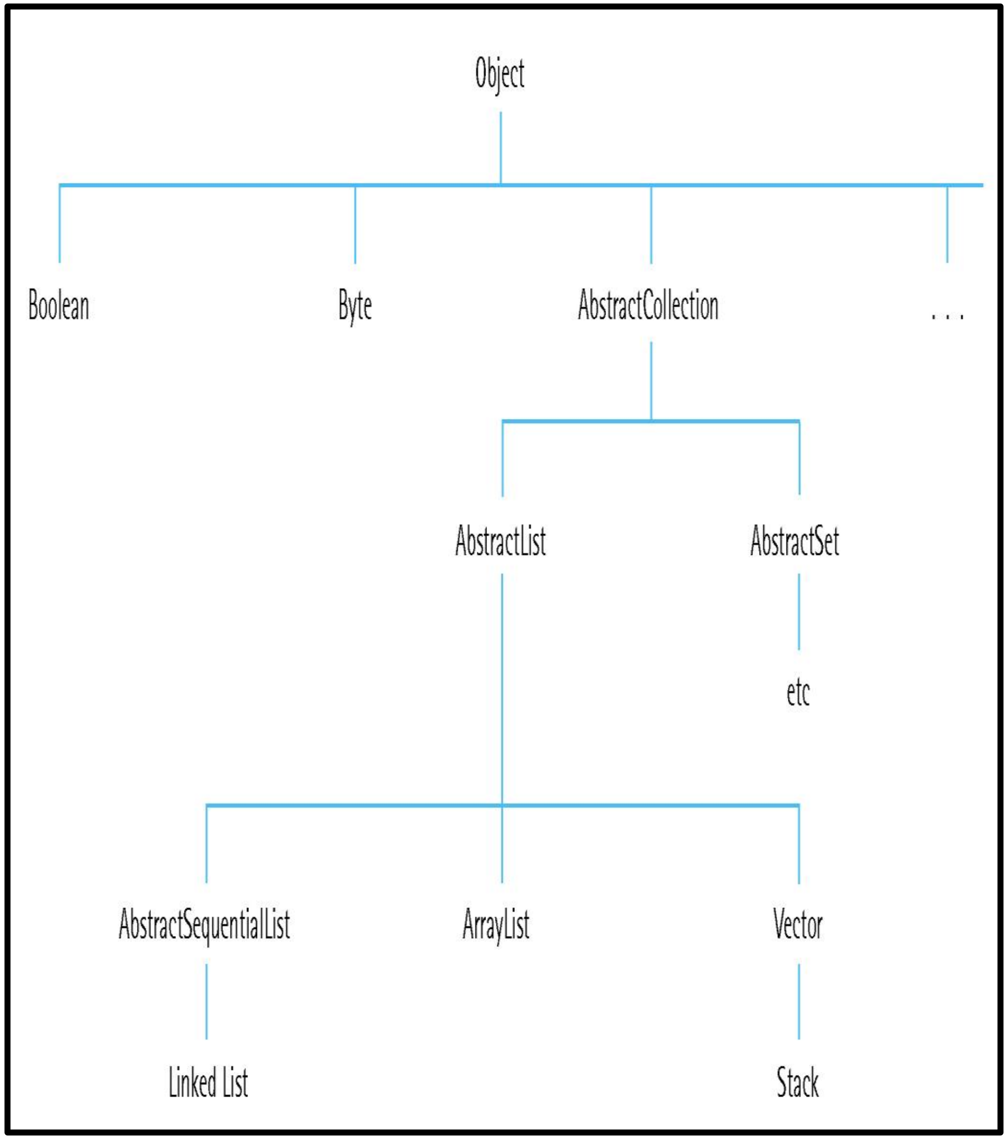
Chapter 8
Binary Search Trees

Chapter 8: Binary Search Trees

8.1 Trees

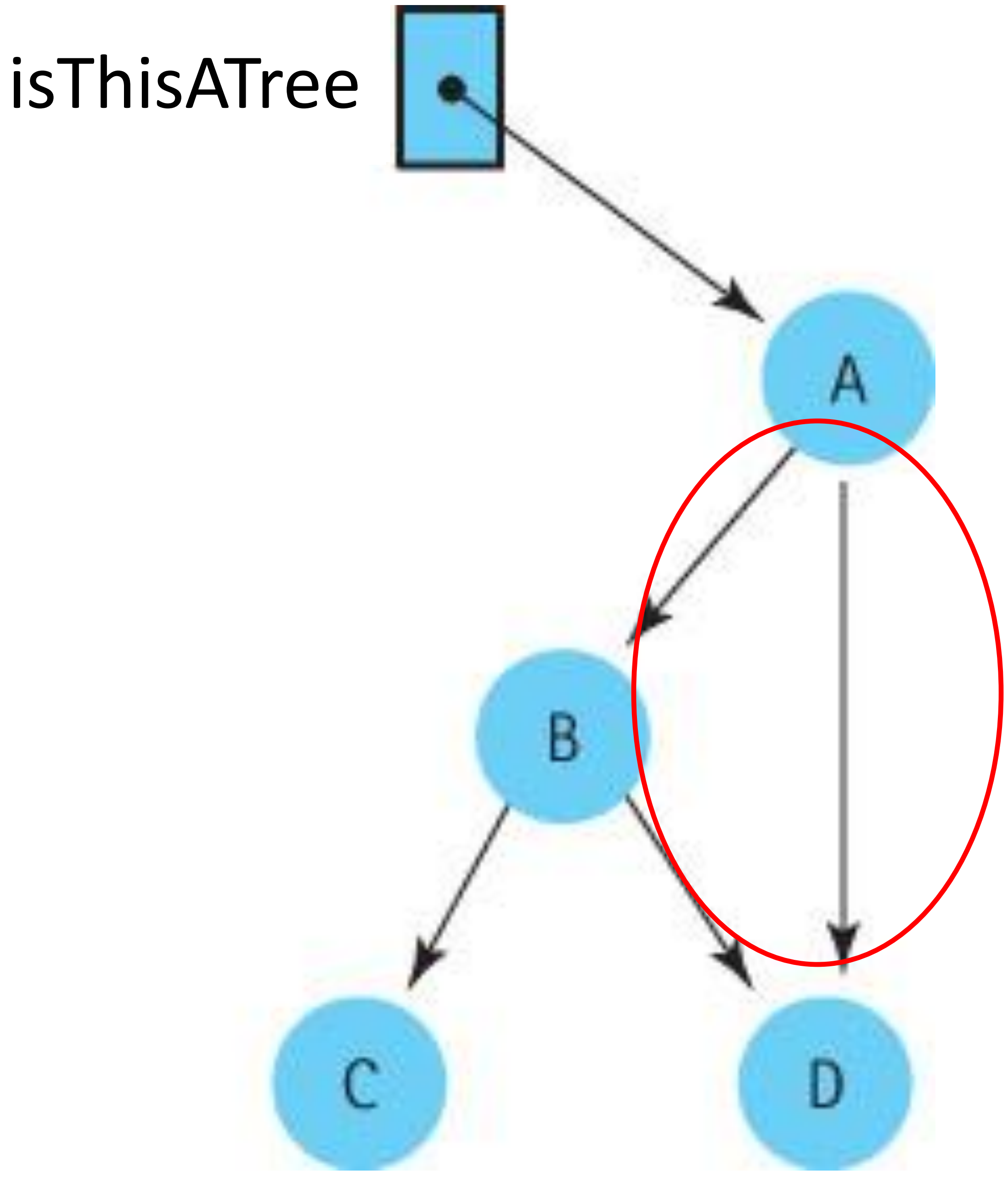
- A **tree** is a nonlinear structure in which each node is capable of having many successor nodes, called *children*.

- Trees are useful for representing hierarchical relationships among data items.



Definitions

- **Tree** A structure with a unique starting node (the root), in which each node is capable of having many child nodes, and in which a unique path exists from the root to every other node.
- **Root** The top node of a tree structure; a node with no parent


isThisATree
A
B
C
D

Definitions

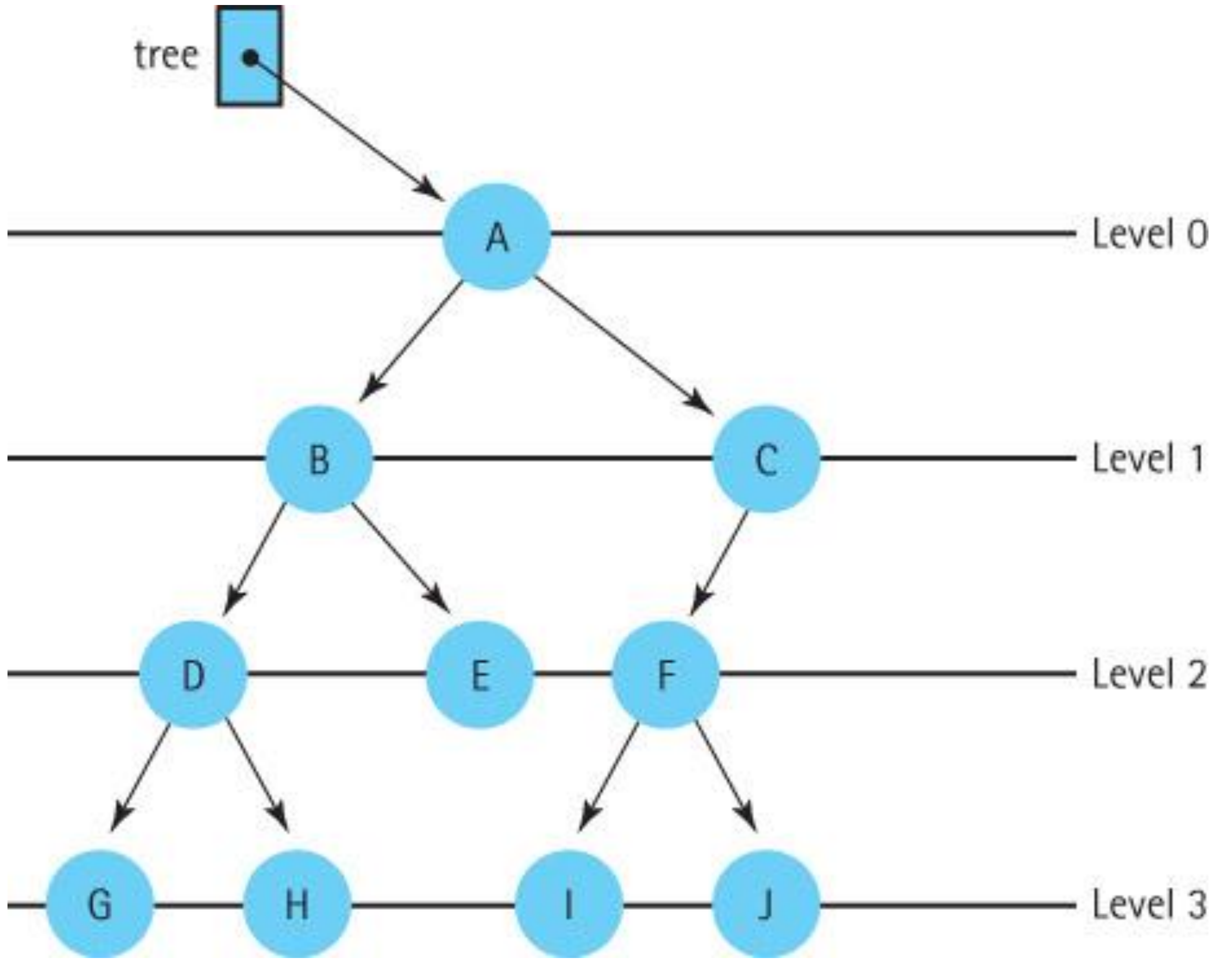
- **Binary tree**

  A tree in which each node is capable of having two child nodes: left and right

- **Leaf**

  A tree node that has no children

A Binary Tree



Is this a tree?

