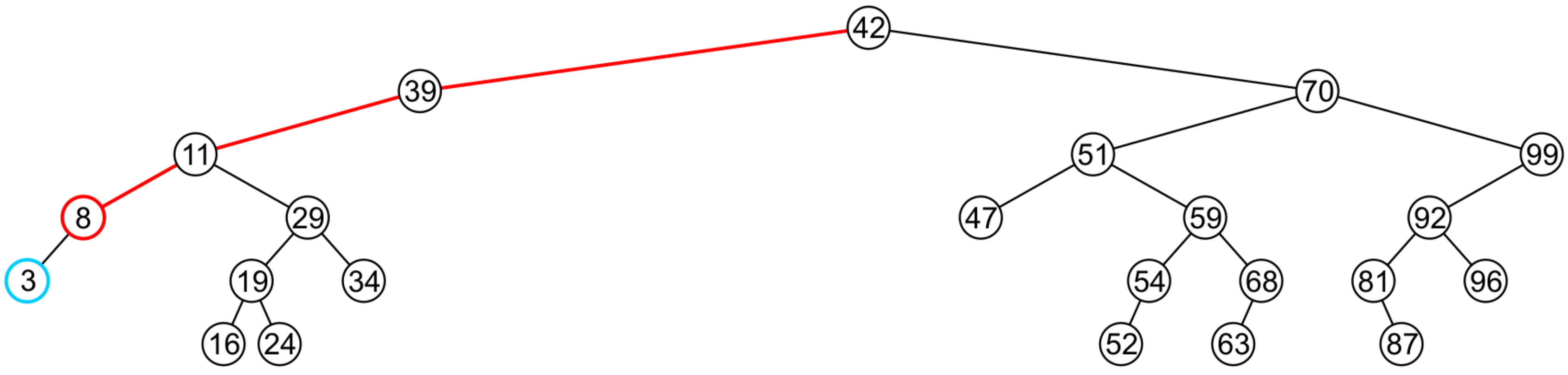
The node is deleted and `right_tree` of 39 is set to `nullptr`



Erase

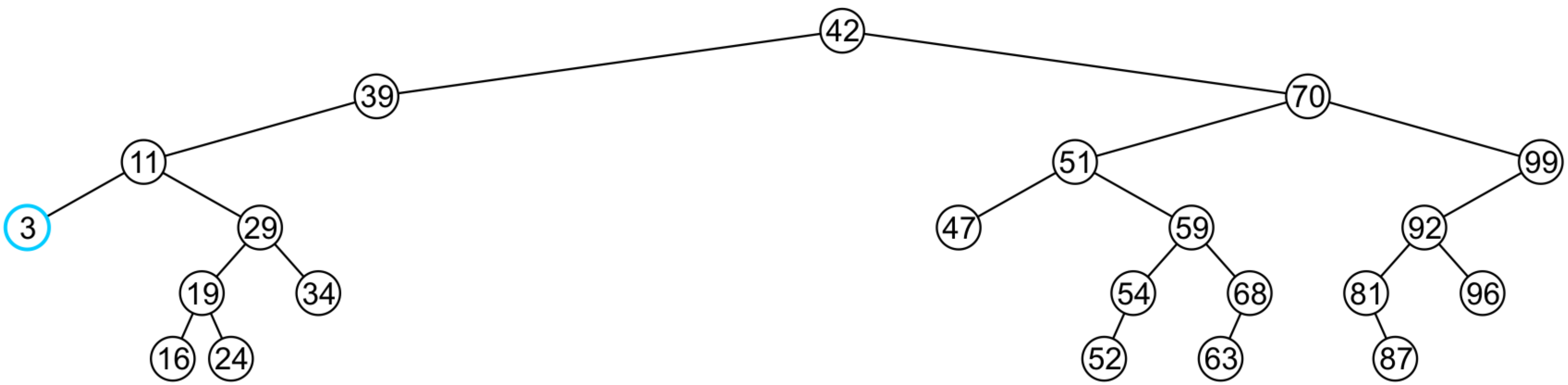
If a node has only one child, we can simply promote the sub-tree associated with the child

