CS-2001
Data Structures
Fall 2023

Tree

Rizwan Ul Haq

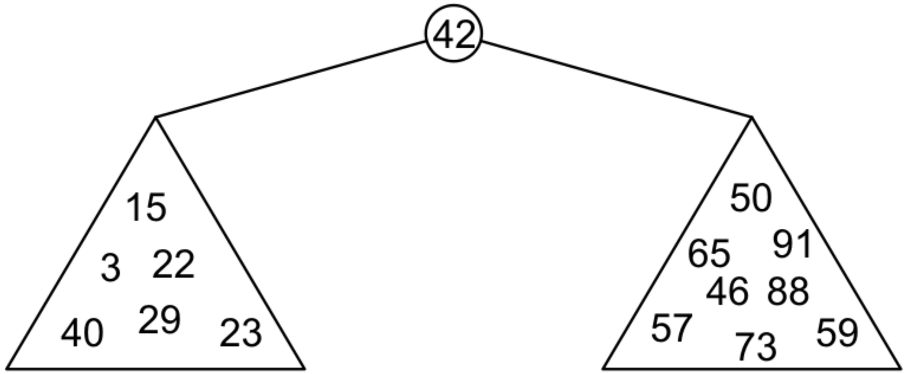
Assistant Professor

FAST-NU

rizwan.haq@nu.edu.pk

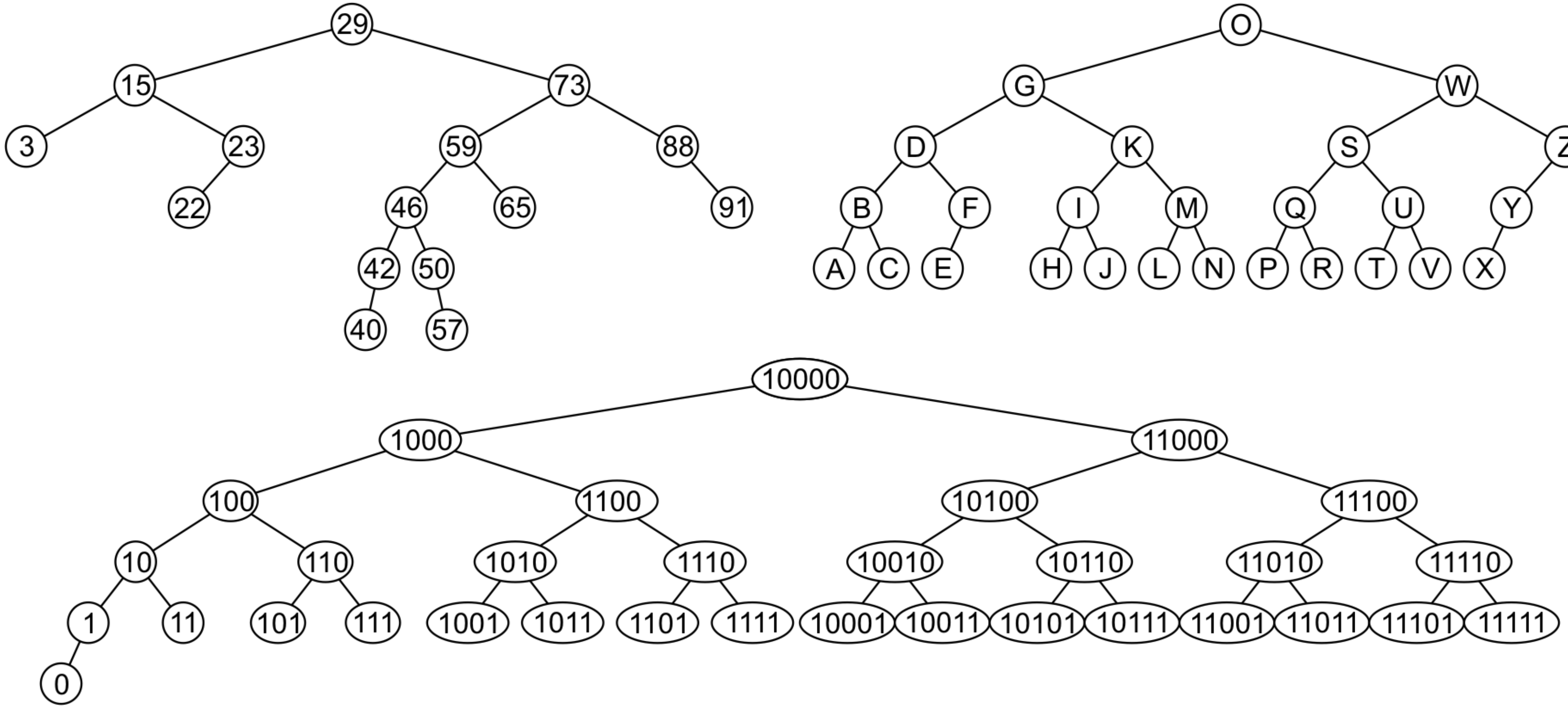
Binary Search Tree (BST)

- 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

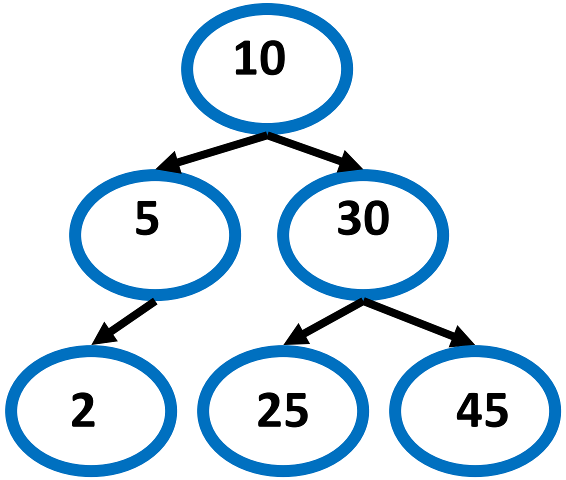


subtrees will themselves be binary search trees

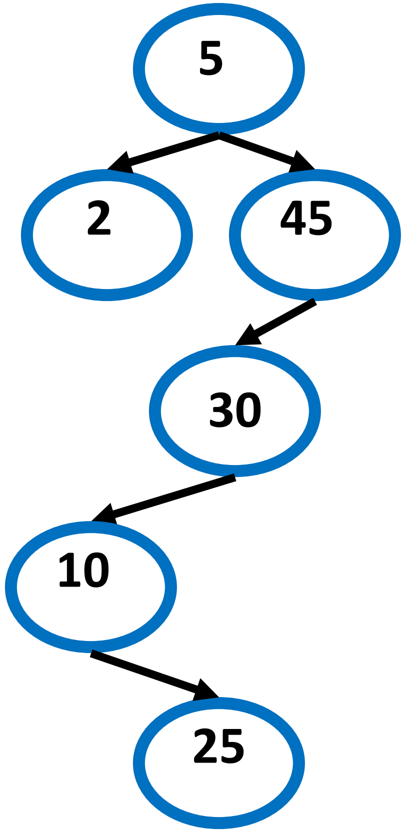
Binary Search Tree (BST) – Example (1)



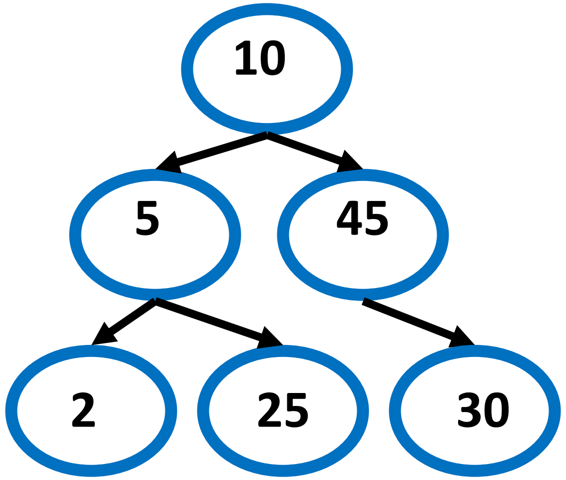
Binary Search Tree (BST) – Example (2)