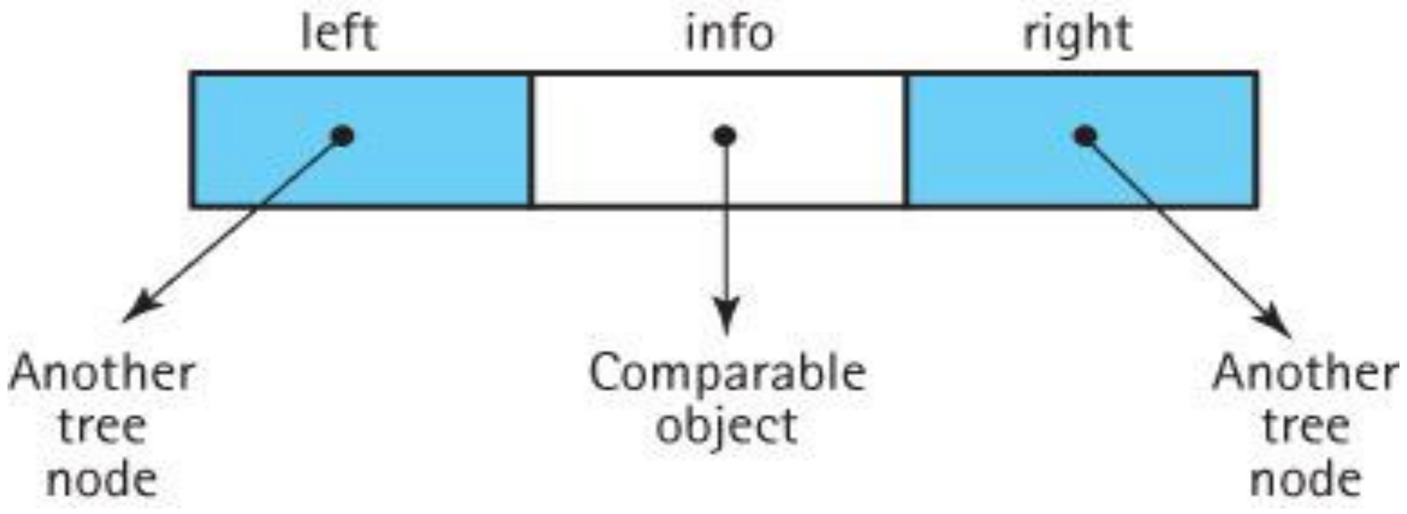
What kind of tree is it?

Which node is the root node? A

Which nodes are the leaf nodes? G, H, I, J

Implementation Level: Basics

- We define `BSTNode.java` in our `support` package to provide nodes for our binary search trees
- Visually, a `BSTNode` object is:



BSTNode.java

instance variables:

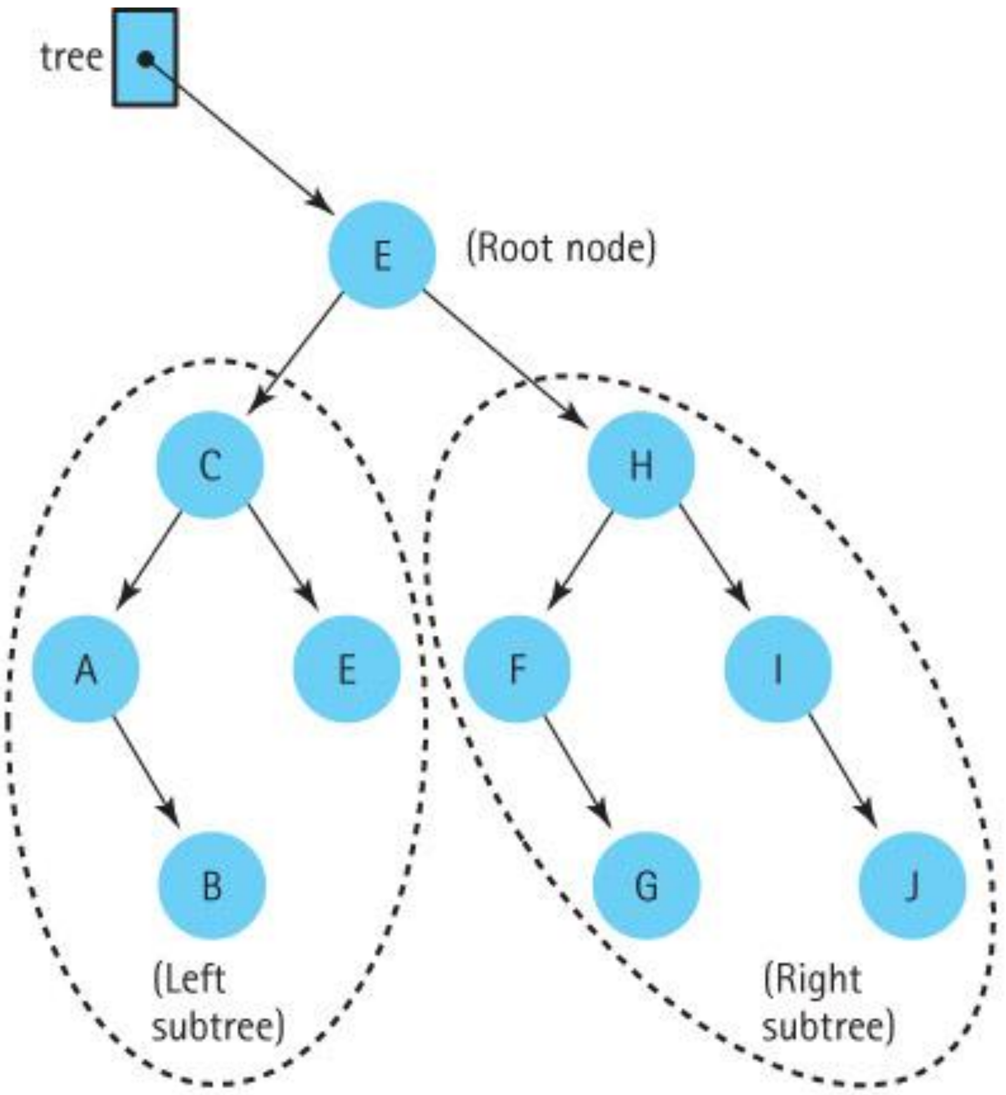
```
protected T info;                 // The info in a BST node
protected BSTNode<T> left;        // A link to the left child node
protected BSTNode<T> right;       // A link to the right child node
```

Constructor:

```
public BSTNode(T info)
{
  this.info = info;
  left = null;
  right = null;
}
```

Plus it includes the standard setters and getters, see BSTNode.java

A Binary Search Tree



All values in the left subtree are less than or equal to the value in the root node.

All values in the right subtree are greater than the value in the root node.

Definition

- **Binary search tree**
  - A binary tree in which the key value in any node is
    - greater than or equal to the key value in its left child and any of that child nodes descendants (the nodes in the left subtree)
    - less than the key value in its right child and any of that nodes descendants (the nodes in the right subtree)

Traversal Definitions

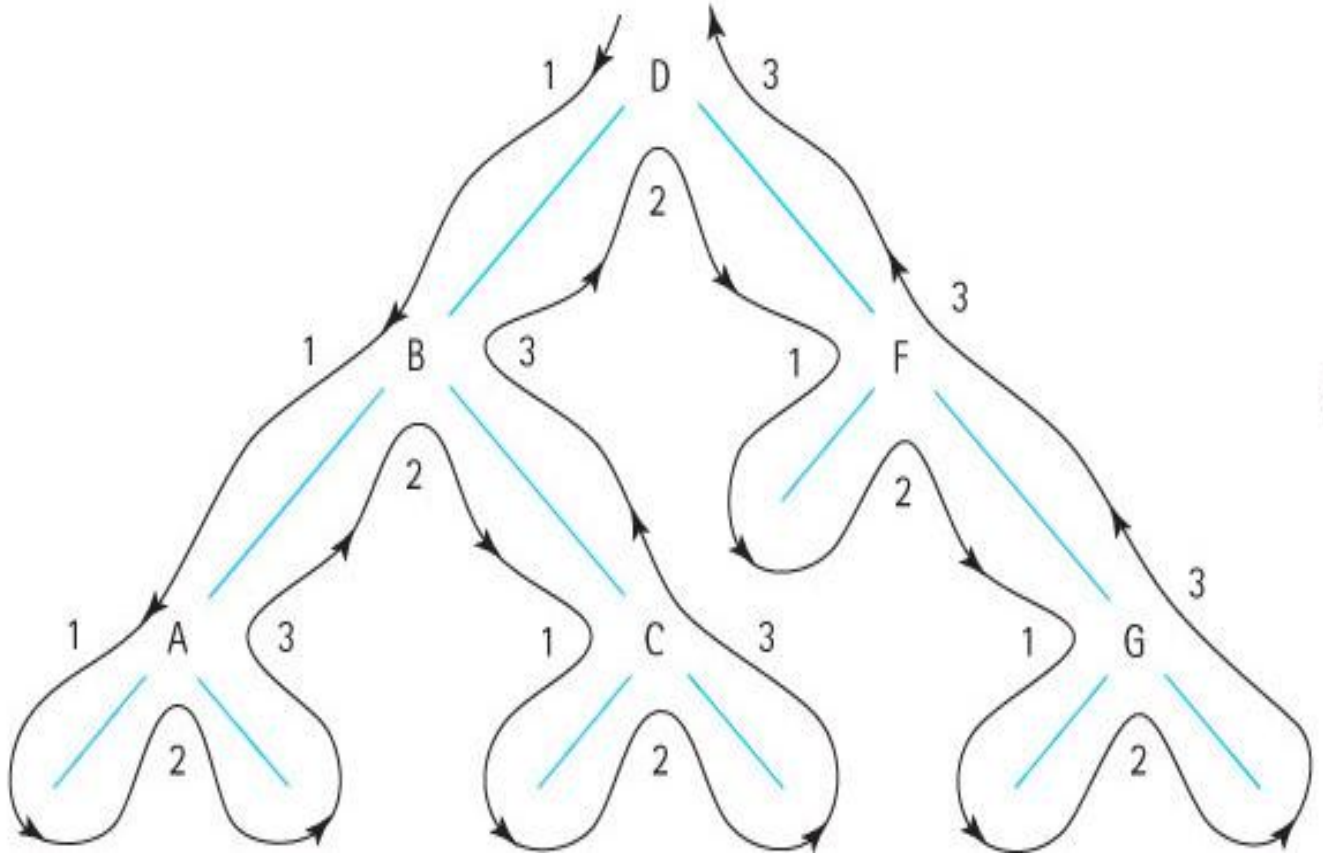
- **Preorder traversal**:
  - visit the root,
  - visit the left subtree,
  - visit the right subtree

- **Inorder traversal**:
  - visit the left subtree
  - visit the root
  - visit the right subtree

- **Postorder traversal**:
  - visit the left subtree
  - visit the right subtree
  - visit the root

