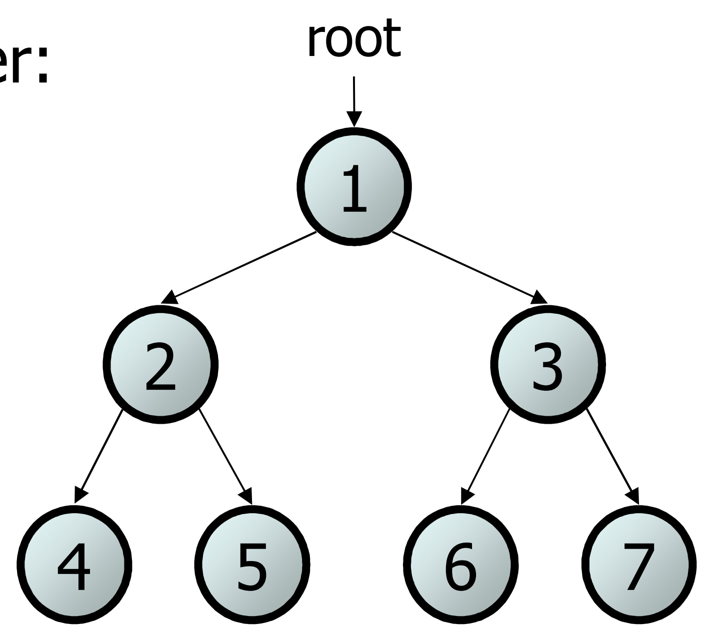
Binary Trees

Trees

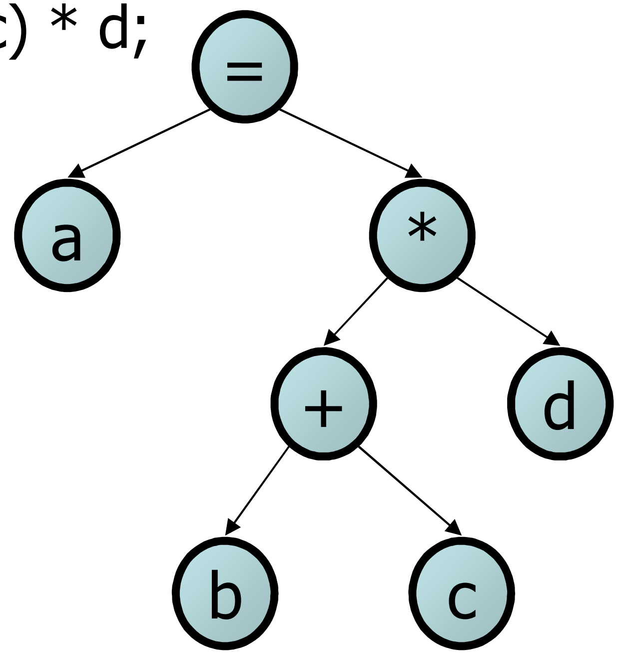
- **tree**: A directed, acyclic structure of linked nodes.
  - *directed* : Has one-way links between nodes.
  - *acyclic* : No path wraps back around to the same node twice.
  - binary tree: One where each node has at most two children.

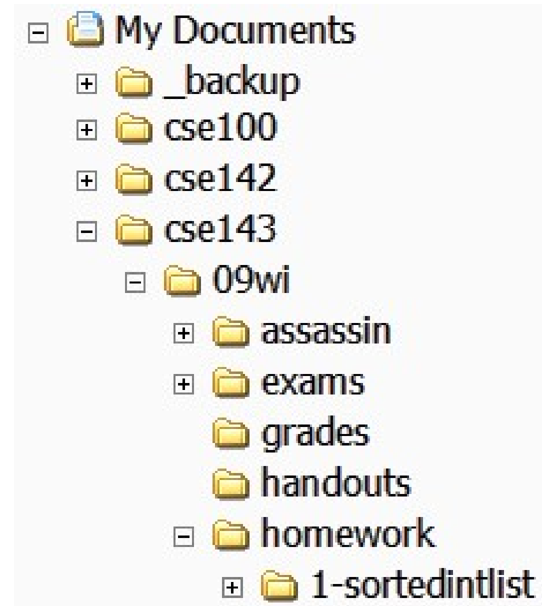
- A **binary tree** can be defined as either:
  - empty (`null`), or
  - a **root** node that contains:
    - **data**,
    - a **left** subtree, and
    - a **right** subtree.
      - (The left and/or right subtree could be empty.)



Trees in computer science

- folders/files on a computer
- family genealogy; organizational charts
- AI: decision trees
- compilers: parse tree
  - a = (b + c) * d;





Programming with trees

- Trees are a mixture of linked lists and recursion
  - considered very elegant (perhaps beautiful!) by Ceng nerds
  - difficult for novices to master

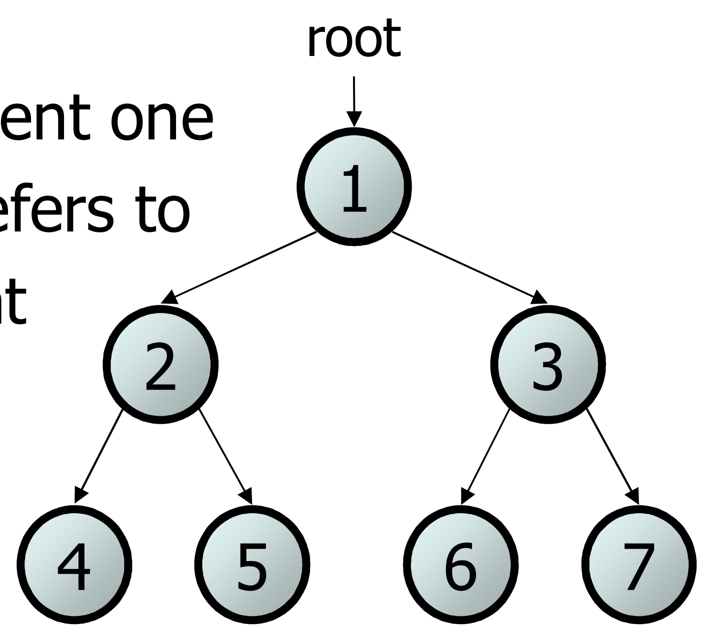
- Common student remark #1:
  - "My code doesn't work, and I don't know why."

- Common student remark #2:
  - "My code works, and I don't know why."

Terminology

- **node**: an object containing a data value and left/right children
- **root**: topmost node of a tree
- **leaf**: a node that has no children
- **branch**: any internal node; neither the root nor a leaf

- **parent**: a node that refers to the current one
- **child**: a node that the current node refers to
- **sibling**: a node with a common parent



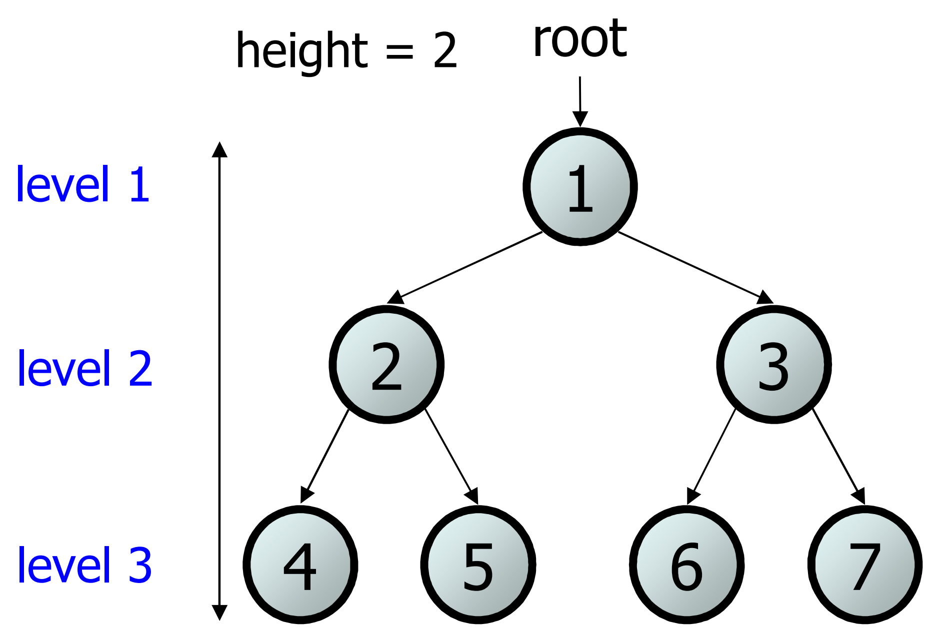
Terminology (cont.)