BST



BST



Not BST

BST 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 search(int);
      void remove(int);
      void showNodesInOrder() { displayInOrder(root); }
      void showNodesPreOrder() { displayPreOrder(root); }
      void showNodesPostOrder() { displayPostOrder(root); }
};
```

Basic Structure of Binary Search Tree

- The root pointer is the pointer to the binary search tree. This is similar to the head pointer in a linked list

- The root pointer will point to the first node in the tree, or to NULL (if the tree is empty)

- It is initialized in the constructor

- The destructor calls *destroySubTree*, a private member function, that recursively deletes all the nodes in the tree

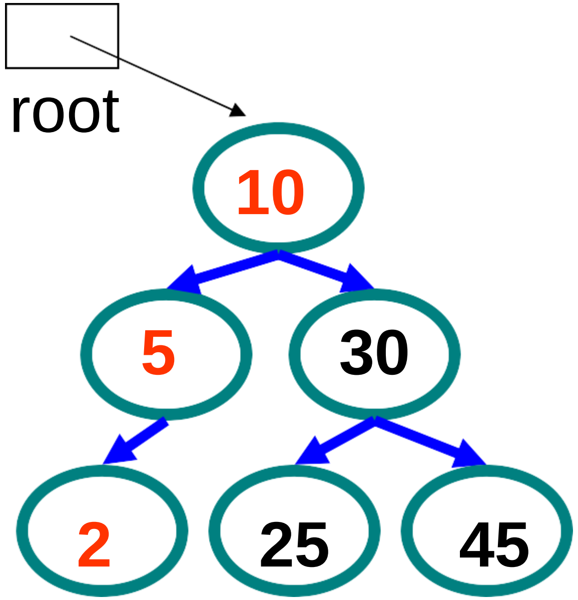
Finding a node in a binary search tree

- Recall that a BST has the following key property (invariant):
  - Smaller values in left subtree
  - Larger values in right subtree

- For searching for a node, make use of this property

Example Binary Searches (search)

- search(2)



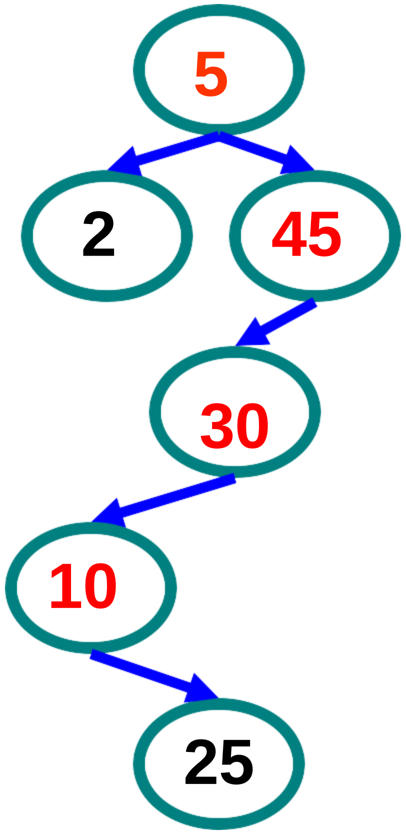
10 > 2, left

5 > 2, left

2 = 2, found

Example Binary Searches (search)

- seach(10)



10 > 5, right

10 < 45, left

10 < 30, left

10 = 10, found

Finding a node - Implementation

- The IntBinaryTree class has a public member function called search, that returns **true** if a value is found in the tree, or **false** otherwise.
- The function starts at the **root** node, and **traverses** the tree, until it **finds** the search value, or runs out of nodes

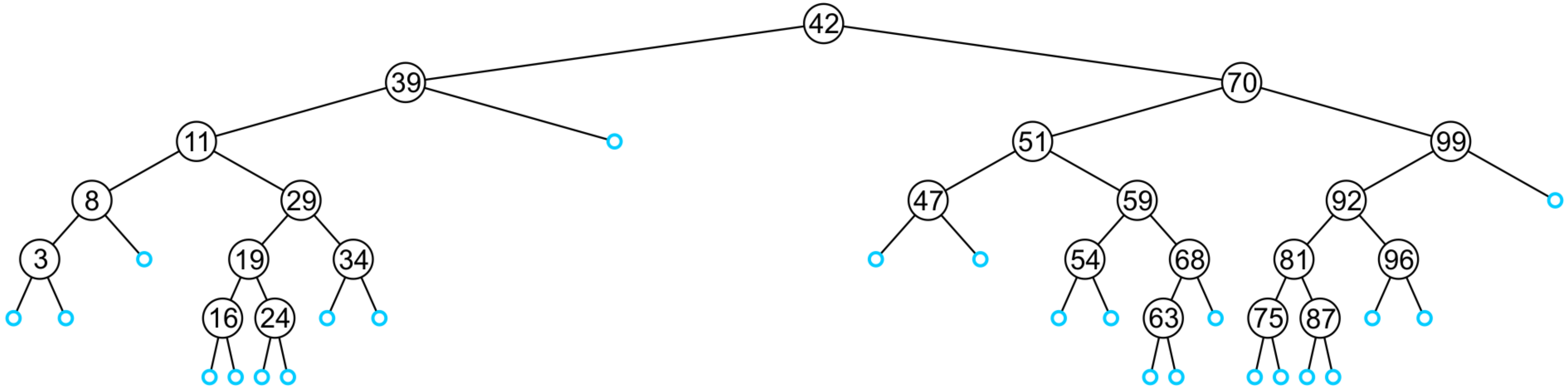
```
bool IntBinaryTree::searchNode(int num)
{
    TreeNode *nodePtr = root;
    while (nodePtr!=NULL)
    {
        if (nodePtr->value == num)
            return true;
        else if (num < nodePtr->value)
            nodePtr = nodePtr->left;
        else
            nodePtr = nodePtr->right;
    }
    return false;
}
```

Recursive Search of Binary Tree

```
Node *Find( Node *n, int key) {
   if (n == NULL)// Not found
      return( n );
   else if (n->data == key) //Found it
      return( n );
   else if (n->data > key)// In left subtree
      return Find( n->left, key );
   else// In right subtree
      return Find( n->right, key );
}

