CS 304 Lecture 8

Binary search trees

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

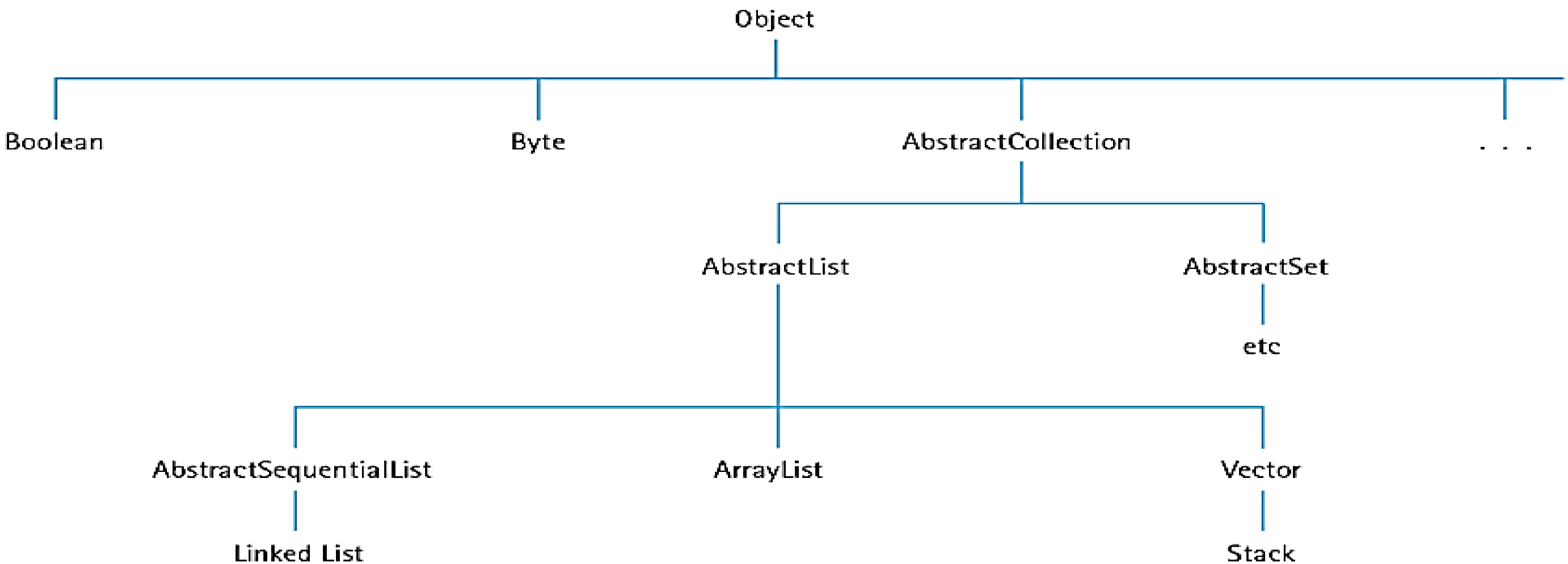
Northeastern Illinois University

Chicago, Illinois, 60625

27 October 2016

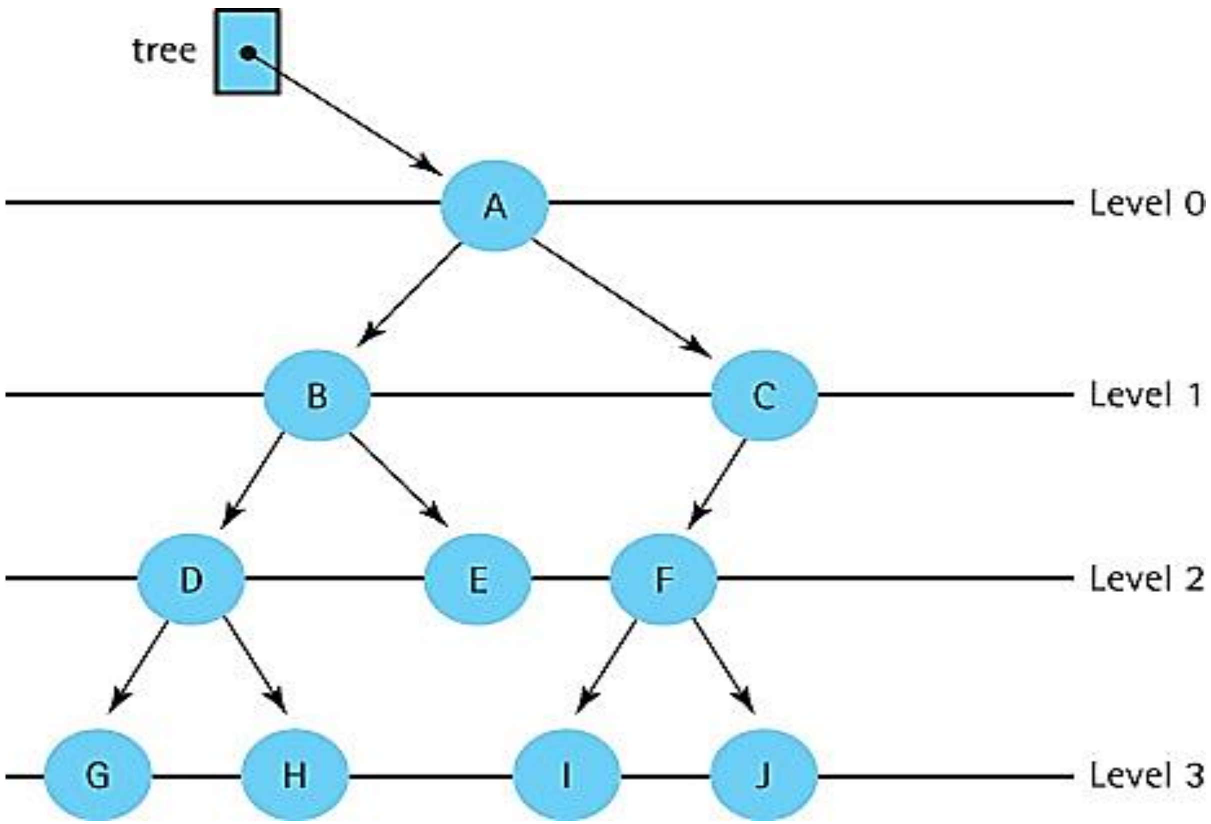
Trees

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



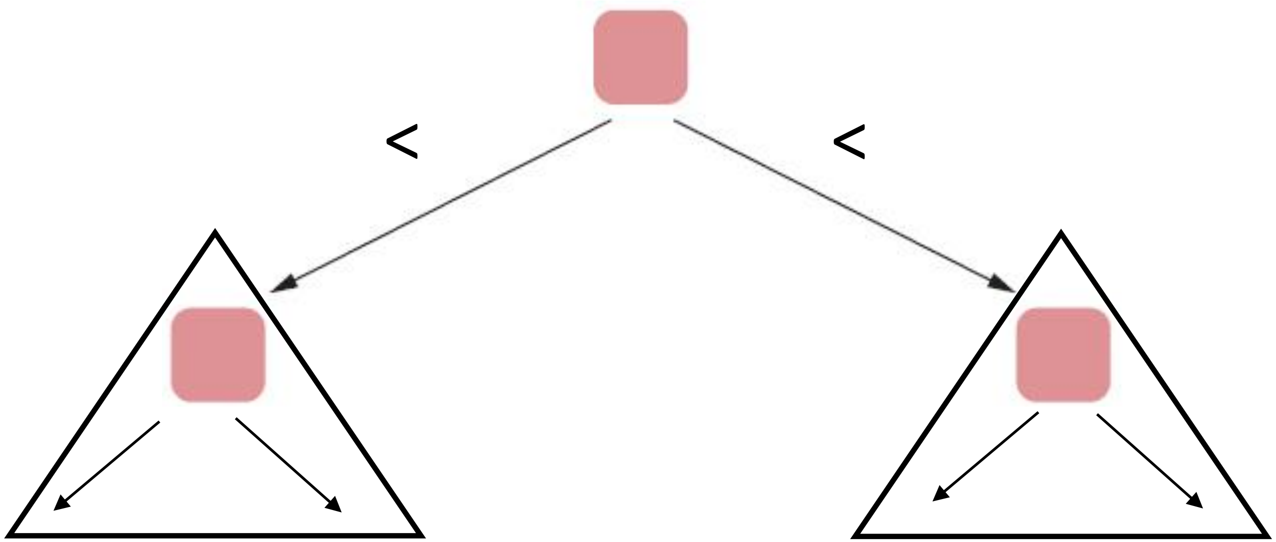
Binary trees

- **Binary tree** - A tree in which each node is capable of having two child nodes, a left child node and a right child node.
- **Leaf** - A tree node that has no children.

