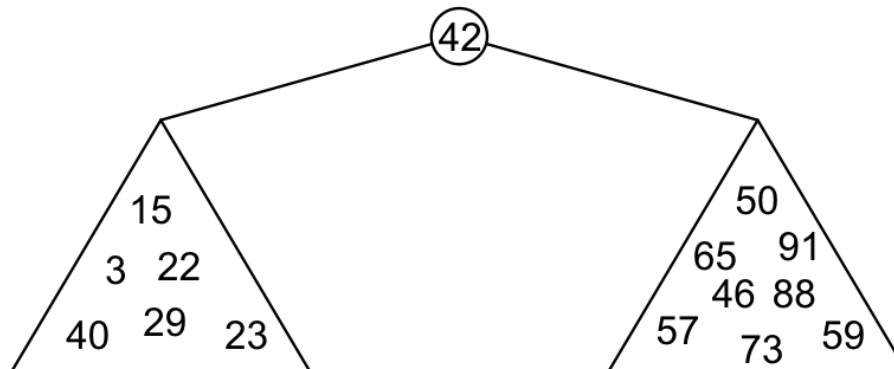


Data Structures

17. Binary Search Tree

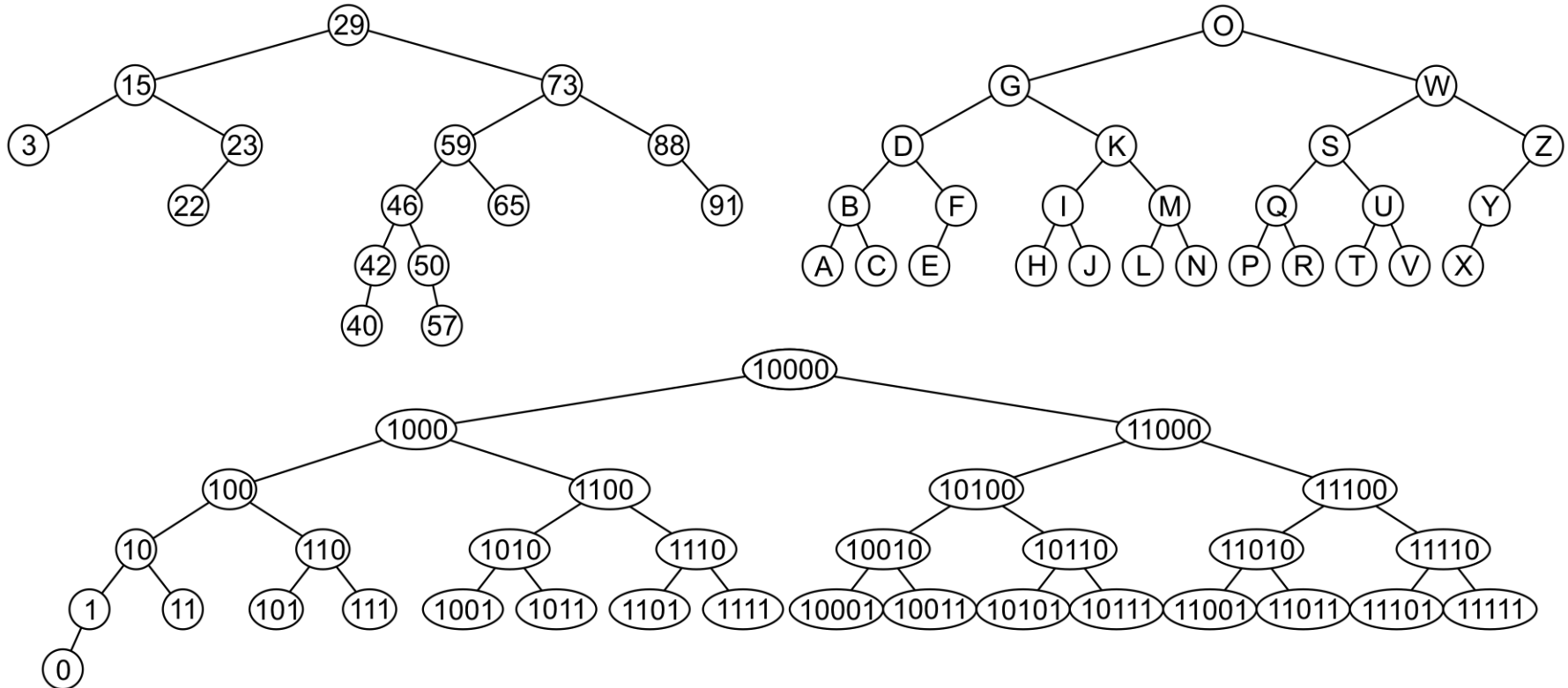
Binary Search Tree (BST)

- With a binary tree, we can dictate an order on the two children
- Binary Search Tree (BST) defines the following order:
 - All elements in the **left sub-tree to be less** than the element stored in the root node, and
 - All elements in the **right sub-tree to be greater** than the element in the root object

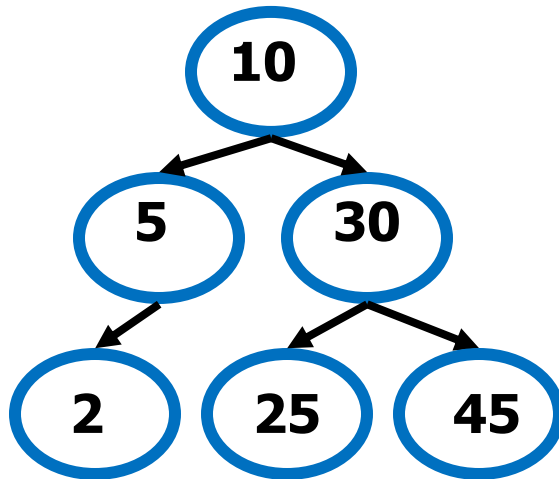


subtrees will
themselves be
binary search trees

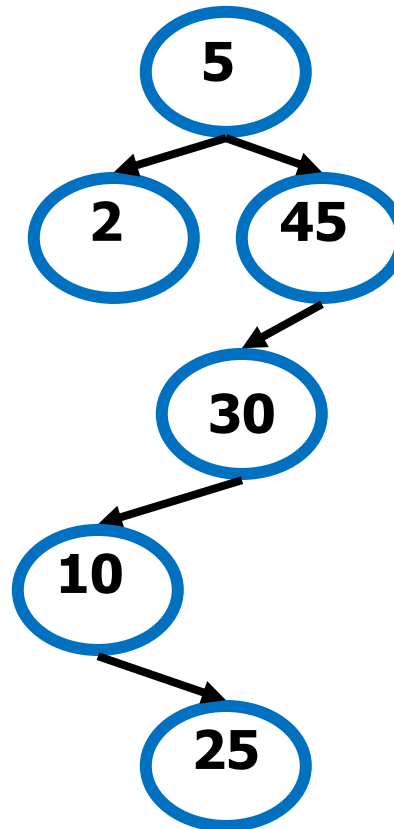
Binary Search Tree (BST) – Example (1)



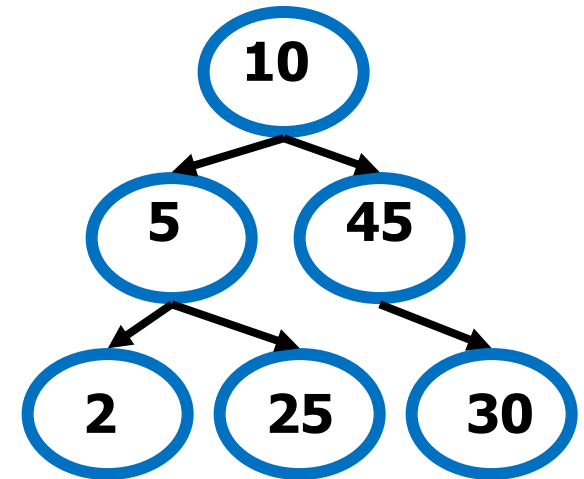
Binary Search Tree (BST) – Example (2)



BST



BST



Not BST

BST

BST Operations

- Many operations one can perform on a binary search tree
 - Creating a binary search tree
 - Finding a node in a binary search tree
 - Inserting a node into a binary search tree
 - Deleting a node in a binary search tree
 - Traversing a binary search tree
- In the following, we will examine the algorithms and examples for all of the above operations

Creating BST

- A simple class that implements a binary tree to store integer values
 - A class called IntBinaryTree
- Node of binary search tree

```
struct TreeNode
{
    int value;
    TreeNode *left;
    TreeNode *right;
};
```

Creating BST – Class Definition

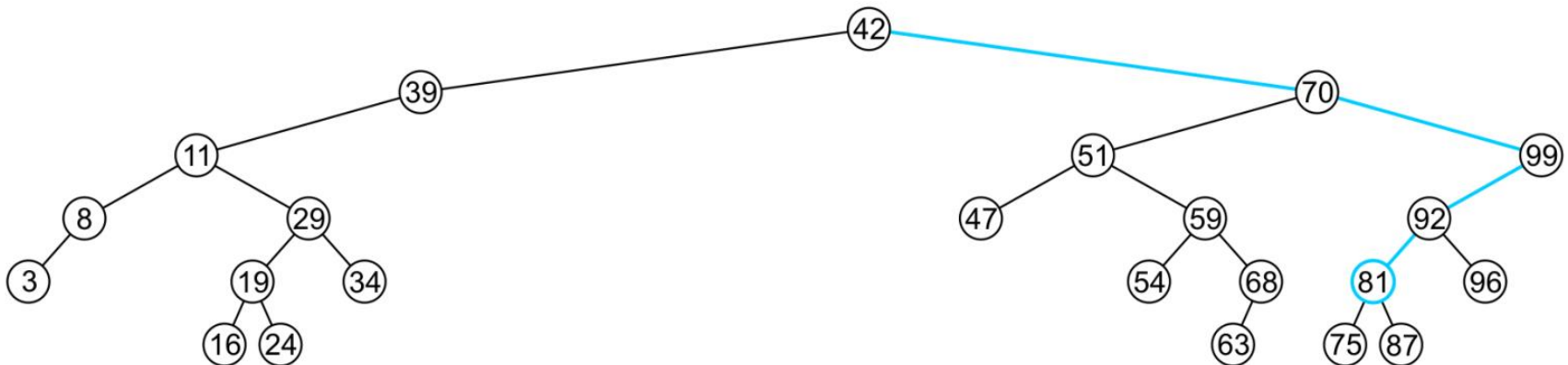
```
class IntBinaryTree {
    private:
        TreeNode *root; // Pointer to the root of BST

        void destroySubTree(TreeNode *); //Recursively delete all tree nodes
        void deleteNode(int, TreeNode *&);
        void makeDeletion(TreeNode *&);
        void displayInOrder(TreeNode *);
        void displayPreOrder(TreeNode *);
        void displayPostOrder(TreeNode *);

    public:
        IntBinaryTree()                { root = NULL; }
        ~IntBinaryTree()                { destroySubTree(root); }
        void insertNode(int);
        bool find(int);
        void remove(int num);           { deleteNode( num, root); }
        void showNodesInOrder()         { displayInOrder(root); }
        void showNodesPreOrder()        { displayPreOrder(root); }
        void showNodesPostOrder()       { displayPostOrder(root); }
};
```