- **subtree**: the tree of nodes reachable to the left/right from the current node

- **height**: length of the longest path to a leaf from the given node

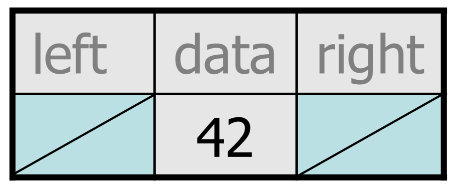
- **depth**: length of the path from the root to a given node

- **full tree**: one where every branch has 2 children

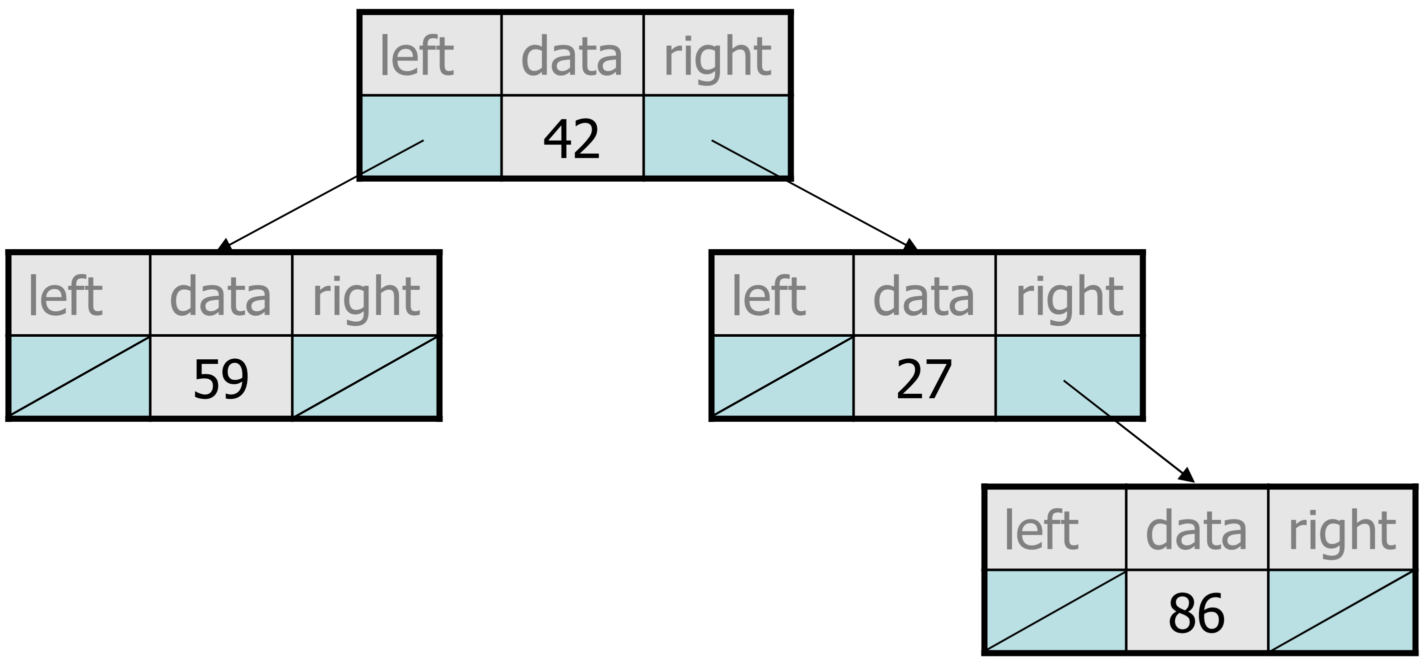


A tree node for integers

- A basic **tree node object** stores data and refers to left/right



- Multiple nodes can be linked together into a larger tree



IntTreeNode class

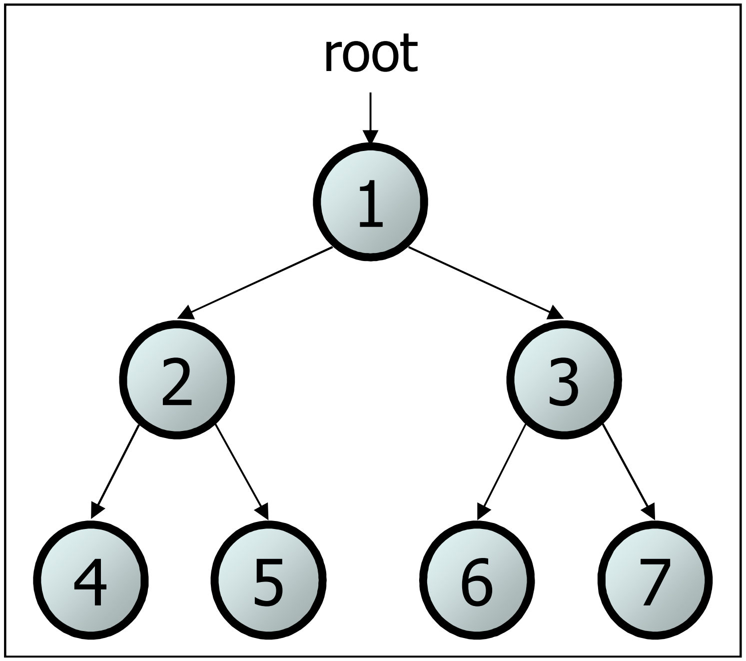
```
// An IntTreeNode object is one node in a binary tree of ints.
class IntTreeNode {
    public:
        int data;                 // data stored at this node
        IntTreeNode *left;        // reference to left subtree
        IntTreeNode *right;       // reference to right subtree

    // Constructs a leaf node with the given data.
    IntTreeNode(int val) {
        data = val;
        left = nullptr;
        right = nullptr;
    }

    // Constructs a branch node with the given data and links.
    IntTreeNode(int val, IntTreeNode *l, IntTreeNode *r) {
        data = val;
        left = l;
        right = r;
    }
}
```

IntTree class

```
// An IntTree object represents an entire binary tree of ints.
class IntTree {
    private:
        IntTreeNode *root;    // null for an empty tree

    public:
        methods
}
```

- Client code talks to the `IntTree`, not to the node objects inside it

- Methods of the `IntTree` create and manipulate the nodes, their data and links between them



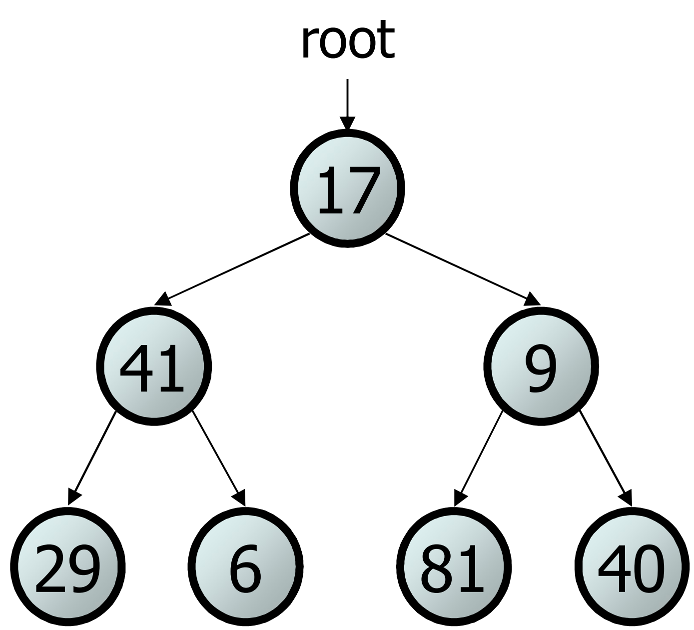
`IntTree` constructor

- Assume we have the following constructors:

```
IntTree(IntTreeNode *r)
IntTree(int height)
```

  - The 2nd constructor creates a tree and fills it with nodes with random data values from 1..100 until it is full at the given height.

```
IntTree tree(2);
```

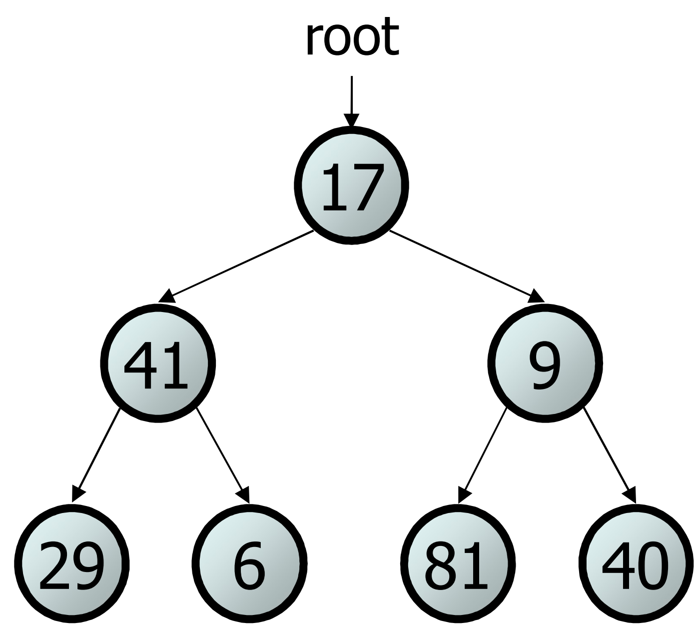


Printing a tree

- Add a method `print` to the `IntTree` class that prints the elements of the tree, separated by spaces.
  - A node's left subtree should be printed before it, and its right subtree should be printed after it.

  - Example: `tree.print();`

    `29 41 6 17 81 9 40`



Solution

```
// An IntTree object represents an entire binary tree of ints.
class IntTree {
  public:
    void print() {
        print(root);
        cout << endl;   // end the line of output
    }
// other methods
...
  private:
    IntTreeNode *root;   // null for an empty tree

    void print(IntTreeNode *r) {
      // (base case is implicitly to do nothing on null)
      if (r != null) {
        // recursive case: print left, center, right
        print(r->left);
        cout << r->data << " ";
        print(r->right);
      }
    }
}
```

Style for tree methods

```
class IntTree {
  public:
        type  function_name(parameters)  {
              function_name (root, parameters);
        }


  private:
       IntTreeNode * root;
       type  function_name (IntTreeNode *r, parameters)  {
         ...
       }
};
```

- Tree methods are often implemented recursively
  - with a public/private pair
  - the private version accepts the root node to process