//Function call
Node * n = Find( root, 5);
```

Inserting a Node in BST

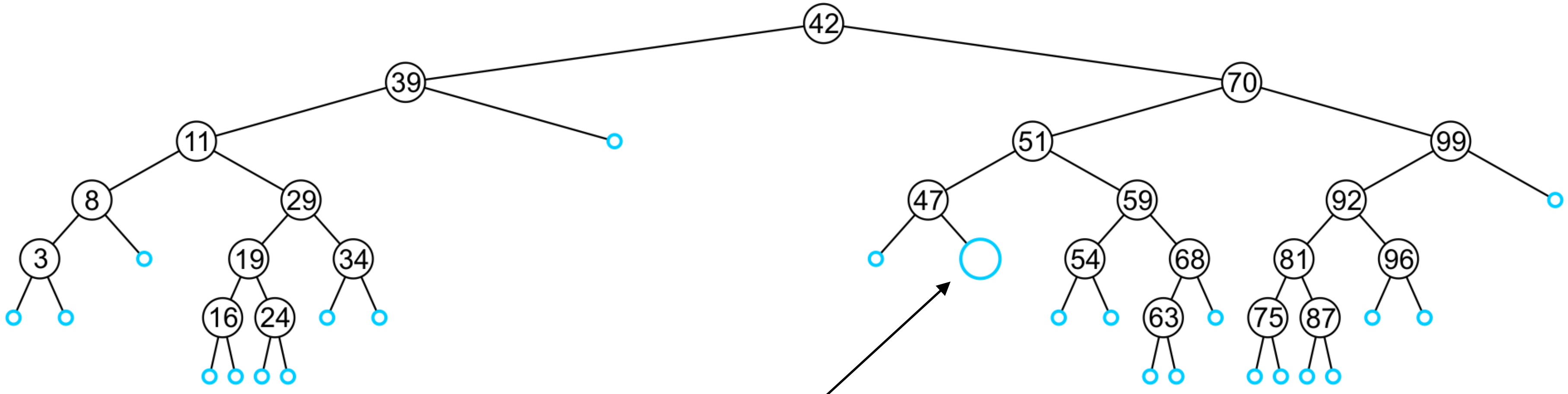
- An insertion will be performed at a leaf node
    - Any empty node is a possible location for an insertion



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

Inserting a Node in BST

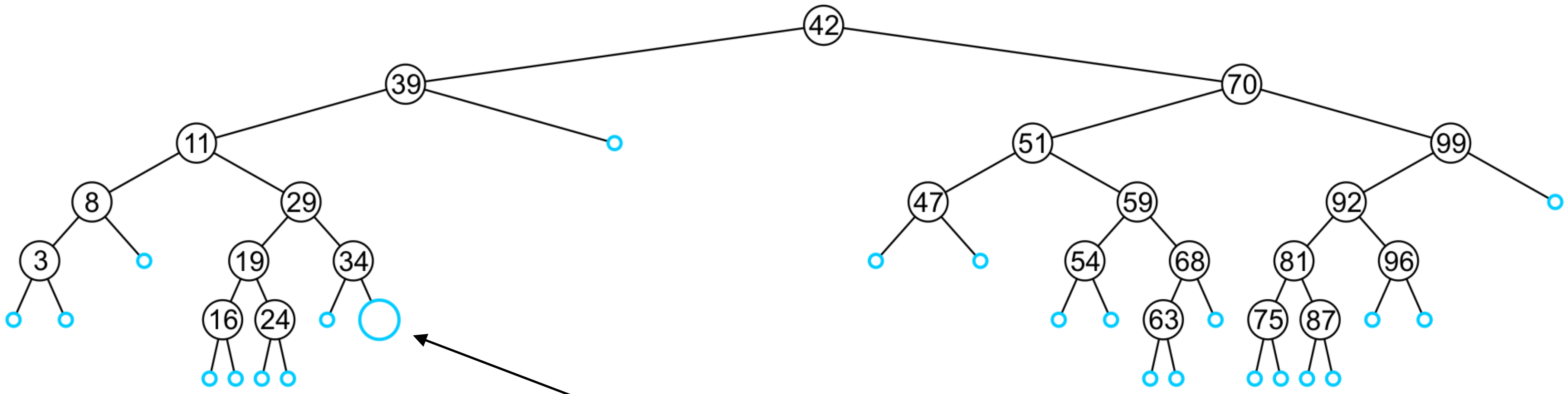
- Which values can be held by empty node?



This node may hold
48, 49, or 50

Inserting a Node in BST

- Which values can be held by empty node?



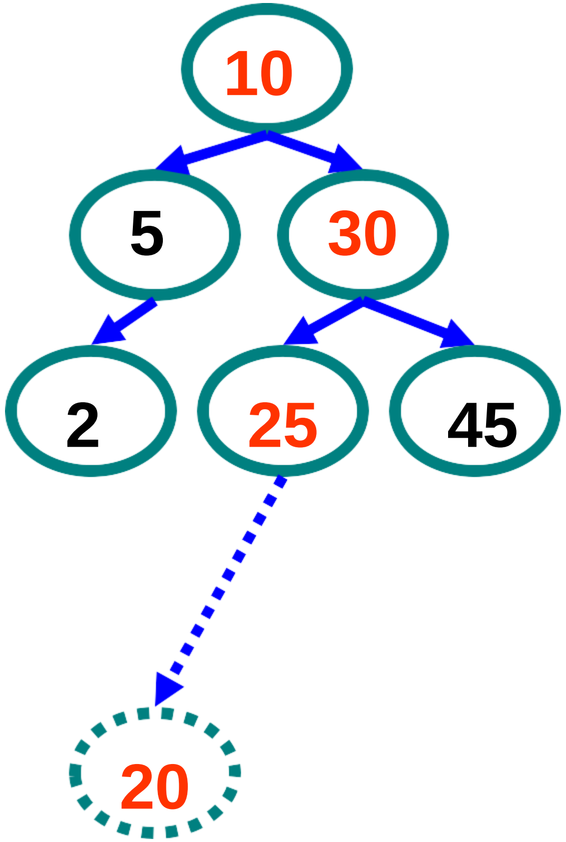
This node may hold 35, 36, 37 or 38

Binary Search Tree – Insertion

- **Algorithm**
  1. Perform search for value X
  2. Search will end at node Y **(if X not in tree as no duplicates allowed)**
  3. If X < Y, insert new leaf X as new left subtree for Y
  4. If X > Y, insert new leaf X as new right subtree for Y
- Observations
  - Insertions may unbalance tree

Example Insertion

- insert (20)



10 < 20, right
30 > 20, left
25 > 20, left

Insert 20 on left

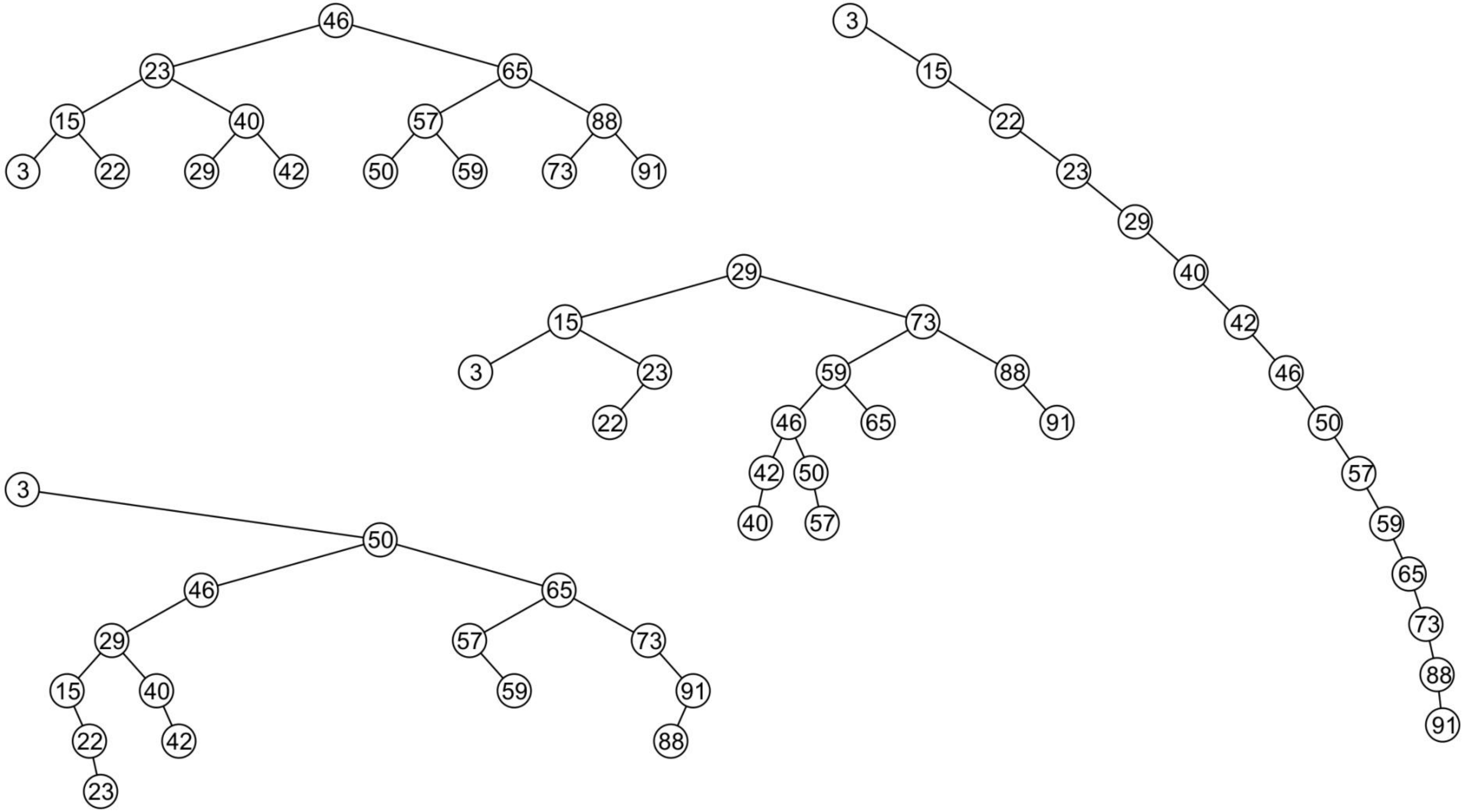
Insertion implementation

Note, we assume that our binary tree will store no duplicate values.