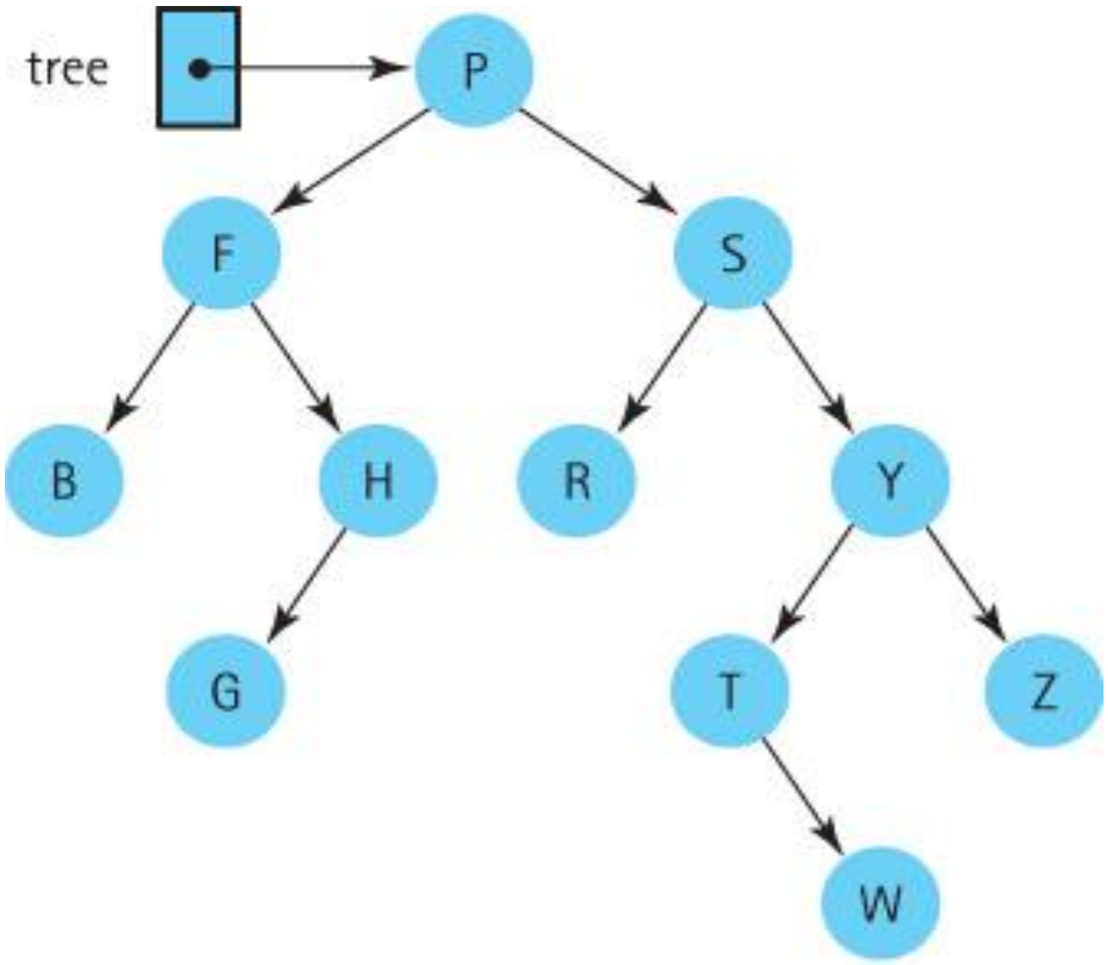
Visualizing Binary Tree Traversals

The extended tree



Preorder: DBACFG
Inorder: ABCDFG
Postorder: ACBGFD

Three Binary Tree Traversals



Inorder:
Preorder:
Postorder:

8.2 The Logical Level

- Specify our Binary Search Tree ADT.
- Use the Java interface construct to write the specification.
- Our binary search tree specification is very similar to our sorted list specification.

Design Decisions

- Duplicate elements are allowed
- Our binary search trees are unbounded
- Null elements are NOT allowed
- All three binary tree traversals are supported, with parameters used to indicate choice
- Binary search tree elements are objects of a class that implements the `Comparable` interface

```
// Returns a negative integer, zero, or a positive
// integer as this object is less than, equal to,
// or greater than the specified object.
public int compareTo(T o);
```

BSTInterface.java

- isEmpty
- size
- contains
- remove
- get
- add
- reset
- getNext

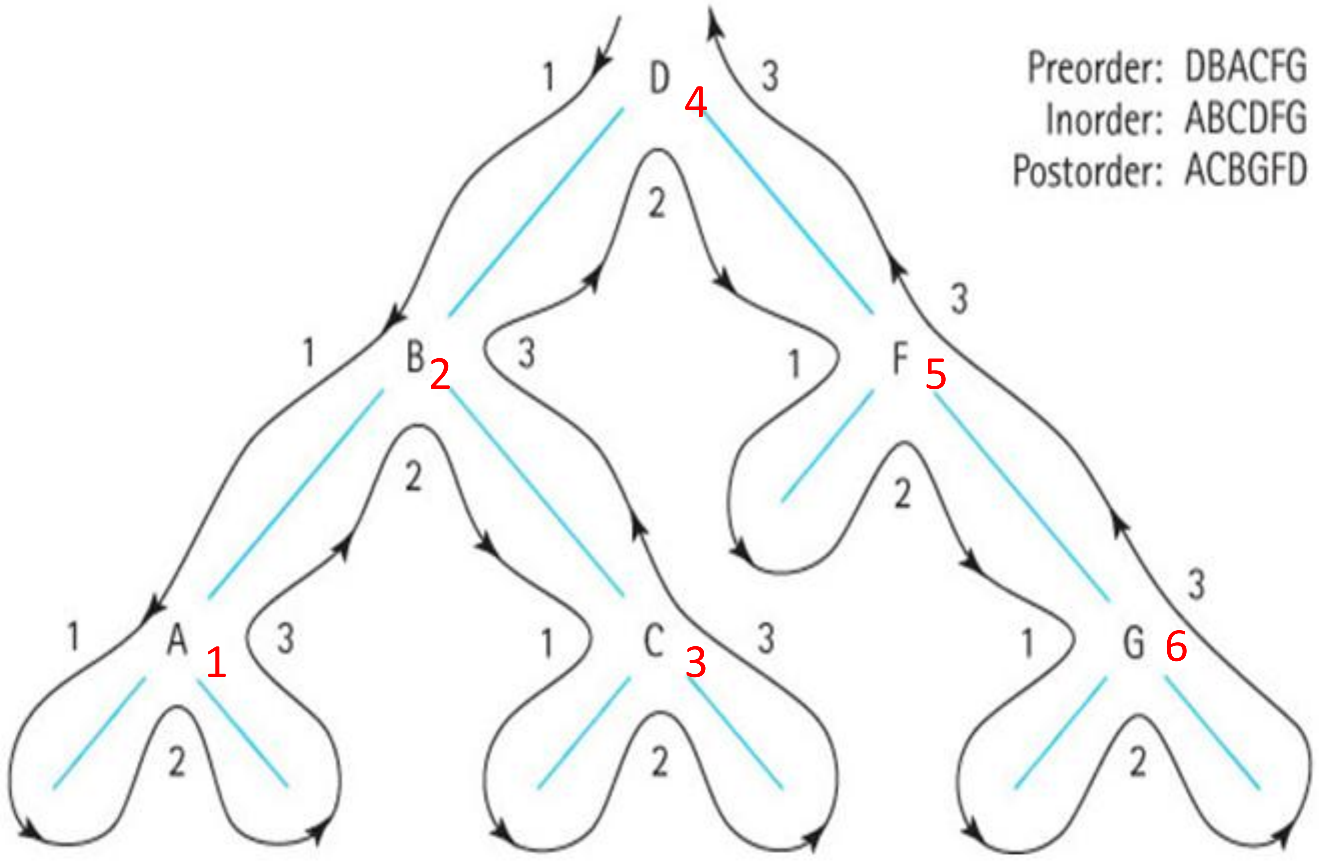
  - See BSTInterface.java
  - See BinarySearchTree.java

8.3 The Application Level

- Our Binary Search Tree ADT is very similar to our Sorted List ADT, from a functional point of view.
- How do we to use a binary search tree to support the golf score application originally presented in Section 6.5 where it used the Sorted List ADT.
- Demo GolfApp2.java
- Lab – use the GolfApp2.App to verify preOrder, inOrder and postOrder traversals. See next slide.

Test Pre-, In-, and Post-order with GolfApp2.java

Think about how the tree is made, i.e. which node is made first!!!



8.4 Implementation Level BinarySearchTree.java

```
package ch08.trees;

import ch05.queues.*;
import ch03.stacks.*;
import support.BSTNode;

public class BinarySearchTree<T extends Comparable<T>> implements BSTInterface<T>
{
  protected BSTNode<T> root;   // reference to the root of this BST
  boolean found;               // used by remove

  // for traversals
  protected LinkUnbndQueue<T> inOrderQueue;    // queue of info
  protected LinkUnbndQueue<T> preOrderQueue;   // queue of info
  protected LinkUnbndQueue<T> postOrderQueue;  // queue of info

  public BinarySearchTree()
  // Creates an empty BST object.
  {
    root = null;
  }
. . .
```

See BinarySearchTree.java
isEmpty is a simple Observer method.
What about the others?

8.5 Iterative vs. Recursive Method Invocations

- Trees are inherently recursive;
  - They consist of subtrees

- Look at recursive and iterative approaches to the `size` method

- Benefits of recursion versus iteration for this problem

Recursive Approach

- We create a public method, `size`, that calls a private recursive method, recSize and passes it a reference to the root of the tree.

```
// Returns the number of elements in this BST.
public int size()
{
  return recSize(root);
}
```

- We design the `recSize` method to return the number of nodes in the subtree referenced by the argument passed to it.
- Note that the number of nodes in a tree is:
  - *number of nodes in left subtree + number of nodes in right subtree + 1*

`recSize` Algorithm Version 1

```
***recSize(tree): returns int***
if (tree.getLeft( ) is null) AND (tree.getRight( ) is null)
   return 1
else
   return recSize(tree.getLeft( )) + recSize(tree.getRight( )) + 1
```

What is wrong with the above method?

- It crashes
- When does it crash?
  - Attempting to access `tree.left` when `tree` is `null`.

`recSize` Algorithm Version 2

```
recSize(tree): returns int
if (tree.getLeft( ) is null) AND (tree.getRight( ) is null)
   return 1
else if tree.getLeft( ) is null
   return recSize(tree.getRight( )) + 1
else if tree.getRight( ) is null
   return recSize(tree.getLeft( )) + 1
else return recSize(tree.getLeft( )) + recSize(tree.getRight( )) + 1
```

And now has the problem been solved?

No, An empty tree still causes a crash!

recSize Algorithm Version 3

```
recSize(tree): returns int    Version 3
if tree is null
   return 0
else if (tree.getLeft( ) is null) AND (tree.getRight( ) is null)
   return 1
else if tree.getLeft( ) is null
   return recSize(tree.getRight( )) + 1
else if tree.getRight( ) is null
   return recSize(tree.getLeft( )) + 1
```

Does this pseudocode solve the problem?

Yes, but the code can be simplified by collapsing the two base cases into one.

`recSize` Algorithm Version 4

```
***recSize(tree): returns int    Version 4***
if tree is null
   return 0
else
   return recSize(tree.getLeft( )) + recSize(tree.getRight( )) + 1
```

- Works and is “simple”.
- This example illustrates two important points about recursion with trees:
  - always check for the empty tree first
  - leaf nodes do not need to be treated as separate cases.

The `recSize` Code

```
// Returns the number of elements in tree.
private int recSize(BSTNode<T> tree)
{
  if (tree == null)
    return 0;
  else
    return recSize(tree.getLeft()) + recSize(tree.getRight()) + 1;
}
```

Iterative Version

- We use a stack to hold nodes we have encountered but not yet processed
- We must be careful that we process each node in the tree exactly once. We follow these rules:
  - Process a node immediately after removing it from the stack.
  - Do not process nodes at any other time.
  - Once a node is removed from the stack, do not push it back onto the stack.