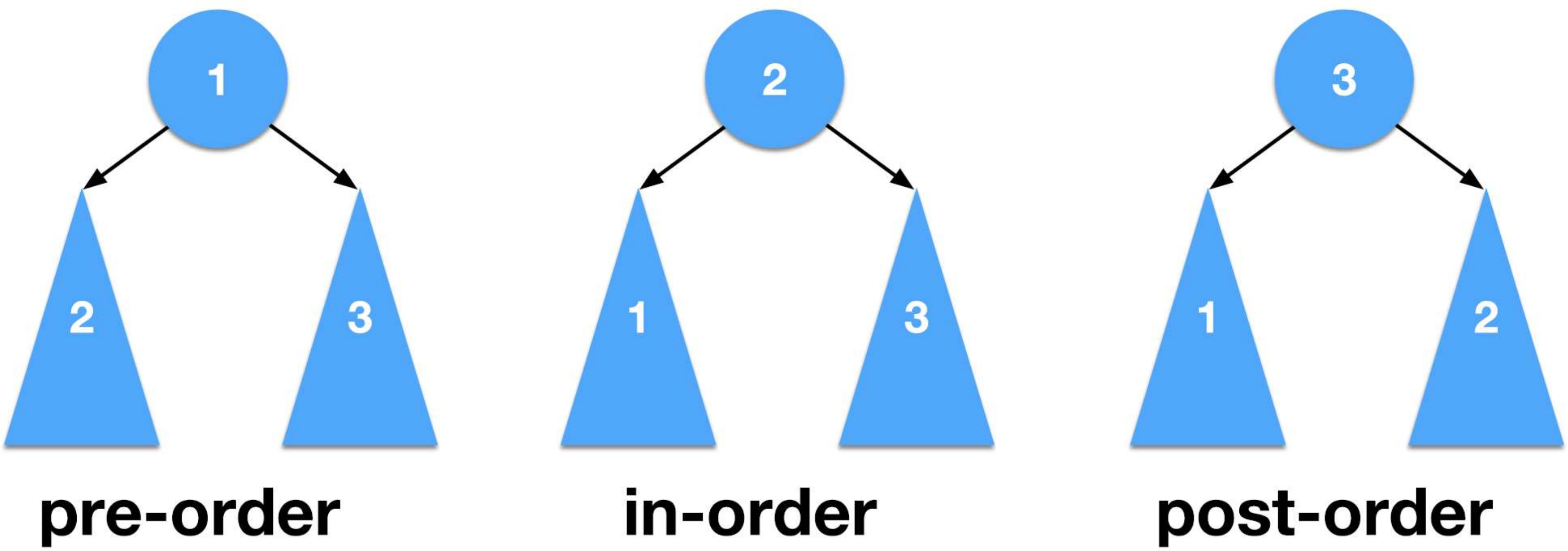
Traversals

- **traversal**: An examination of the elements of a tree.
  - A pattern used in many tree algorithms and methods

- Common orderings for traversals:
  - **pre-order**: visit the *current* node, visit the left subtree, then visit the right subtree
  - **in-order**: visit the left subtree, visit the *current* node, then visit the right subtree
  - **post-order**: visit the left subtree, visit the right subtree, then visit the *current* node

Traversing a Binary Tree

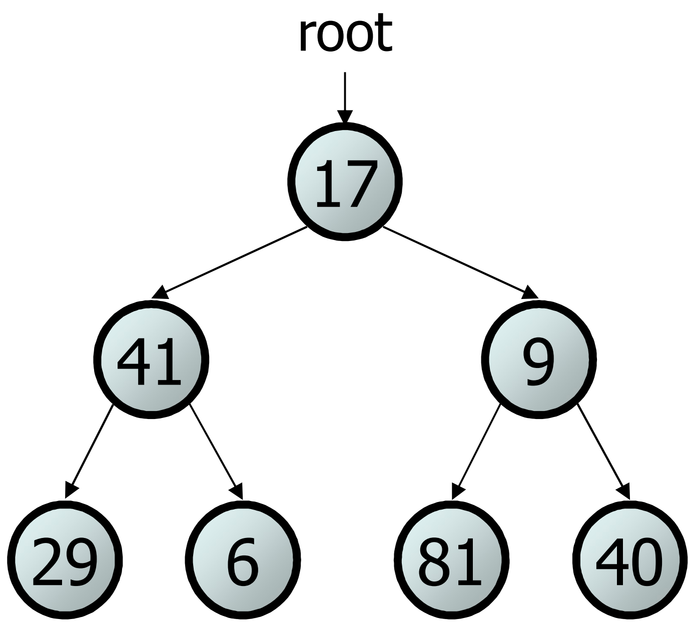
- Comparing the tree traversal methods:



(The numbers above refer to the order of traversal.)

- The subtrees are traversed **recursively**!

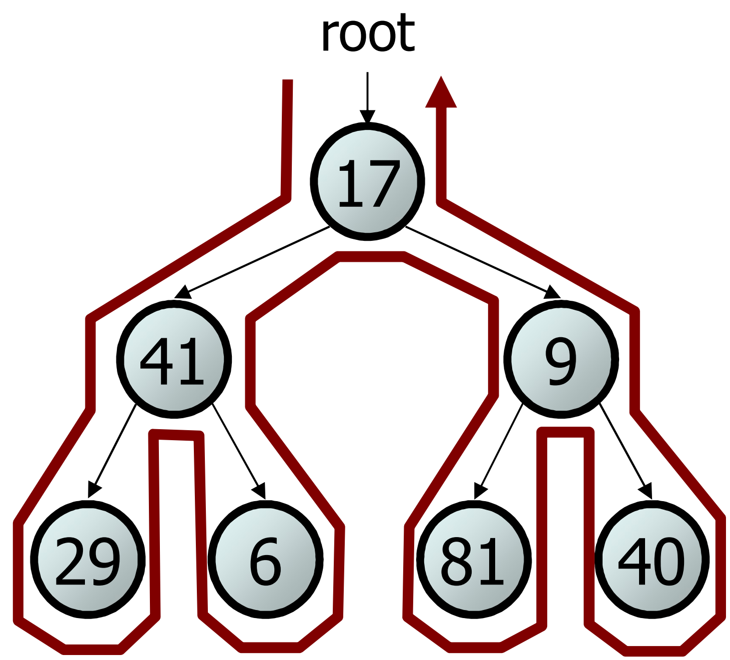
Traversal example



- pre-order: `17 41 29 6 9 81 40`
- in-order: `29 41 6 17 81 9 40`
- post-order: `29 6 41 81 40 9 17`

Traversal trick

- To quickly generate a traversal:
  - Trace a path around the tree.
  - As you pass a node on the proper side, process it.
    - pre-order: left side
    - in-order: bottom
    - post-order: right side



- pre-order: 17 41 29 6 9 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17

Exercise 1

- Give pre-, in-, and post-order traversals for the following tree:

  - Pre-order:

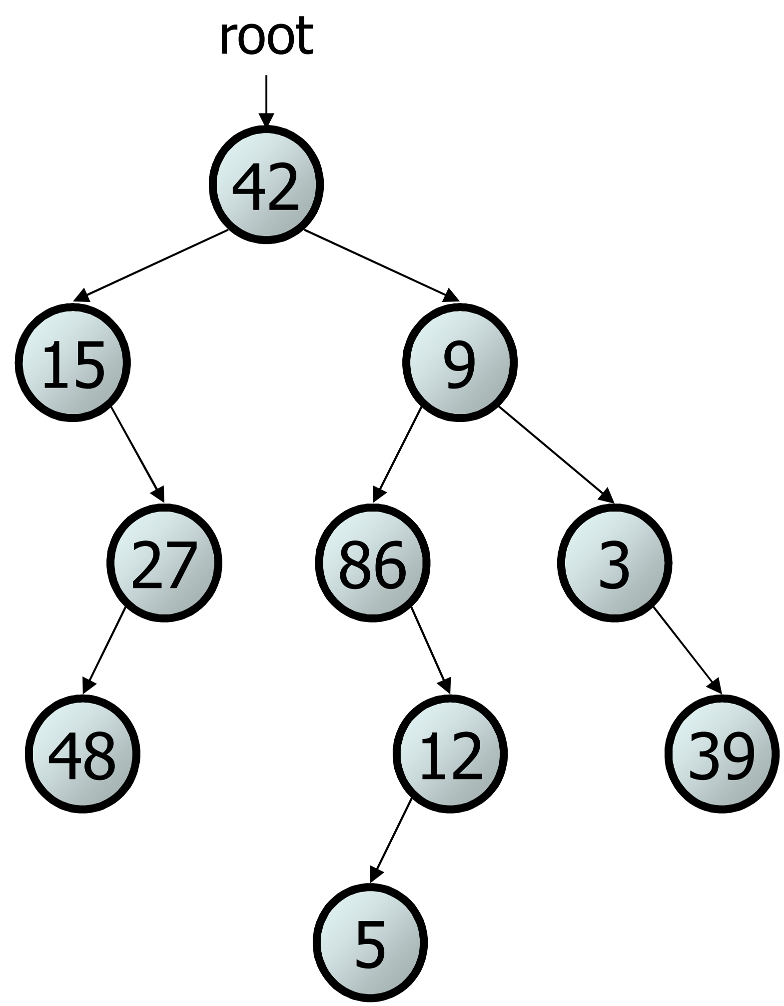
    42 15 27 48 9 86 12 5 3 39

  - In-order:

    15 48 27 42 86 5 12 9 3 39

  - Post-order:

    48 27 15 5 12 86 39 3 42



Preorder traversal

```
void preorder(IntTreeNode *r) {

  if (r != nullptr) {
    cout << r->data << " ";
    preorder(r->left);
    preorder(r->right);
  }

}
```

Postorder Traversal

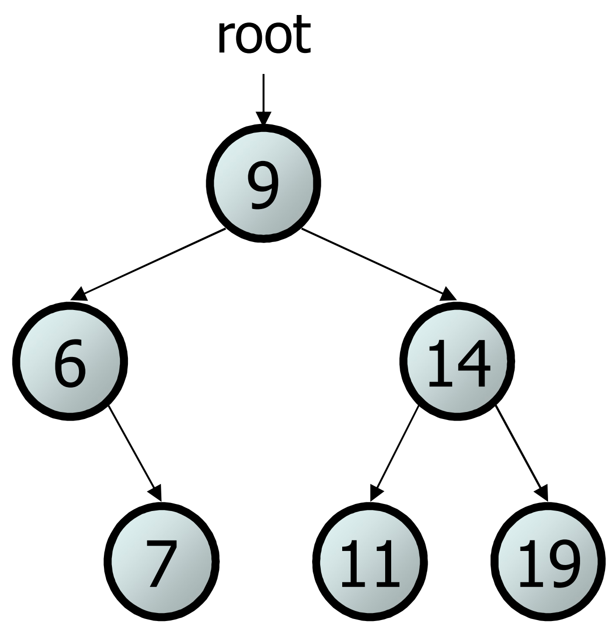
```
void postorder(IntTreeNode *r) {

  if (r != nullptr) {
    postorder(r->left);
    postorder(r->right);
    cout << r->data << " ";
  }
}
```

Exercise

- Add a method named `printSideways` to the `IntTree` class that prints the tree in a sideways indented format, with right nodes above roots above left nodes, with each level 4 spaces more indented than the one above it.