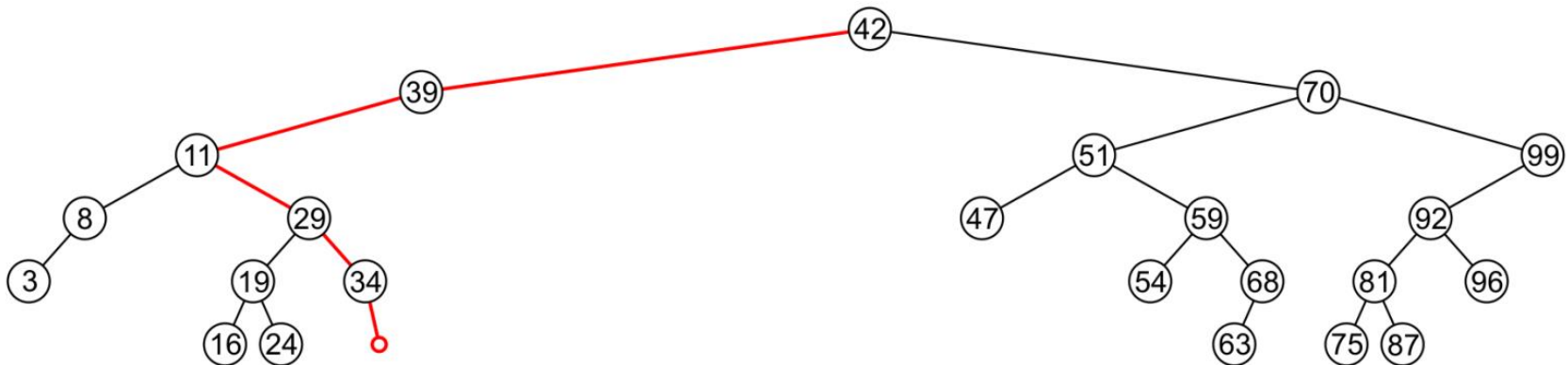
Finding a Node in BST

- Recall that a BST has the following key property (invariant):
 - Smaller values in left sub-tree
 - Larger values in right sub-tree
- For example: find (81)
 - Returns true if found



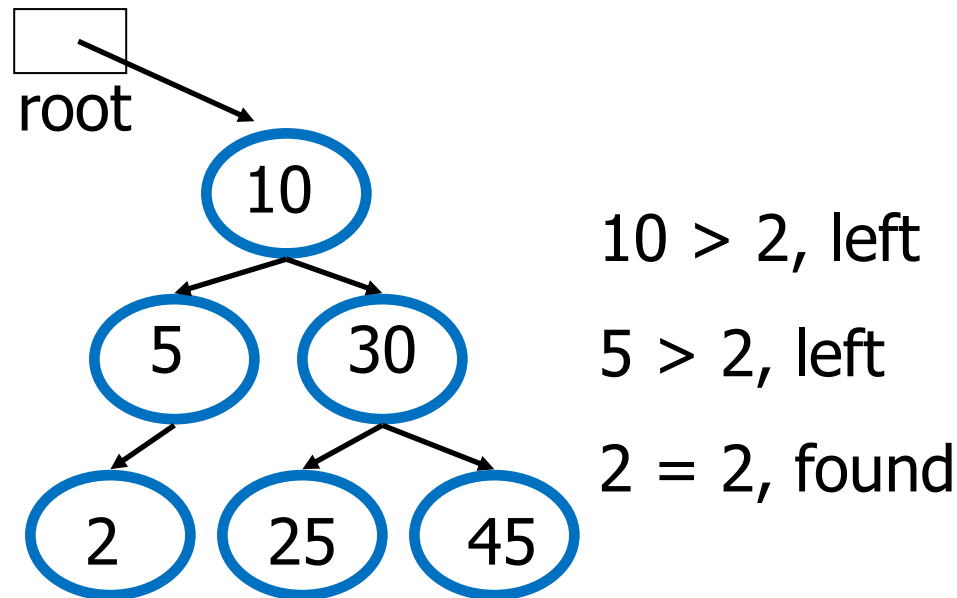
Finding a Node in BST

- Recall that a BST has the following key property (invariant):
 - Smaller values in left sub-tree
 - Larger values in right sub-tree
- For example: find (36)
 - Returns false if not found



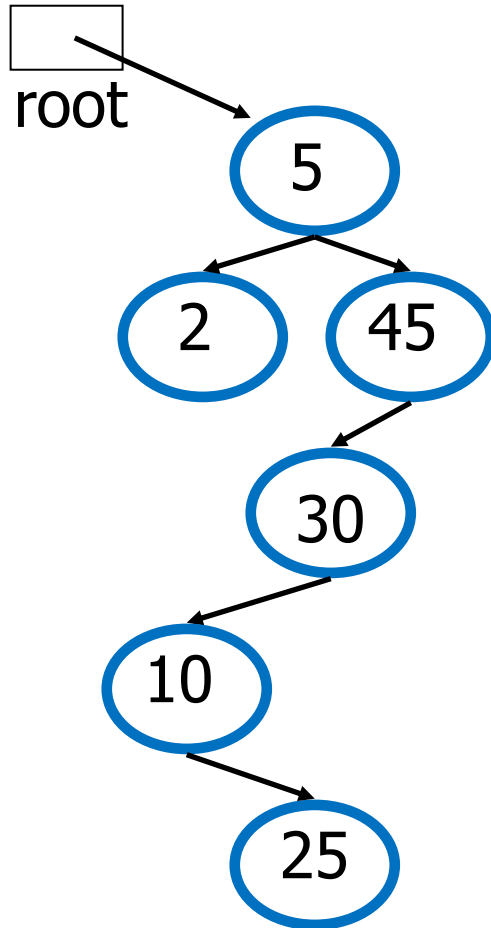
Finding a Node in BST

- Example: `find(2)`



Finding a Node in BST

- Example: `find(25)`



$5 < 25$, right

$45 > 25$, left

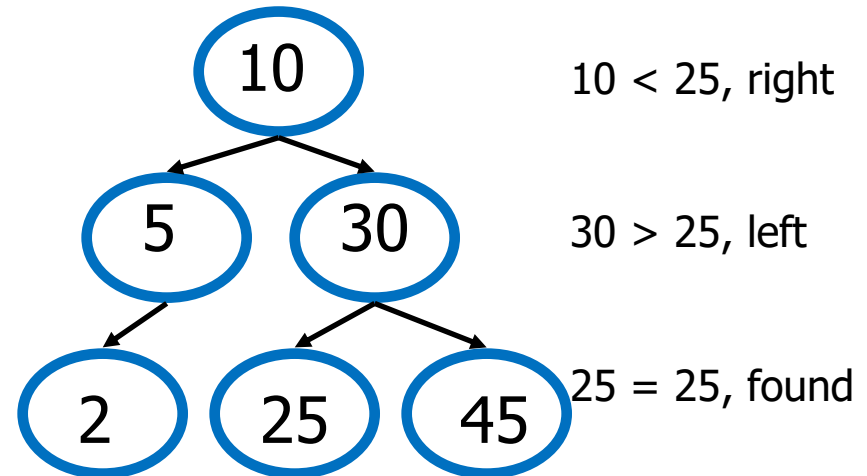
$30 > 25$, left

$10 < 25$, right

$25 = 25$, found

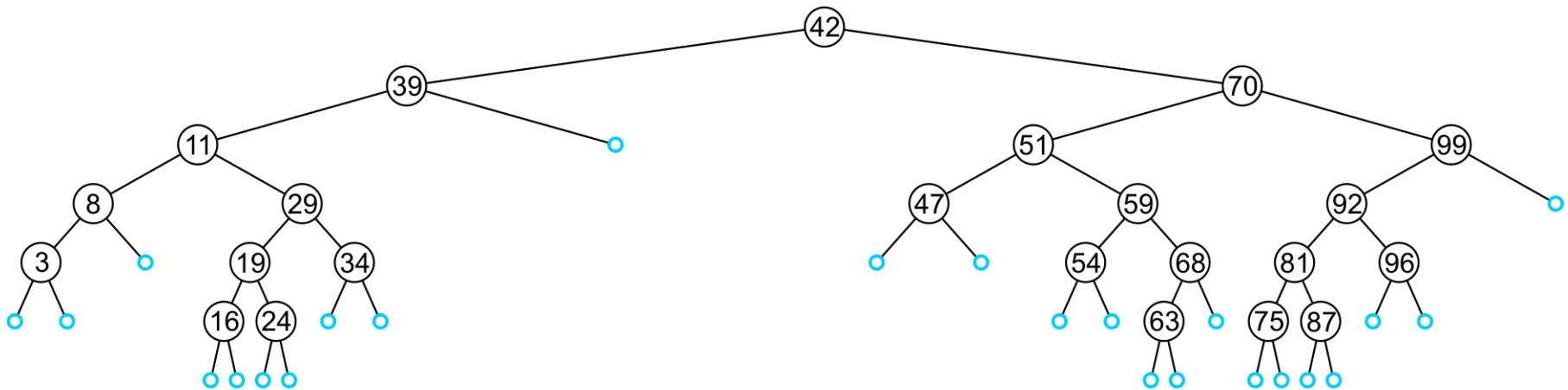
Finding a Node in BST – Implementation

```
bool IntBinaryTree::find(int num){  
    // The function starts from the root  
    TreeNode *nodePtr = root;  
  
    while (nodePtr) {  
        if (nodePtr->value == num)  
            return true; // value is found  
        else if (num < nodePtr->value)  
            nodePtr = nodePtr->left;  
        else  
            nodePtr = nodePtr->right;  
    }  
    return false; // value not found  
}
```



Inserting a Node in BST

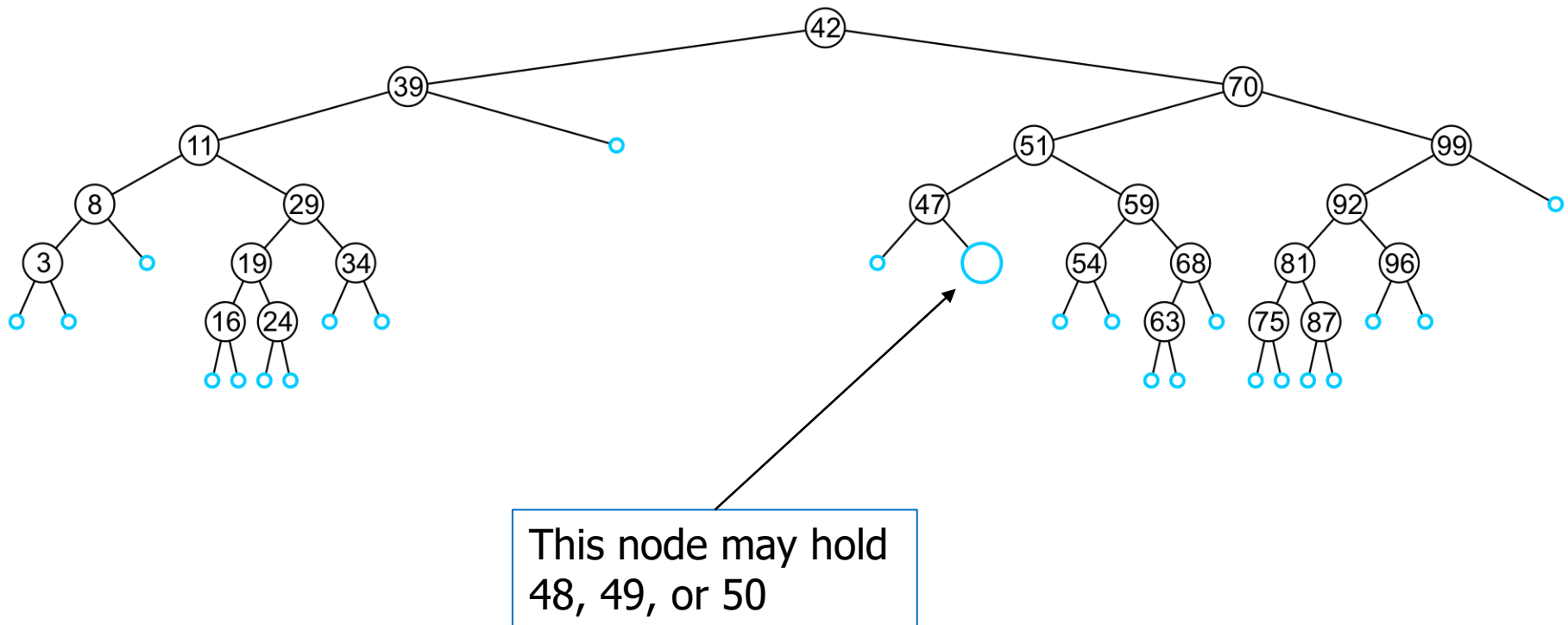
- An insertion will be performed at a leaf node
 - Any empty node is a possible location for an insertion