Algorithm for the iterative approach

```
size returns int
Set count to 0
if the tree is not empty
    Instantiate a stack
    Push the root of the tree onto the stack
    while the stack is not empty
        Set currNode to top of stack
        Pop the stack
        Increment count
        if currNode has a left child
            Push currNode’s left child onto the stack
        if currNode has a right child
            Push currNode’s right child onto the stack
return count
```

Code for the iterative approach

```
// Returns the number of elements in this BST.
public int size()
{
  int count = 0;
  if (root != null)
  {
    LinkedStack<BSTNode<T>> hold = new LinkedStack<BSTNode<T>> ();
    BSTNode<T> currNode;
    hold.push(root);
    while (!hold.isEmpty())
    {
      currNode = hold.top();
      hold.pop();
      count++;
      if (currNode.getLeft() != null)
        hold.push(currNode.getLeft());
      if (currNode.getRight() != null)
        hold.push(currNode.getRight());
    }
  }
  return count;
}
```

Recursion or Iteration?

- *Is the depth of recursion relatively shallow?*
  - Yes
- *Is the recursive solution shorter or clearer than the nonrecursive version?*
  - Yes
- *Is the recursive version much less efficient than the nonrecursive version?*
  - No
- Is this a good use of recursion.
  - Yes

8.6 The Implementation Level: Remaining Operations

- In this section, we use recursion to implement the remaining Binary Search Tree operations
  - `contains`
  - `get`
  - `add`
  - `remove`
  - `iteration`

The `contains` operation

- We implement `contains` using a private recursive method called `recContains` which
  - is passed the element we are searching for and a reference to a subtree in which to search
  - checks to see if the element searched for is in the root
    - if not, compares the element with the root and look in either the left or the right subtree

The `contains` method

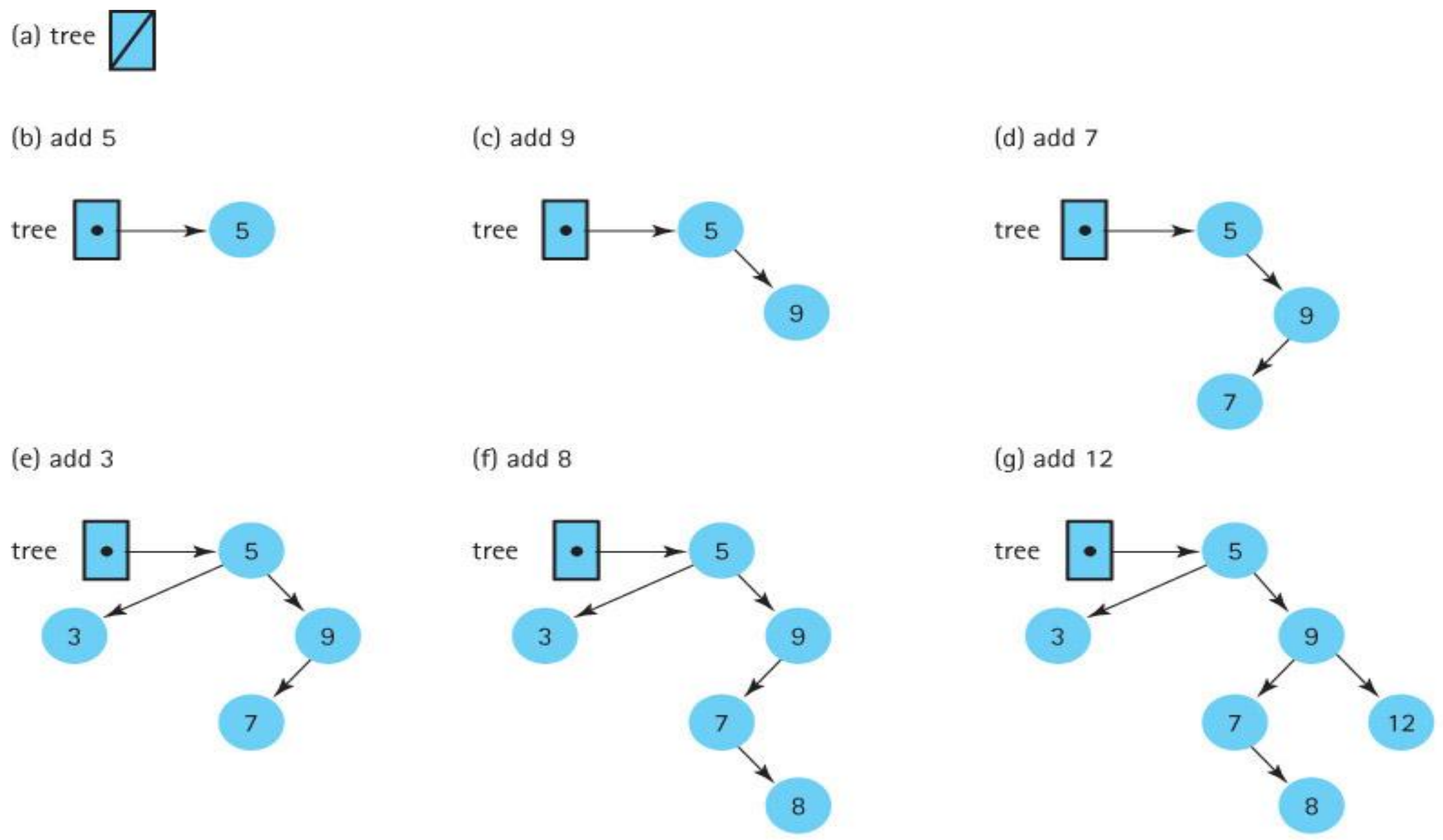
```
// Returns true if tree contains an element e such that
// e.equals(element), otherwise returns false.
private boolean recContains(T element, BSTNode<T> tree)
{
  if (tree == null)
    return false;       // element is not found
  else if (element.compareTo(tree.getInfo()) < 0)
    return recContains(element, tree.getLeft());   // Search left subtree
  else if (element.compareTo(tree.getInfo()) > 0)
    return recContains(element, tree.getRight());  // Search right subtree
  else
    return true;        // element is found
}

// Returns true if this BST contains an element e such that
// e.equals(element), otherwise returns false.
public boolean contains (T element)
{
  return recContains(element, root);
}
```

The `get` Method is very similar

```
// Returns an element e from tree such that e.equals(element);
// if no such element exists returns null.
private T recGet(T element, BSTNode<T> tree)
{
  if (tree == null)
    return null;             // element is not found
  else if (element.compareTo(tree.getInfo()) < 0)
    return recGet(element, tree.getLeft());          // get from left subtree
  else
  if (element.compareTo(tree.getInfo()) > 0)
    return recGet(element, tree.getRight());         // get from right subtree
  else
    return tree.getInfo();  // element is found
}

// Returns an element e from this BST such that e.equals(element);
// if no such element exists returns null.
public T get(T element)
{
  return recGet(element, root);
}
```

The `add` operation

- A new node is always inserted into its appropriate position in the tree as a leaf



The `add` operation

- The `add` method invokes the recursive method, `recAdd`, and passes it the element to be added plus a reference to the root of the tree.

```
// Adds element to this BST. The tree retains its BST property.
public void add (T element)
{
  root = recAdd(element, root);
}
```

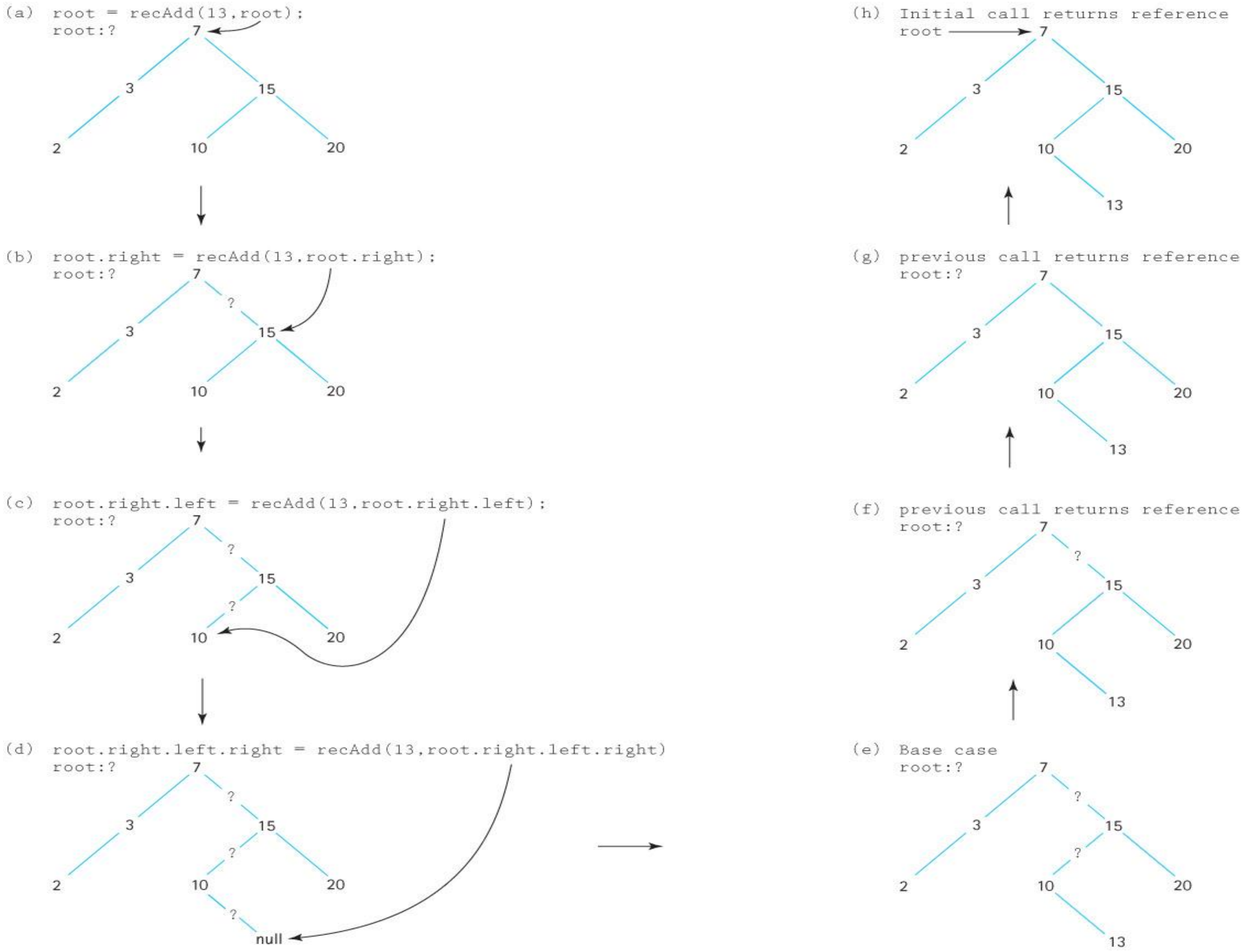
- The call to `recAdd` returns a `BSTNode`. It returns a reference to the new tree, that is, to the tree that includes element. The statement

  ```
  root = recAdd(element, root);
  ```

  can be interpreted as “Set the reference of the root of this tree to the root of the tree that is generated when element is added to this tree.”