```
void IntBinaryTree::insertNode(int num)
{
      TreeNode *newNode, // Pointer to a new node
            *nodePtr; // Pointer to traverse the tree
      // Create a new node
      newNode = new TreeNode {value, NULL, NULL};
      if (!root)// Is the tree empty?
            root = newNode;
      else{
            nodePtr = root;
            while (nodePtr != NULL
            {
               if (num < nodePtr->value){
                  if (nodePtr->left)
                        nodePtr = nodePtr->left;
                  else{
                        nodePtr->left = newNode;
                        break ;
                  }//using break is not a good way to terminate think of alternative method
            }
            else if (num > nodePtr->value){
               if (nodePtr->right)
                  nodePtr = nodePtr->right;
               else{
                  nodePtr->right = newNode;
                  break ;
               }
            }
            else{
               cout << "Duplicate value found in tree.\n";
               break;
            }//end of else
         }//end of while loop
      }//end of else corresponding to empty check
}//end of function
```

Insertion in BST: An Observation

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



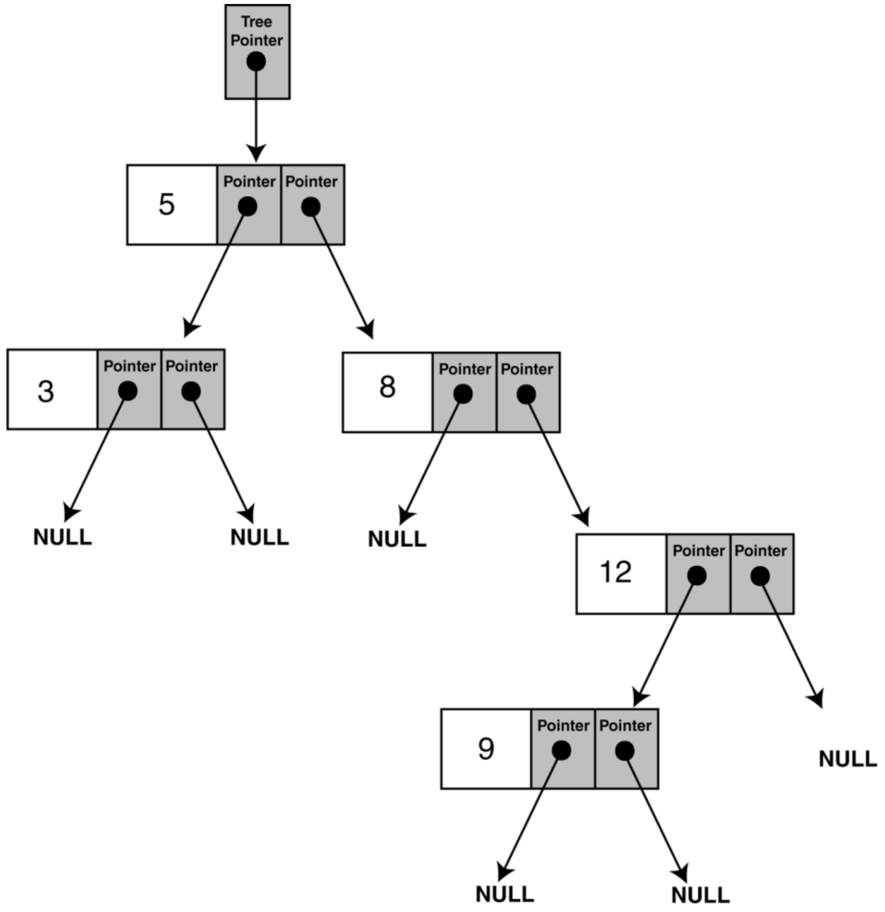
Using a BST

```
// This program builds a binary tree with 5 nodes.

#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.

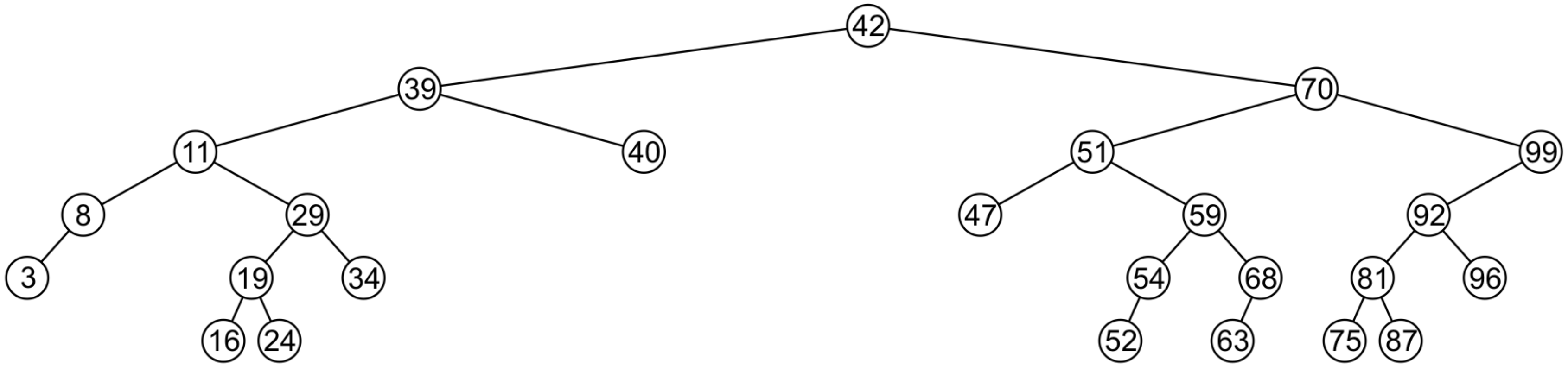
Using a BST

- 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

- Removes a specified item from the BST and adjusts the tree.
- Uses a binary search to locate the target item:
    - Starting at the root it probes down the tree till it finds the target or reaches a leaf node (target not in the tree)
    - Removal of a node must not leave a ‘gap’ in the tree

Removal in BST - Pseudocode

- method remove (key)
- if the tree is empty return false
- Attempt to locate the node containing the target using the binary search algorithm
- **if** the target is not found return false

  else the target is found, so remove its node:

    Case 1: if the node has 2 empty subtrees replace the link in the parent with null

    Case 2: if the node has no left child link the parent of the node to the right (non-empty) subtree

    Case 3: if the node has no right child link the parent of the target to the left (non-empty) subtree

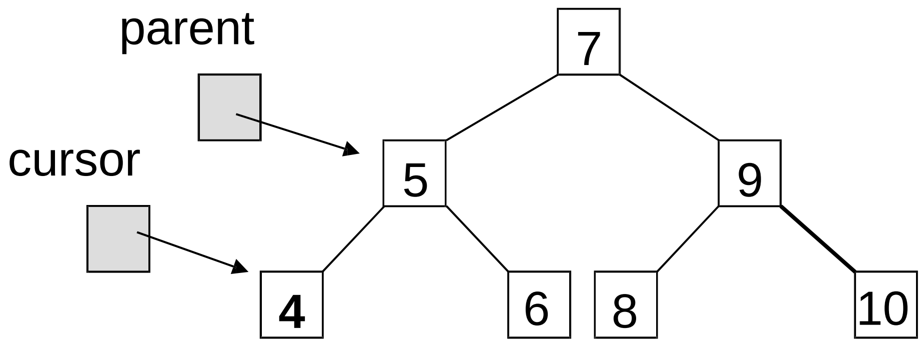
    Case 4: if the node has a left and a right subtree replace the node’s value with the max value in the left subtree delete the max node in the left subtree

    OR

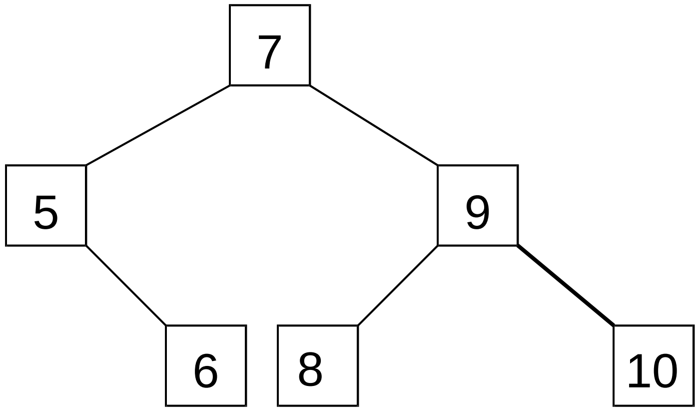
    Find a position in the right subtree of p to attach its left subtree. Left most node in the right subtree of node p (successor of p). Attach the right subtree of node p to its parent