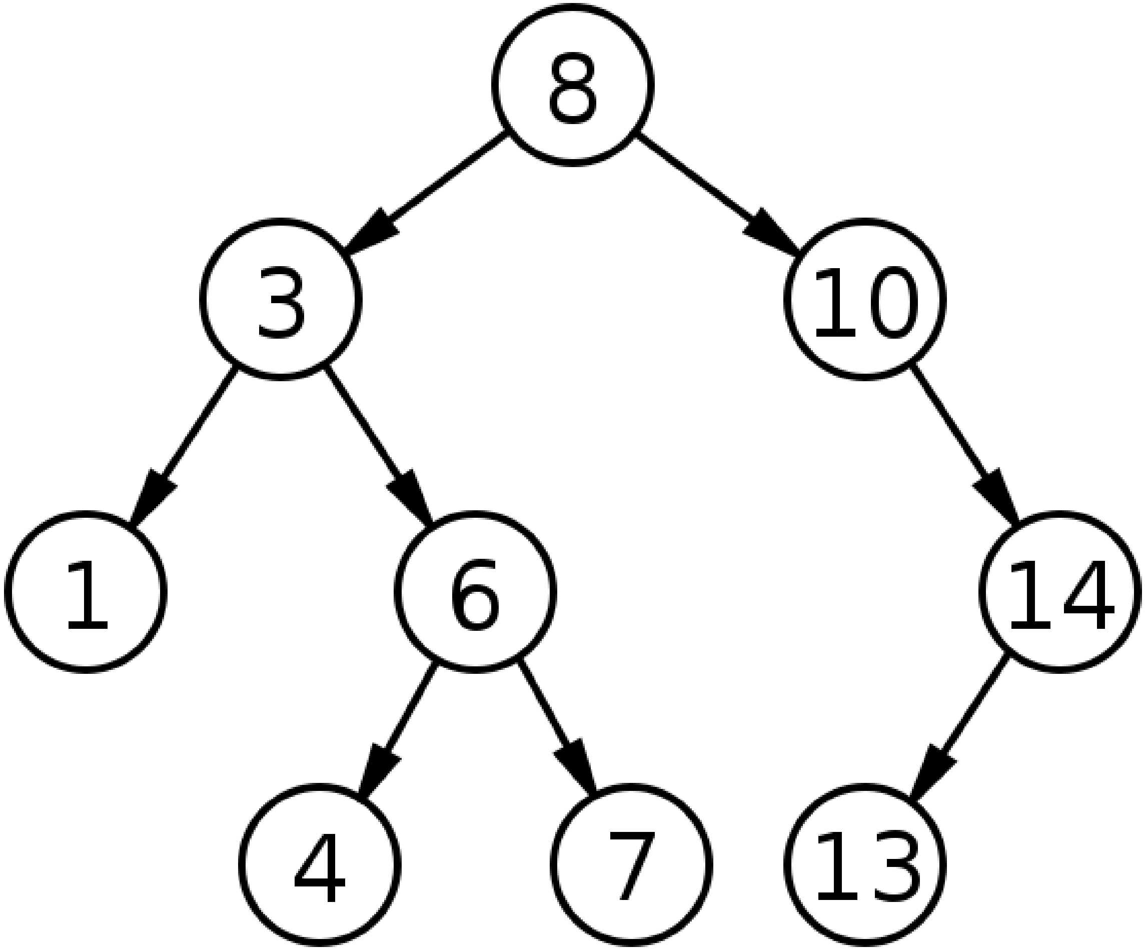


Binary search trees

- **Binary search tree** - A special kind of binary tree, used for quick lookup.
- A binary search tree maintains this property:
  - The data values of all descendants to the left of any node are less than the data value stored in that node, and all descendants to the right have greater data values.



Binary search trees

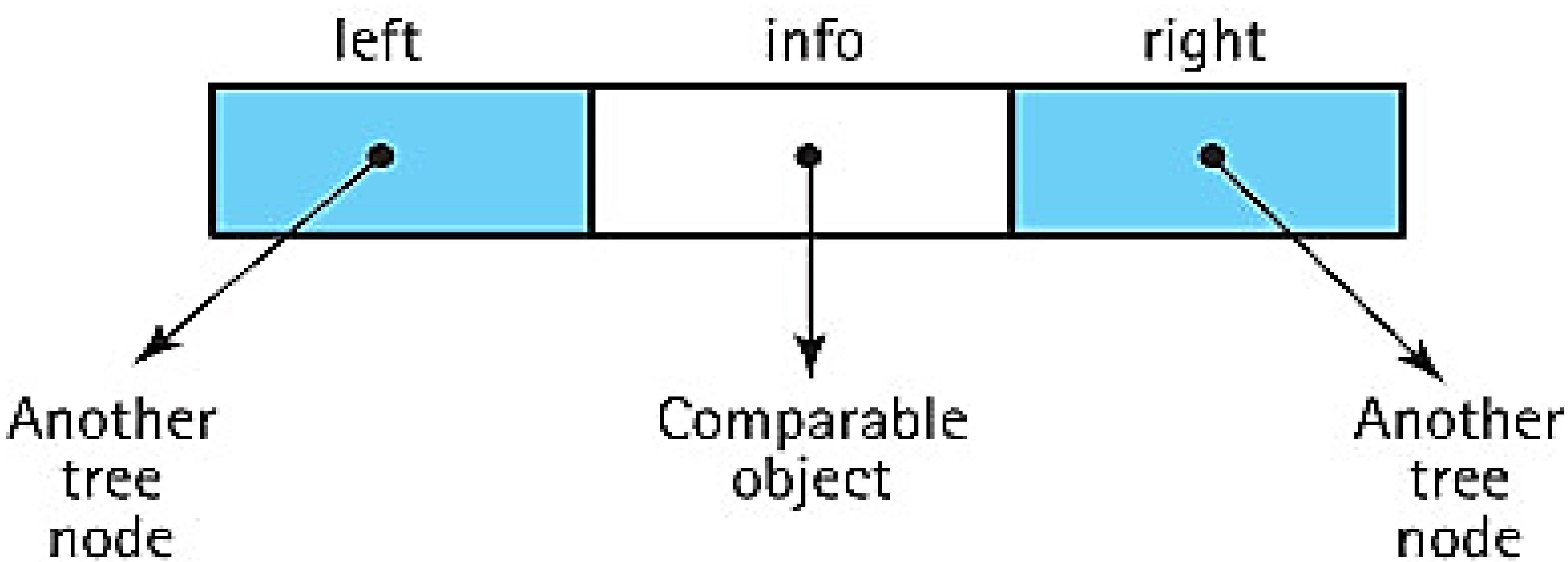


BST implementation

- The BST ADT should support the following operations:
  - Adding a node into the tree, retaining the BST property;
  - Removing a node from the tree, retaining the BST property;
  - Checking for the existence of a node;
  - Counting the number of nodes in the tree;
  - Traversing the tree.