- Values which may be inserted at any empty node depend on the surrounding nodes

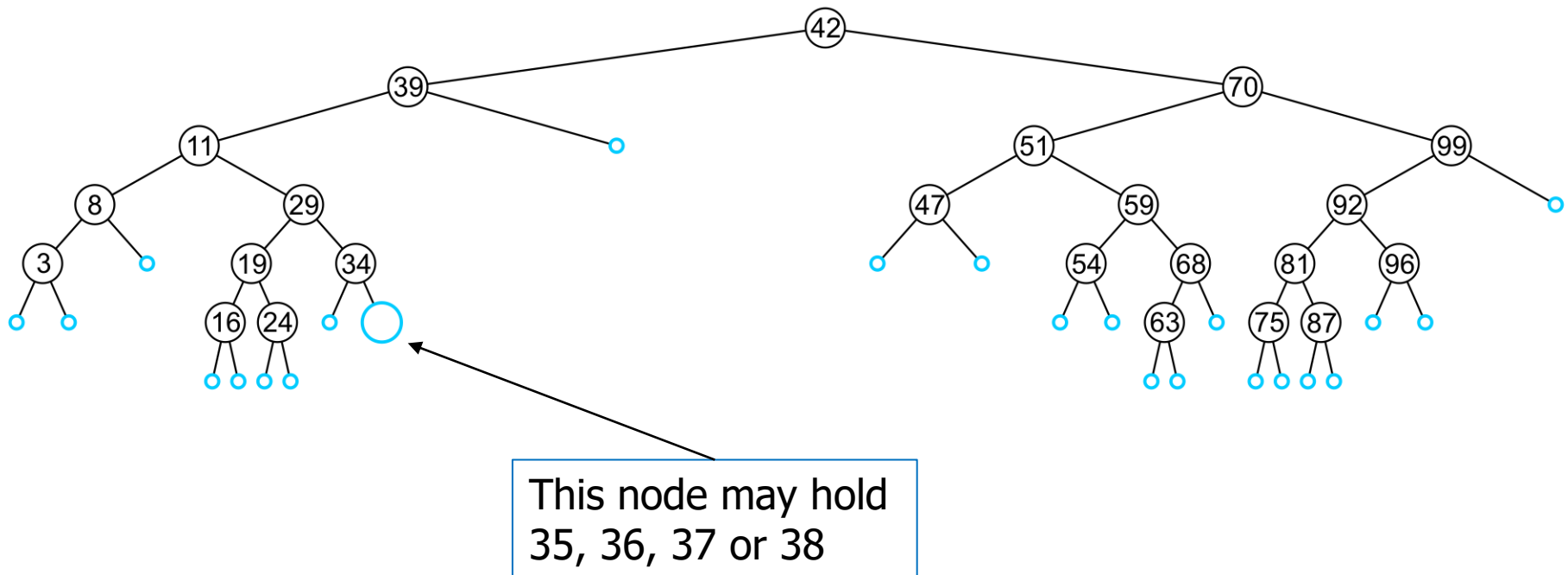
Inserting a Node in BST

- Which values can be held by empty node?



Inserting a Node in BST

- Which values can be held by empty node?

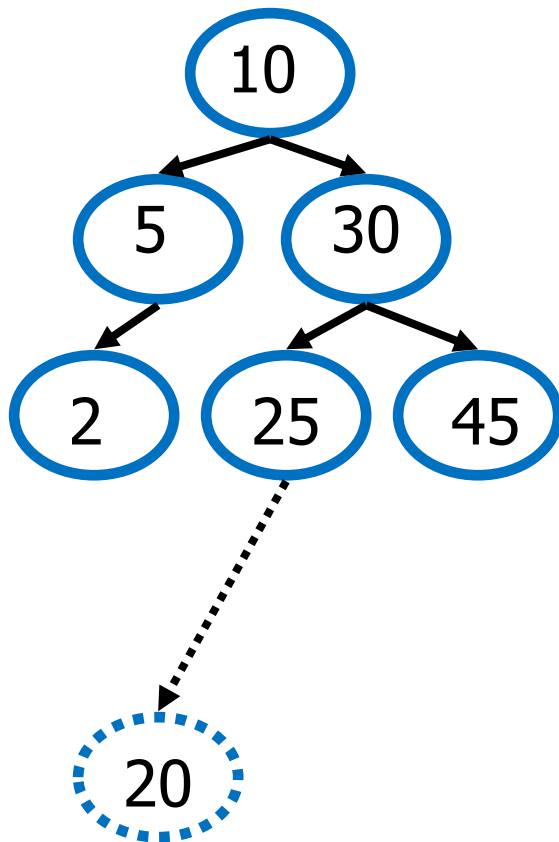


Inserting a Node in BST – Algorithm

- Like find, algorithm will step through the tree
 - If algorithm find the object already in the tree, it will return
 - The object is already in the binary search tree (no duplicates)
 - Otherwise, algorithm will arrive at an empty node
 - The object will be inserted into that location
- Why no duplicates?
 - In reality, it is seldom the case where duplicate elements in a BST must be stored as separate entities

Inserting a Node in BST – Example

- `insertNode(20)`



$10 < 20$, right

$30 > 20$, left

$25 > 20$, left

Insert 20 on left

Inserting a Node in BST – Implementation

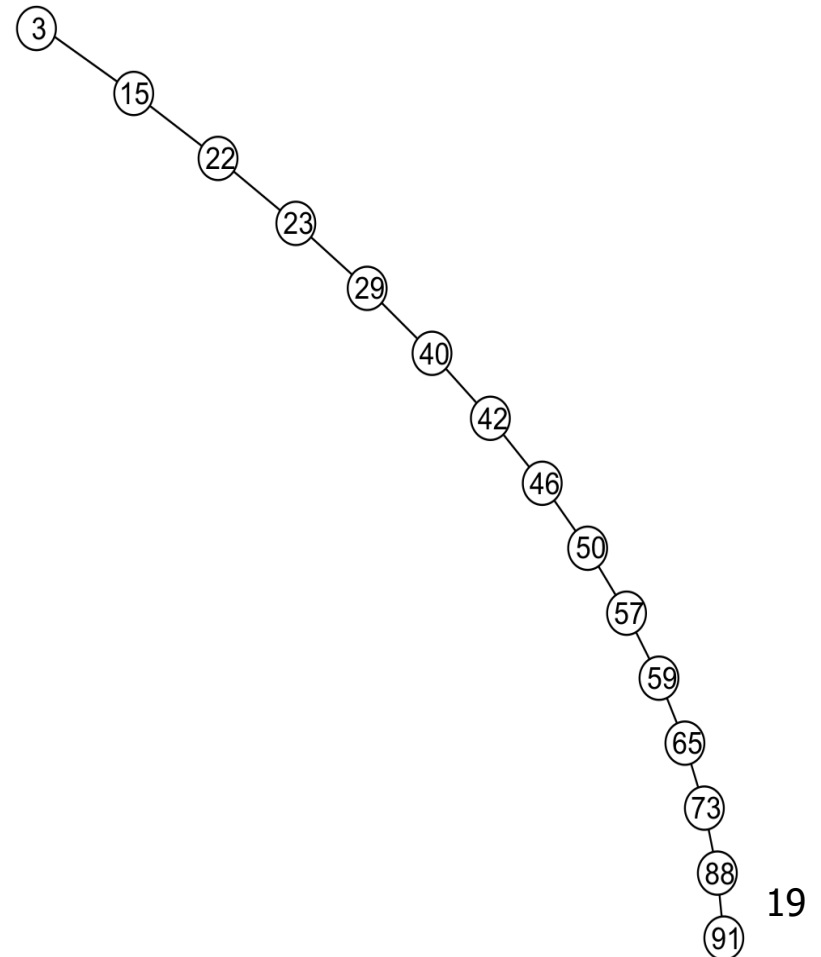
```
void IntBinaryTree::insertNode(int num) {
    TreeNode *newNode, *nodePtr; // Pointer to create new node & traverse tree

    newNode = new TreeNode; // Create a new node
    newNode->value = num;
    newNode->left = newNode->right = NULL;

    if (!root) root = newNode; // If tree is empty.
    else { // Tree is not empty
        nodePtr = root;
        while (nodePtr != NULL) {
            if (num < nodePtr->value) { // Left subtree
                if (nodePtr->left) { nodePtr = nodePtr->left; }
                else { nodePtr->left = newNode; break; }
            }
            else if (num > nodePtr->value) { // Right subtree
                if (nodePtr->right) nodePtr = nodePtr->right;
                else { nodePtr->right = newNode; break; }
            }
            else { cout << "Duplicate value found in tree.\n"; break; }
        }
    }
}
```

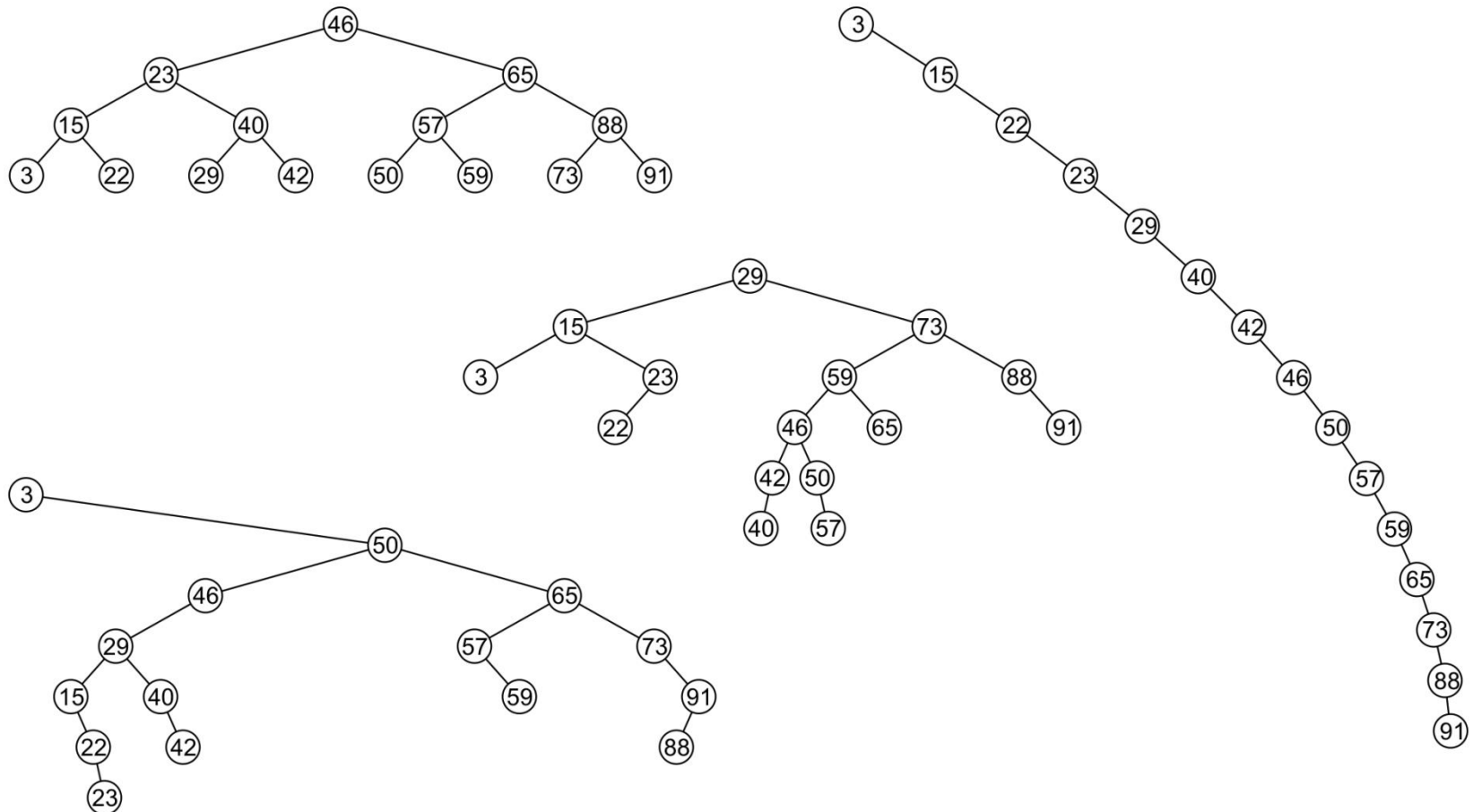
Inserting a Node in BST – Observation (1)