Removal in BST: Example

- Case 1: removing a node with 2 EMPTY SUBTREES

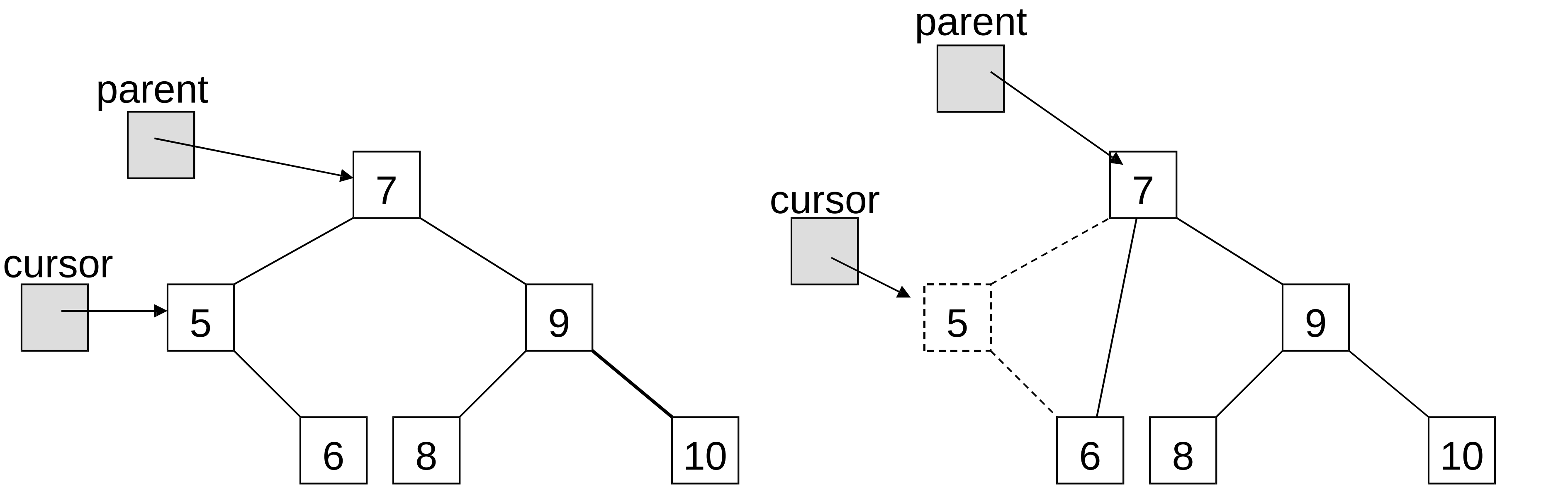


Removing 4
replace the link in the parent with `null`



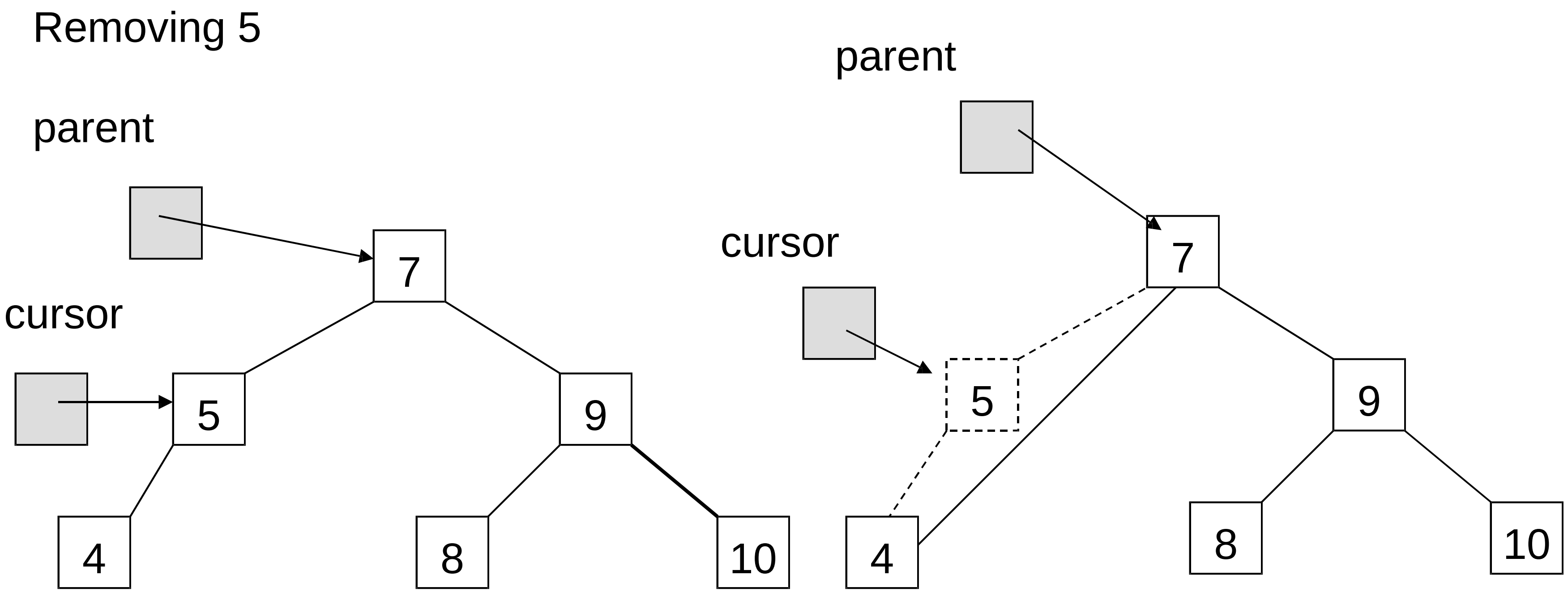
Removal in BST: Example

- **Case 2:** removing a node with 1 EMPTY SUBTREE the node has no left child: link the parent of the node to the right (non-empty) subtree