– Consider removing 8 which has one left child



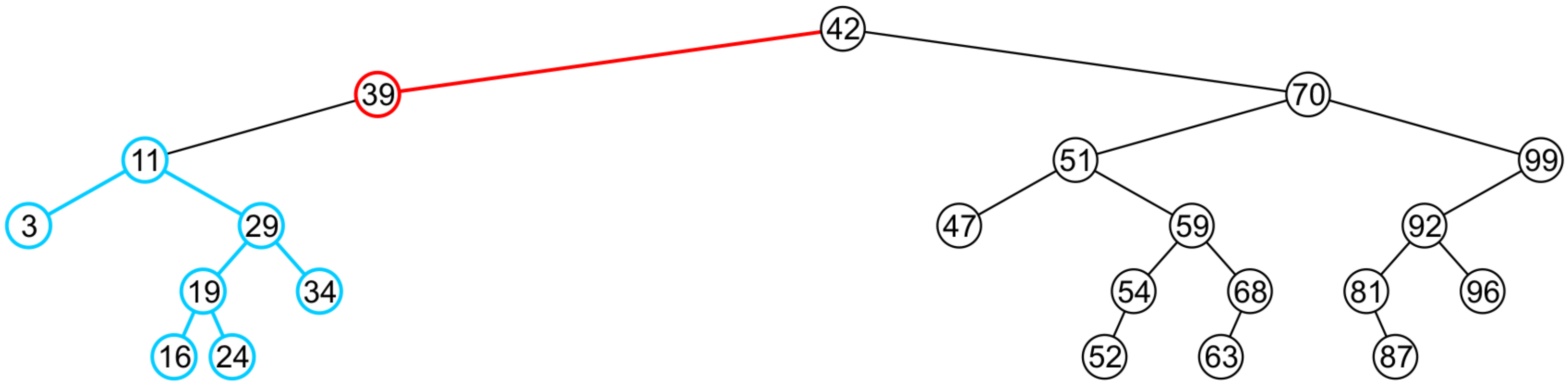
Erase

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



Erase

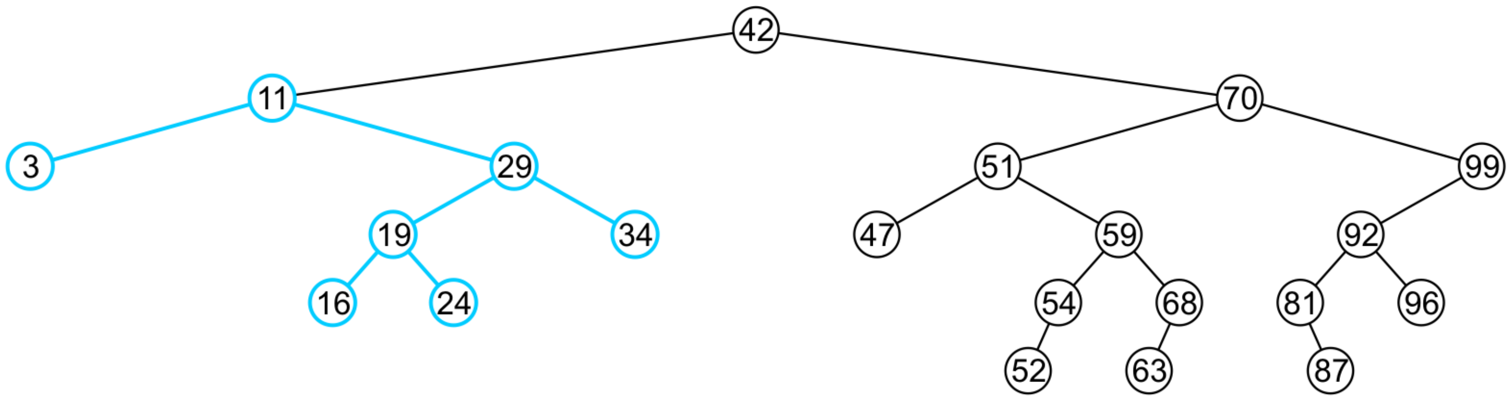
- There is no difference in promoting a single node or a sub-tree
  - To remove 39, it has a single child 11



Erase

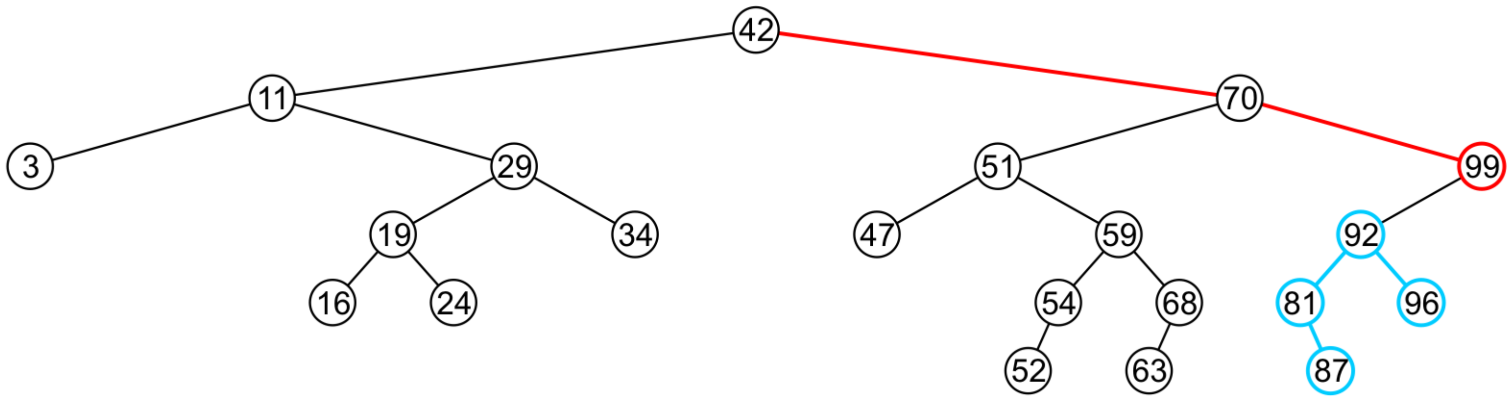
The node containing 39 is deleted and `left_node` of 42 is updated to point to 11