- Insertion may unbalance the tree
- It is possible to construct degenerate BST
 - The example is equivalent to a linked list



Inserting a Node in BST – Observation (2)

- All these binary search trees store the same data
 - Resultant tree depends on the order in which the values are inserted



Using BST (1)

```
// This program builds a binary tree with 5 nodes.  
// The SearchNode function determines if the  
// value 3 is in the tree.
```

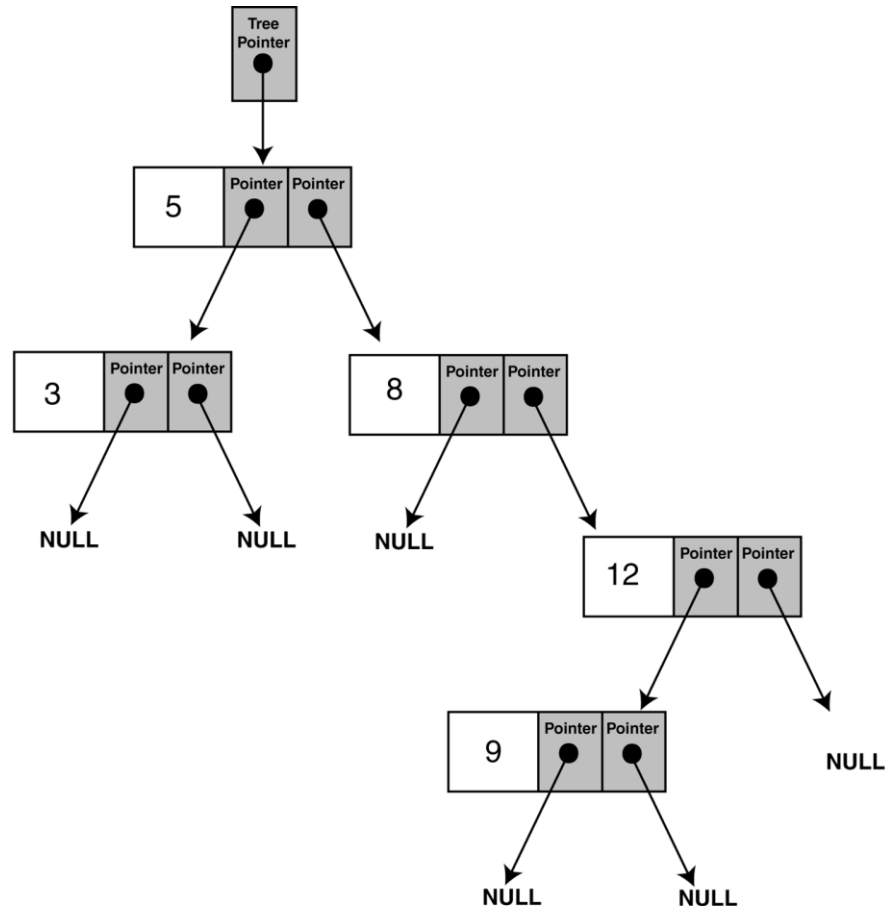
```
#include <iostream.h>  
#include "IntBinaryTree.h"
```

```
void main(void)  
{  
    IntBinaryTree tree;  
  
    cout << "Inserting nodes.\n";  
    tree.insertNode(5);  
    tree.insertNode(8);  
    tree.insertNode(3);  
    tree.insertNode(12);  
    tree.insertNode(9);  
    if (tree.find(3))  
        cout << "3 is found in the tree.\n";  
    else  
        cout << "3 was not found in the tree.\n";  
}
```

Output:
Inserting nodes.
3 is found in the tree.

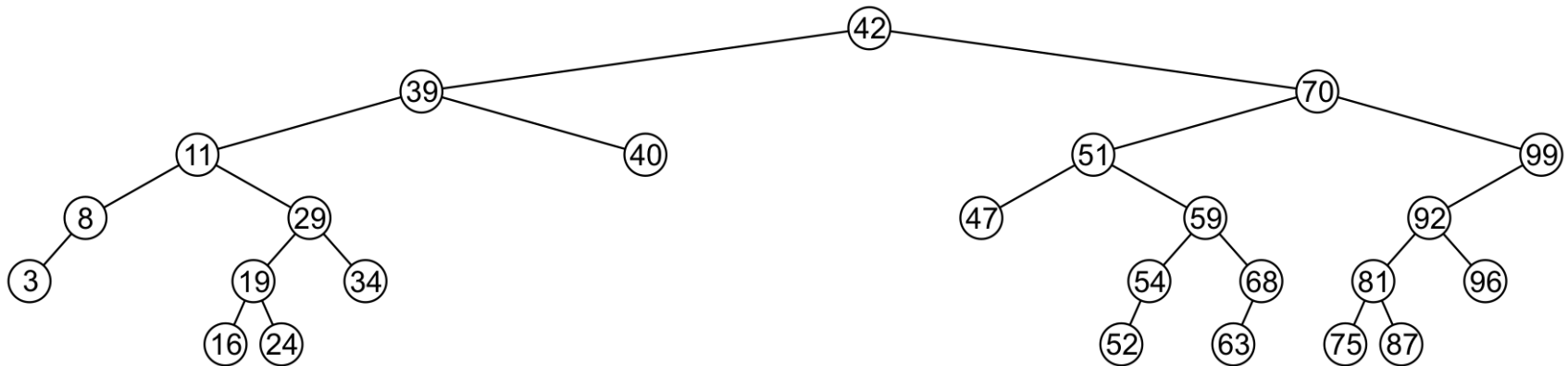
Using BST (2)

- Structure of binary tree built by the program



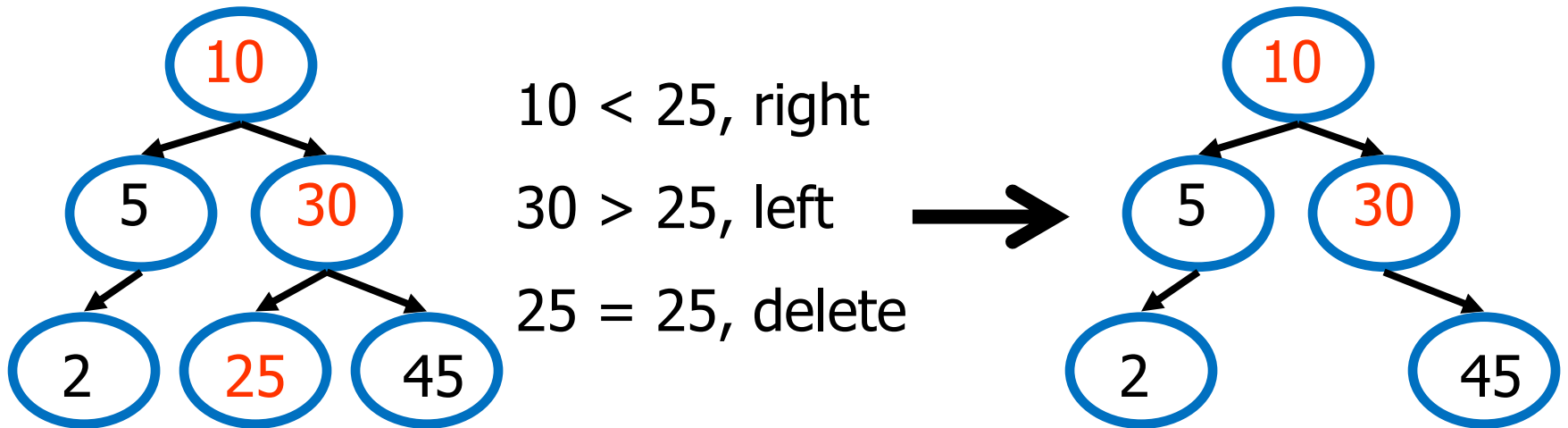
Deleting a Node

- 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)



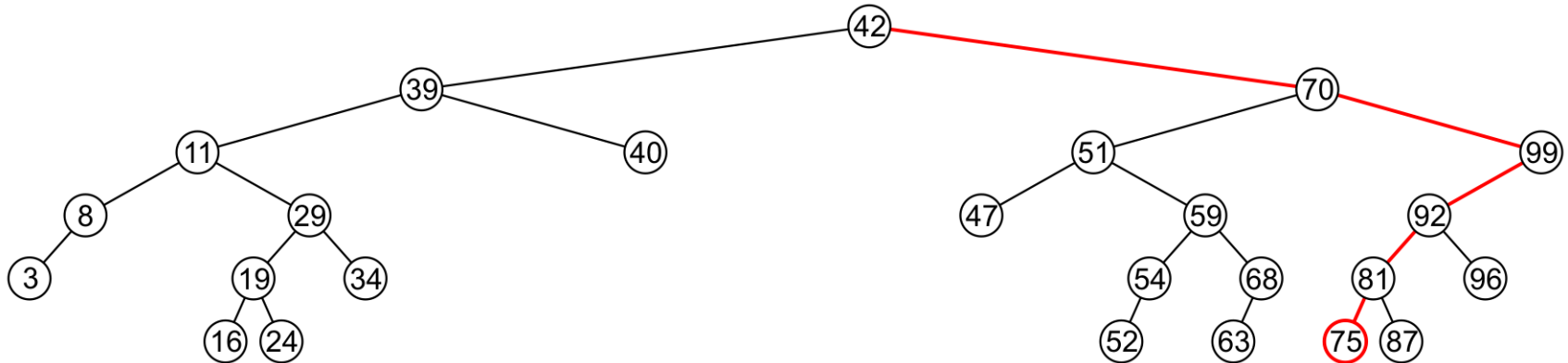
Deleting a Node – Leaf Node

- Deleting a leaf node is easy
 - Find its parent
 - Set the child pointer that links to it to NULL
 - Free the node's memory
- Consider deleting node containing 25