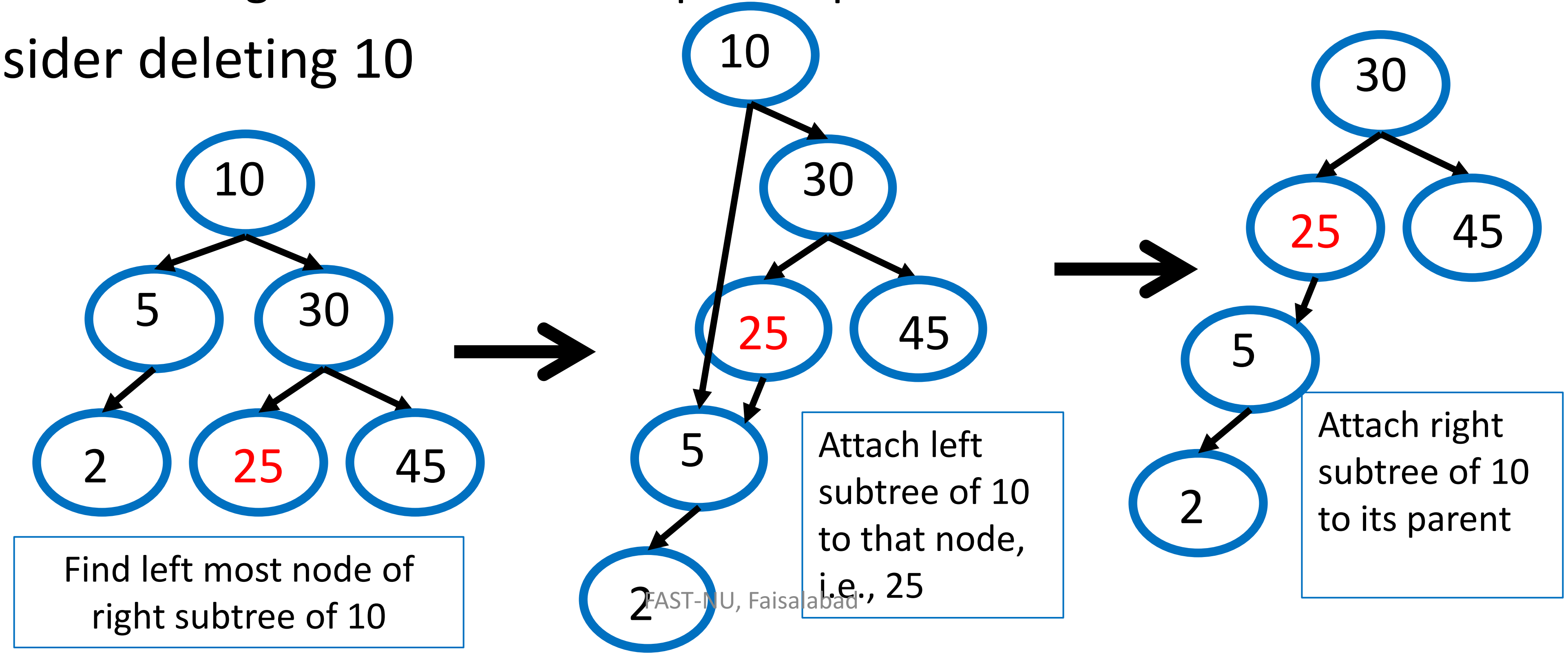
Removal in BST: Example

- **Case 3:** removing a node with 1 EMPTY SUBTREE the node has no right child: link the parent of the node to the left (non-empty) subtree



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 (immediate successor of p)
  - Attach the right subtree of node p to its parent
- Consider deleting 10



Pointers Review

```cpp
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;
}
```

```cpp
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:

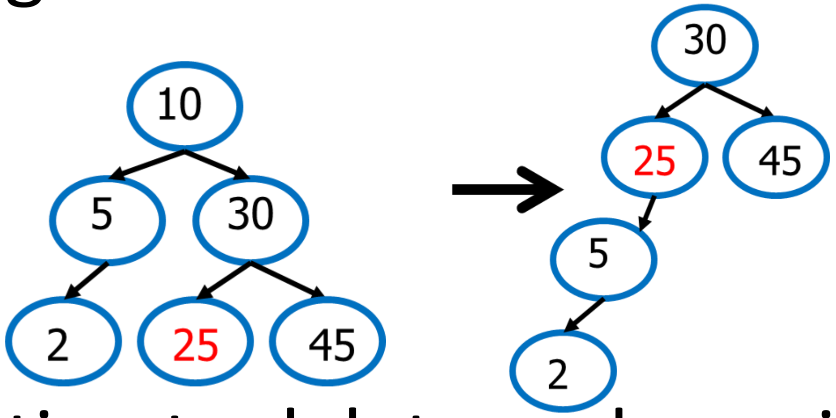
- The declaration of the `nodePtr` parameter: `TreeNode *&nodePtr;`
- `nodePtr` is a reference to a pointer to a `TreeNode` structure
    - Any action performed on `nodePtr` is actually performed on the argument passed into `nodePtr`

Deleting a Node – Implementation

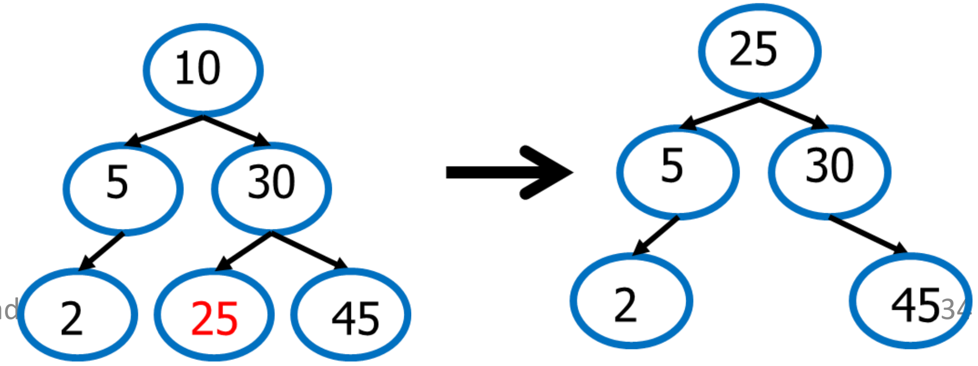
```
void IntBinaryTree::makeDeletion(TreeNode *&nodePtr) {
   TreeNode *tempNodePtr; // Temporary pointer
   if (nodePtr->right == NULL) { // case for leaf and one (left) child
      tempNodePtr = nodePtr;
      nodePtr = nodePtr->left; // Reattach the left child
      delete tempNodePtr;
   }
   else 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

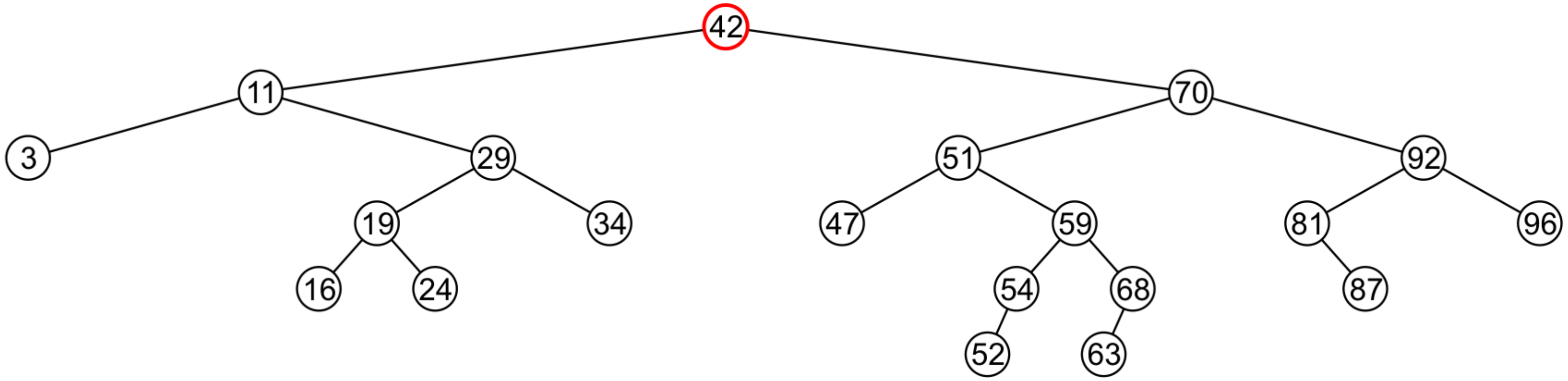


- A better Solution to delete node p with two children
  - Replace node p with the minimum object in the right subtree
  - Delete that object from the right subtree



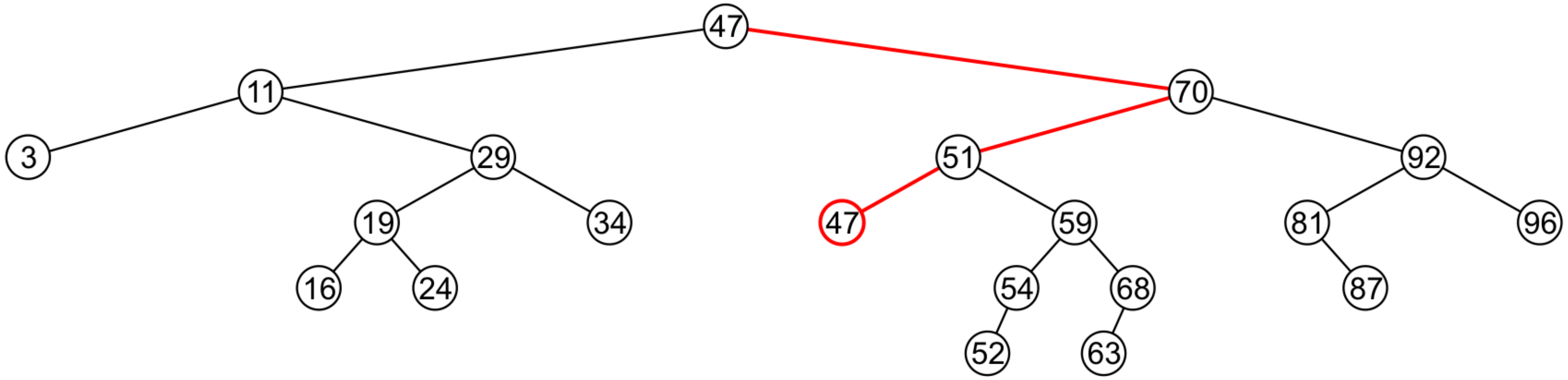
Deleting a Node Two Children – Better Solution