BST implementation

- The BST implementation makes use of the **`BSTNode`** class.
- Visually, a **`BSTNode`** object is:



BST implementation

- 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.

BST implementation

- The incomplete **BinarySearchTree** class

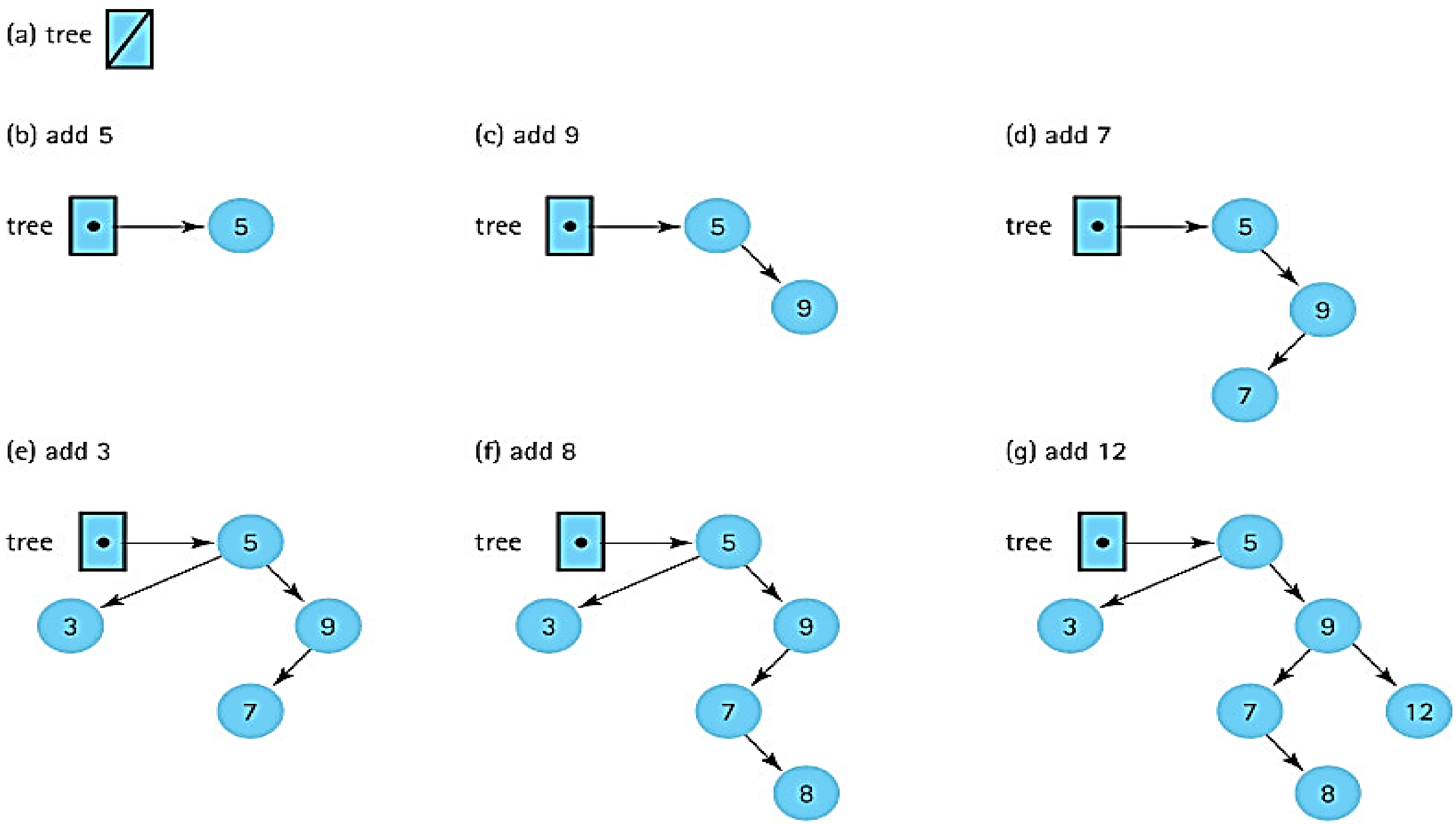
```
public class BinarySearchTree<T>
{
   protected BSTNode<T> root; // reference to the root of the BST

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

   boolean isEmpty();  // Returns true if this BST is empty;
                       // otherwise, returns false.
   int size();  // Returns the number of elements in this BST.
   boolean contains (T element); // Checks if this BST contains
                                 // the element.
   boolean remove (T element);   // Removes an element.
   void add (T element);  // Adds an element to this BST. The
                          // tree retains its BST property.
}
```

The `add` method

- First off, let us think about how to add a node into a BST.
- Just keep in mind - a new node is always inserted into its appropriate position in the tree as a leaf.



The **add** method

- 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.

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

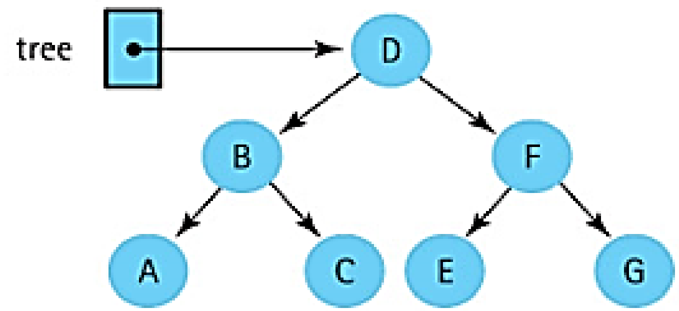
can be interpreted as "Set the reference of the root of this tree to the root of the tree that is generated when element is added to this tree."

The `add` method

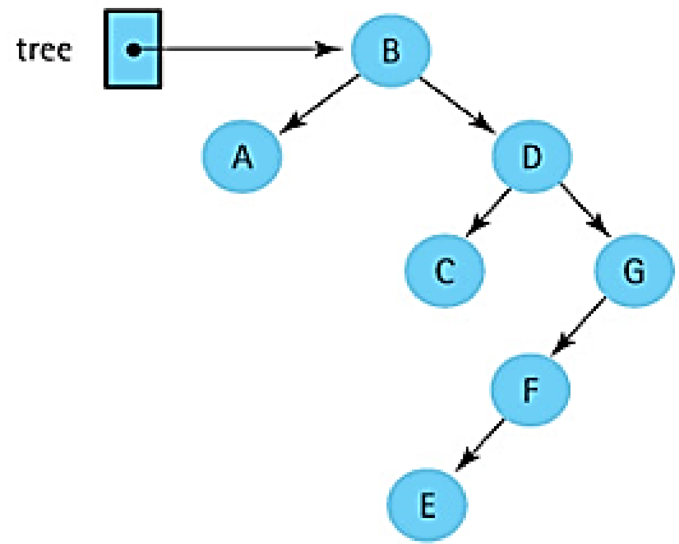
```
private BSTNode<T> recAdd(T element, BSTNode<T> tree)
// Adds element to tree; tree retains its BST property.
{
  if (tree == null)
    // Addition place found
    tree = new BSTNode<T>(element);
  else if (element.compareTo(tree.getInfo()) <= 0)
    // Add in the left subtree
    tree.setLeft(recAdd(element,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

The `add` method

- Insertion order and tree shape



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



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

The `size` method

- Trees are inherently recursive; a tree consists of subtrees.
- We create a public method, **`size`**, that calls a *private* recursive method, **`recSize`** and passes it a reference to the *root* of the tree.