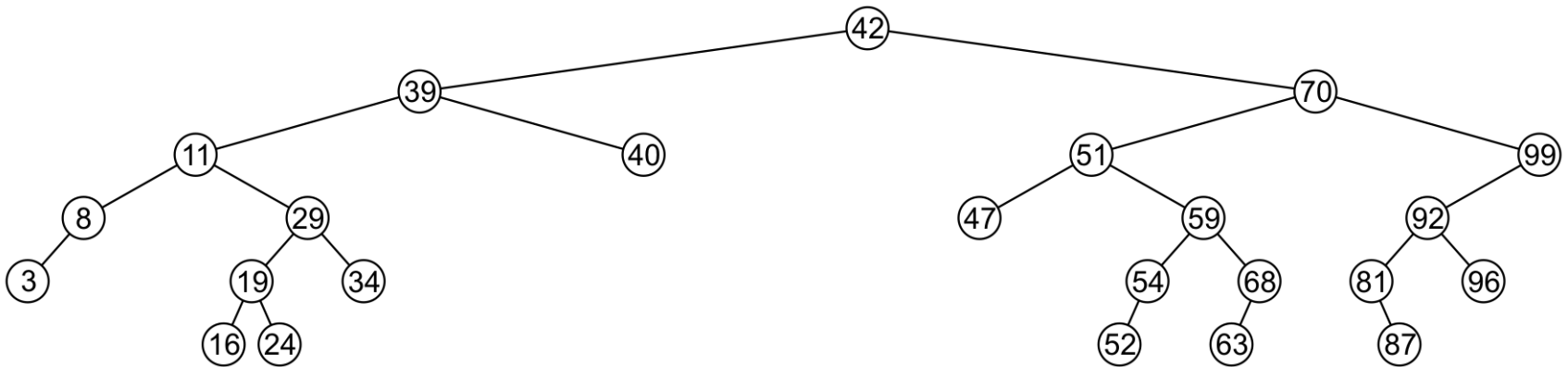
Deleting a Node – Leaf Node

- Consider deleting node containing 75



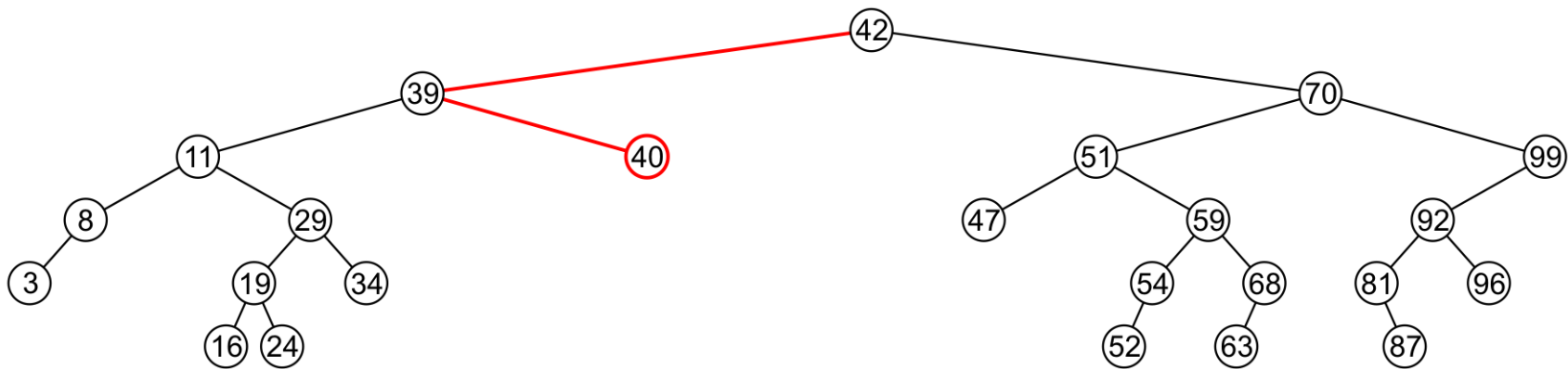
Deleting a Node – Leaf Node

- Consider deleting node containing 75
 - The node is deleted and left child of 81 is set to NULL



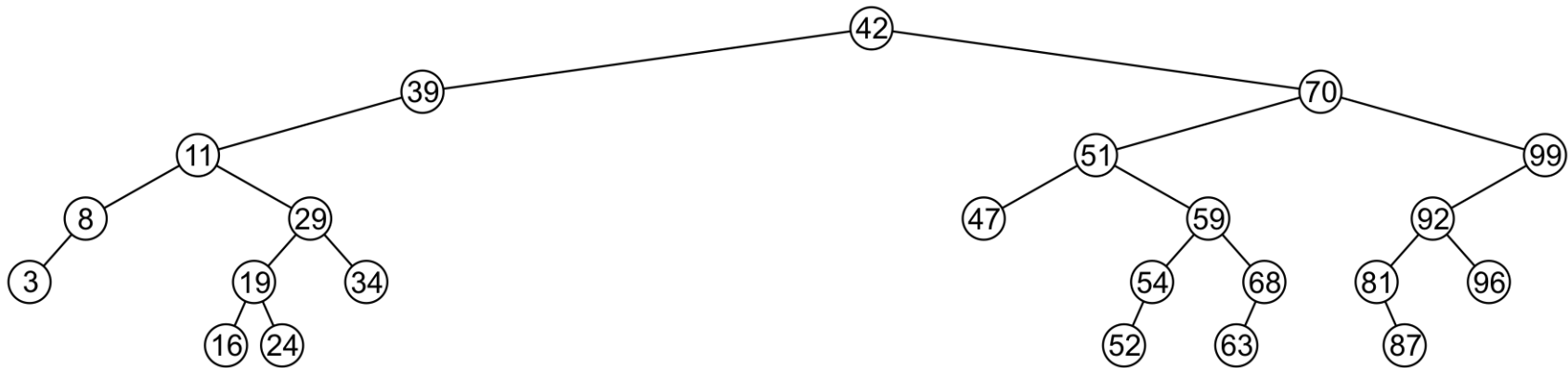
Deleting a Node – Leaf Node

- Consider deleting node containing 40



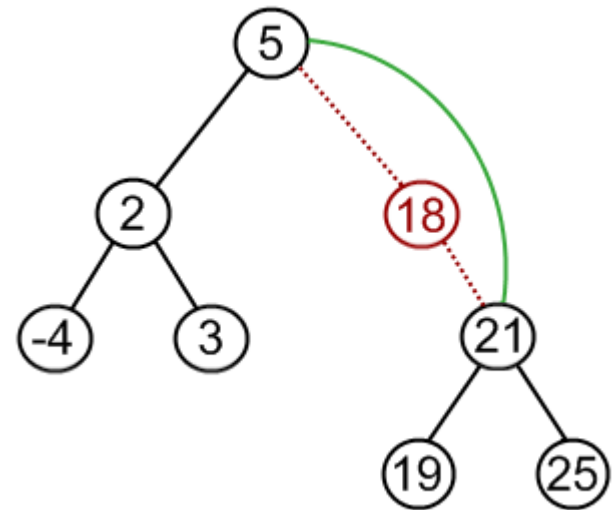
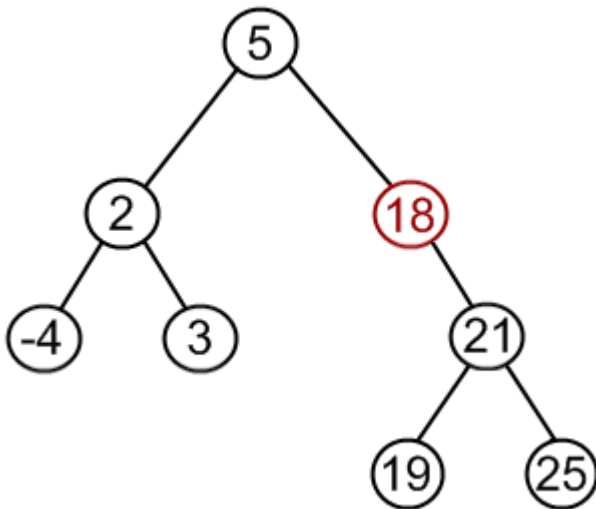
Deleting a Node – Leaf Node

- Consider deleting node containing 40
 - Node is deleted and right child of 39 is set to NULL



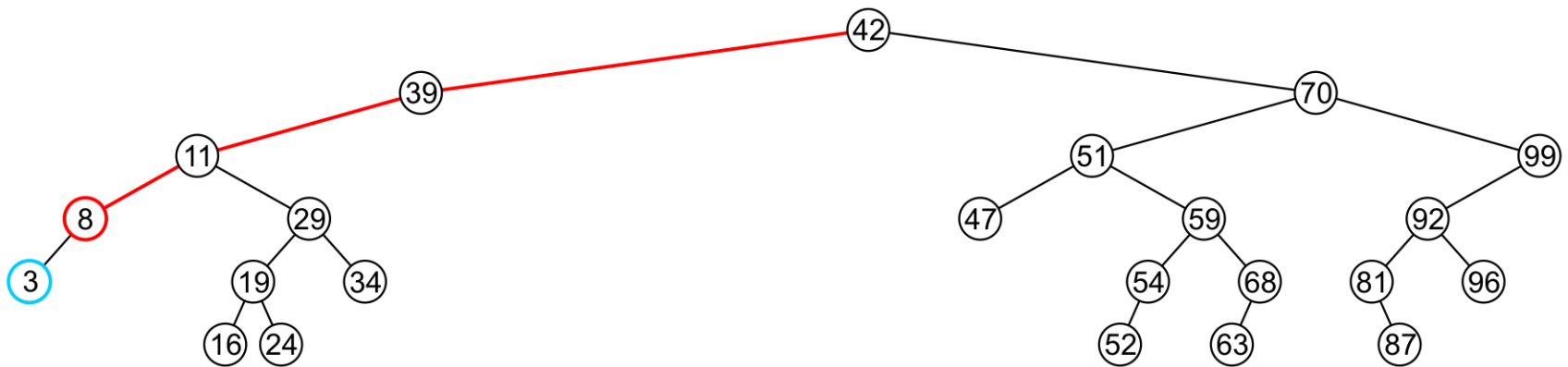
Deleting a Node – Node With Child

- If a node has only one child (left or right)
 - Simply promote the subtree associated with the child
- Consider deleting 18 which has one right child
 - Node 18 is deleted and right tree of node 5 is update to point to 21



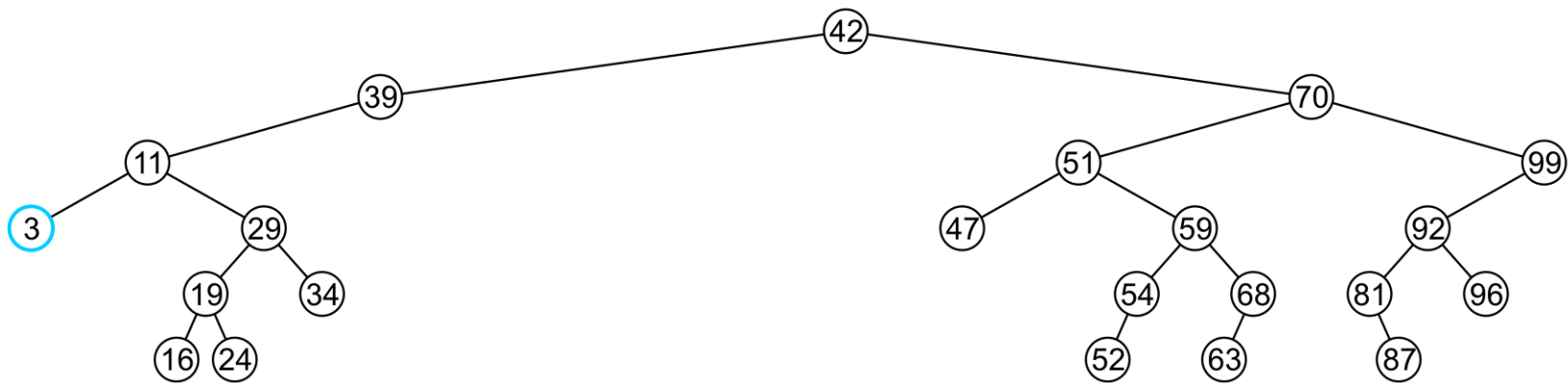
Deleting a Node – Node With Child

- Consider deleting 8 which has one left child



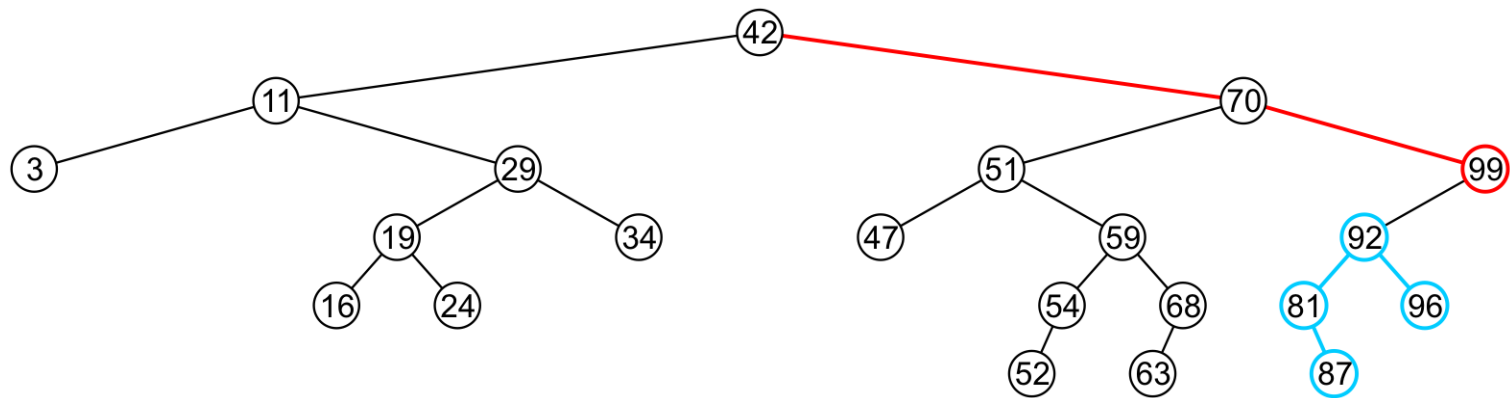
Deleting a Node – Node With Child

- Consider deleting 8 which has one left child
 - Node 8 is deleted and the left tree of 11 is updated to point to 3