– Notice that order is still maintained



Erase

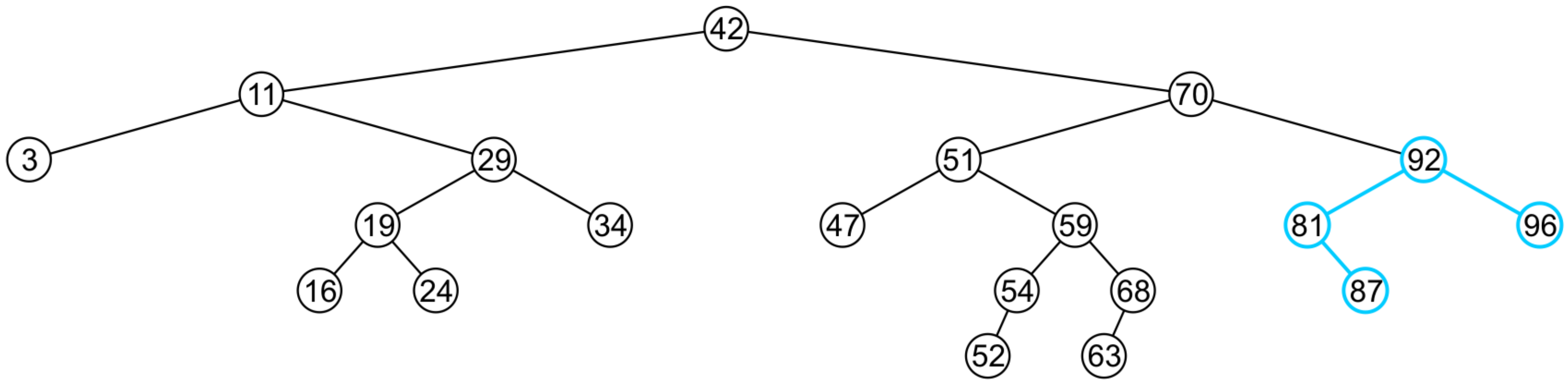
Consider erasing the node containing 99



Erase

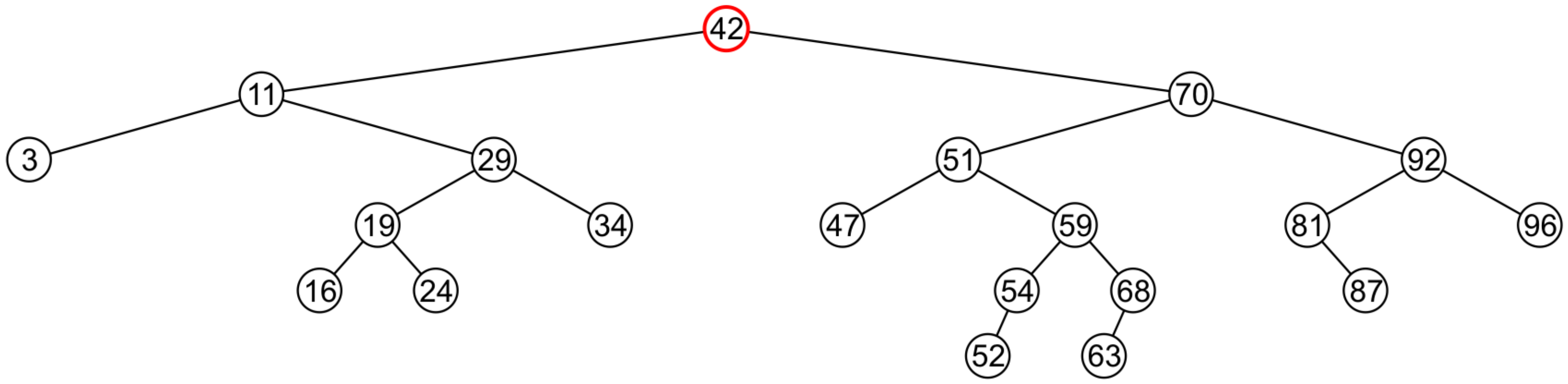
The node is deleted and the left sub-tree is promoted:

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



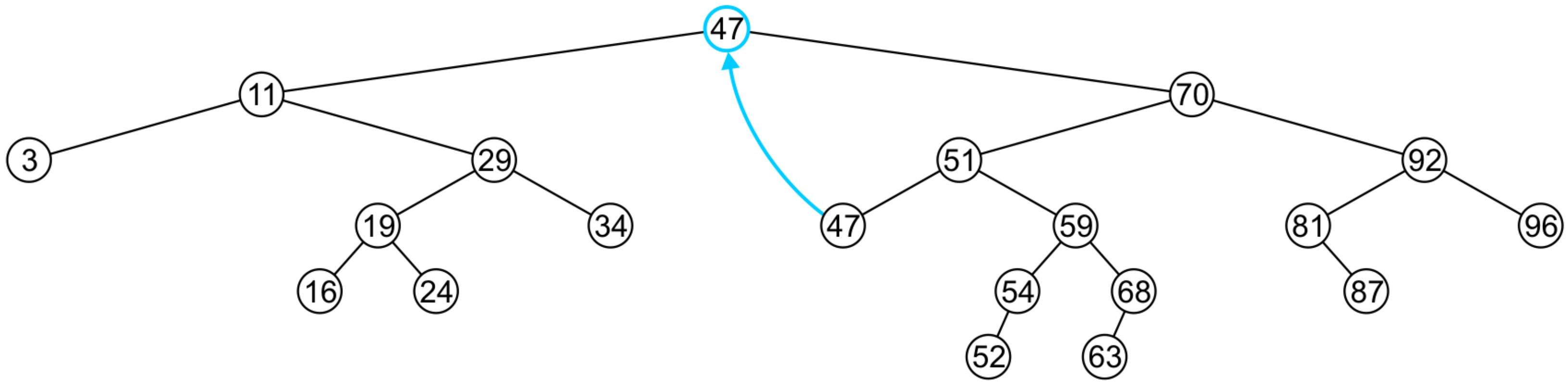
Erase

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



Erase

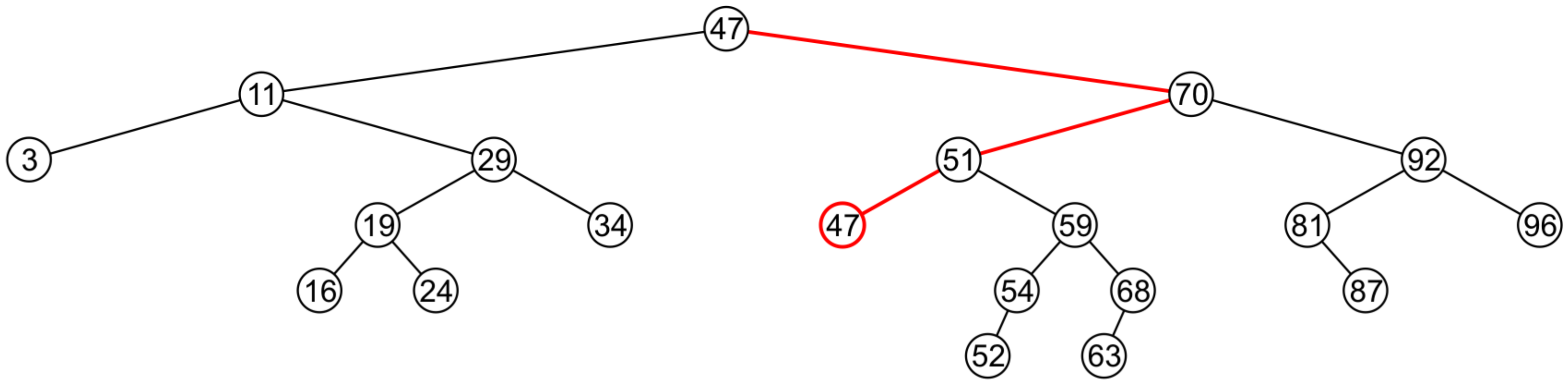
- In this case, we replace 42 with 47
  - We temporarily have two copies of 47 in the tree



Erase

We now recursively erase 47 from the right sub-tree

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

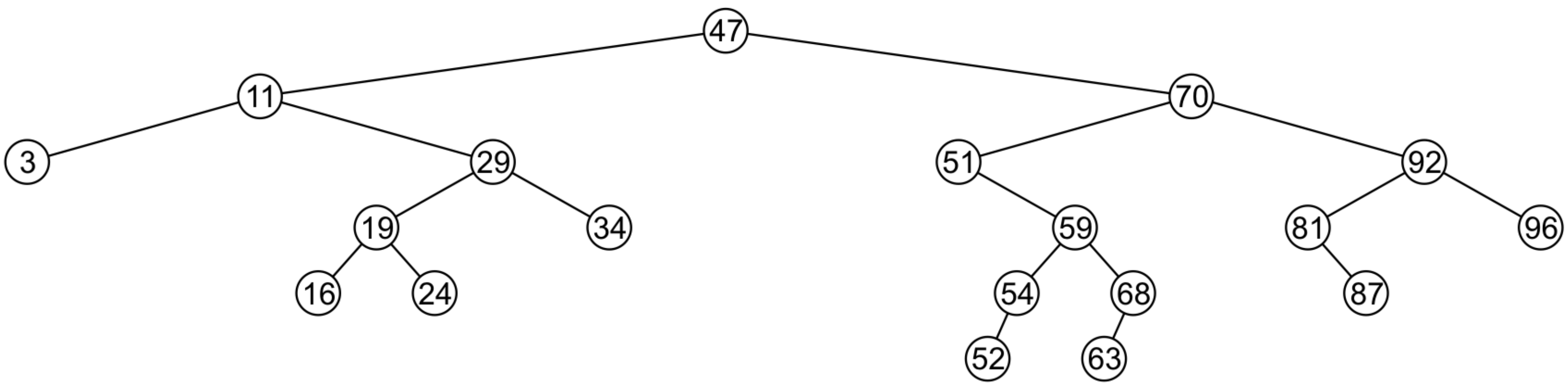


Erase

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

– Notice that the tree is still sorted:

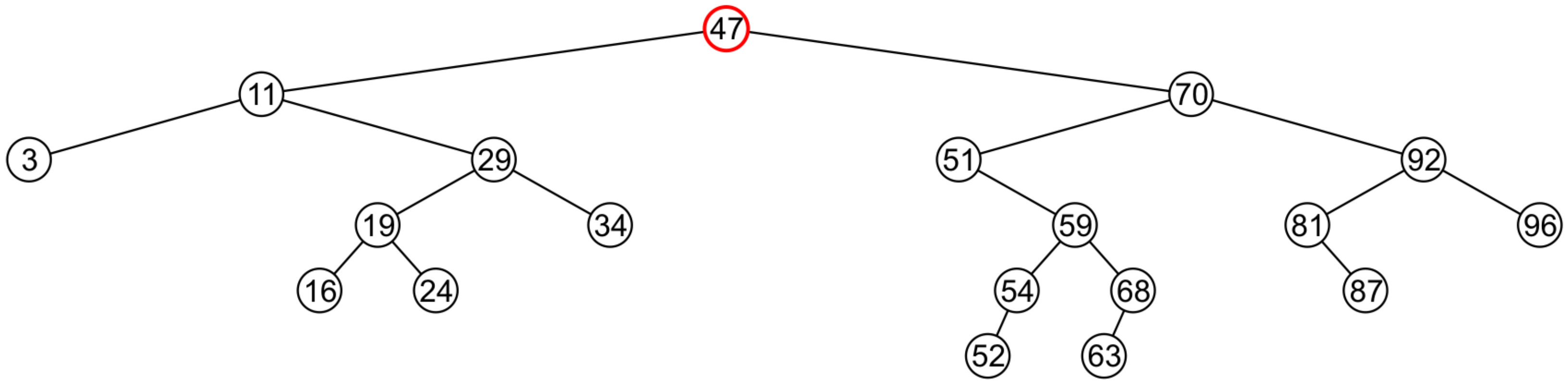
47 was the least object in the right sub-tree