```
public int size()
// Returns the number of elements in this BST.
{
  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 = the number of nodes in left subtree + the number of nodes in right subtree + 1

The `size` method

- `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
```

- The corresponding method would crash in three cases:
  - When **`tree`** is **`null`** and we try to access **`tree.left`** or **`tree.right`**.
  - When **`tree.left`** is **`null`** but **`tree.right`** is not **`null`** and **`recSize(tree.getLeft())`** is invoked.
  - When **`tree.right`** is **`null`** but **`tree.left`** is not **`null`** and **`recSize(tree.getRight())`** is invoked.

The `size` method

- `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
```

- An initially empty tree still causes a crash.

The **size** method

- **recSize** Algorithm - version 3

```
recSize(tree): returns int
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
else
    return recSize(tree.getLeft()) + recSize(tree.getRight())
           + 1
```

- Works, but can be simplified. We can collapse the two base cases into one. There is no need to make the leaf node a special case.

The `size` method

- `recSize` Algorithm - version 4

```
recSize(tree): returns int
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 `size` method

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

The `size` method

- The iterative version uses a stack to hold nodes it has 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.

The `size` method

```
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 the top of the 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
```

The `size` method

- 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.
  - This is a good use of recursion.

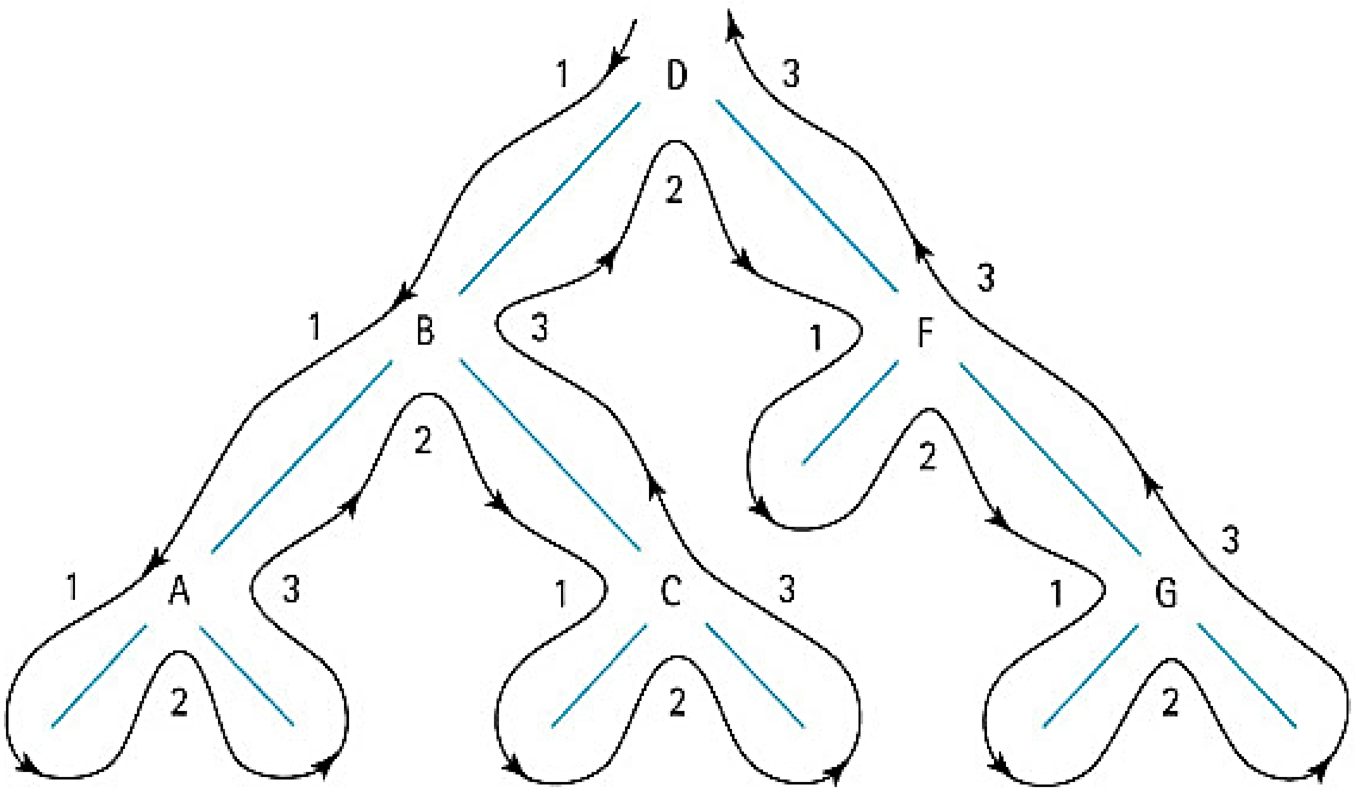
BST traversals

- Tree traversal is a form of graph traversal and refers to the process of visiting each node in a tree data structure, exactly once.
- Three steps to a traversal
  1. Visit the current node
  2. Traverse its left subtree
  3. Traverse its right subtree
- There are three types of tree traversals:
  - Pre-order traversal: (1) -> (2) -> (3)
  - In-order traversal: (2) -> (1) -> (3)
  - Post-order traversal: (2) -> (3) -> (1)

BST traversals

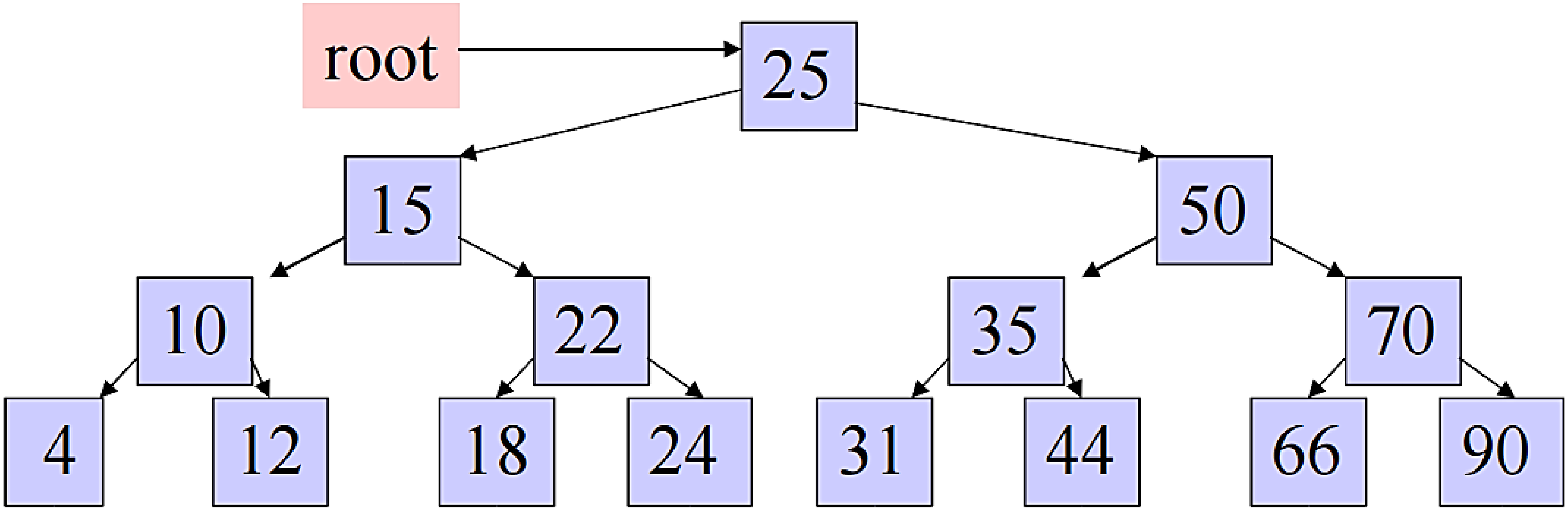
- **Pre-order traversal**: root -> left subtree -> right subtree
- **In-order traversal**: left subtree -> root -> right subtree
- **Post-order traversal**: left subtree -> right subtree -> root

The extended tree



Preorder: DBACFG
Inorder: ABCDFG
Postorder: ACBGFD

BST traversals



- In what order does the *in-order traversal* visit the nodes?

  4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

- In what order does the *pre-order traversal* visit the nodes?

  25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

- In what order does the *post-order traversal* visit the nodes?

  4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

BST traversals

- Using recursion to traverse BSTs can be very easy:

```
private void inOrder(BSTNode<T> tree)
{
    if (tree != null)
    {
        // Traverses the left subtree
        inOrder(tree.getLeft());

        // Visits and prints the value of the current node
        System.out.println(tree.getInfo());

        // Traverses the right subtree
        inOrder(tree.getRight());
    }
}
```

The `contains` method

- The **`contains`** operation checks if a given element is in the BST. It uses a *private* recursive method called **`recContains`**.
  - It is passed the element we are searching for and a reference to a subtree in which to search.
  - It first checks to see if the element searched for is in the root - if it is not, it compares the element with the root and looks in either the left or the right subtree, depending on the relationships between the values of the root and the element that we are looking for.

The `contains` method

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

The **remove** method

- The **remove** method is the most complicated of the binary search tree operations.
- We must ensure that when we remove an element, the binary search tree property is maintained.
- The **remove** method invokes a recursive method **recRemove**:

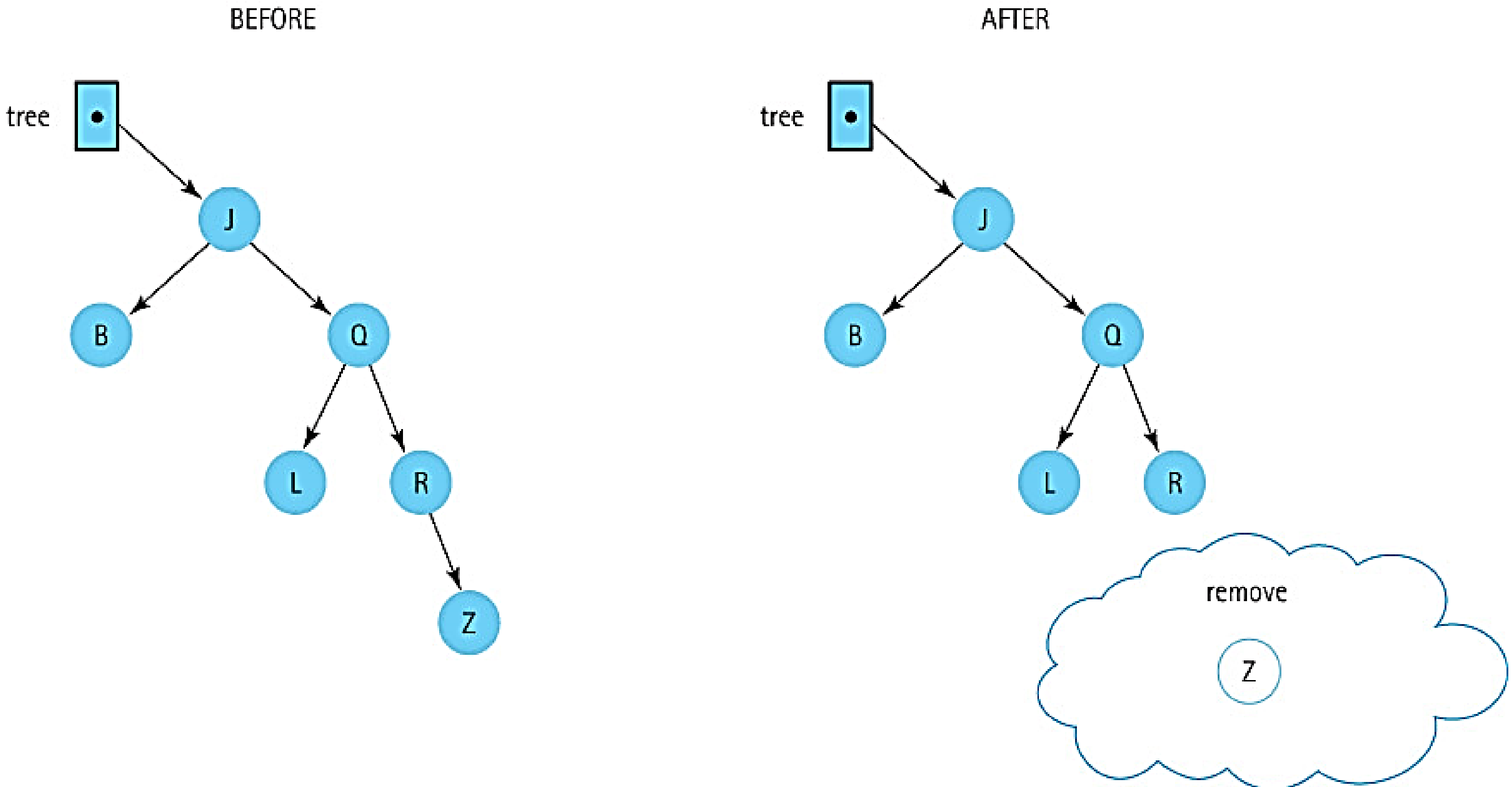
```
public boolean remove (T element)
// Removes an element e from this BST such that
// e.equals(element) and returns true; if no such element
// exists returns false.
{
  root = recRemove(element, root);
  return found;
}
```

The `remove` method