  - Example: Output from the tree below:

```
        19
    14
        11
9
        7
    6
```



Exercise solution

```
// Prints the tree in a sideways indented format.
void printSideways() {
    printSideways(root, "");
}

void printSideways(IntTreeNode *r, string indent) {
    if (r != nullptr) {
        printSideways(r->right, indent + "    ");
        cout << indent << r-> data) << endl;
        printSideways(r->left, indent + "    ");
    }
}
```

Finding the maximum value in a binary tree

```
class IntTree {
  public:
...
    int getMax (){
      return getMax (root);
    }
...

  private:
       IntTreeNode * root;
...
       int getMax(IntTreeNode *r){
            ...
       }
};
```

Finding the maximum value in a binary tree

```
 int getMax(IntTreeNode *r){
   int root_val, left, right, max;
   max = -1; // Assuming all values are positive integers
   if (r!= nullptr) {
      root_val = r ->data;
      left = getMax(r->left);
      right = getMax(r->right);
      // Find the largest of the three values.
      if (left > right)
         max = left;
      else
         max = right;
      if (root_val > max)
         max = root_val;
      }
      return max;
}
```

Adding up all values in a Binary Tree

```
int find_sum(IntTreeNode *r){
   if (r==nullptr)
      return 0;
   else
      return (r->data +
              find_sum(r->left) + find_sum(r->right) );
}
```

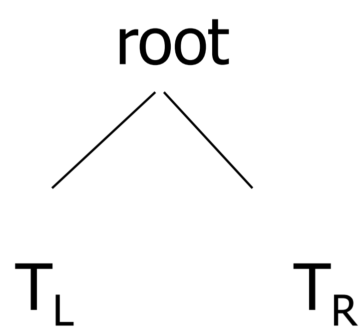
Exercise

Add a method `count_leaves` to the `IntTree` class that counts the leaves of a binary tree.

```
public:
    int count_leaves (){
        return count_leaves (root);
    }
private:
    int count_leaves(IntTreeNode *r){
    // TODO


    }
```

Height of Binary Tree

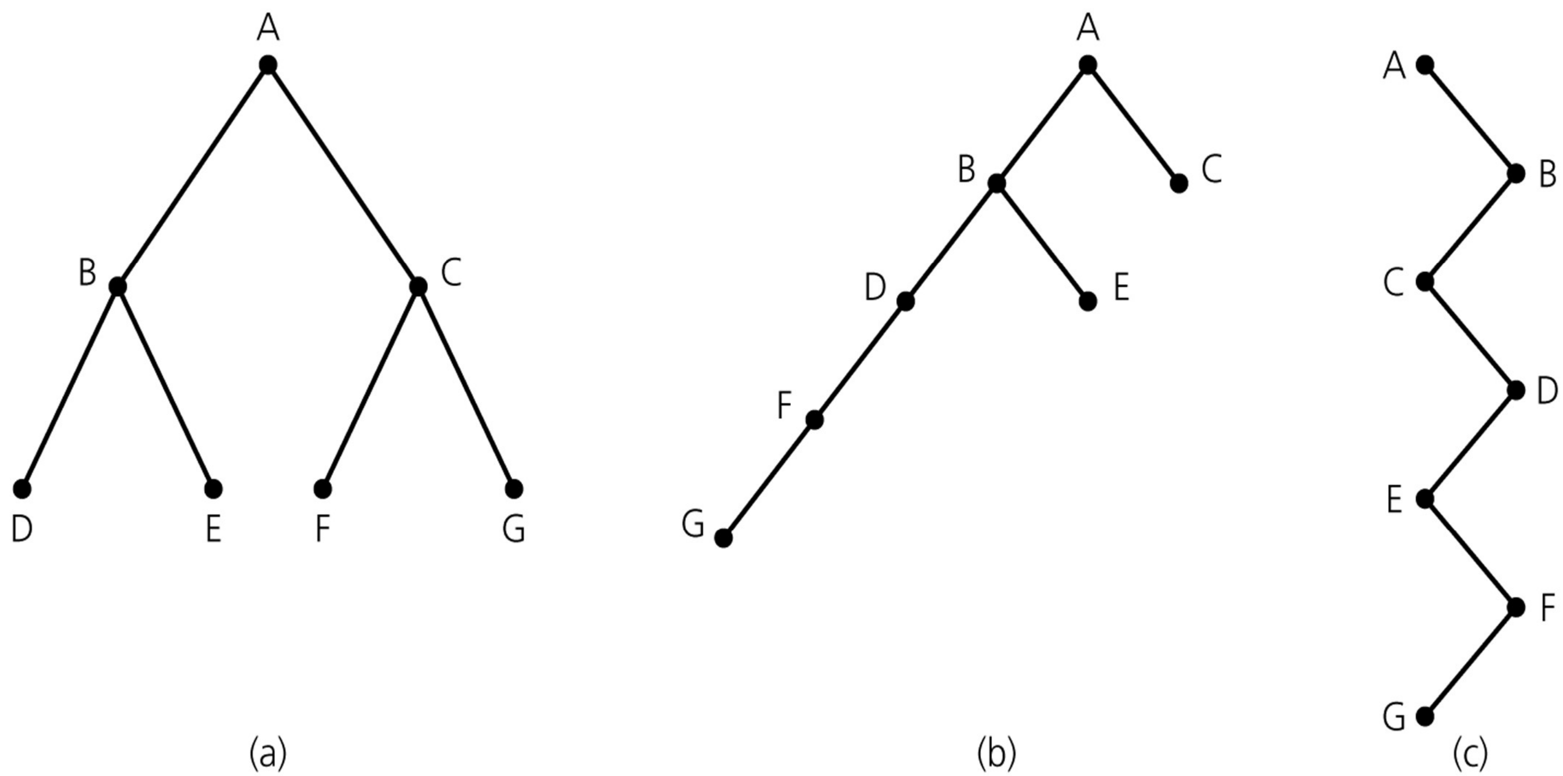
- The height of a binary tree T can be defined *recursively* as:
  - If T is empty, its height is -1.
  - If T is non-empty tree, then since T is of the form



the height of T is 1 greater than the height of its root's taller subtree; i.e.

$$\mathbf{height(T)} = \mathbf{1 + max}\{\text{height}(T_L), \text{height}(T_R)\}$$

Height of Binary Tree (cont.)

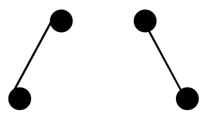


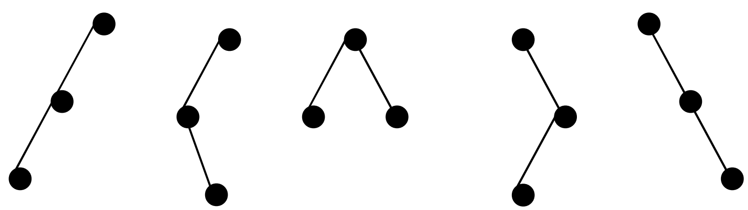
Binary trees with the same nodes but different heights

Number of Binary trees with Same # of Nodes

n=0 → empty tree

n=1 →  (1 tree)

n=2 →  (2 trees)

n=3 →  (5 trees)

In general:

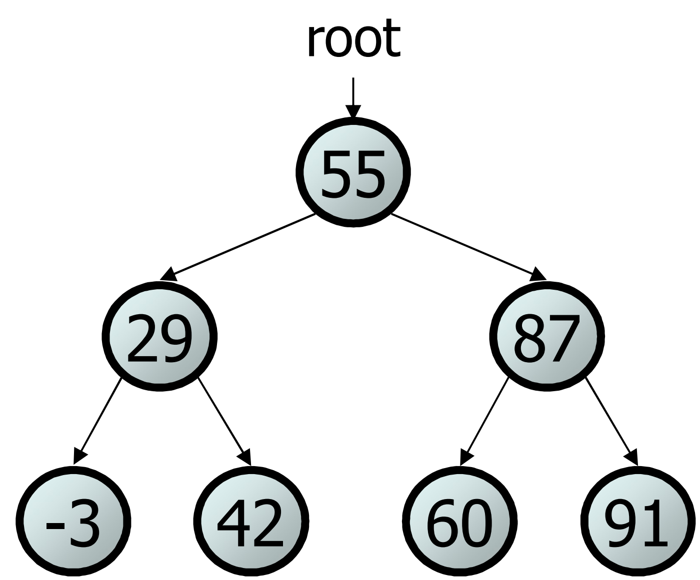
Catalan number C(n) = (2n)!/(n+1)!n!