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



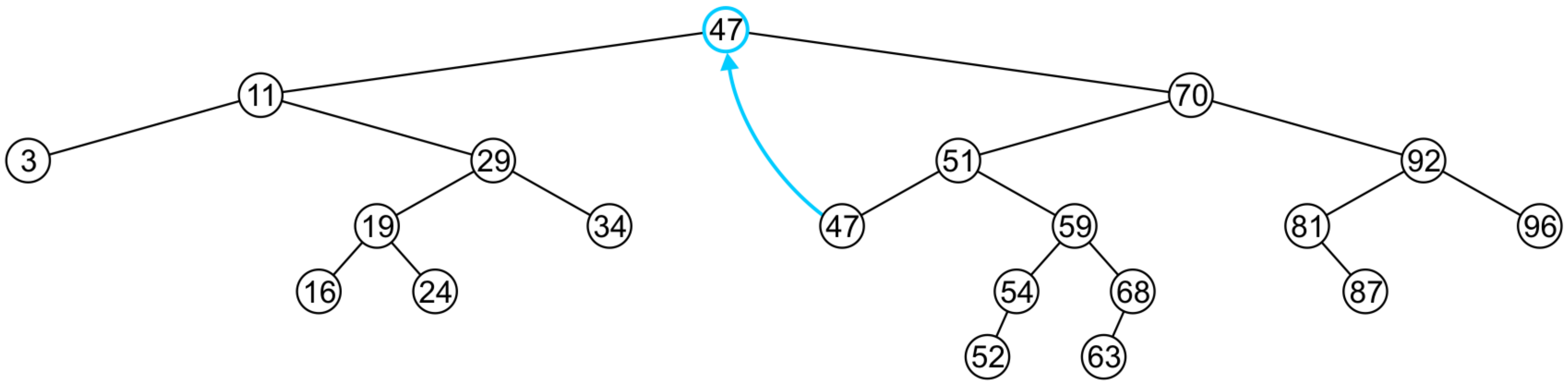
Deleting a Node Two Children – Better Solution

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



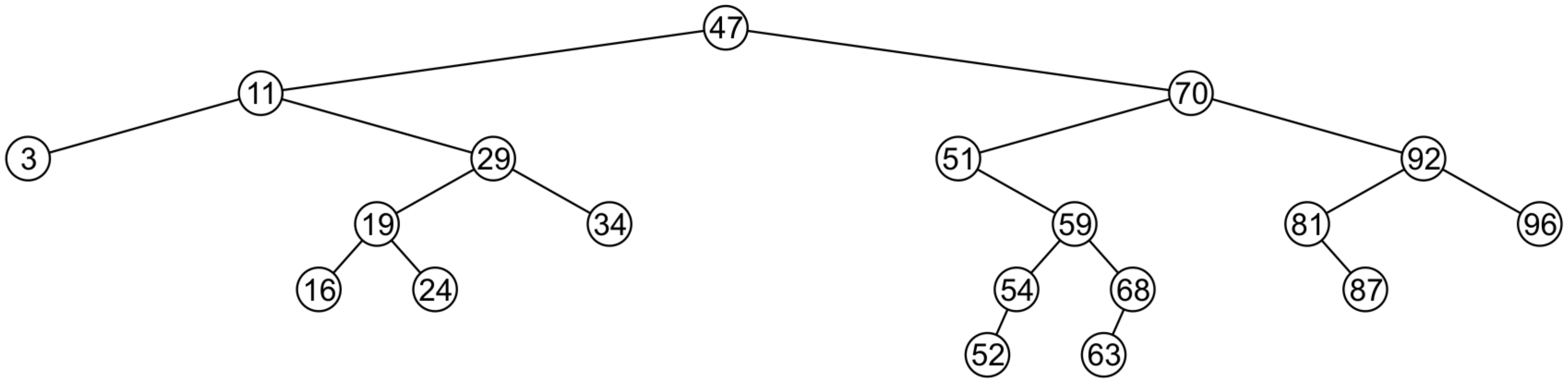
Deleting a Node Two Children – Better Solution

- 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



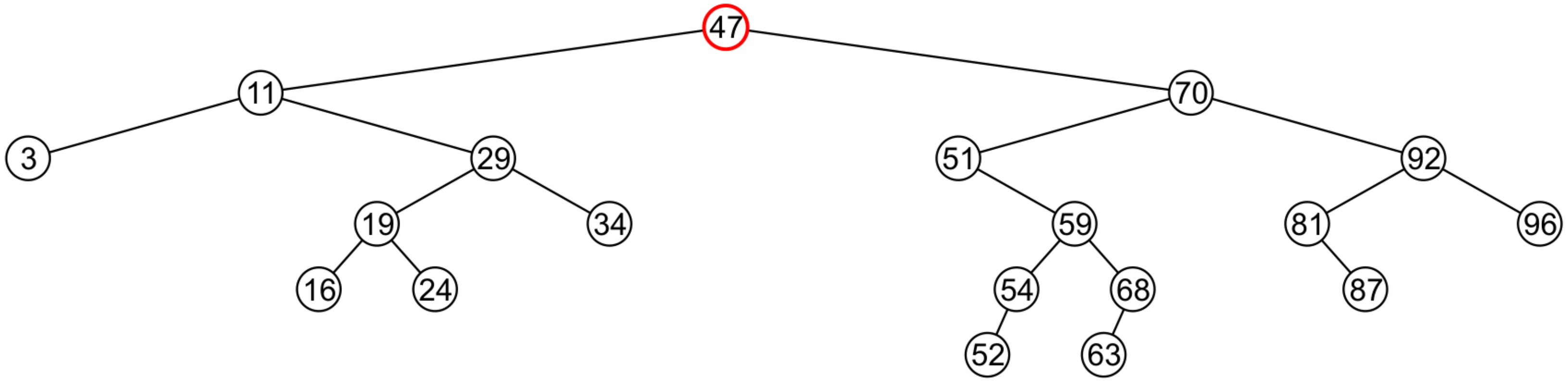
Deleting a Node Two Children – Better Solution

- 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



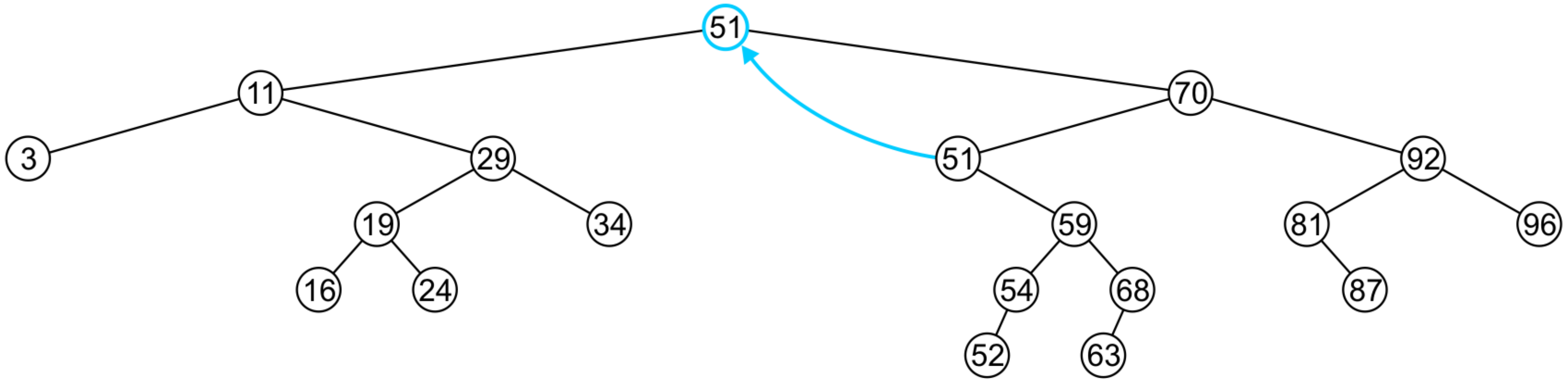
Deleting a Node Two Children – Better Solution