The `add` method

```
// Adds element to tree; tree retains its BST property.
private BSTNode<T> recAdd(T element, BSTNode<T> tree)
{
  if (tree == null)
    // Addition place found
    tree = new BSTNode<T>(element);
  else if (element.compareTo(tree.getInfo()) <= 0)
    tree.setLeft(recAdd(element, tree.getLeft()));     // Add in left subtree
  else
    tree.setRight(recAdd(element, tree.getRight()));   // Add in right subtree
  return tree;
}

// Adds element to this BST. The tree retains its BST property.
public void add (T element)
{
  root = recAdd(element, root);
}
```


(a) root = recAdd(13,root);
root:?
(b) root.right = recAdd(13,root.right);
root:?
(c) root.right.left = recAdd(13,root.right.left);
root:?
(d) root.right.left.right = recAdd(13,root.right.left.right)
root:?
null
(e) Base case
root:?
(f) previous call returns reference
root:?
(g) previous call returns reference
root:?
(h) Initial call returns reference
root
7
3
15
2
10
20
13
?

Insertion Order and Tree Shape

tree

(a) Input: D B F A C E G

(b) Input: B A D C G F E

(c) Input: A B C D E F G

The `remove` Operation

- The most complicated of the binary search tree operations.
- We must ensure when we remove an element and maintain the binary search tree property.

The code for `remove`:

- The set up for the remove operation is the same as that for the add operation.
- The private `recRemove` method is invoked from the public `remove` method with arguments equal to the element to be removed and the subtree to remove it from.
- The recursive method returns a reference to the revised tree
- The `remove` method returns the boolean value stored in `found`, indicating the result of the remove operation.

```
// Removes an element e from this BST such that e.equals(element)
// and returns true; if no such element exists returns false.
public boolean remove (T element)
{
  root = recRemove(element, root);
  return found;
}
```

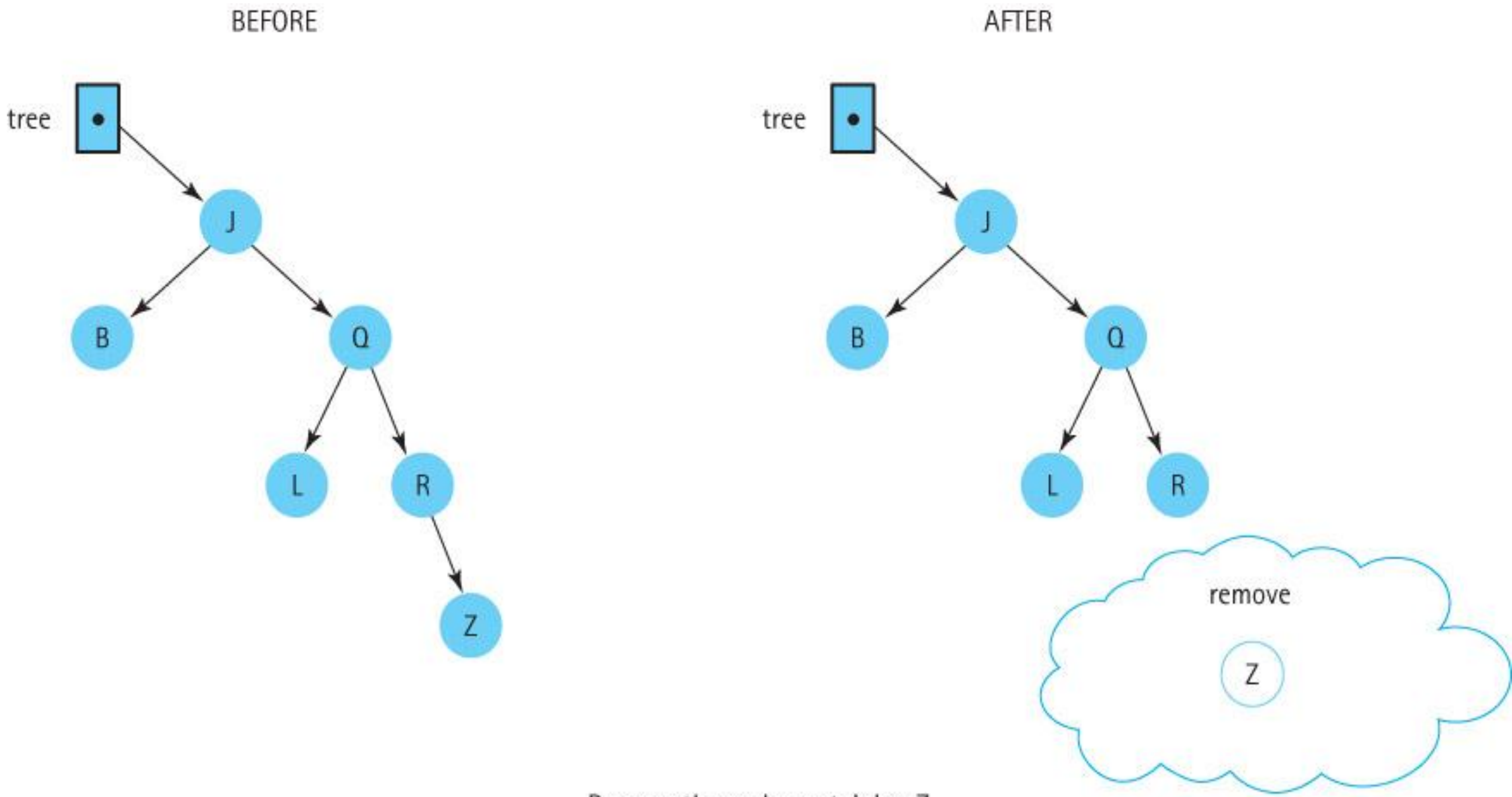
The `recRemove` method

```
// Removes an element e from tree such that e.equals(element)
// and returns true; if no such element exists returns false.
private BSTNode<T> recRemove(T element, BSTNode<T> tree)
{
  if (tree == null)
    found = false;
  else if (element.compareTo(tree.getInfo()) < 0)
    tree.setLeft(recRemove(element, tree.getLeft()));
  else if (element.compareTo(tree.getInfo()) > 0)
    tree.setRight(recRemove(element, tree.getRight()));
  else
  {
    tree = removeNode(tree);
    found = true;
  }
  return tree;
}
```

Three cases for the `removeNode` operation

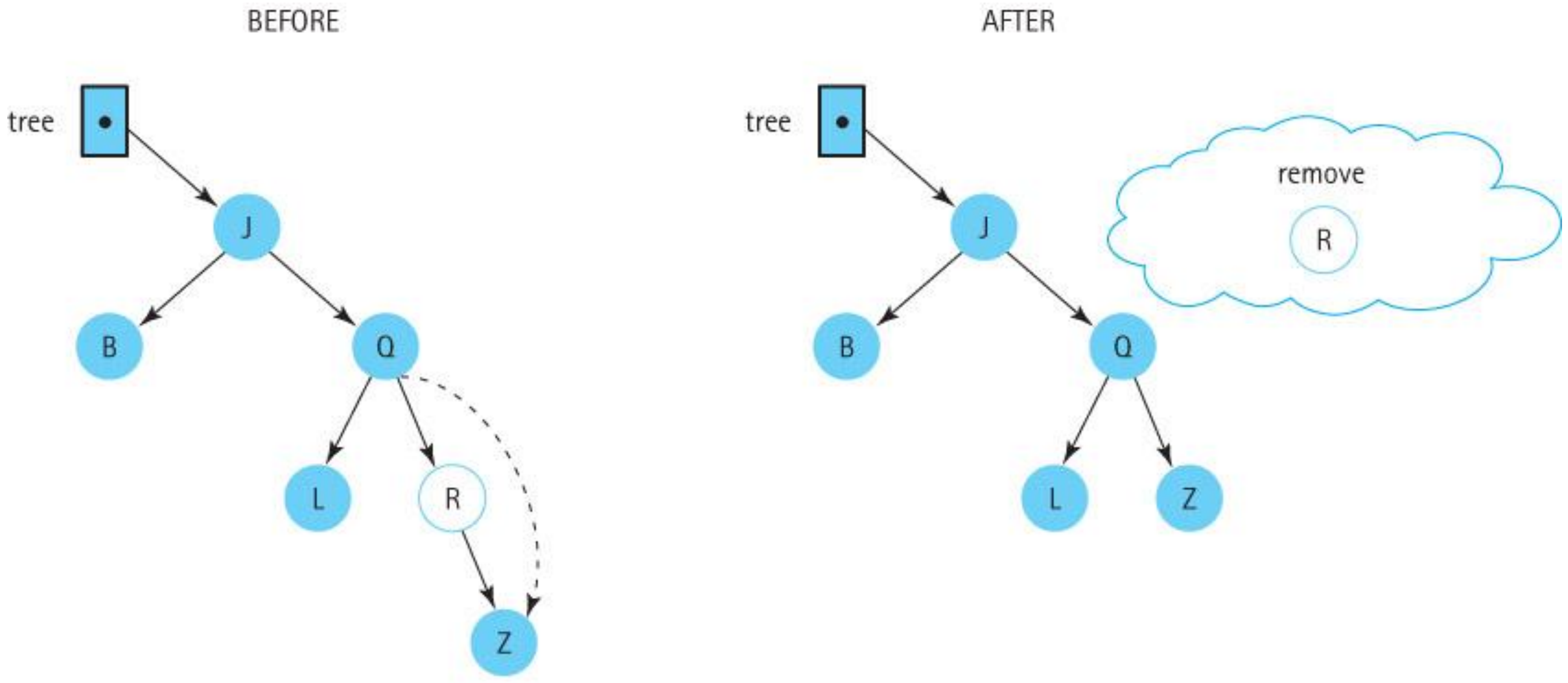
- Removing a leaf (no children): removing a leaf is simply a matter of setting the appropriate link of its parent to null.
- Removing a node with only one child: make the reference from the parent skip over the removed node and point instead to the child of the node we intend to remove
- Removing a node with two children: replaces the node’s info with the info from another node in the tree so that the search property is retained - then remove this other node