Erase

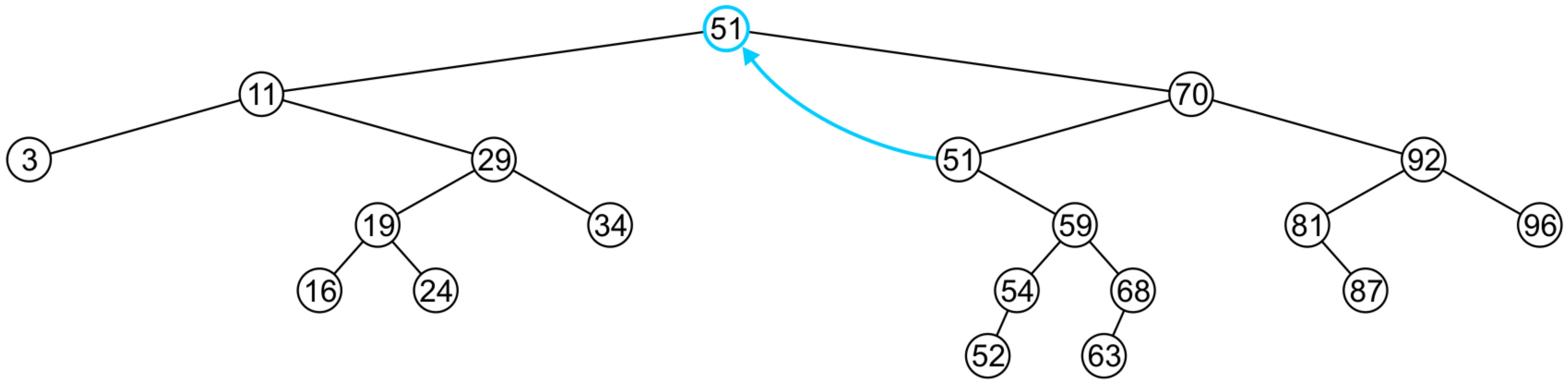
Suppose we want to erase the root 47 again:

- 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



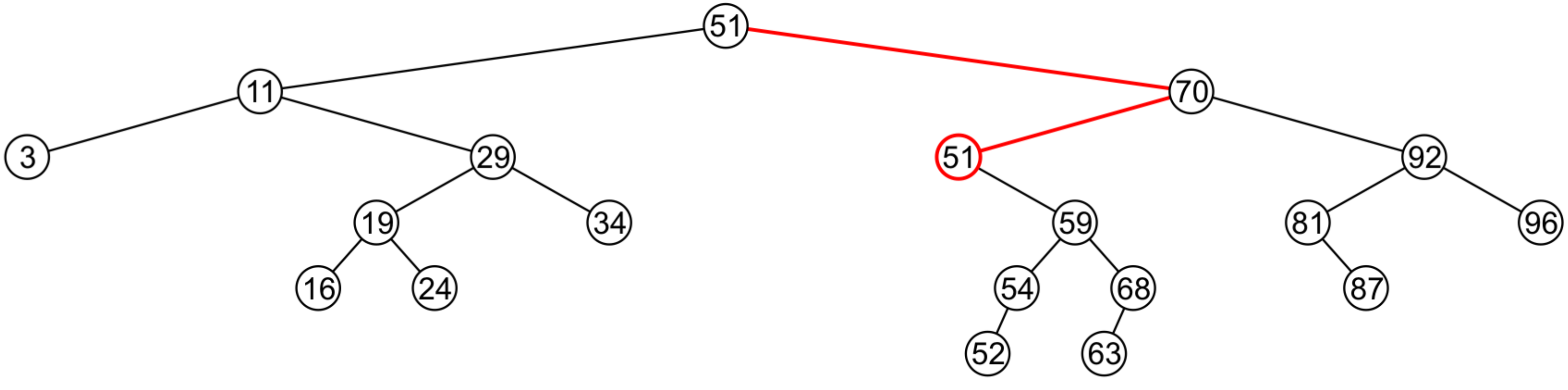
Erase

We copy 51 from the right sub-tree



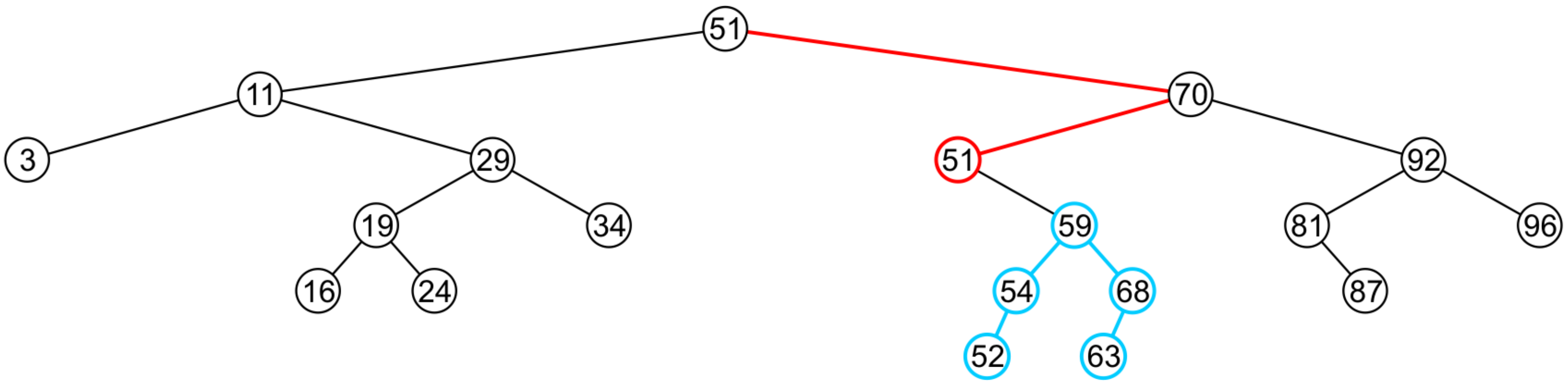
Erase

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



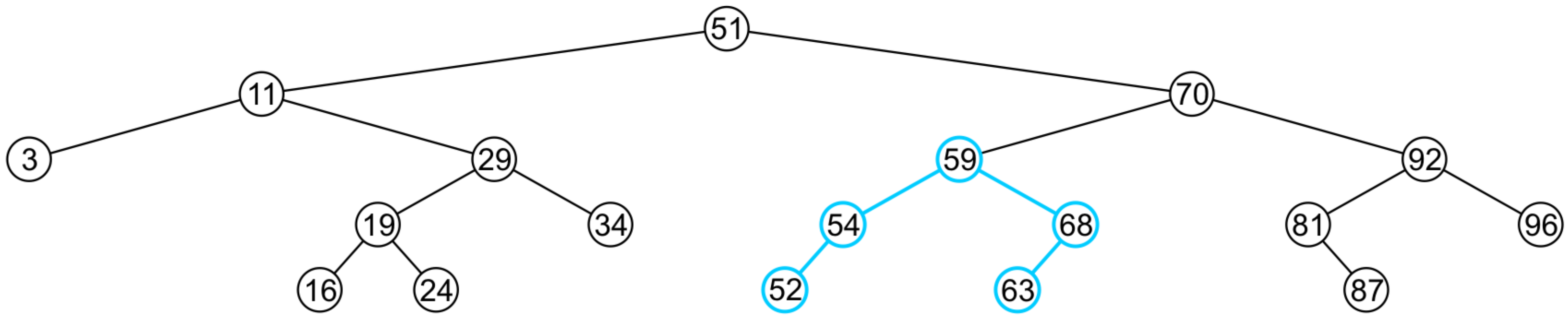
Erase

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



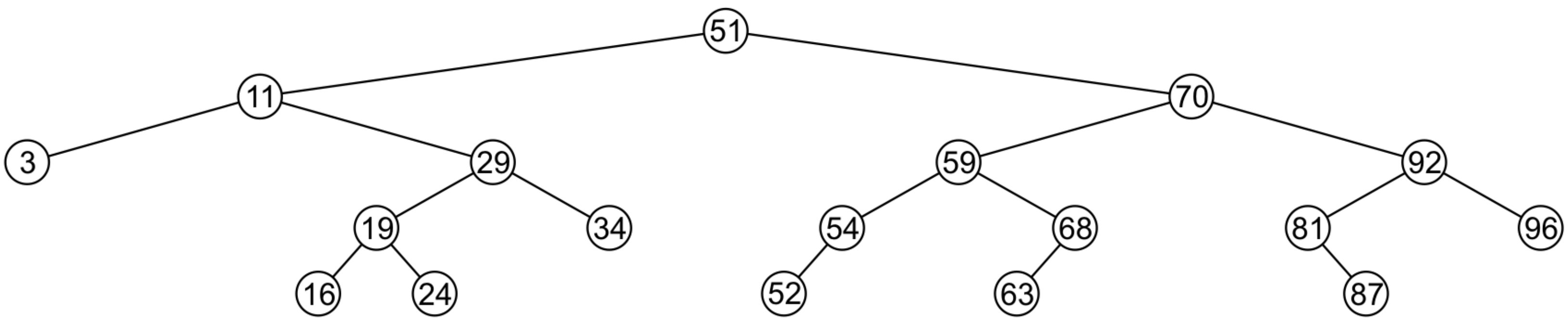
Erase

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

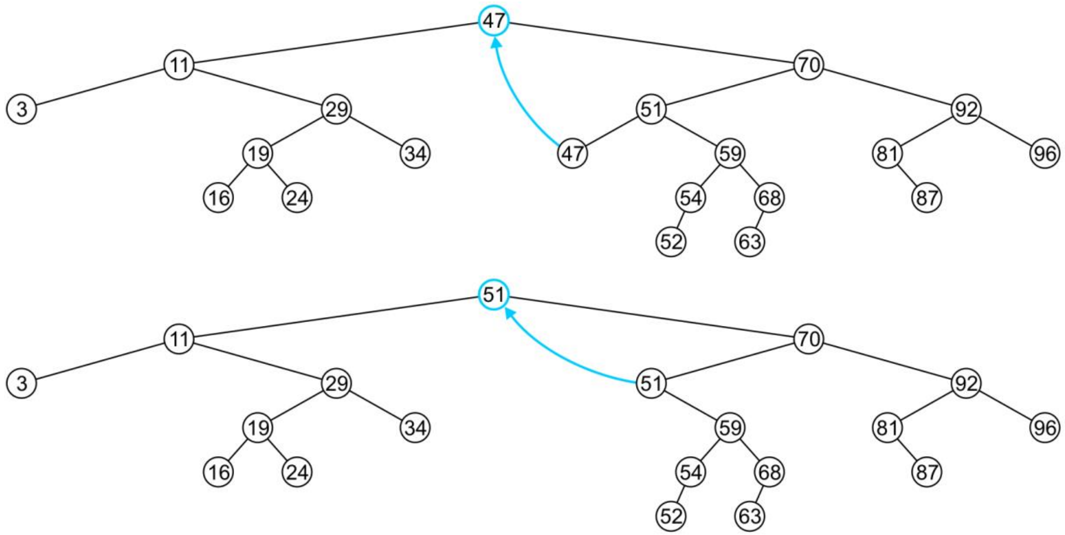