```
private BSTNode<T> recRemove(T element, BSTNode<T> tree)
// Removes an element e from tree such that e.equals(element)
// and returns true; if no such element exists returns false.
{
  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;
}
```

The **`remove`** method

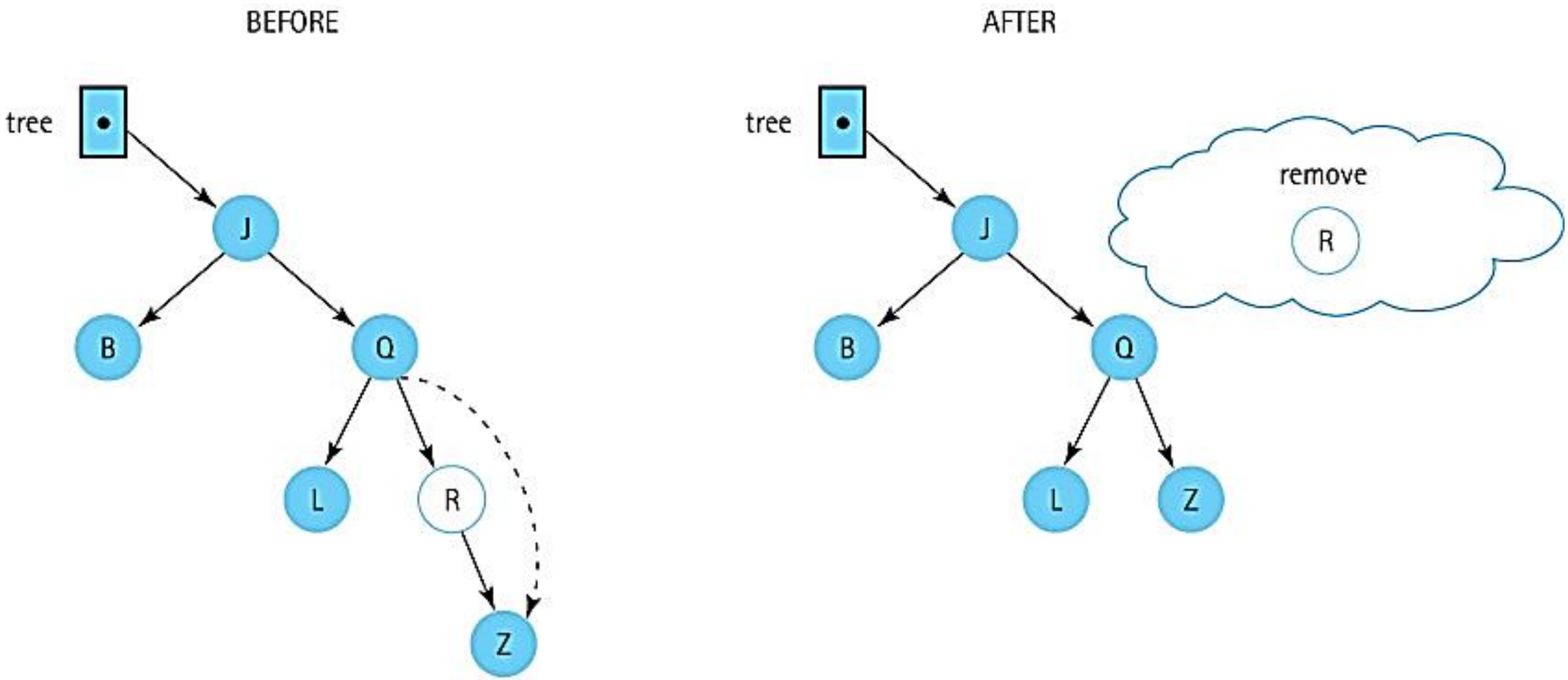
- There are three cases for the **`removeNode`** operation:
  - Removing a leaf (no children): removing a leaf is simply a matter of setting the appropriate link of its parent to **`null`**.



Remove the node containing Z

The `remove` method

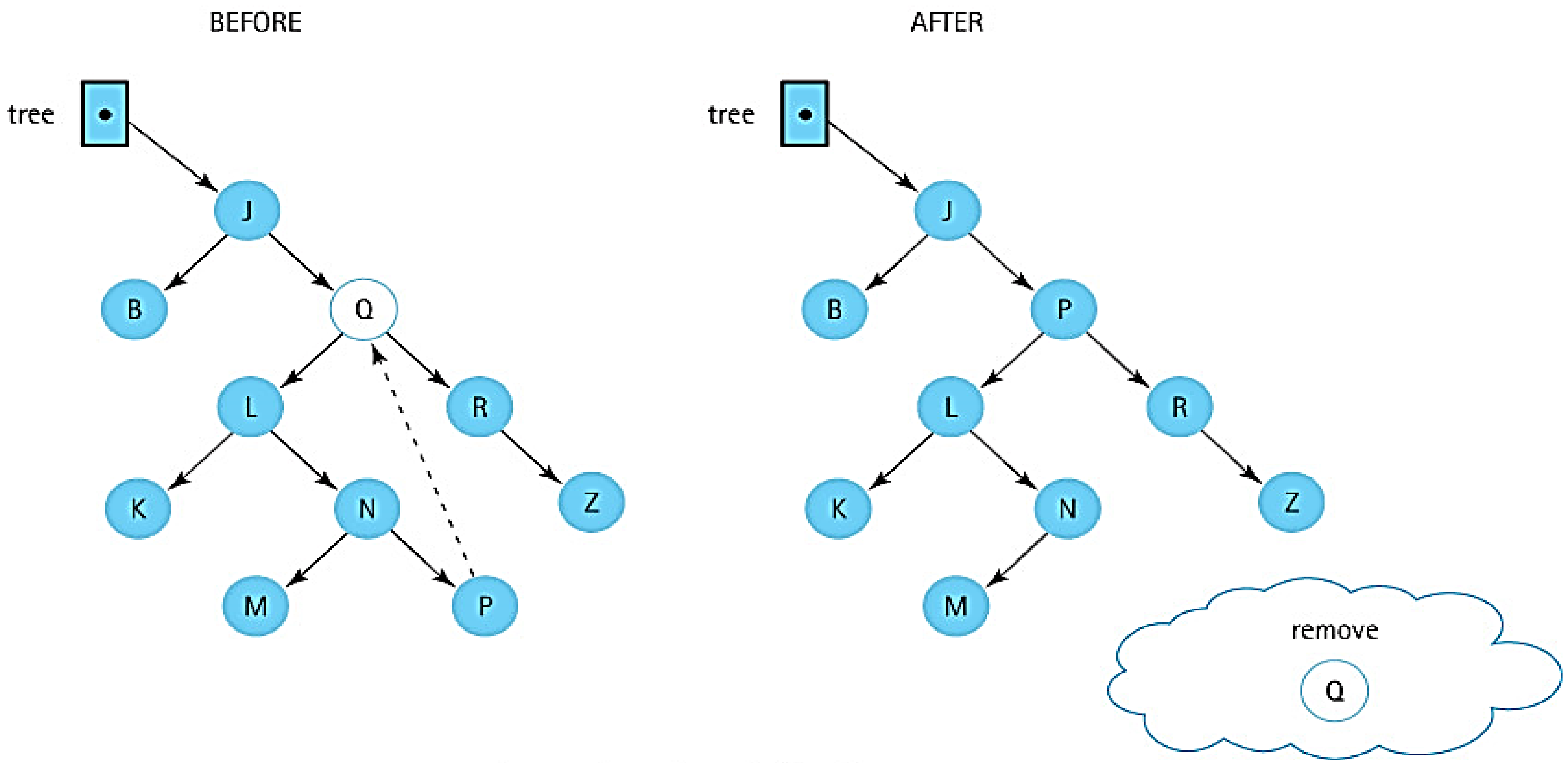
- There are three cases for the `removeNode` operation:
  - Removing a node with only one child: make the reference from the parent skip over the removed node and point instead to the child of the node we intend to remove.



Remove the node containing R

The `remove` method

- There are three cases for the `removeNode` operation:
  - 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.



Remove the node containing Q

The `remove` method