Different number of structurally different Binary trees is : Catalan(N)
Different number of Binary Trees: N! Catalan(N)

Binary Search Trees

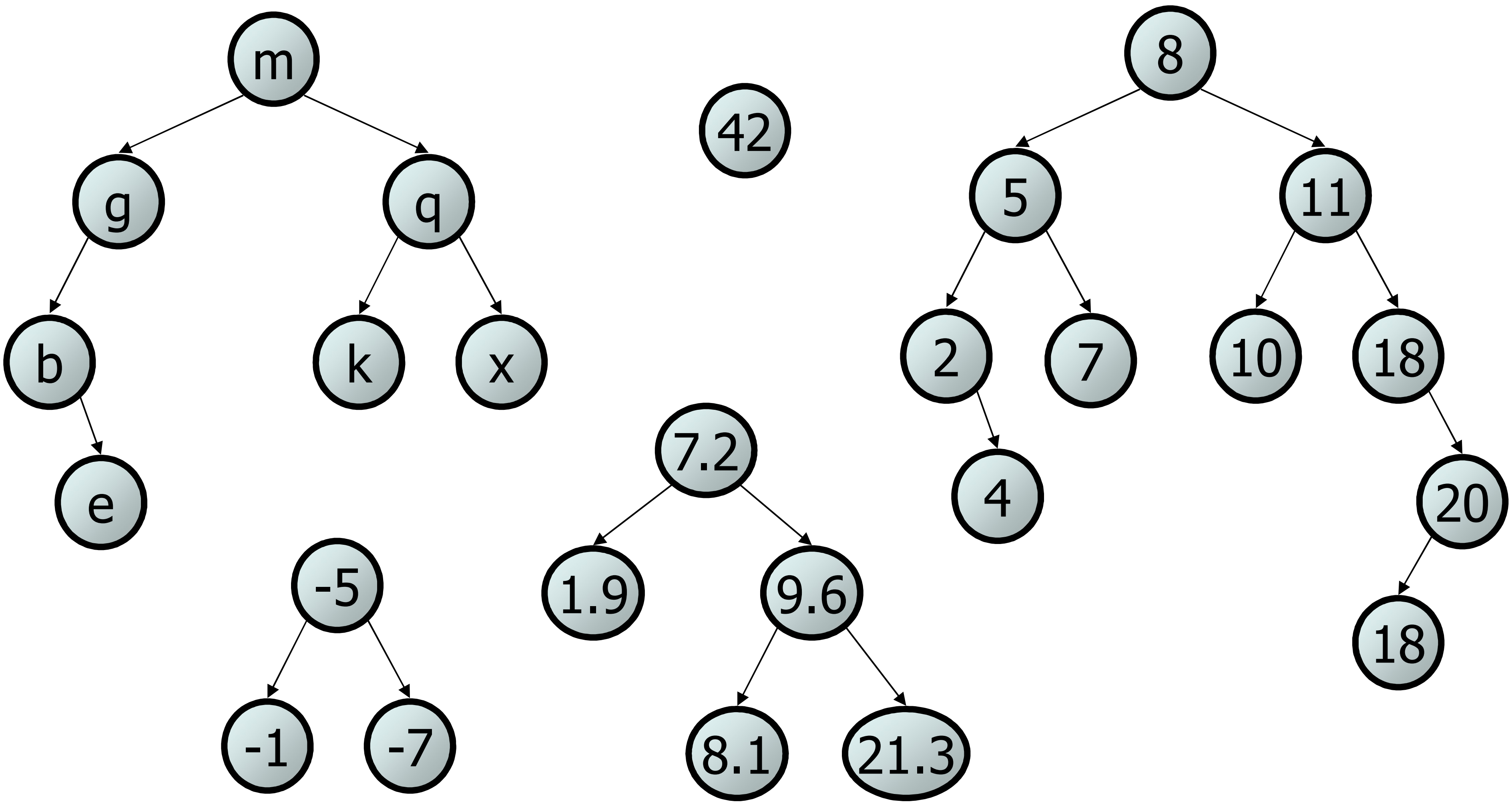
- **binary search tree** (BST) is a binary tree that is either:
  - empty (`null`), or
  - a root node R such that:
    - every element of R's left subtree contains data "less than" R's data,
    - every element of R's right subtree contains data "greater than" R's,
    - R's left and right subtrees are also binary search trees.

- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.



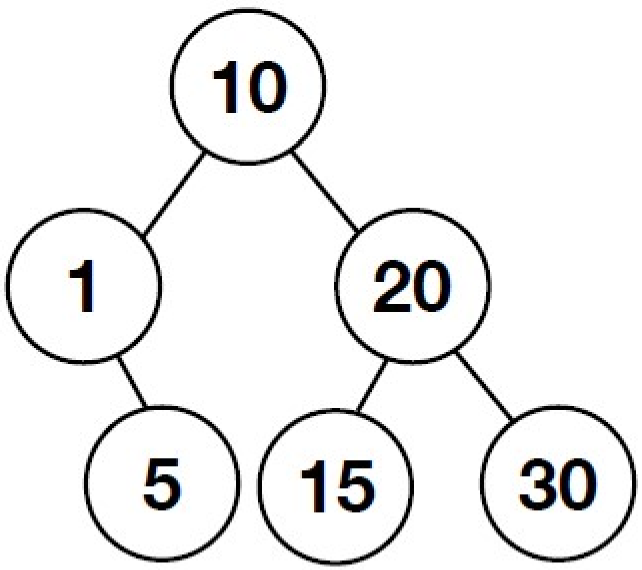
Exercise

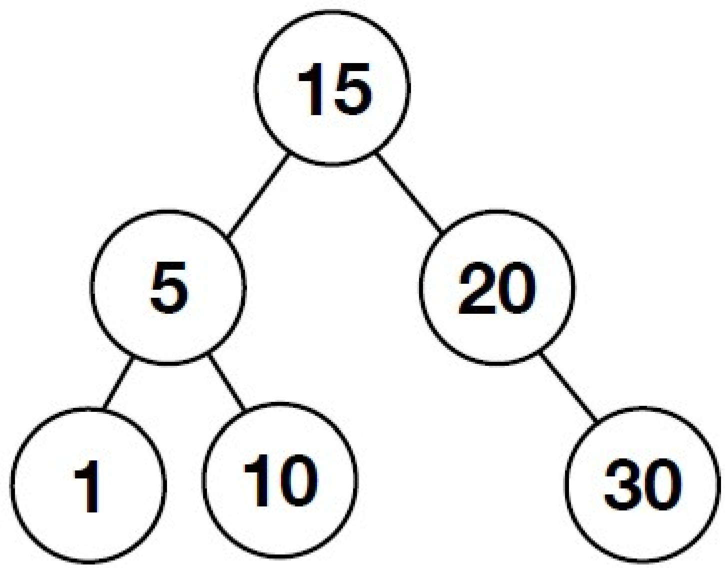
- Which of the trees shown are legal binary search trees?



Inorder traversal of BST

- Let’s work out the **in-order traversal** results of the following two valid BSTs.





- For both, in order traversal gives the same result:

1, 5, 10, 15, 20, 30. This is clearly sorted!

Hey! these are all different things

Please do not confuse them

- **Binary Search:**
  an algorithm on a sorted <u>array</u>.

- **Binary Tree**
  a tree where nodes have no more than 2 children.

- **Binary Search Tree**
  a binary tree with a special ordering property

Search in a BST

- However, Binary Search and BST are related, because the way you search in a BST is similar to performing a binary search in an ordered array.
- Find 31

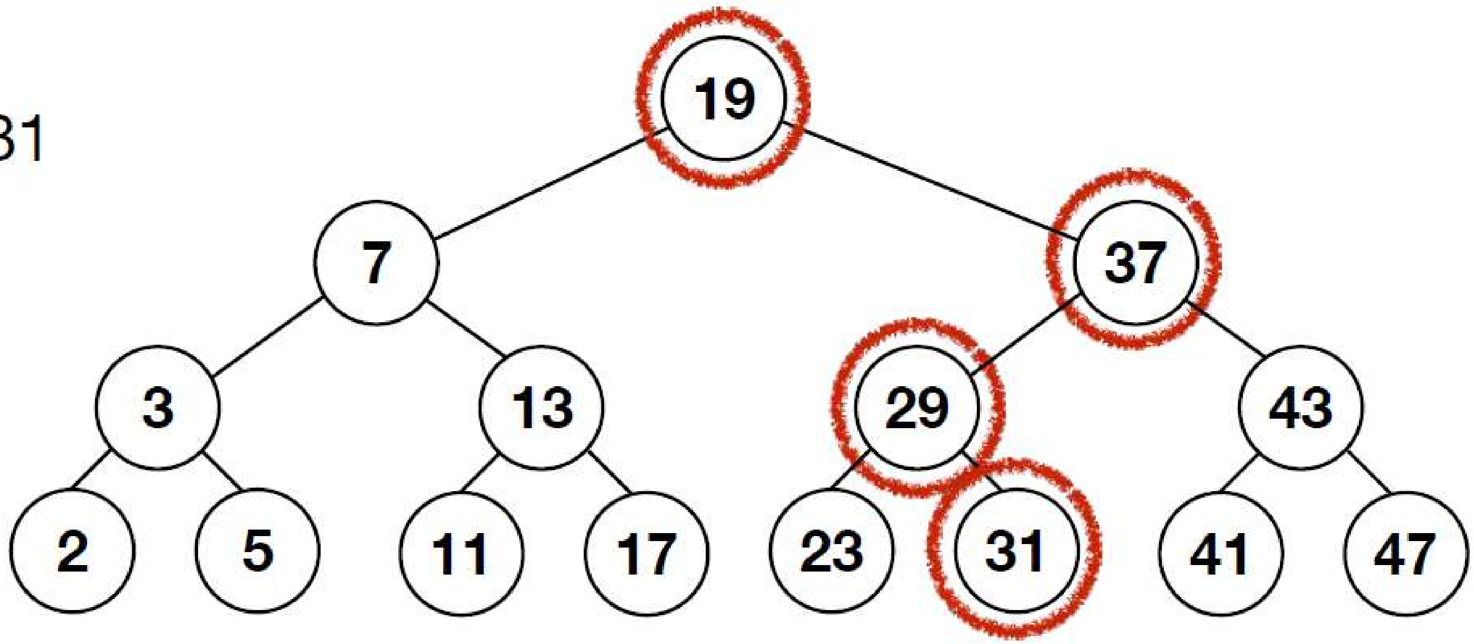
2	3	5	7	11	13	17	**19**	23	29	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	31	**37**	41	43	47
2	3	5	7	11	13	17	19	23	**29**	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	**31**	37	41	43	47

Search in a BST

- Find 31

2	3	5	7	11	13	17	**19**	23	29	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	31	**37**	41	43	47
2	3	5	7	11	13	17	19	23	**29**	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	**31**	37	41	43	47

- Find 31



What is the maximum number of nodes you would need to examine to perform any search?