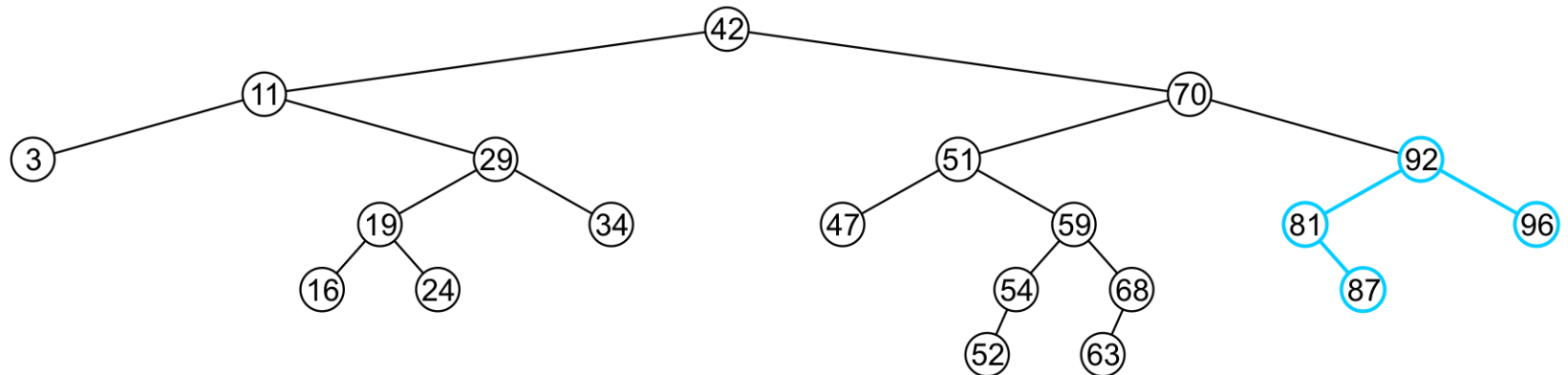
Deleting a Node – Node With Child

- Consider deleting the node containing 99



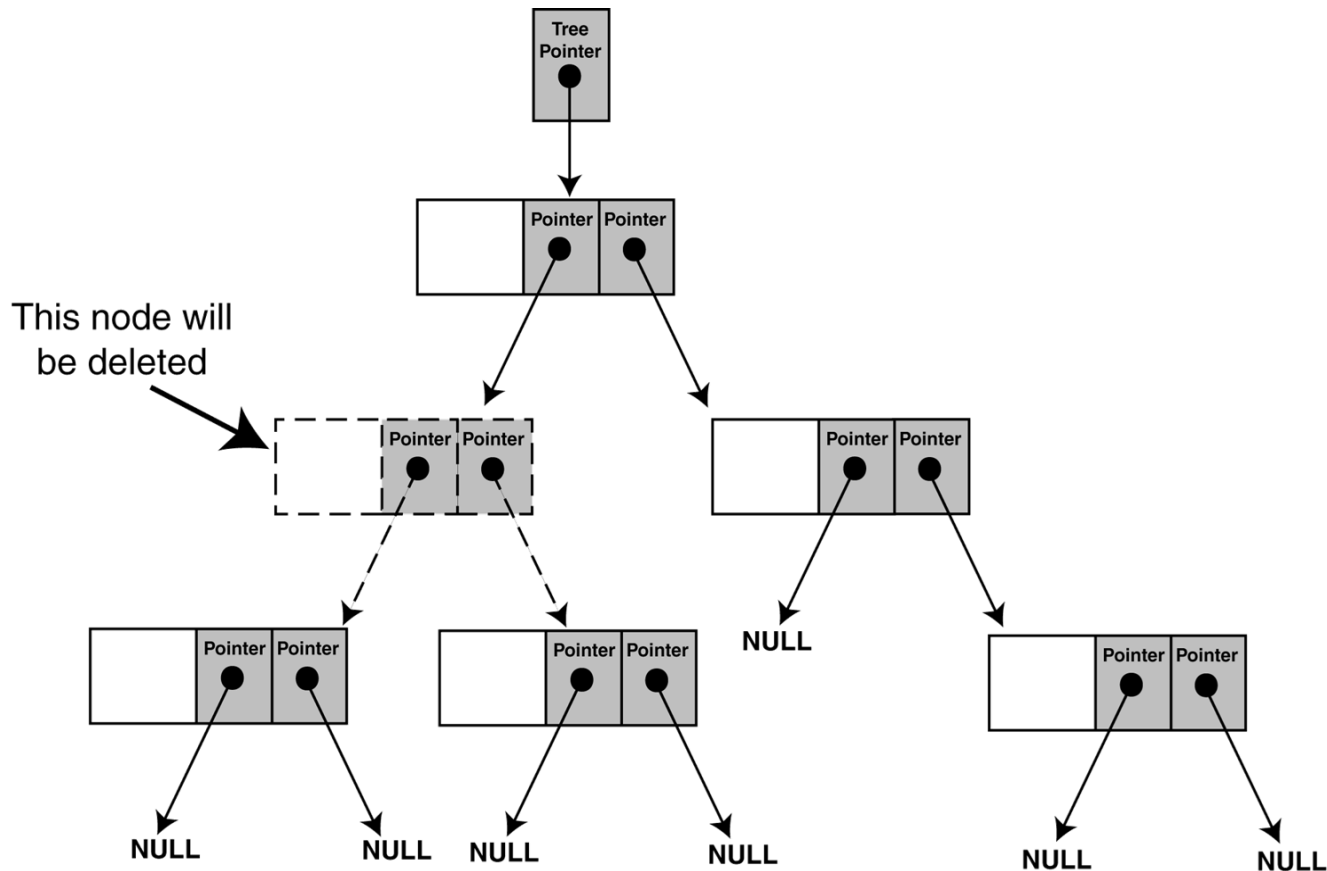
Deleting a Node – Node With Child

- Consider deleting the node containing 99
 - The right tree of 70 is set to point to node 92
 - Again, the order of the tree is maintained



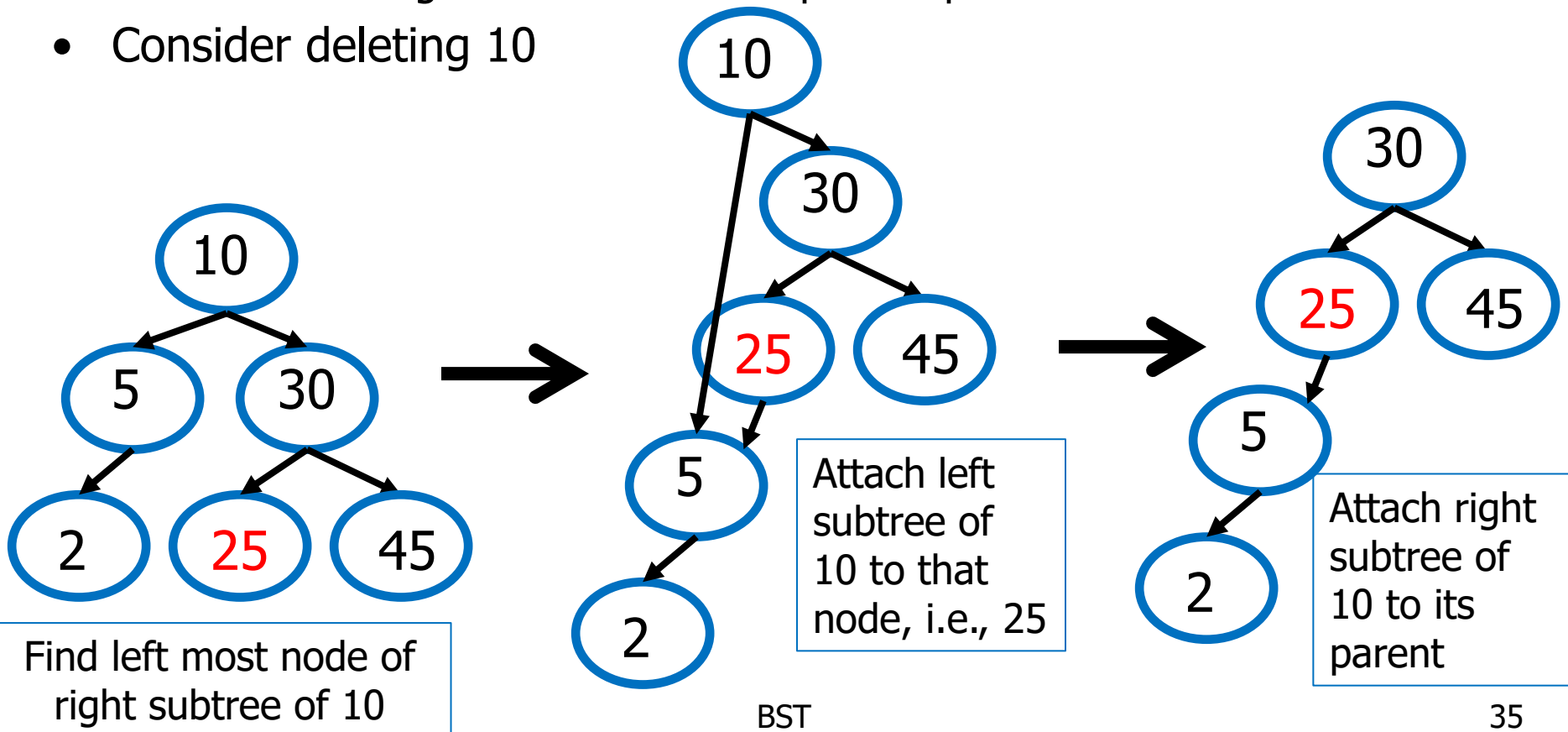
Deleting a Node – Node With Two Children

- The problem is not as easily solved if the node has two children



Deleting a Node – Node With Two Children

- Suppose node p with two children has to be deleted
 - Find a position in the right subtree of p to attach its left subtree
 - Left most node in the right subtree of node p
 - Attach the right subtree of node p to its parent
- Consider deleting 10



Pointers Review

- Pointer to Pointer versus reference to Pointer?

```
int g_One=1;

void func(int* pInt);

int main()
{
    int nvar=2;
    int* pvar=&nvar;
    func(pvar);
    std::cout<<*pvar<<std::endl;
    return 0;
}

void func(int* pInt)
{
    pInt=&g_One;
}
```

```
int g_One=1;

void func(int*& rpInt);

int main()
{
    int nvar=2;
    int* pvar=&nvar;
    func(pvar);
    std::cout<<*pvar<<std::endl;
    return 0;
}

void func(int*& rpInt)
{
    rpInt=&g_One;
}
```

Deleting a Node – Implementation

```
class IntBinaryTree {  
    private:  
        TreeNode *root; // Pointer to the root of BST  
  
        void destroySubTree(TreeNode *); //Recursively delete all tree nodes  
        void deleteNode(int, TreeNode *&);  
        void makeDeletion(TreeNode *&);  
        void displayInOrder(TreeNode *);  
        void displayPreOrder(TreeNode *);  
        void displayPostOrder(TreeNode *);  
  
    public:  
        IntBinaryTree() { root = NULL; }  
        ~IntBinaryTree() { destroySubTree(root); }  
        void insertNode(int);  
        bool find(int);  
        void remove(int num); { deleteNode( num, root)}  
        void showNodesInOrder() { displayInOrder(root); }  
        void showNodesPreOrder() { displayPreOrder(root); }  
        void showNodesPostOrder() { displayPostOrder(root); }  
};
```

The argument passed to the remove function is the value of the node to be deleted.