```
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
```

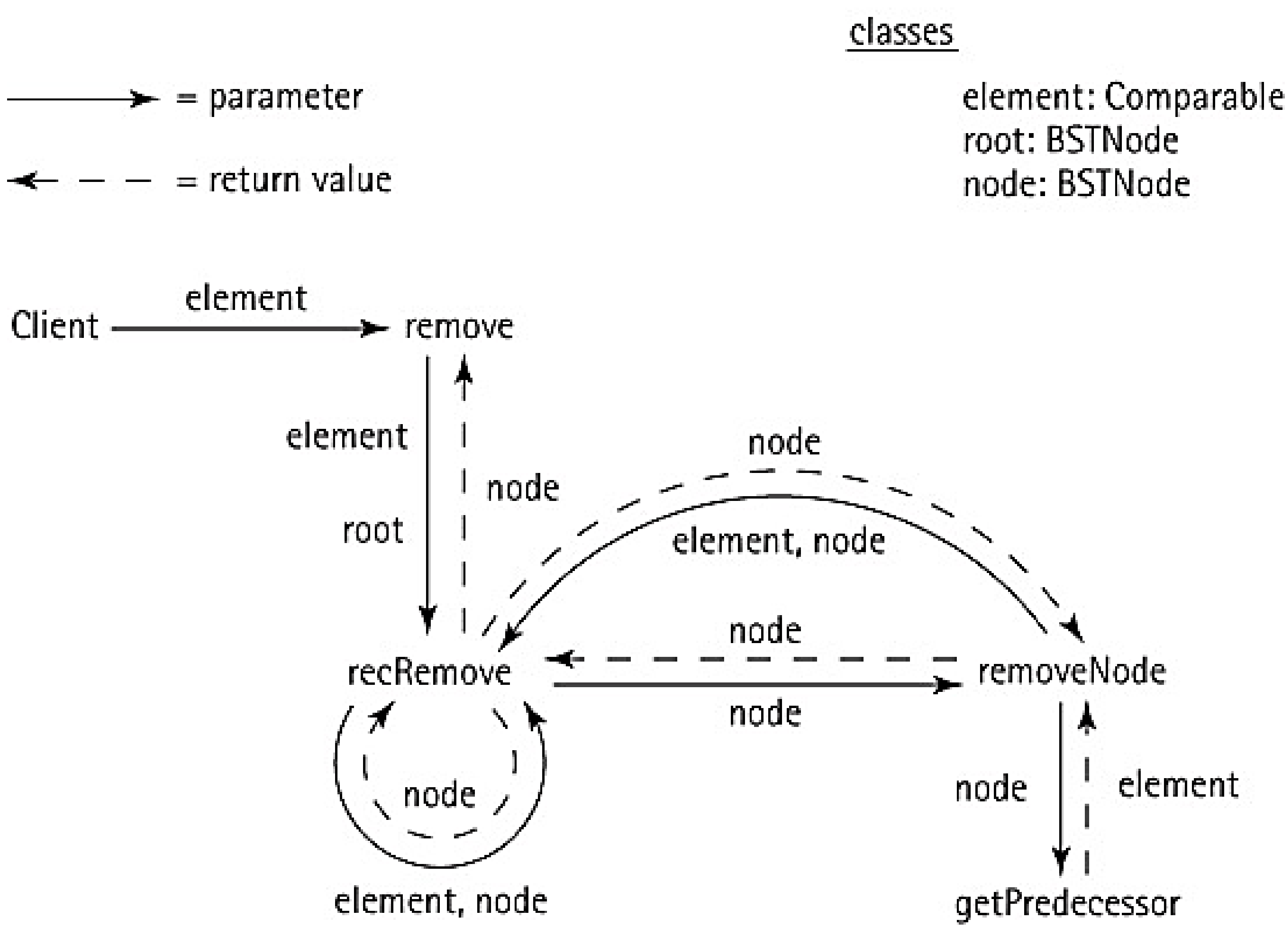
The `remove` method

- 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.

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

- On top of the predecessor, we can also use the successor of the node to replace it. The logical successor is the minimum value (in the *leftmost node*) in tree's *right subtree*.

The `remove` method



Comparing binary search trees to linear lists

	Binary Search Tree	Array-based Sorted List	Linked List
Class constructor	O(1)	O(N)	O(1)
`isEmpty`	O(1)	O(1)	O(1)
`contains`	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(1)
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)

Balancing a BST

- 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.
- A beneficial addition to our binary search tree ADT operations is a **`balance`** operation.
- The specification of the operation is:

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

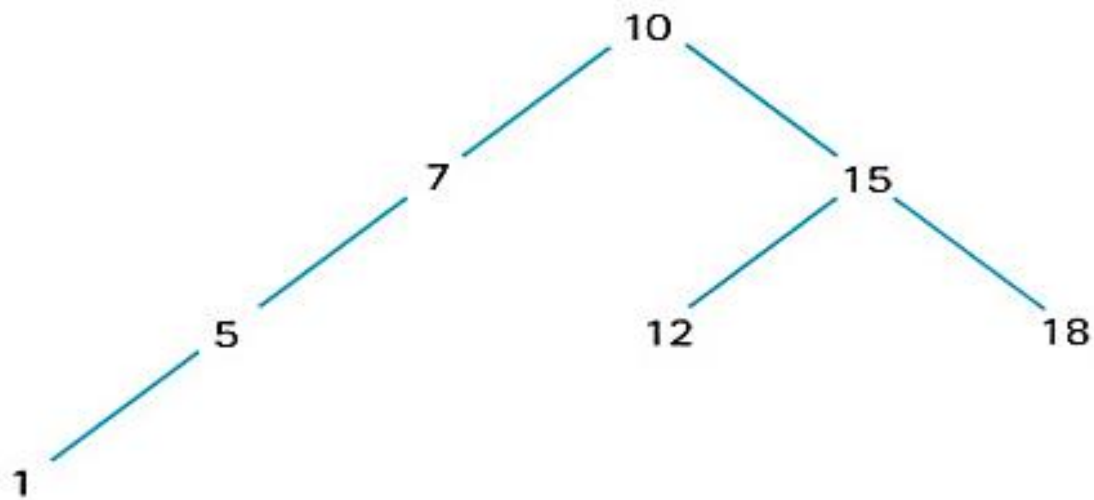
Balancing a BST

- 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.