Search in a BST

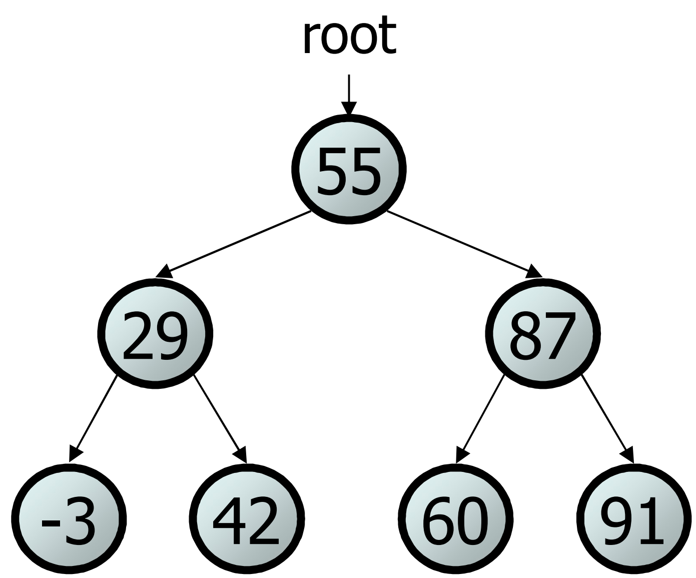
- To summarize, you start from the root node, then choose to go left or right depending on the comparison result. The search ends when either you've found the target or you've reached a leaf.
- The maximum number of steps is the **tree height**.
- As in binary search, search in BST can achieve O(log N) time. However, this requires the BST to be balanced (i.e. the height should be small).
- If you have a poorly constructed BST (e.g. degenerated to a linked list), you won't get the O(log N) performance!

Binary Search Tree Class

- Convert the `IntTree` class into a `SearchTree` class.
  - The elements of the tree will constitute a legal binary search tree.

- Add a method `contains` to the `SearchTree` class that searches the tree for a given integer, returning `true` if found.

  - If a `SearchTree` variable `tree` referred to the tree below, the following calls would have these results:
    - `tree.contains(29)` → `true`
    - `tree.contains(55)` → `true`
    - `tree.contains(63)` → `false`
    - `tree.contains(35)` → `false`



Method `contains`

```
// Returns whether this tree contains the given integer.
public:
    bool contains(int val){
        return contains(root, val);
    }
private:
   bool contains(IntTreeNode *r, int val){
      if (r == nullptr)
         return false;
      else {
         if (r->data == val)
            return true;
         else if (r->data > val)
            return contains(r->left,val);
         else return contains(r->right,val);
       }
     }
```

Adding to a BST

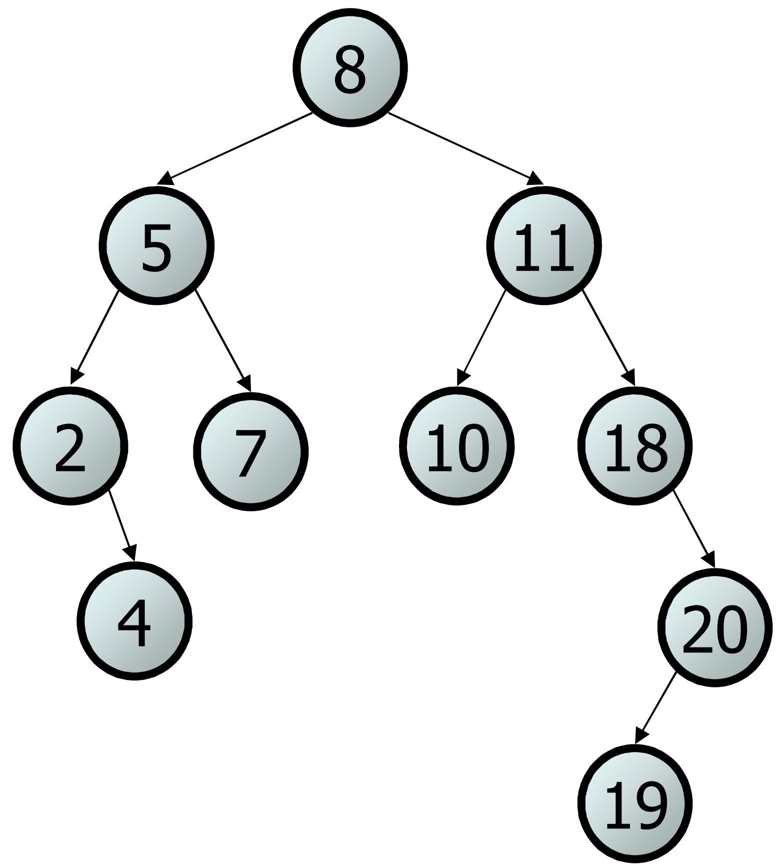
- Suppose we want to add the value 14 to the BST below.
  - Where should the new node be added?

- Where would we add the value 3?

- Where would we add 7?

- If the tree is empty, where should a new value be added?

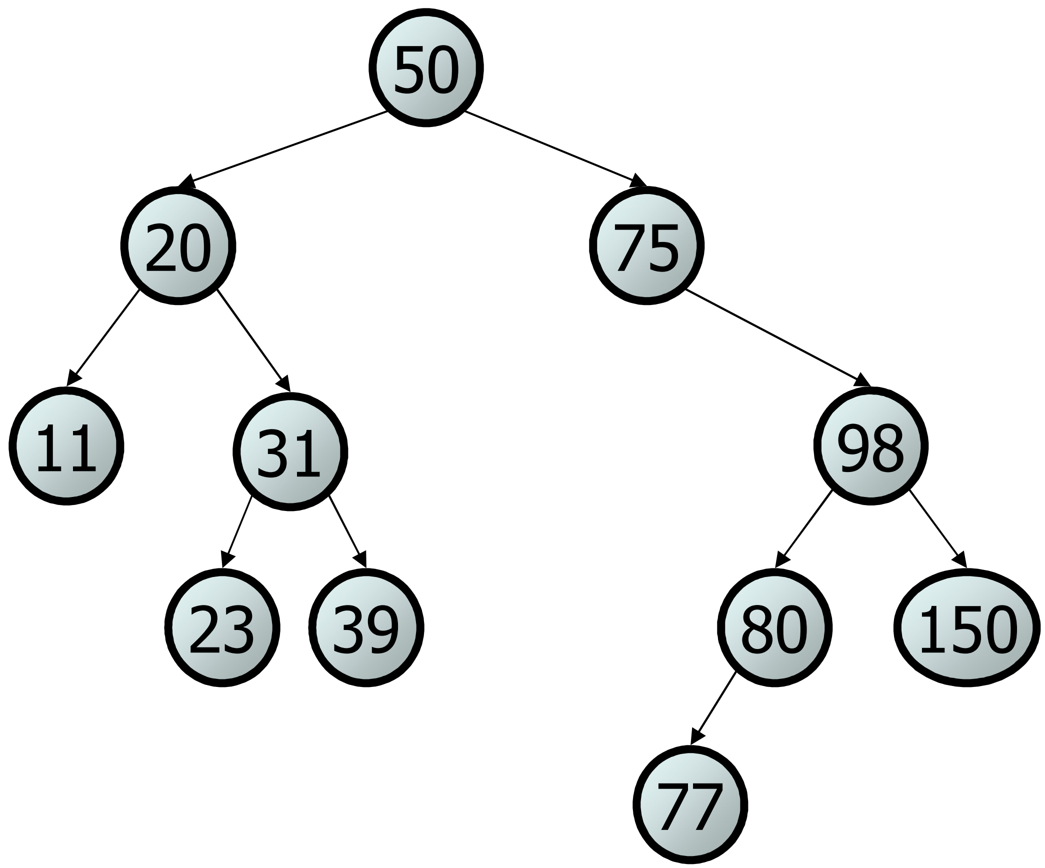
- What is the general algorithm?



Adding exercise

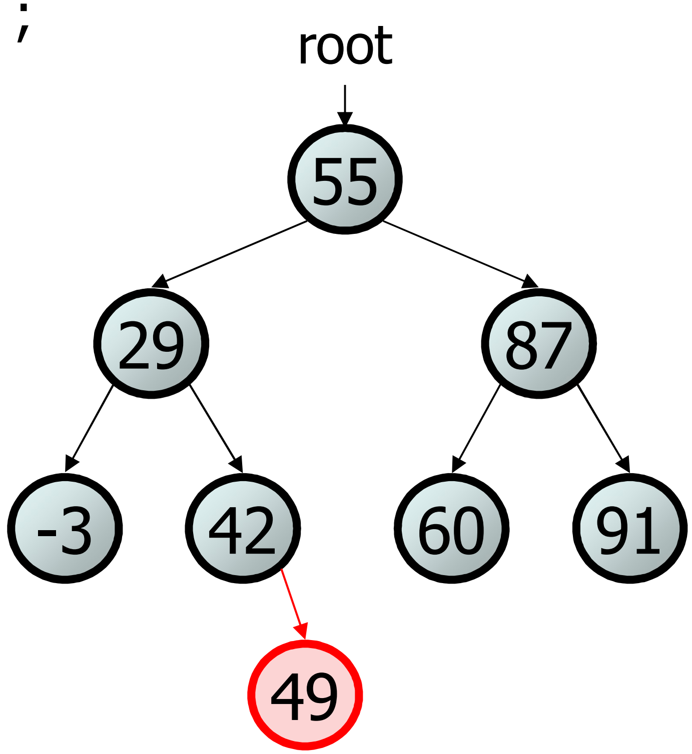
- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50
20
75
98
80
31
150
39
23
11
77