Deleting a Node – Implementation

```
void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr)
{
    if (nodePtr == NULL) // node does not exist in the tree
        cout << num << " not found.\n";
    else if (num < nodePtr->value)
        deleteNode(num, nodePtr->left); // find in left subtree
    else if (num > nodePtr->value)
        deleteNode(num, nodePtr->right); // find in right subtree
    else // num == nodePtr->value i.e. node is found
        makeDeletion(nodePtr); // actually deletes node from BST
}
```

Note:

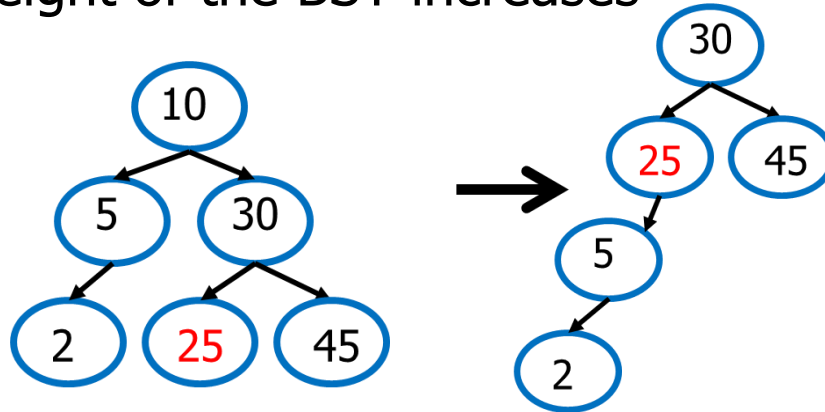
- The declaration of the `nodePtr` parameter: `TreeNode *&nodePtr;`
- `nodePtr` is a reference to a pointer to a `TreeNode` structure
 - Any action performed on `nodePtr` is actually performed on the argument passed into `nodePtr`

Deleting a Node – Implementation

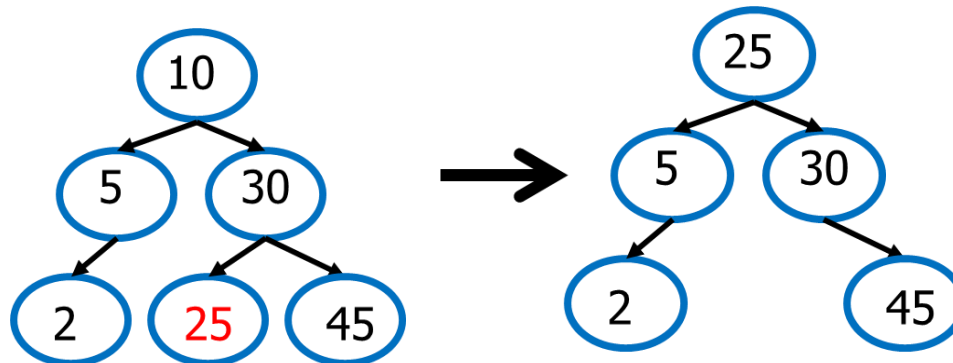
```
void IntBinaryTree::makeDeletion(TreeNode *&nodePtr) {
    TreeNode *tempNodePtr; // Temporary pointer
    if (nodePtr->right == NULL) { // case for leaf and one (left) child
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->left; // Reattach the left child
        delete tempNodePtr;
    }
    else if (nodePtr->left == NULL) { // case for one (right) child
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->right; // Reattach the right child
        delete tempNodePtr;
    }
    else { // case for two children.
        tempNodePtr = nodePtr->right; // Move one node to the right
        while (tempNodePtr->left) { // Go to the extreme left node
            tempNodePtr = tempNodePtr->left;
        }
        tempNodePtr->left = nodePtr->left; // Reattach the left subtree
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->right; // Reattach the right subtree
        delete tempNodePtr;
    }
}
```

Deleting a Node – Node With Two Children

- Problem: Height of the BST increases

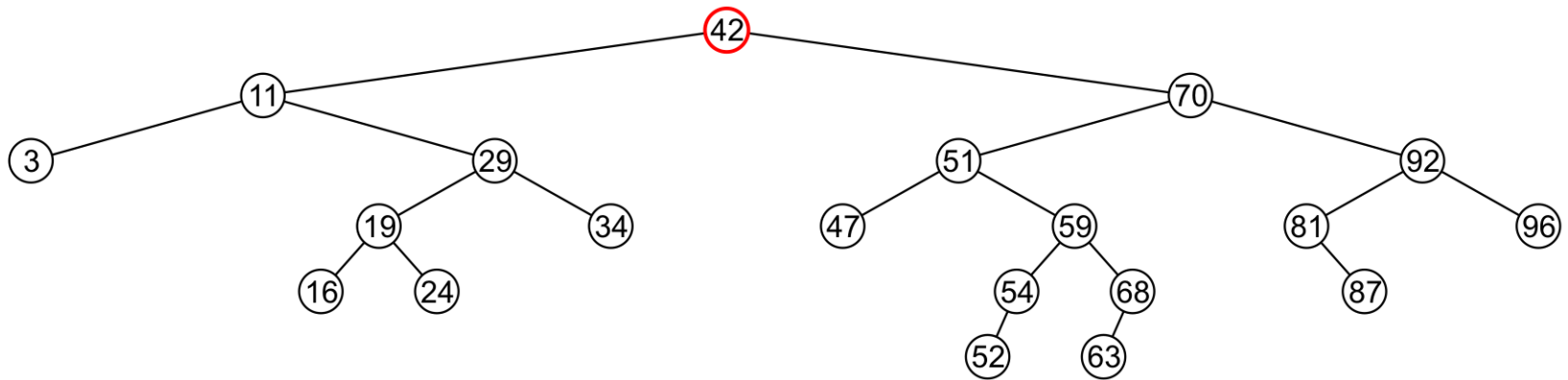


- A better Solution to delete node p with two children
 - Replace node p with the minimum object in the right subtree
 - Delete that object from the right subtree



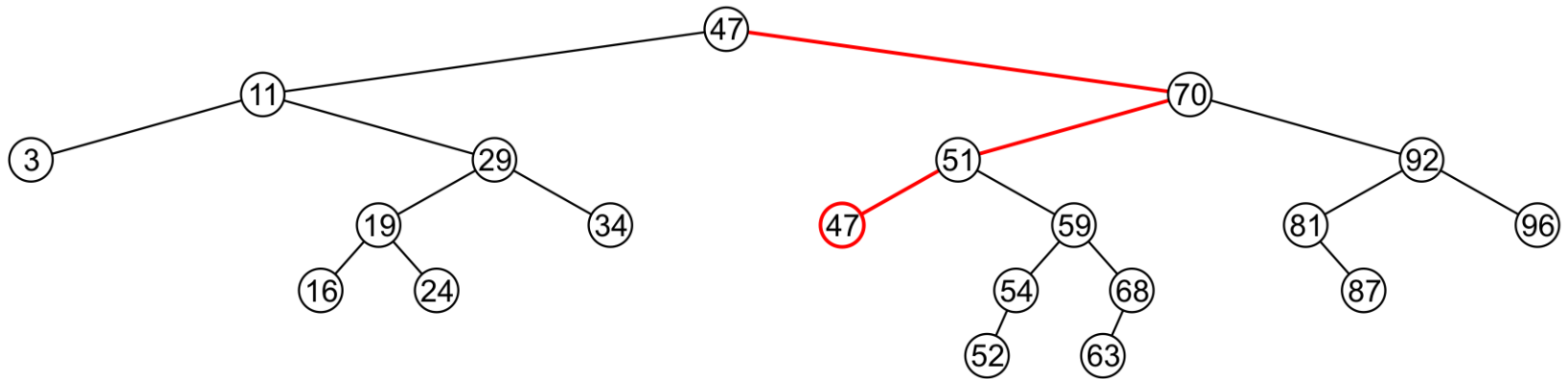
Deleting a Node Two Children – Better Solution

- Consider the problem of deleting a full node, e.g., 42



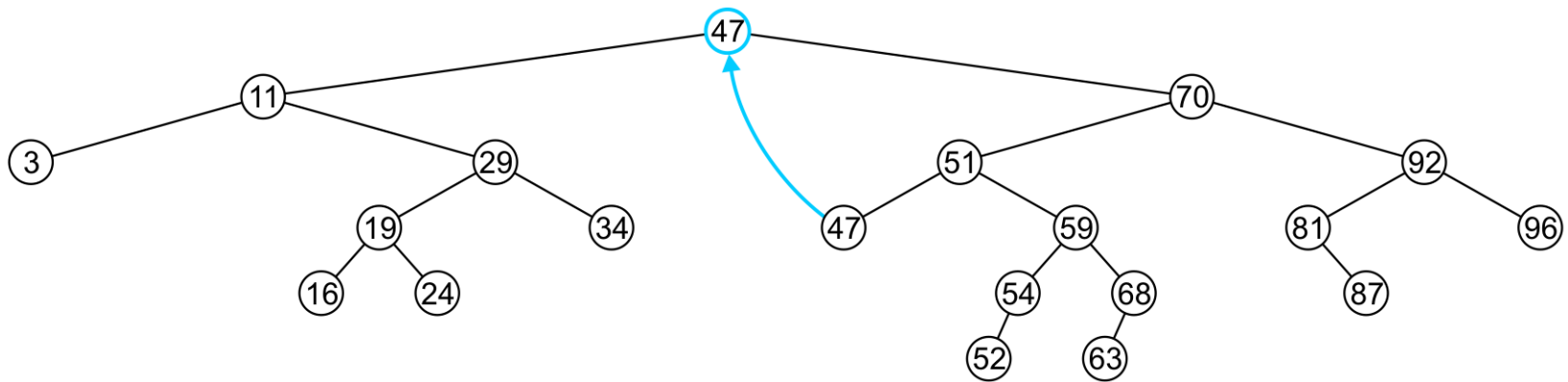
Deleting a Node Two Children – Better Solution

- Consider the problem of deleting a full node, e.g., 42
 - Find minimum object in the right subtree, i.e., 47



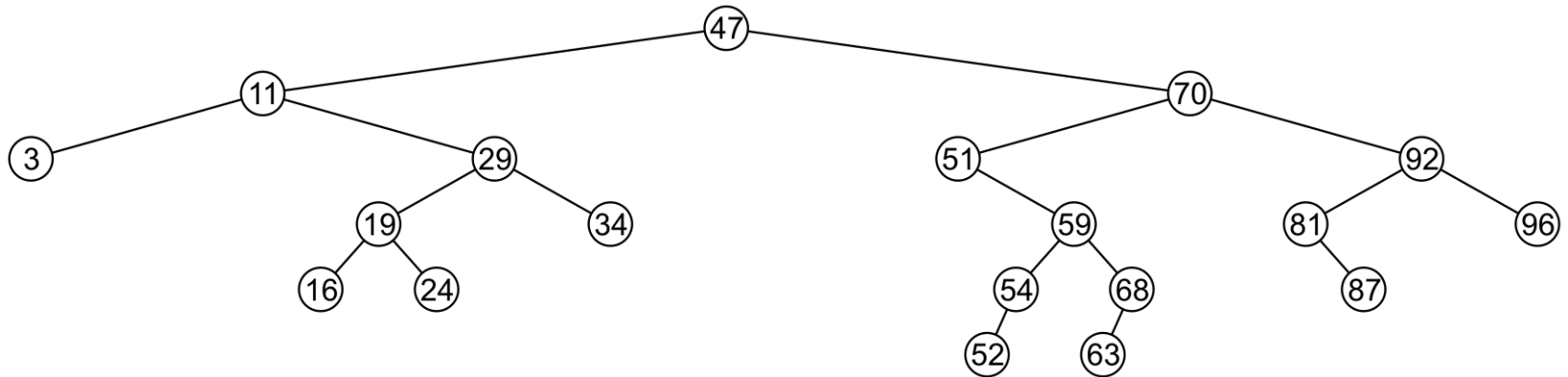
Deleting a Node Two Children – Better Solution