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



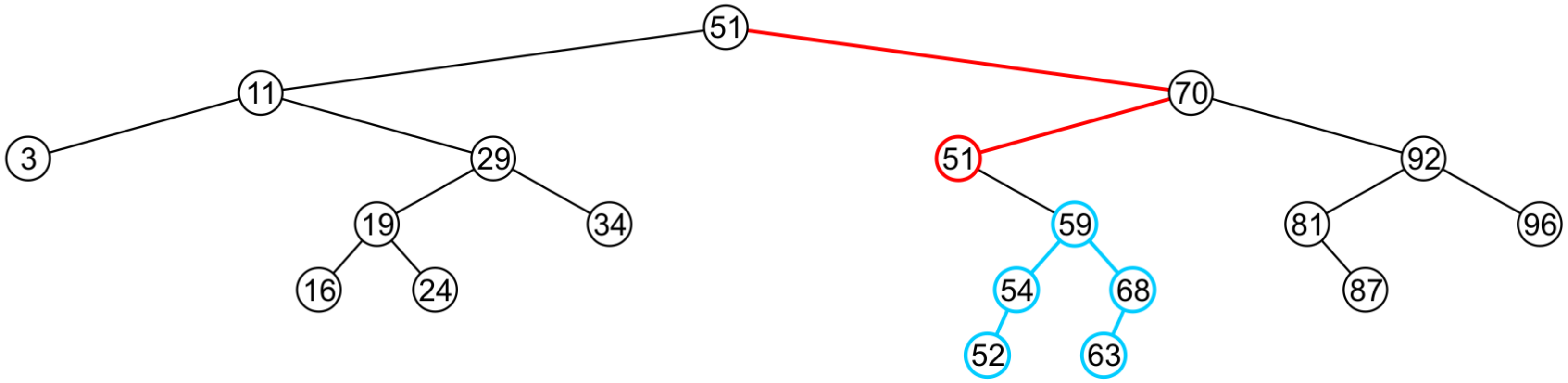
Deleting a Node Two Children – Better Solution

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



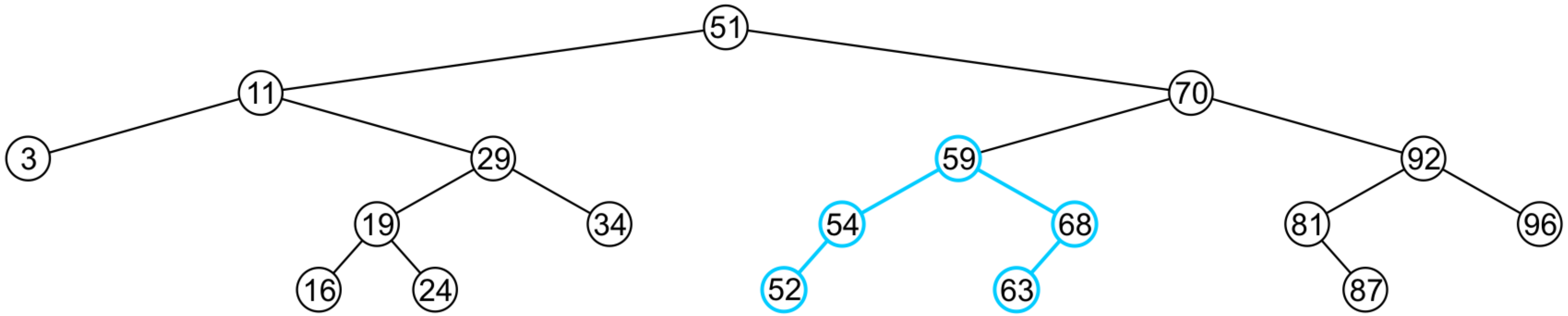
Deleting a Node Two Children – Better Solution

- 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



Removal in BST – A better approach

```
void IntBinaryTree::deleteNode(int val, Node*& node)
{
  if (node == NULL)
    return ;
  else if (val < node->value){
    deleteNode(val, node->left);
  }
  else {
    deleteNode(val, node->right);
  }
  else{
    if (val == node->data) {
      if (node->left == nullptr) {
       //This will execute if the node
       //has only right child or no child
        Node* temp = node;
        node = node->right;
        delete temp;
      }
      else if (node->right == nullptr)
      //This will execute if the
      {//node has only left child
        Node* temp = node;
        node = node->left;
        delete temp;
      }
      else  //This will execute if the node has
          //two children
      {
        Node* temp = findMaxFromLeft(node->left);
        int data = temp->data;
        deleteNode(temp->data, root);
        node->data = data;
      }
    }
  }
}
```

Another implementation of BST deletion

```
Node* IntBinaryTree::findMaxFromLeft(Node* node)
{
    while (node && node->right != nullptr)
      node = node->right;
    return node;
}
void IntBinaryTree::remove(int val)
{
    root = deleteNode(val, root);
}
```

Another implementation of BST deletion

```
Node* IntBinaryTree::deleteNode(int val, Node*
node){
  if (node == NULL)
    return node;
  else
  {
    if (val == node->data) {
        //This will execute if the node
        //has only right child or no child
      if (node->left == nullptr) {
        Node* temp = node->right;
        delete node;
        return temp;
      }
      else if (node->right == nullptr) {
            //This will execute if the
            //node has only left child
        Node* temp = node->left;
        delete node;
        return temp;
      }
      else {
        //This will execute if the
        //node has two children

        Node* temp = findMaxFromLeft(node->left);
        node->data = temp->data;
        node->left = deleteNode(temp->data, node->left);
      }
    }
    else if (val < node->value){
      node->left =
        deleteNode(val, node->left);
    }
    else{
      node->right =
        deleteNode(val, node->right);
    }
  }
  return node;
}
```

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();
}
```

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);
      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 algorithms.
#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

Binary Tree - Terminologies

- Similar vs. copy of a tree
- Full/strictly binary tree
- Complete tree
- Issues with BST

Applications of Binary Tree

- File system (e.g. UNIX, DOS, etc.)
- Expression trees
- Leaves represents operands, operators are at non- leaf nodes

Example:

- Each traversal order yields respective expression

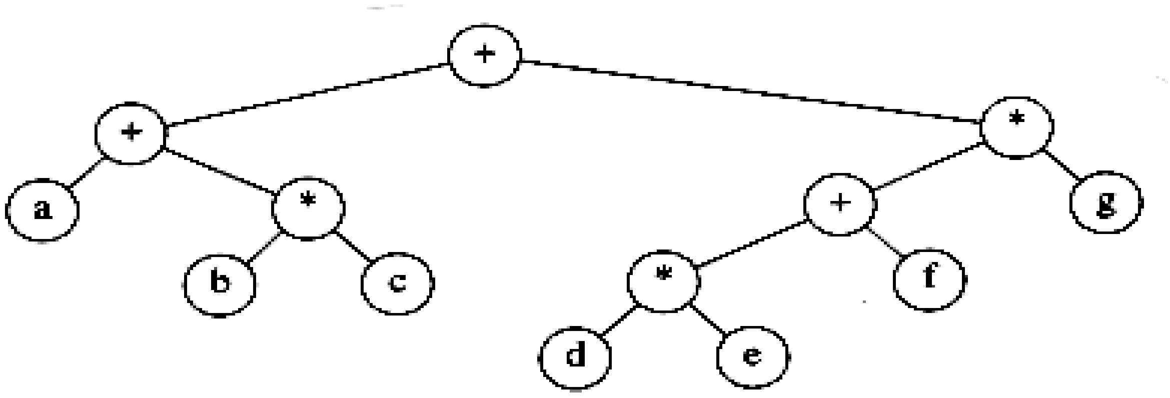


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Constructing an expression tree

- Algorithm: (build tree from postfix expression)
- Example
  1. Read postfix expression, one symbol at a time
  2. If the symbol is an operand, create one-node tree and push a pointer to it onto stack.
  3. If the symbol is an operator, pop two tree pointers T1 (popped first) and T2 from stack, form a new tree with operator as a root and T2 & T1 (left & right respectively) as its children.
  4. Push the new tree pointer on the stack