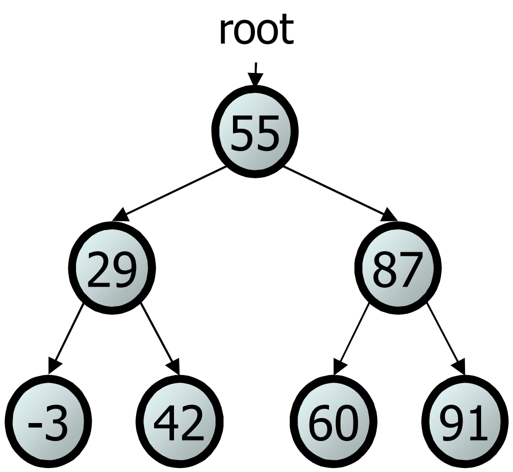
Implementing add

- Let's add a method `add` to the `SearchTree` class that adds a given integer value to the tree. Assume that the elements of the `SearchTree` constitute a legal binary search tree, and add the new value in the appropriate place to maintain ordering.
  - `tree.add(49);`



Code

```
// Adds the given value to this BST in sorted order.
public:
 void add(int value) {
    add(root, value);
}

private:
 void add(IntTreeNode *&r, int value) {
    if (r == nullptr)
        r = new IntTreeNode(value);
    else if (r->data > value)
        add(r->left, value);
    else if (r->data < value)
        add(r->right, value);
    // else a duplicate

 }
```

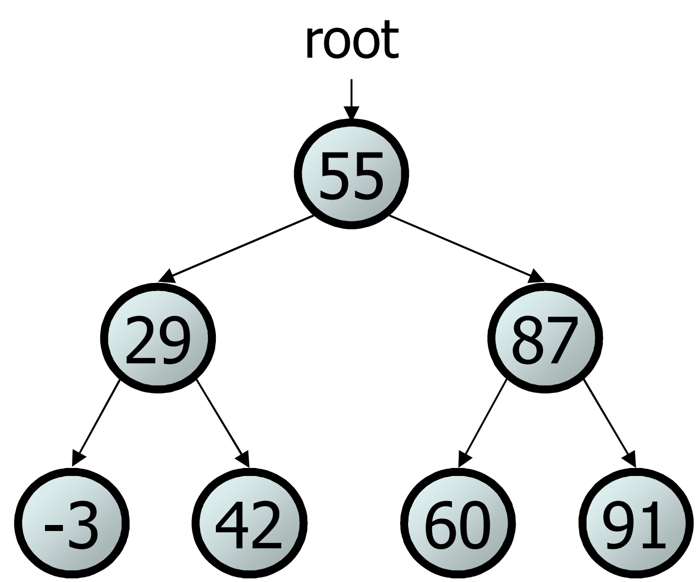


- Think about the case when `r` is a leaf...

Exercise

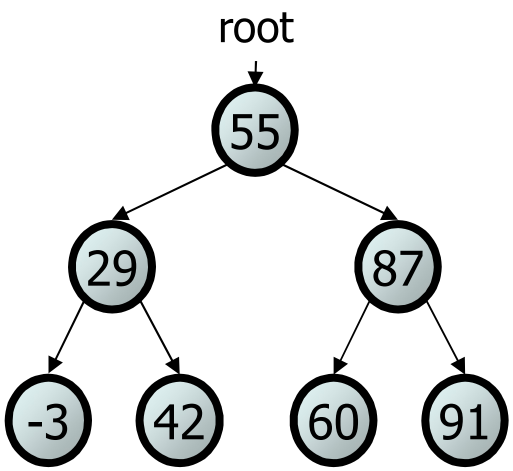
- Add a method `getMin` to the `IntTree` class that returns the minimum integer value from the tree. Assume that the elements of the `IntTree` constitute a legal binary search tree. Throw a `NoSuchElementException` if the tree is empty.

```
int min = tree.getMin();  // -3
```



Solution

```
// Returns the minimum value from this BST.
// Throws a NoSuchElementException if the tree is empty.
public:
  int getMin() {
    if (root == nullptr)
        throw new NoSuchElementException();
    return getMin(root);
  }

private:
  int getMin(IntTreeNode* r) {
    if (r->left == nullptr)
        return r->data;
    else
        return getMin(r->left);
  }
```



Find max: Iterative method

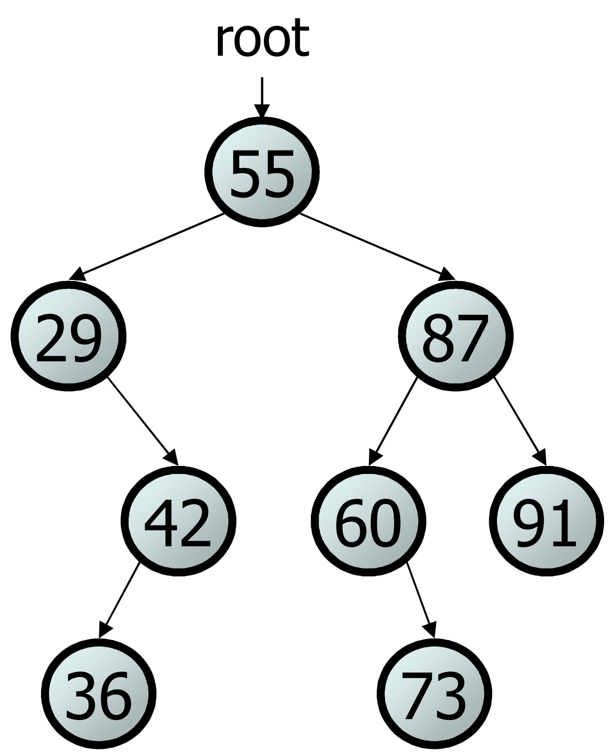
```
// Returns the largest value from this BST.
// Throws a NoSuchElementException if the tree is
  empty.
public:
   int getMax() {
      return getMax(root);
   }
private:
    int getMax(IntTreeNode *r){
        if (r == nullptr)
           throw new NoSuchElementException();
        while (r->right!= nullptr)
           r = r->right;
        return r->data;
     }
```

Removing from a BST

Possible cases for the node to be removed:

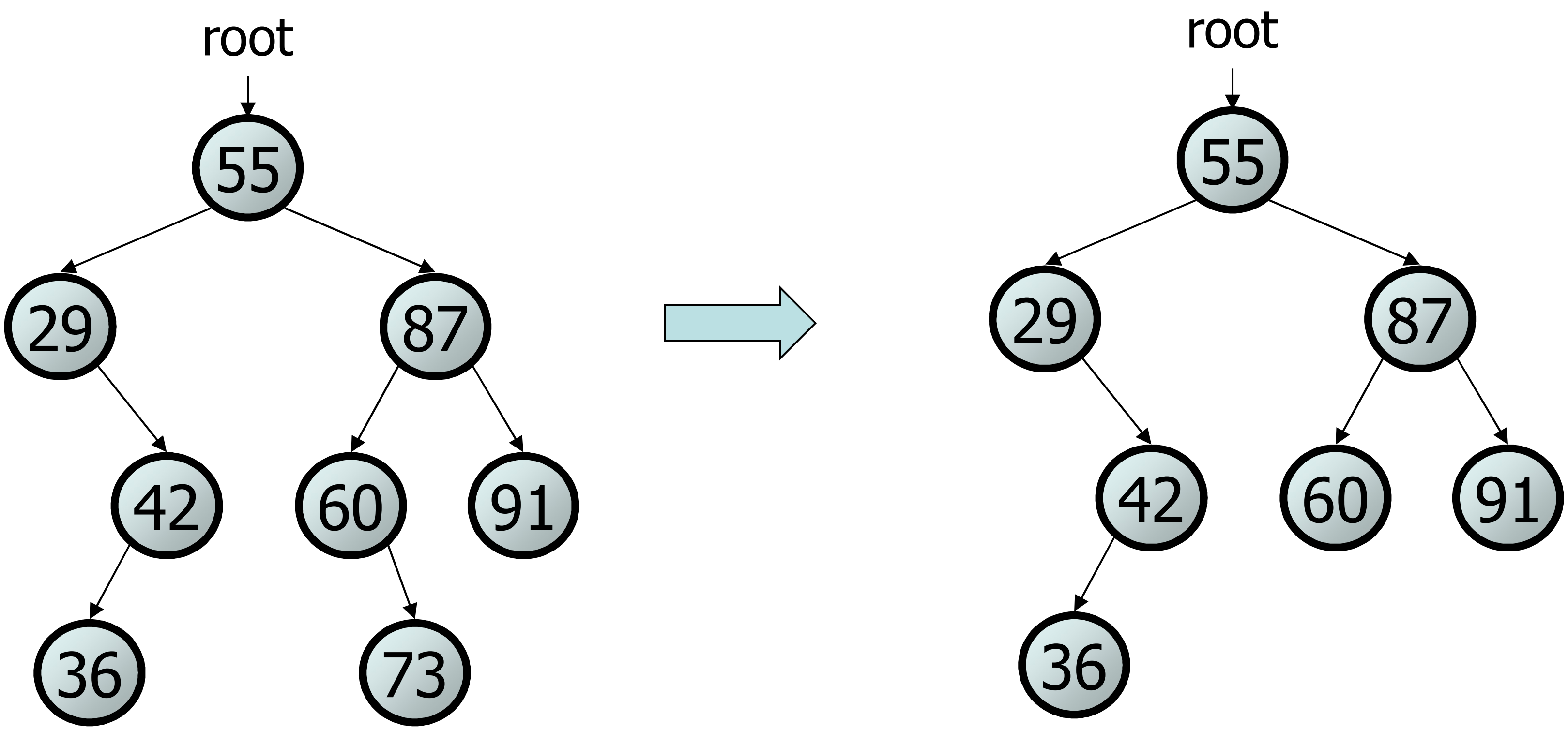
1. a leaf
2. a node with only one child (left or right child)
3. a node with both children