Erase

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

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

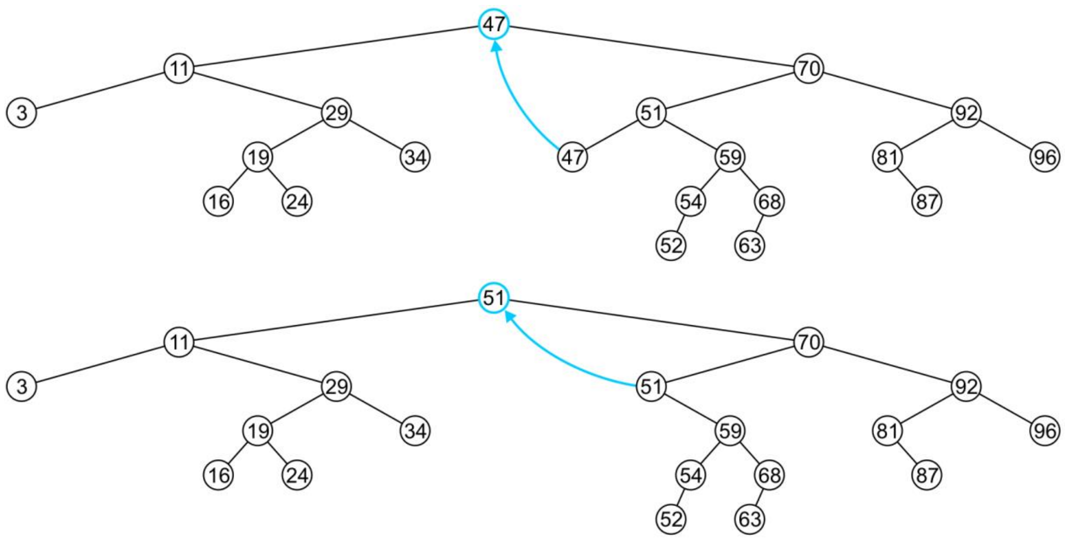
Is it possible, in removing a full node, to promote a child with two children?



Erase

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

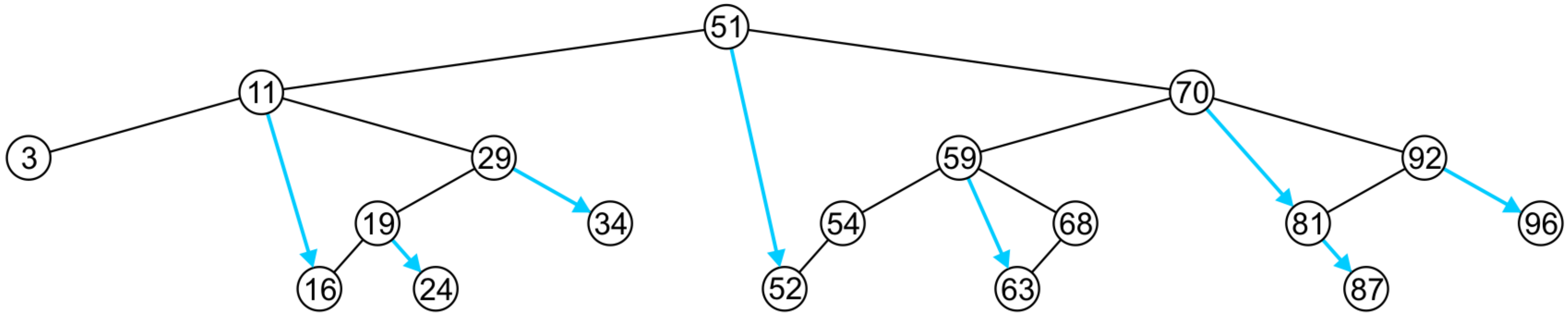
– If that node had a left sub-tree, that sub-tree would contain a smaller value



Previous and Next Objects

To find the next largest object:

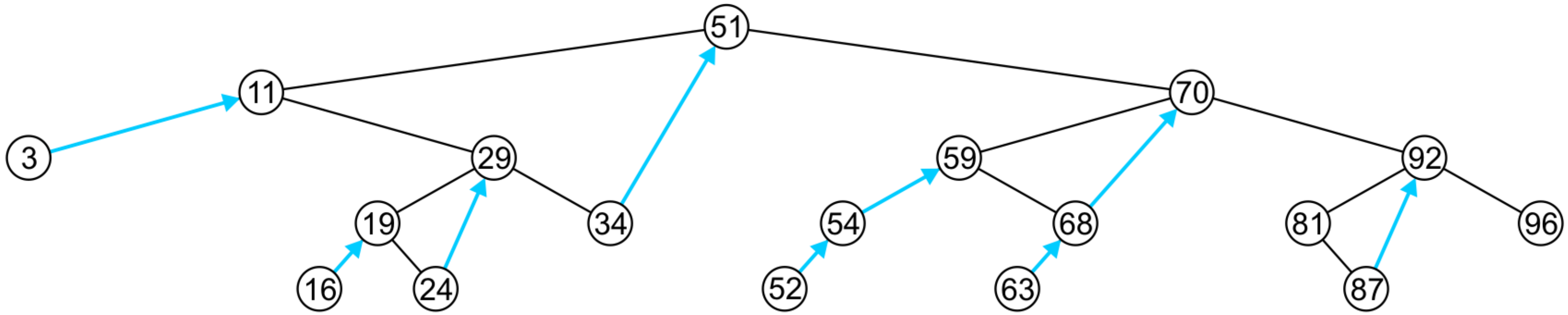
– If the node has a right sub-tree, the minimum object in that sub-tree is the next-largest object



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



- Go up and right to find this

Lazy Deletion

- Lazy deletion can work well for a BST
  - Simpler
  - Can do "real deletions" later as a batch
  - Some inserts can just "undelete" a tree node

- But
  - Can waste space and slow down find operations
  - Make some operations more complicated:
    - e.g., **`findMin`** and **`findMax`**?

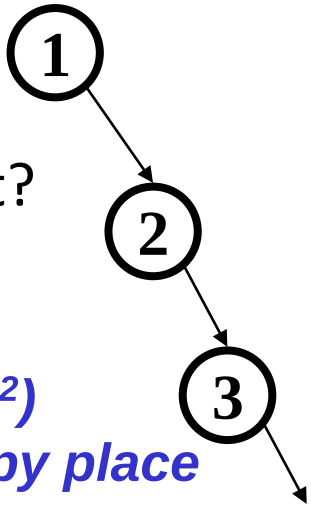
Finding the k^{th} Object

Another operation on sorted lists may be finding the k^{th} largest object

- Recall that k goes from 0 to $n - 1$
- If the left-sub-tree has $\ell = k$ entries, return the current node,
- If the left sub-tree has $\ell > k$ entries, return the k^{th} entry of the left sub-tree,
- Otherwise, the left sub-tree has $\ell < k$ entries, so return the $(k - \ell - 1)^{\text{th}}$ entry of the right sub-tree

BuildTree for BST

- Let's consider **`buildTree`**
  - Insert all, starting from an empty tree
- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
  - If inserted in given order, what is the tree?
  - What big-O runtime for this kind of sorted input?
    $1 + 2 + 3 + . . . + n = n(n+1)/2$
  - Is inserting in the reverse order any better?



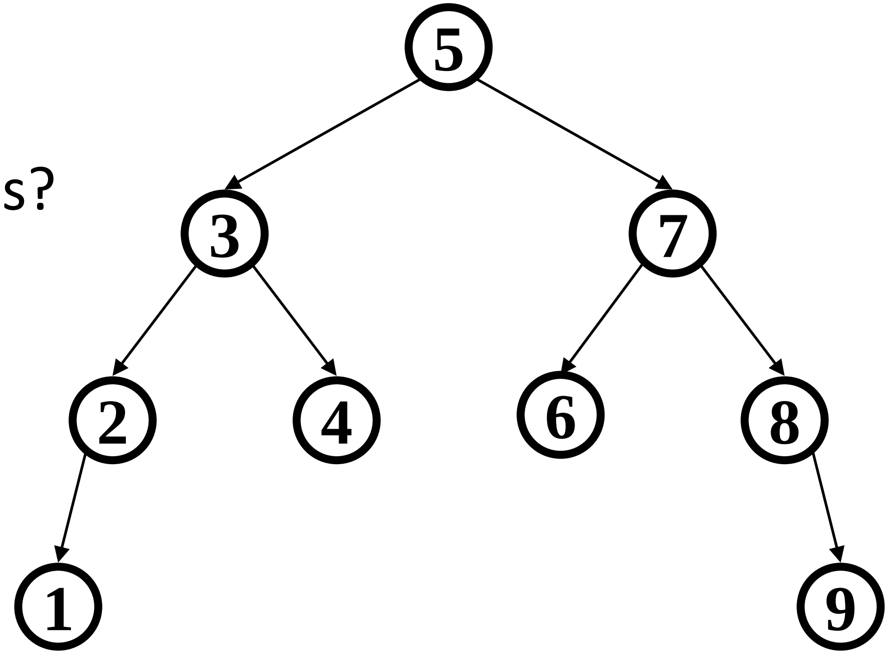
$O(n^2)$
Not a happy place

BuildTree for BST

- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
- What if we could somehow re-arrange them
  - median first, then left median, right median, etc.
  - 5, 3, 7, 2, 1, 4, 8, 6, 9
  - What tree does that give us?
  - What big-O runtime?

O(n log n), definitely better

  - **So the order the values come in is important!**



Complexity of Building a Binary Search Tree

- Worst case: $O(n^2)$
- Best case: $O(n \log n)$
- We do better by keeping the tree balanced.