Removing a Leaf Node



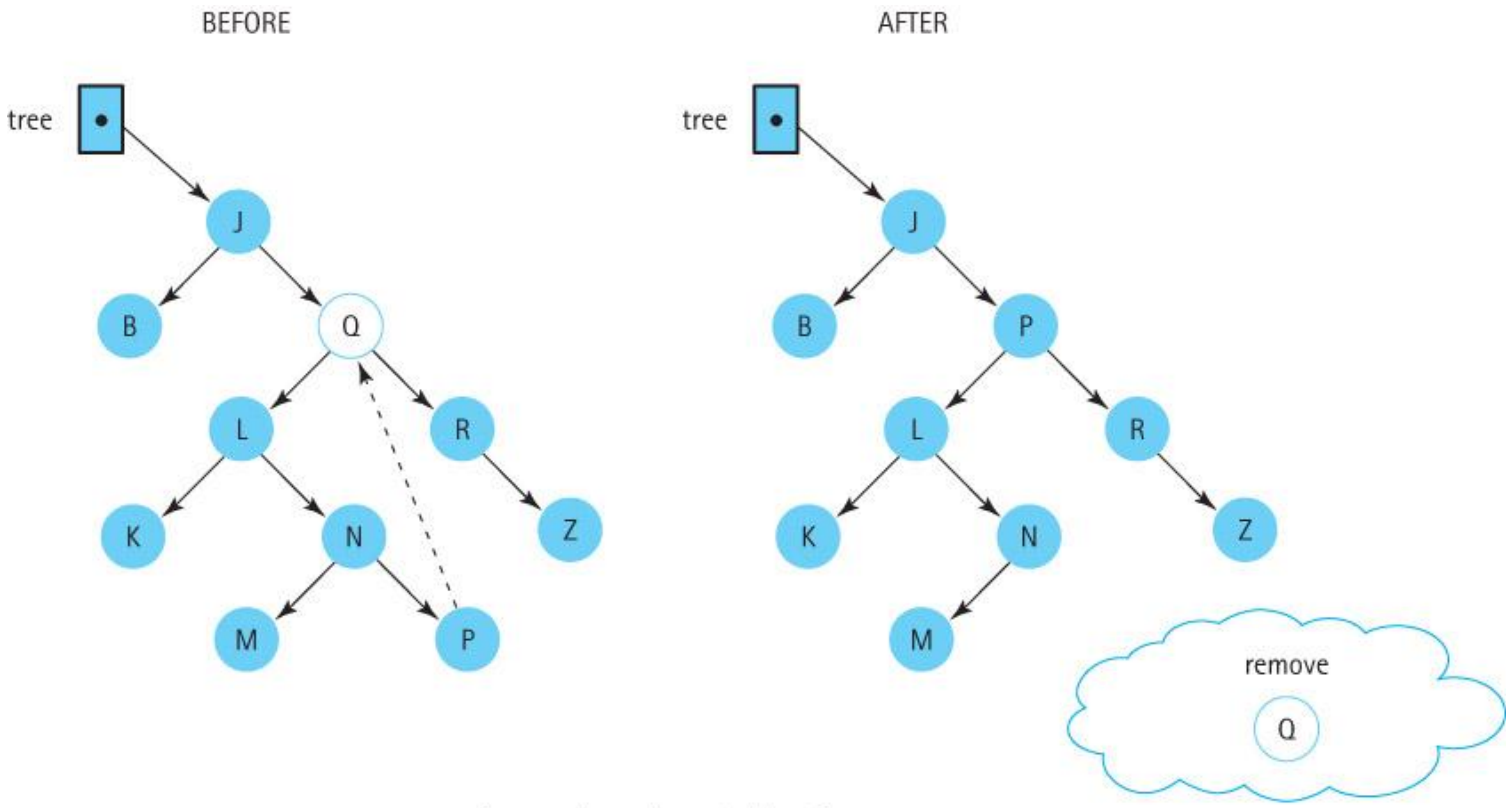
Remove the node containing Z

Removing a node with one child



Remove the node containing R

Removing a Node With Two Children



Remove the node containing Q

The Remove Node Algorithm

```
removeNode (tree): returns BSTNode
if (tree.getLeft( ) is null) AND (tree.getRight( ) is null)
   return null
else if tree.getLeft( ) is null
   return tree.getRight( )
else if tree.getRight( ) is null
   return tree.getLeft( )
else
   Find predecessor
   tree.setInfo(predecessor.getInfo( ))
   tree.setLeft(recRemove(predecessor.getInfo( ), tree.getLeft( )))
  return tree
```

Can this code be simplified?

Note: We can remove one of the tests if we notice that the action taken when the left child reference is null also takes care of the case in which both child references are null. When the left child reference is null, the right child reference is returned. If the right child reference is also null, then null is returned, which is what we want if they are both null.

The removeNode method

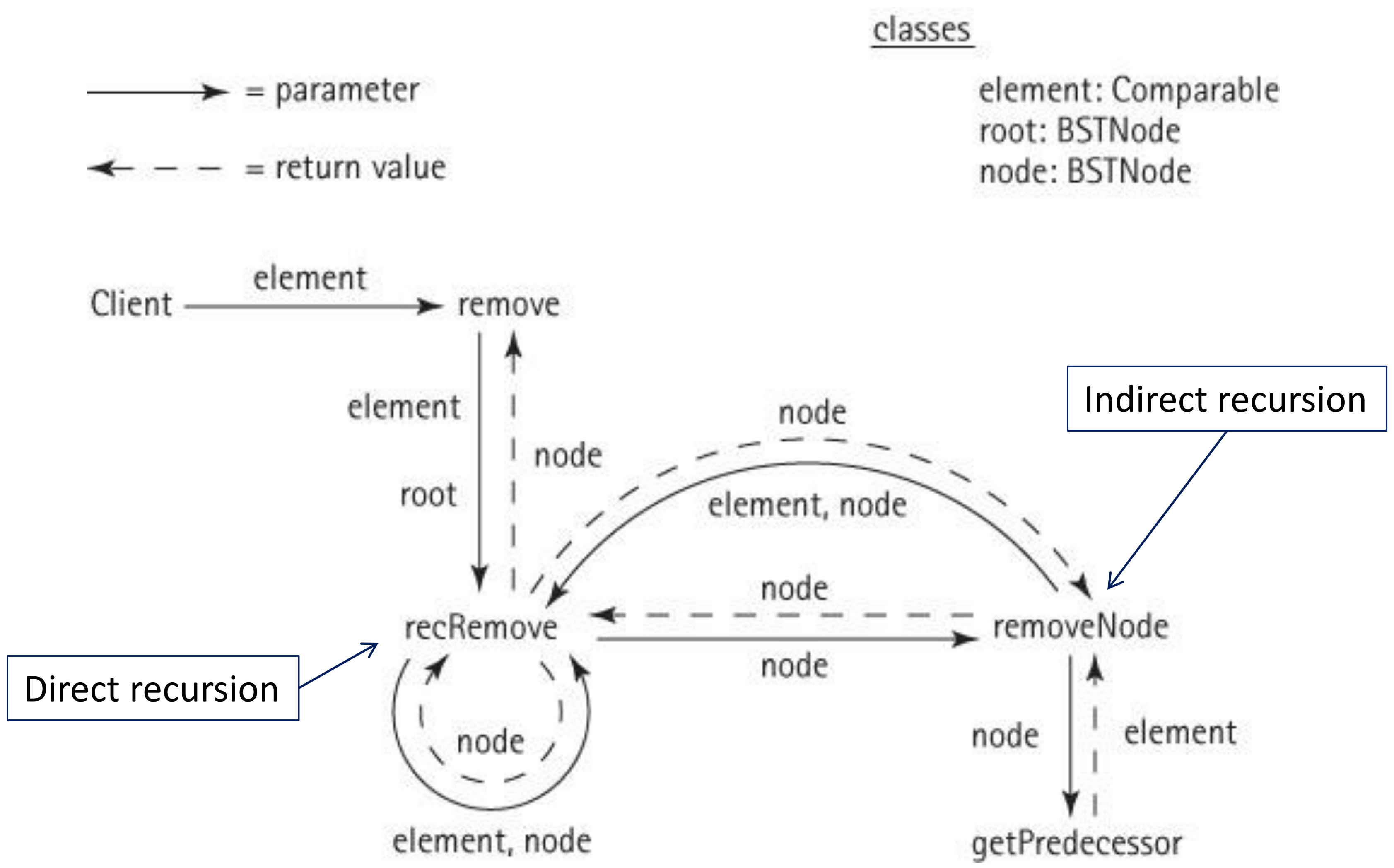
```
// Removes the information at the node referenced by tree
// The user's data in the node referenced to by tree is no
// longer in the tree.  If tree is a leaf node or has only
// a non-null child pointer, the node pointed to by tree is
// removed; otherwise, the user's data is replaced by its
// logical predecessor and the predecessor's node is removed
private BSTNode<T> removeNode(BSTNode<T> tree)
{
  T data;

  if (tree.getLeft() == null)
    return tree.getRight();
  else if (tree.getRight() == null)
    return tree.getLeft();
  else
  {
    data = getPredecessor(tree.getLeft());
    tree.setInfo(data);
    tree.setLeft(recRemove(data, tree.getLeft()));  // Remove predecessor node
    return tree;
  }
}
```

The `getPredecessor` method

- The logical predecessor is the maximum value in tree’s left subtree.
- The maximum value in a binary search tree is in its rightmost node.
- Therefore, given tree’s left subtree, we just keep moving right until the right child is null.
- When this occurs, we return the info reference of the node.

```
// Returns the information held in the rightmost node in tree
private T getPredecessor(BSTNode<T> tree)
{
  while (tree.getRight() != null)
    tree = tree.getRight();
  return tree.getInfo();
}
```

The Methods Used to Remove a Node



Iteration: Review of Traversal Definitions

- **Preorder traversal:**
  - visit the root
  - visit the left subtree
  - visit the right subtree

- **Inorder traversal:**
  - visit the left subtree
  - visit the root
  - visit the right subtree

- **Postorder traversal:**
  - visit the left subtree
  - visit the right subtree
  - visit the root

Our Iteration Approach

- The client program passes the `reset` and `getNext` methods a parameter indicating which of the three traversal orders to use

- `reset` generates a queue of node contents in the indicated order

Our Iteration Approach *(Cont'd)*

- `getNext` processes the node contents from the appropriate queue

- Each of the traversal orders is supported by a separate queue
  - `protected LinkedUnbndQueue<T> inOrderQueue;`
  - `protected LinkedUnbndQueue<T> preOrderQueue;`
  - `protected LinkedUnbndQueue<T> postOrderQueue;`

Our Iteration Approach *(Cont'd)*

- `reset` calls the `size` method in order to determine how large to make the required queue
- In most cases, a client program that invokes `reset` immediately requires the same information
- We make it easy for the client to obtain the number of nodes, by providing it as the return value of `reset`
- Note: when `getNext` reaches the end of the collection of elements another call to `getNext` results in a run time exception being thrown

The `reset` method

```
public int reset(int orderType)
// Initializes current position for an iteration through this BST
// in orderType order. Returns current number of nodes in the BST.
{
  int numNodes = size();
  if (orderType == INORDER)
  {
    inOrderQueue = new LinkedUnbndQueue<T>(numNodes);
    inOrder(root);
  }
  else
  if (orderType == PREORDER)
  {
    preOrderQueue = new LinkedUnbndQueue<T>(numNodes);
    preOrder(root);
  }
  if (orderType == POSTORDER)
  {
    postOrderQueue = new LinkedUnbndQueue<T>(numNodes);
    postOrder(root);
  }
  return numNodes;
}
```

The `getNext` method

```
// Preconditions: the BST is not empty
//                the BST has been reset for orderType
//                the BST has not been modified since the most recent reset
//                the end of orderType iteration has not been reached
// Returns the element at the current position on this BST and advances
// the value of the current position based on the orderType set by reset.
public T getNext (int orderType)
{
  if (orderType == INORDER)
    return inOrderQueue.dequeue();
  else
  if (orderType == PREORDER)
    return preOrderQueue.dequeue();
  else
  if (orderType == POSTORDER)
    return postOrderQueue.dequeue();
  else return null;
}
```

The Definition of the inOrder Traversal

Method inOrder (tree)
Definition: Enqueues the elements in the binary search tree in order from smallest to largest.

Size:	The number of nodes in the tree whose root is tree
Base Case:	If tree = null, do nothing.
General Case:	Traverse the left subtree in order. Then enqueue tree.getInfo(). Then traverse the right subtree in order.

The `inOrder` method

```
// Initializes inOrderQueue with tree elements in inOrder order
private void inOrder(BSTNode<T> tree)
{
  if (tree != null)
  {
    inOrder(tree.getLeft());
    inOrderQueue.enqueue(tree.getInfo());
    inOrder(tree.getRight());
  }
}
```

The remaining two traversals are approached exactly the same, except the relative order in which they visit the root and the subtrees is changed.

Recursion provides an elegant solution to the binary tree traversal problem.

Testing the Binary Search Tree operations

- The code for the entire `BinarySearchTree` class is included with the rest of the book files.
- It provides both the recursive `size` method, and the iterative version (`size2`).
- We have also included an interactive test driver for the ADT called `ITDBinarySearchTree`.
  - it allows you to create, manipulate, and observe trees containing strings.
  - it also supports a print operation that allows you to indicate one of the traversal orders and “prints” the contents of the tree, in that order.
- You should use the test driver to test the various tree operations.