- Consider the problem of deleting a full node, e.g., 42
 - Find minimum object in the right subtree, i.e., 47
 - Replace 42 with 47



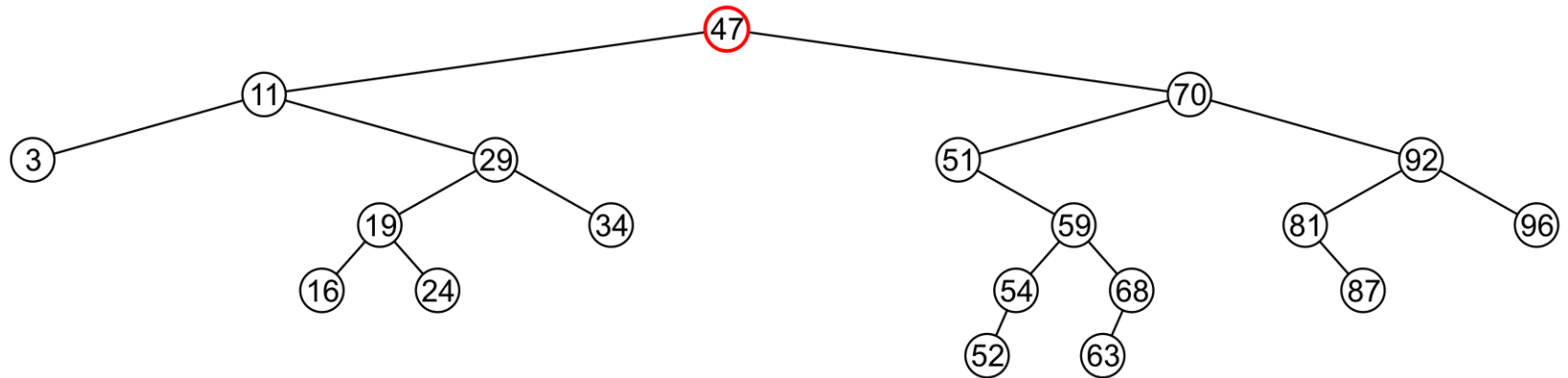
Deleting a Node Two Children – Better Solution

- Consider the problem of deleting a full node, e.g., 42
 - Find minimum object in the right subtree, i.e., 47
 - Replace 42 with 47
 - Delete the leaf node 47



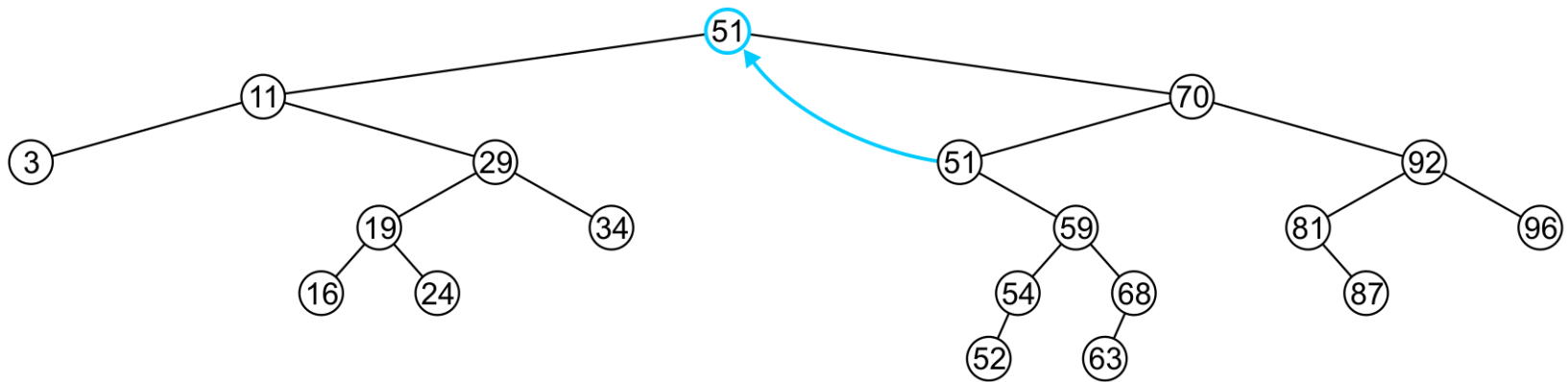
Deleting a Node Two Children – Better Solution

- Consider the problem of deleting a full node, e.g., 47



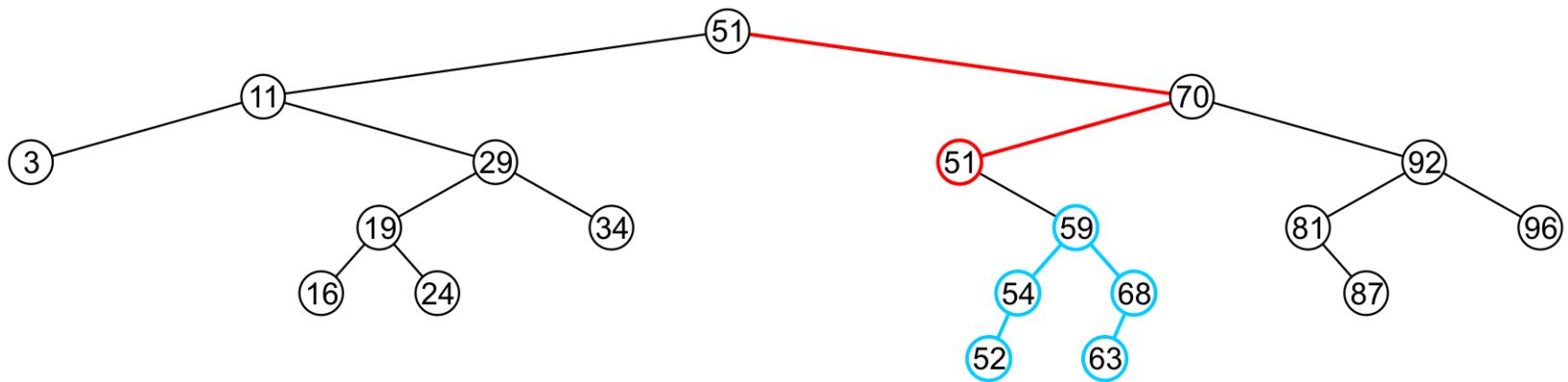
Deleting a Node Two Children – Better Solution

- Consider the problem of deleting a full node, e.g., 47
 - Replace 47 with 51



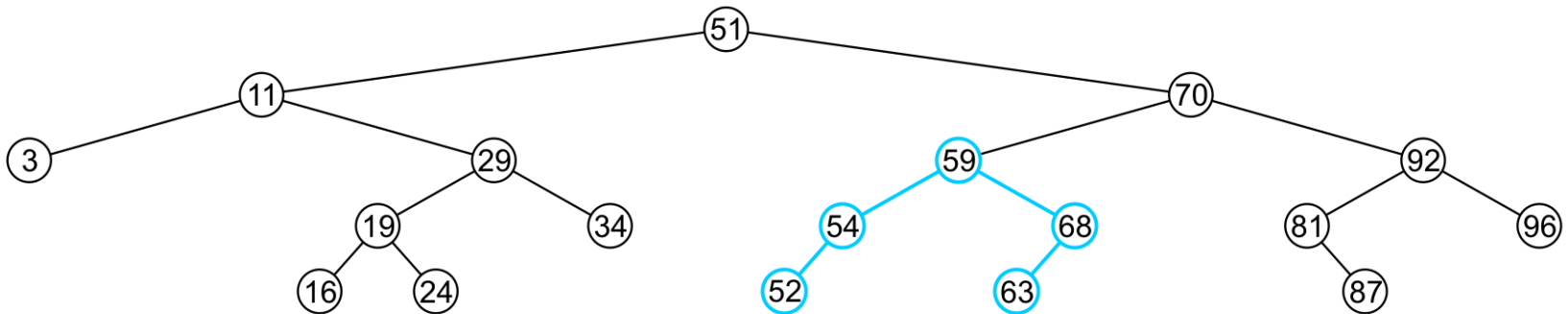
Deleting a Node Two Children – Better Solution

- Consider the problem of deleting a full node, e.g., 47
 - Replace 47 with 51
 - Node 51 is not a leaf node



Deleting a Node Two Children – Better Solution

- Consider the problem of deleting a full node, e.g., 47
 - Replace 47 with 51
 - Node 51 is not a leaf node
 - Assign the left subtree of 70 to point to 59



Using BST

```
// This program builds a binary tree with 5 nodes.
// The DeleteNode function is used to remove two of them.
#include <iostream.h>
#include "IntBinaryTree.h"

void main(void) {
    IntBinaryTree tree;

    cout << "Inserting nodes.\n";
    tree.insertNode(5);
    tree.insertNode(8);
    tree.insertNode(3);
    tree.insertNode(12);
    tree.insertNode(9);

    cout << "Here are the values in the tree:\n";
    tree.showNodesInOrder();
    cout << "Deleting 8...\n";
    tree.remove(8);
    cout << "Deleting 12...\n";
    tree.remove(12);

    cout << "Now, here are the nodes:\n";
    tree.showNodesInOrder();
}
```

Program Output:
Inserting nodes.
Here are the values in
the tree:
3
5
8
9
12

Using BST

```
// This program builds a binary tree with 5 nodes.
// The DeleteNode function is used to remove two of them.
#include <iostream.h>
#include "IntBinaryTree.h"

void main(void) {
    IntBinaryTree tree;

    cout << "Inserting nodes.\n";
    tree.insertNode(5);
    tree.insertNode(8);
    tree.insertNode(3);
    tree.insertNode(12);
    tree.insertNode(9);

    cout << "Here are the values in the tree:\n";
    tree.showNodesInOrder();
    cout << "Deleting 8...\n";
    tree.remove(8);
    cout << "Deleting 12...\n";
    tree.remove(12);

    cout << "Now, here are the nodes:\n";
    tree.showNodesInOrder();
}
```

Program Output:

Inserting nodes.
Here are the values in
the tree:

3
5
8
9
12

Deleting 8...

Deleting 12...

Now, here are the
nodes:

3
5
9

Traversing a Binary Search Tree

```
class IntBinaryTree {  
    private:  
        TreeNode *root; // Pointer to the root of BST  
  
        void destroySubTree(TreeNode *); //Recursively delete all tree nodes  
        void deleteNode(int, TreeNode *&);  
        void makeDeletion(TreeNode *&);  
        void displayInOrder(TreeNode *);  
        void displayPreOrder(TreeNode *);  
        void displayPostOrder(TreeNode *);  
  
    public:  
        IntBinaryTree()                { root = NULL; }  
        ~IntBinaryTree()               { destroySubTree(root); }  
        void insertNode(int);  
        bool find(int);  
        void remove(int);               { deleteNode( num, root); }  
        void showNodesInOrder()         { displayInOrder(root); }  
        void showNodesPreOrder()        { displayPreOrder(root); }  
        void showNodesPostOrder()       { displayPostOrder(root); }  
};
```

→ Recursive implementation as discussed in the slides of Tree Traversal chapter.

Using BST

```
// This program builds a binary tree with 5 nodes.
// The nodes are displayed with inorder, preorder, and postorder
#include <iostream.h>
#include "IntBinaryTree.h"

void main(void)
{
    IntBinaryTree tree;

    cout << "Inserting nodes.\n";
    tree.insertNode(5);
    tree.insertNode(8);
    tree.insertNode(3);
    tree.insertNode(12);
    tree.insertNode(9);

    cout << "Inorder traversal:\n";
    tree.showNodesInOrder();
    cout << "\nPreorder traversal:\n";
    tree.showNodesPreOrder();
    cout << "\nPostorder traversal:\n";
    tree.showNodesPostOrder();
}
```

Program output:

Inserting nodes.

Inorder traversal:

3
5
8
9
12

Preorder traversal:

5
3
8
12
9

Postorder traversal:

3
9
12
8
5

Any Question So Far?