- `tree.remove(73);`
- `tree.remove(29);`
- `tree.remove(42);`
- `tree.remove(55);`



Case 1: Removing a Leaf Node

1. a leaf: replace with null

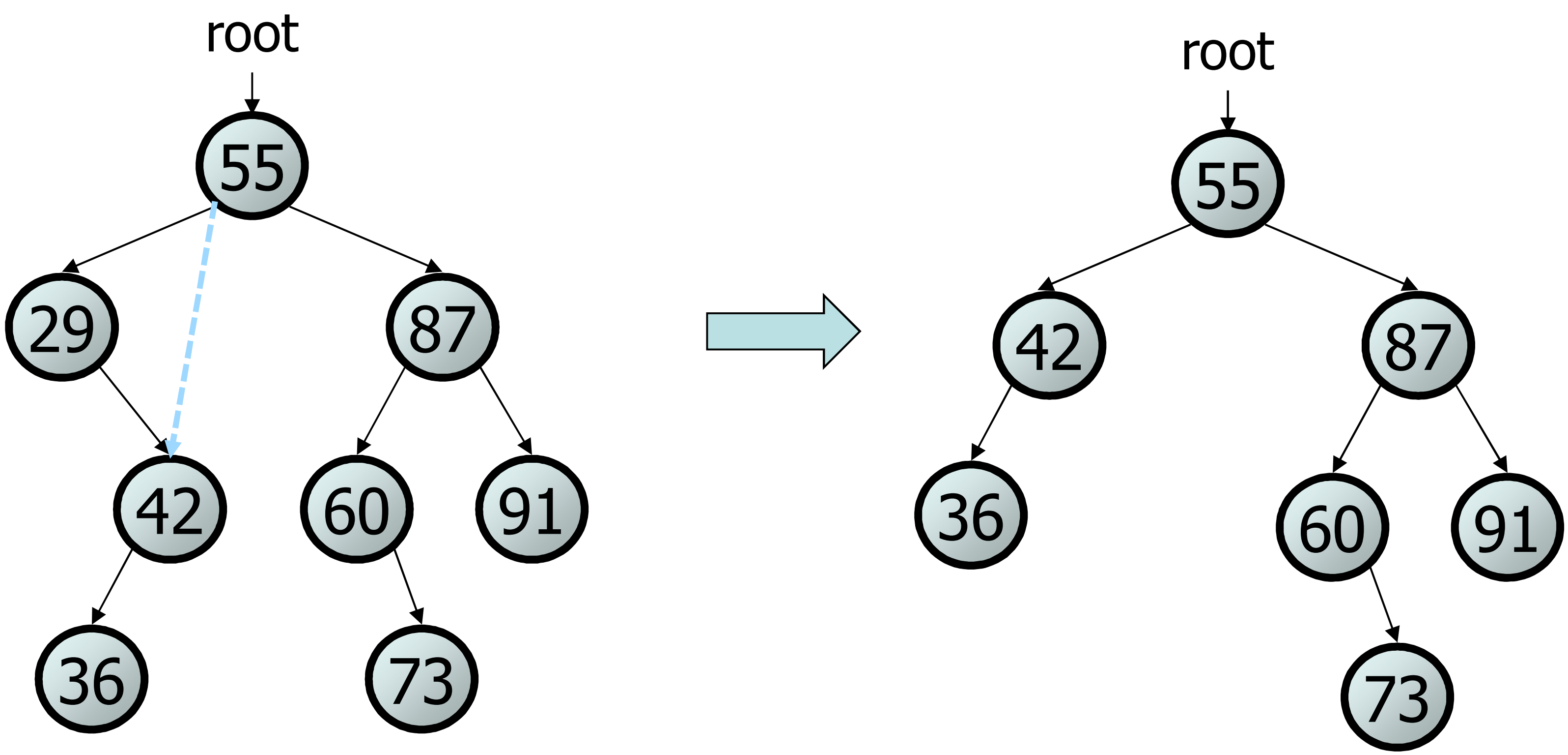



```
tree.remove(73);
```

Case 2: Remove a Node with one child

2.1.a node with a left child only: replace with left child

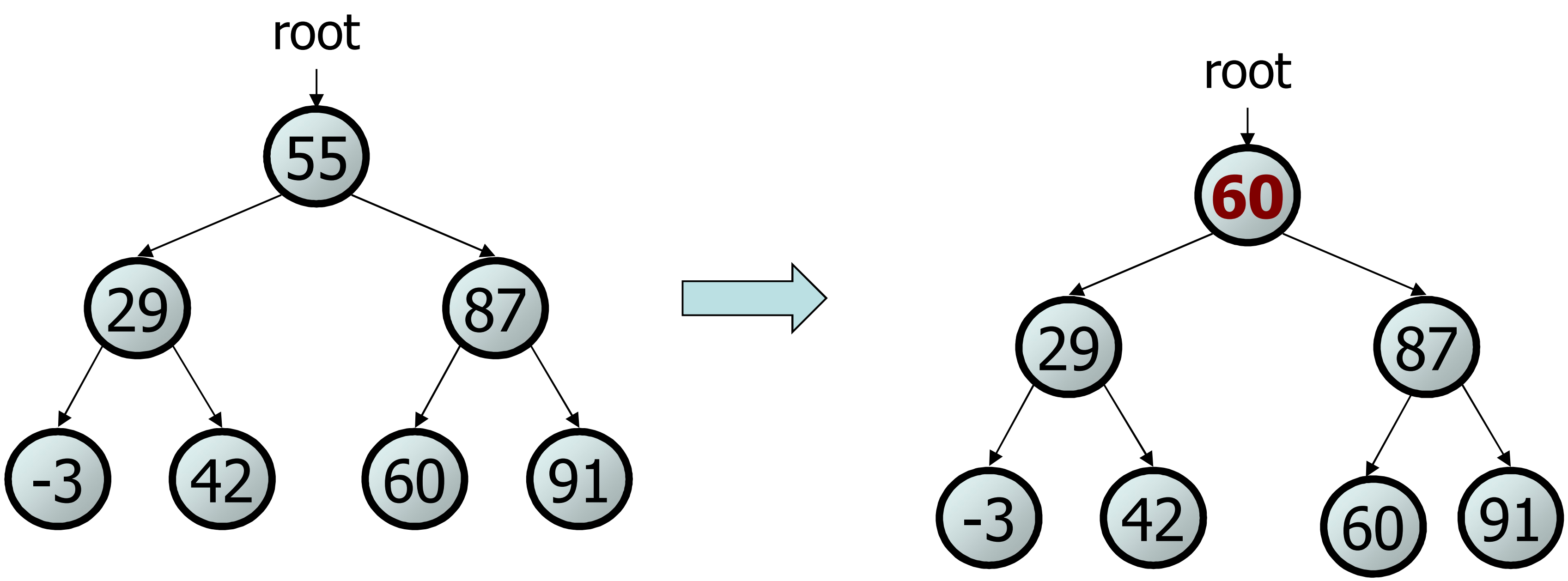
2.2. a node with a right child only: replace with right child




```
tree.remove(29);
```

Case 3: Remove a node with two children

3. a node with **both** children: replace with **min from right**

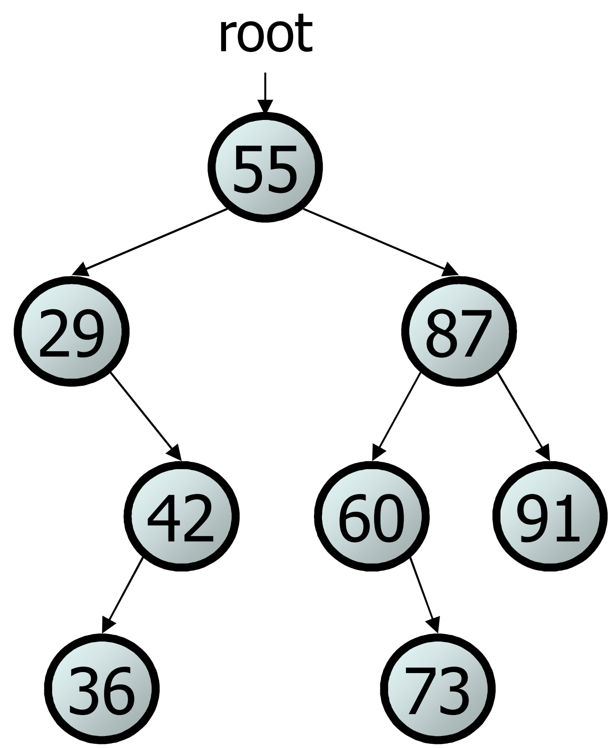



```
tree.remove(55);
```

`remove` method

- Add a method `remove` to the `IntTree` class that removes a given integer value from the tree, if present. Assume that the elements of the `IntTree` constitute a legal binary search tree, and remove the value in such a way as to maintain ordering.

  - `tree.remove(73);`
  - `tree.remove(29);`
  - `tree.remove(87);`
  - `tree.remove(55);`



remove method

```
// Removes the given value from this BST, if it exists.
public:
  void remove(int value) {
     remove(root, value);
}
private:
  void remove(IntTreeNode *& r, int value) {
    if (r == nullptr)
        return;
    else if (r->data > value)
        remove(r->left, value);
    else if (r->data < value)
        remove(r->right , value);
    else    // r->data == value; remove this node
        if (r->left !=nullptr && r->right != nullptr) {
            // case 3: both children; replace w/ min from R
           r->data = getMin(r->right); //copy value here
           remove(r->right, r->data);
        }
        else {// case 2: only child  or case 1: leaf node
           IntTreeNode * oldNode =r;
           r = (r->left != nullptr)? r->left : r->right;
           delete oldNode;
        }
  }
```