8.7 Comparing Binary Search Trees to Linear Lists

	Binary Search Tree	**Array-Based Linear List**	**Linked List**
Class constructor	O(1)	O(N)	O(1)
`isEmpty`	O(1)	O(1)	O(1)
`reset`	O(*N*)	O(1)	O(1)
`getNext`	O(1)	O(1)	O(1)
`contains`	$O(\log_2 N)$	$O(\log_2 N)$	O(*N*)
`get`			
Find	$O(\log_2 N)$	$O(\log_2 N)$	O(*N*)
Process	O(1)	O(1)	O(1)
Total	$O(\log_2 N)$	$O(\log_2 N)$	O(*N*)
`add`			
Find	$O(\log_2 N)$	$O(\log_2 N)$	O(*N*)
Process	O(1)	O(*N*)	O(*N*)
Total	$O(\log_2 N)$	O(*N*)	O(*N*)
`remove`			
Find	$O(\log_2 N)$	$O(\log_2 N)$	O(*N*)
Process	O(1)	O(*N*)	O(1)
Total	$O(\log_2 N)$	O(*N*)	O(*N*)

8.8 Balancing a Binary Search Tree

- In our Big-O analysis of binary search tree operations we assumed our tree was balanced.
- If this assumption is dropped and if we perform a worst-case analysis assuming a completely skewed tree, the efficiency benefits of the binary search tree disappear.
- The time required to perform the contains, get, add, and remove operations is now O(*N*), just as it is for the linked list.

Balancing a Binary Search Tree

- A beneficial addition to our Binary Search Tree ADT operations is a `balance` operation

- The specification of the operation is:

```
public balance();
// Restructures this BST to be optimally balanced
```

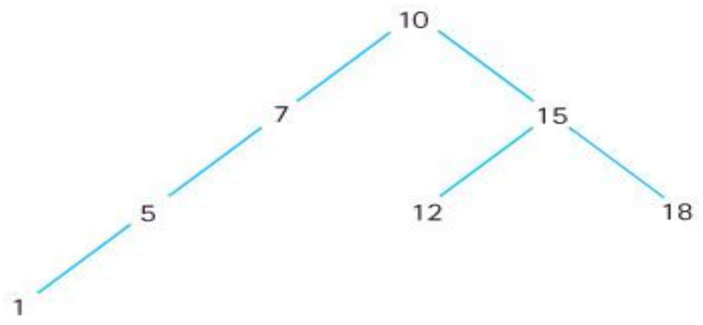
- It is up to the client program to use the `balance` method appropriately

Our Approach

- Basic algorithm:
  - Save the tree information in an array
  - Insert the information from the array back into the tree
- The structure of the new tree depends on the order that we save the information into the array, or the order in which we insert the information back into the tree, or both
- First assume we insert the array elements back into the tree in “index” order

Using inOrder traversal

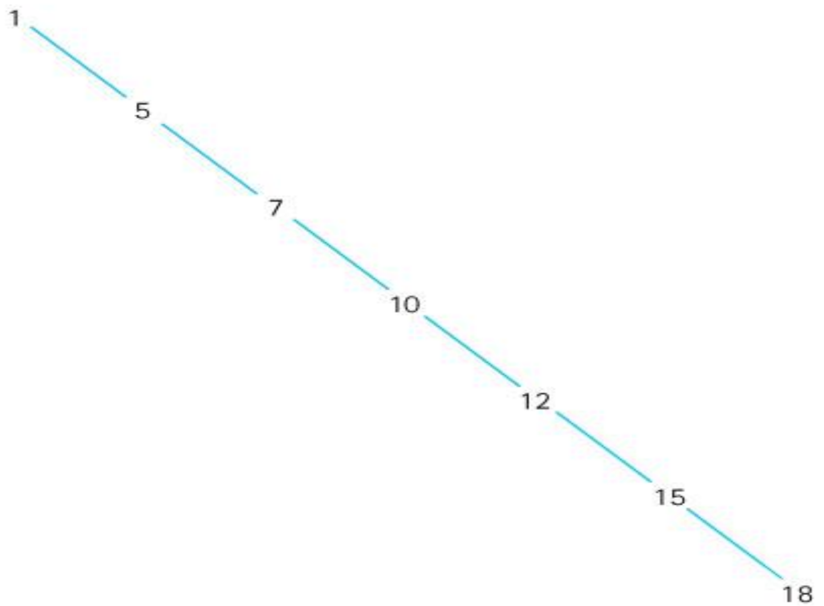
(a) The original tree



(b) The inorder traversal

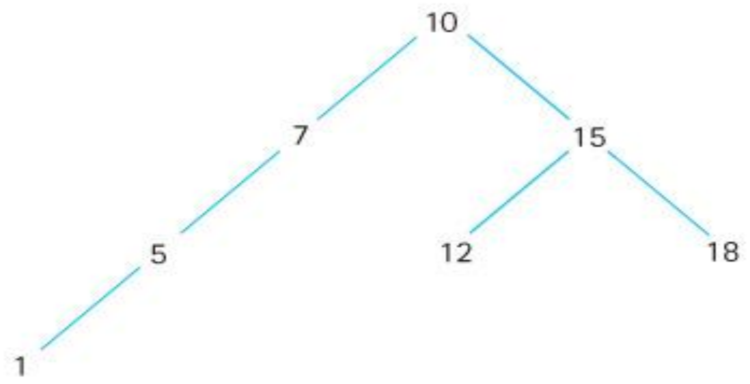
	0	1	2	3	4	5	6
array:	1	5	7	10	12	15	18

(c) The resultant tree if linear traversal of array is used



Using preOrder traversal

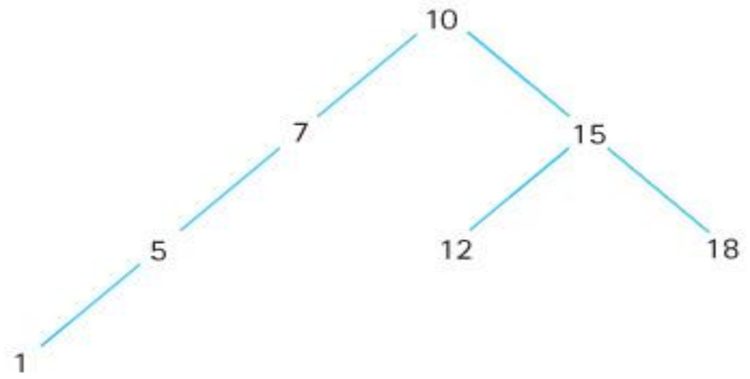
(a) The original tree



(b) The preorder traversal

array:	0	1	2	3	4	5	6
	10	7	5	1	15	12	18

(c) The resultant tree if linear traversal of array is used



To Ensure a Balanced Tree

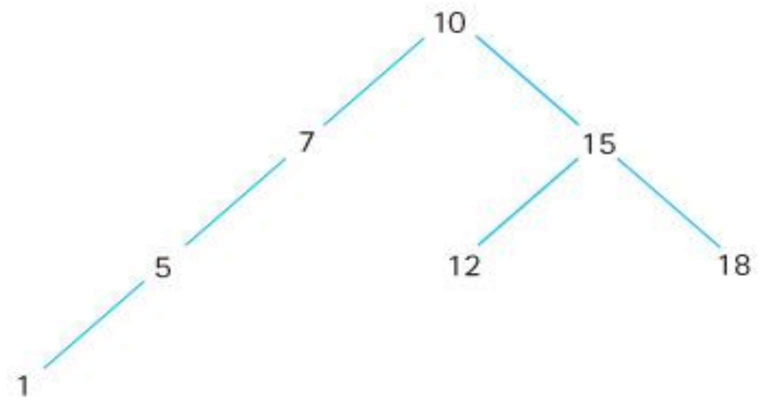
- Even out as much as possible, the number of descendants in each node’s left and right subtrees
- First insert the “middle” item of the inOrder array
  - Then insert the left half of the array using the same approach
  - Then insert the right half of the array using the same approach

Our Balance Tree Algorithm

```
Balance
Set count to tree.reset(INORDER).
For (int index = 0; index < count; index++)
   Set array[index] = tree.getNext(INORDER).
tree = new BinarySearchTree().
tree.InsertTree(0, count - 1)

InsertTree(low, high)
if (low == high)                    // Base case 1
   tree.add(nodes[low]).
else if ((low + 1) == high)         // Base case 2
   tree.add(nodes[low]).
   tree.add(nodes[high]).
else
   mid = (low + high) / 2.
   tree.add(mid).
   tree.InsertTree(low, mid – 1).
   tree.InsertTree(mid + 1, high).
```

Using recursive `insertTree`

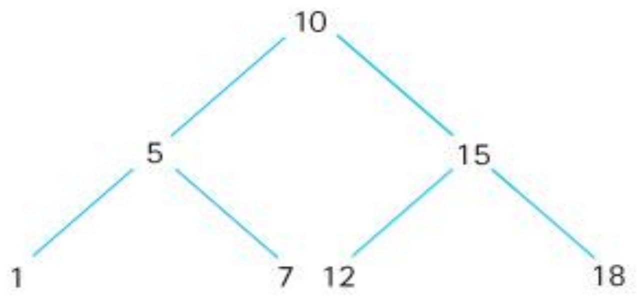
(a) The original tree



(b) The inorder traversal

	0	1	2	3	4	5	6
array:	1	5	7	10	12	15	18

(c) The resultant tree if InsertTree (0,6) is used



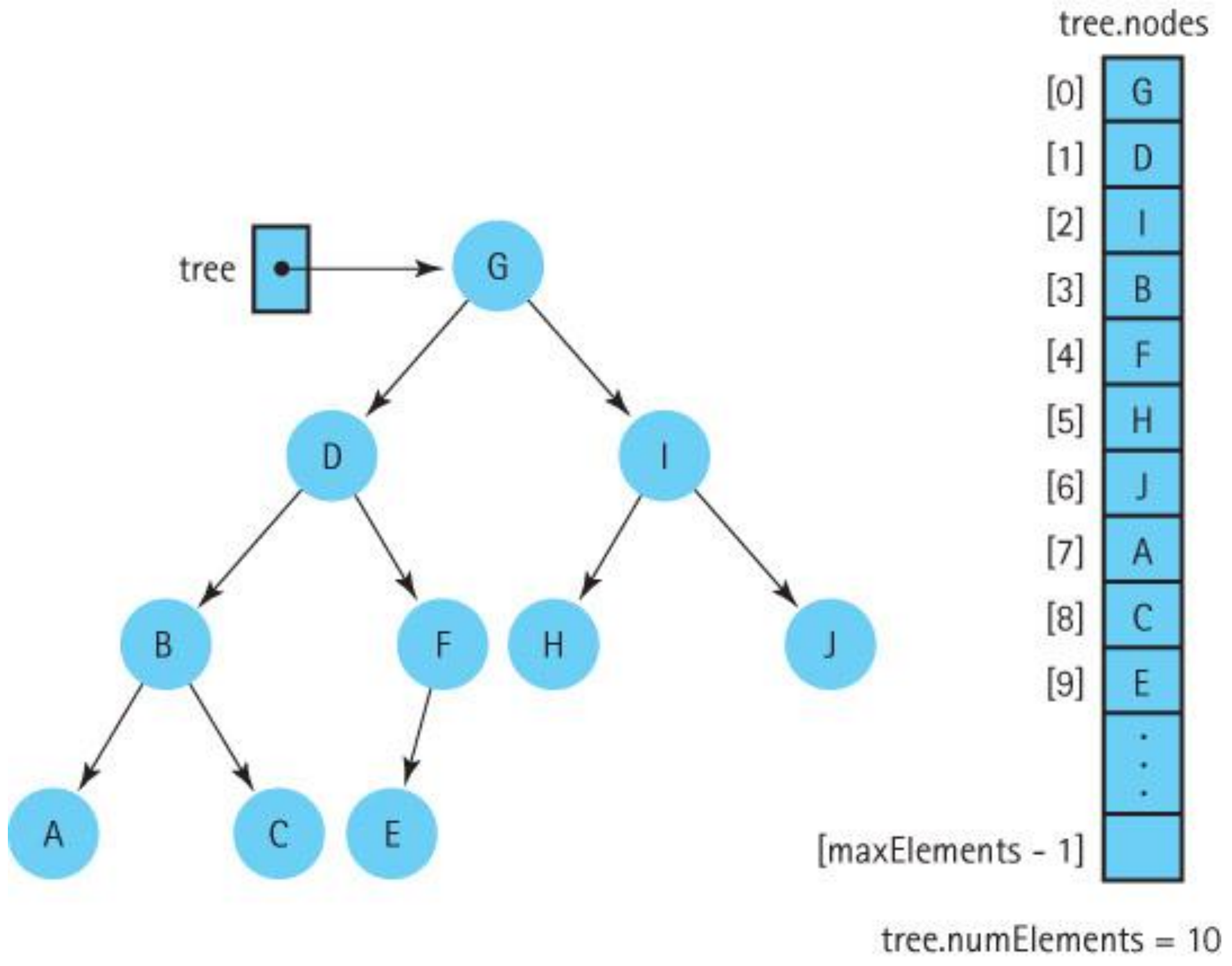
8.9 A Nonlinked Representation of Binary Trees

- A binary tree can be stored in an array in such a way that the relationships in the tree are not physically represented by link members, but are *implicit* in the algorithms that manipulate the tree stored in the array.

- We store the tree elements in the array, level by level, left-to-right. If the number of nodes in the tree is `numElements`, we can package the array and `numElements` into an object

- The tree elements are stored with the root in tree.nodes[0] and the last node in tree.nodes[numElements – 1].

A Binary Tree and Its Array Representation

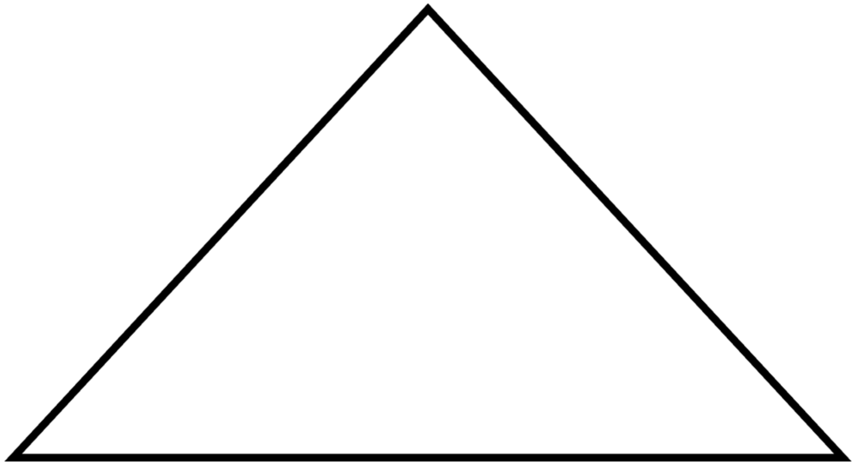


Array Representation continued

- To implement the algorithms that manipulate the tree, we must be able to find the left and right child of a node in the tree:
  - tree.nodes[index] left child is in tree.nodes[index*2 + 1]
  - tree.nodes[index] right child is in tree.nodes[index*2 + 2]
- We can also can determine the location of its parent node:
  - tree.nodes[index]'s parent is in tree.nodes[(index – 1)/2].
- This representation works best, space wise, if the tree is complete

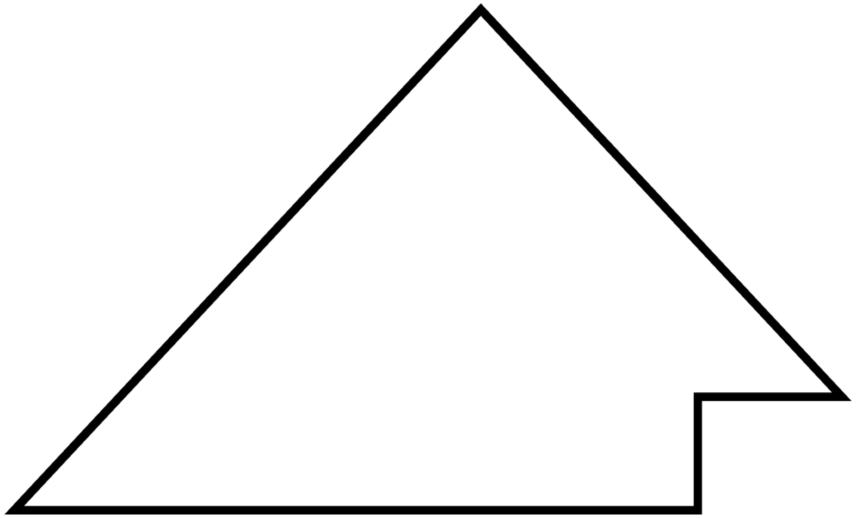
Definitions

- **Full Binary Tree:** A binary tree in which all of the leaves are on the same level and every nonleaf node has two children

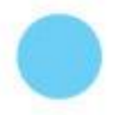


Definitions *(Cont'd)*

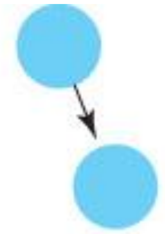
- **Complete Binary Tree:** A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible



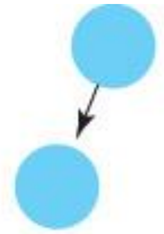
Examples of Different Types of Binary Trees



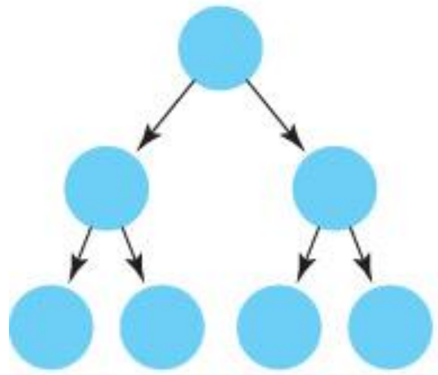
(a) Full and complete



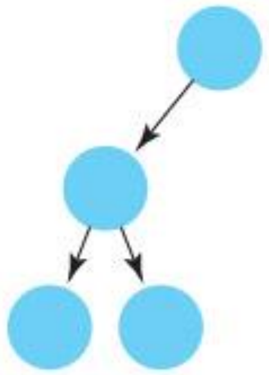
(b) Neither full nor complete



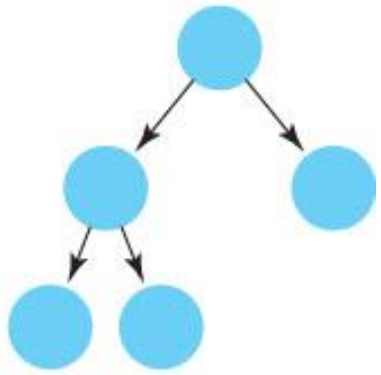
(c) Complete



(d) Full and complete

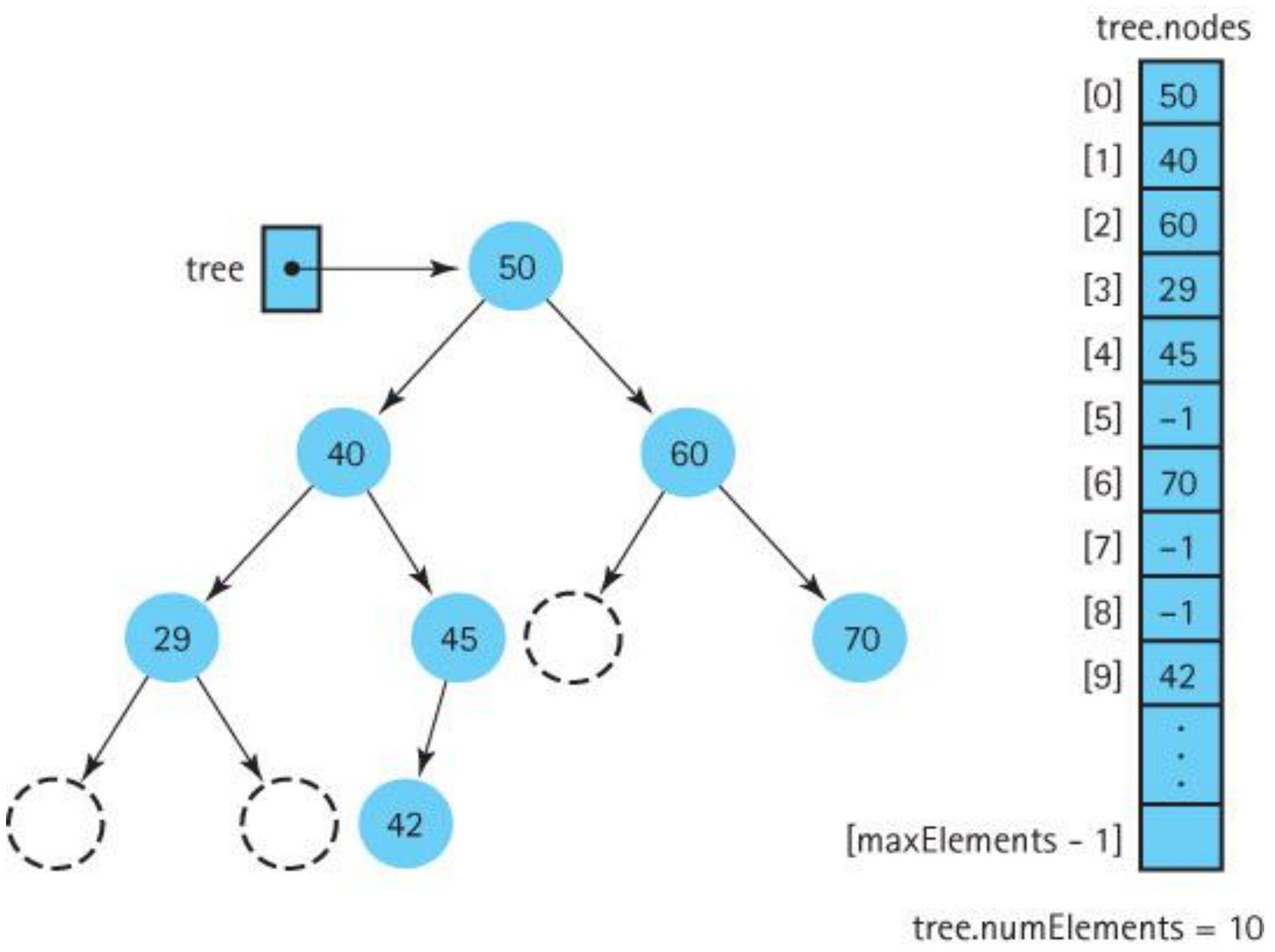


(e) Neither full nor complete



(f) Complete

A Binary Search Tree Stored in an Array with Dummy Values



16. Write a client method that returns the reference to the information in the node with the “smallest” value in the binary search tree.

17. Write a client method that returns the reference to the information in the node with the “largest” value in the binary search tree.

20. Extend the Binary Search Tree ADT to include the public method getRoot that returns a reference to the root of the tree. If the tree is empty, the method should return null.

Extend the Binary Search Tree ADT to include the public leafCount that returns the number of leaf nodes in the tree.

HW

- HW Chapter 8 questions: 8, 44, 48, 49, 51
- HW Chapter 8 coding question: 47

- Benchmark and analysis using methods discussed in this chapter will be required for the final project.