Balancing a BST

- Using **`inOrder`** traversal

(a) The original tree



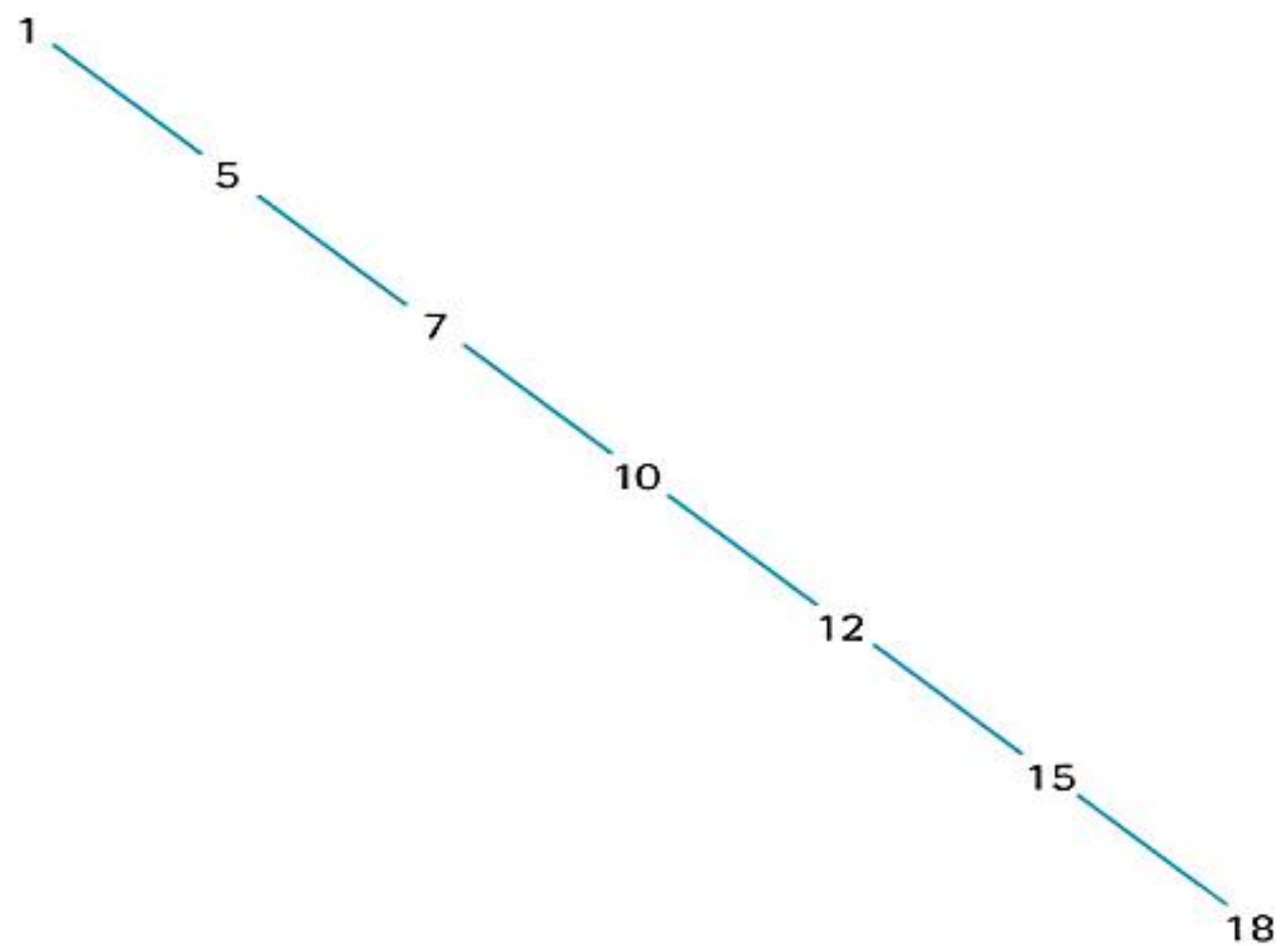
(b) The inorder traversal

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

Balancing a BST

- Using **`inOrder`** traversal

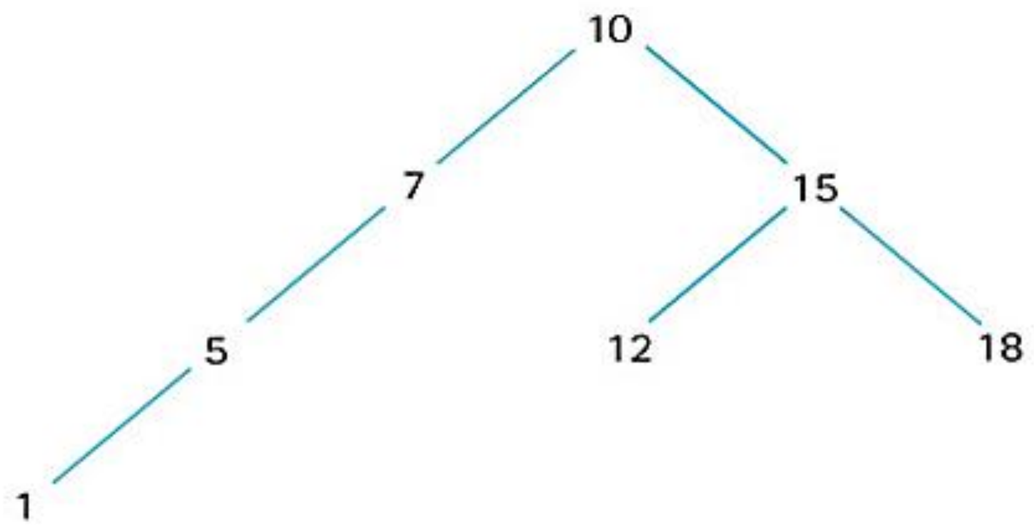
(c) The resultant tree if linear traversal of array is used



Balancing a BST

- Using **`preOrder`** traversal

(a) The original tree

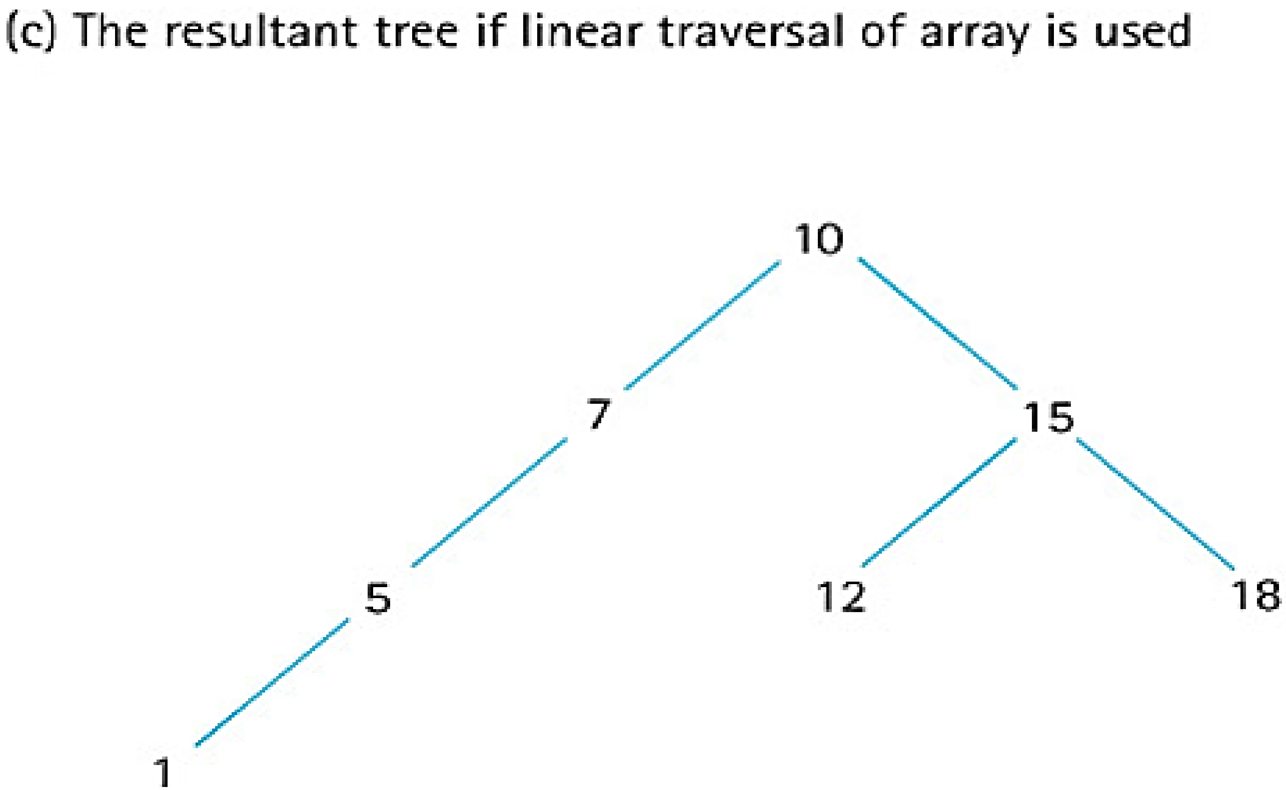


(b) The preorder traversal

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

Balancing a BST

- Using `preOrder` traversal



Balancing a BST

- 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.

Balancing a BST

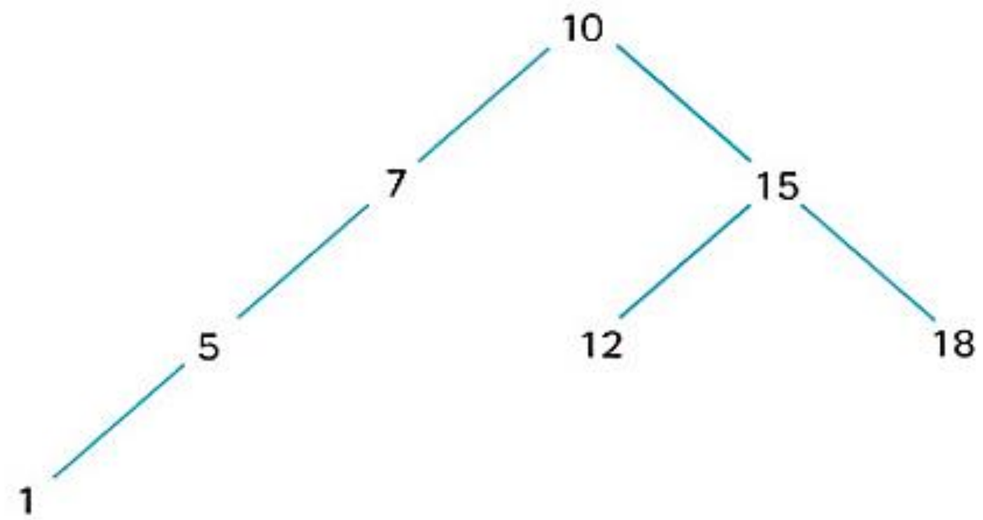
```
Balance
Set count to tree.reset(INORDER)
For (int index = 0; index < count; index++)
    Set array[index] = tree.getNext(INORDER)
tree = new BinarySearchTree()
tree.InsertTree(0, count - 1)

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)
```

Balancing a BST

- Using recursive **`insertTree`**

(a) The original tree

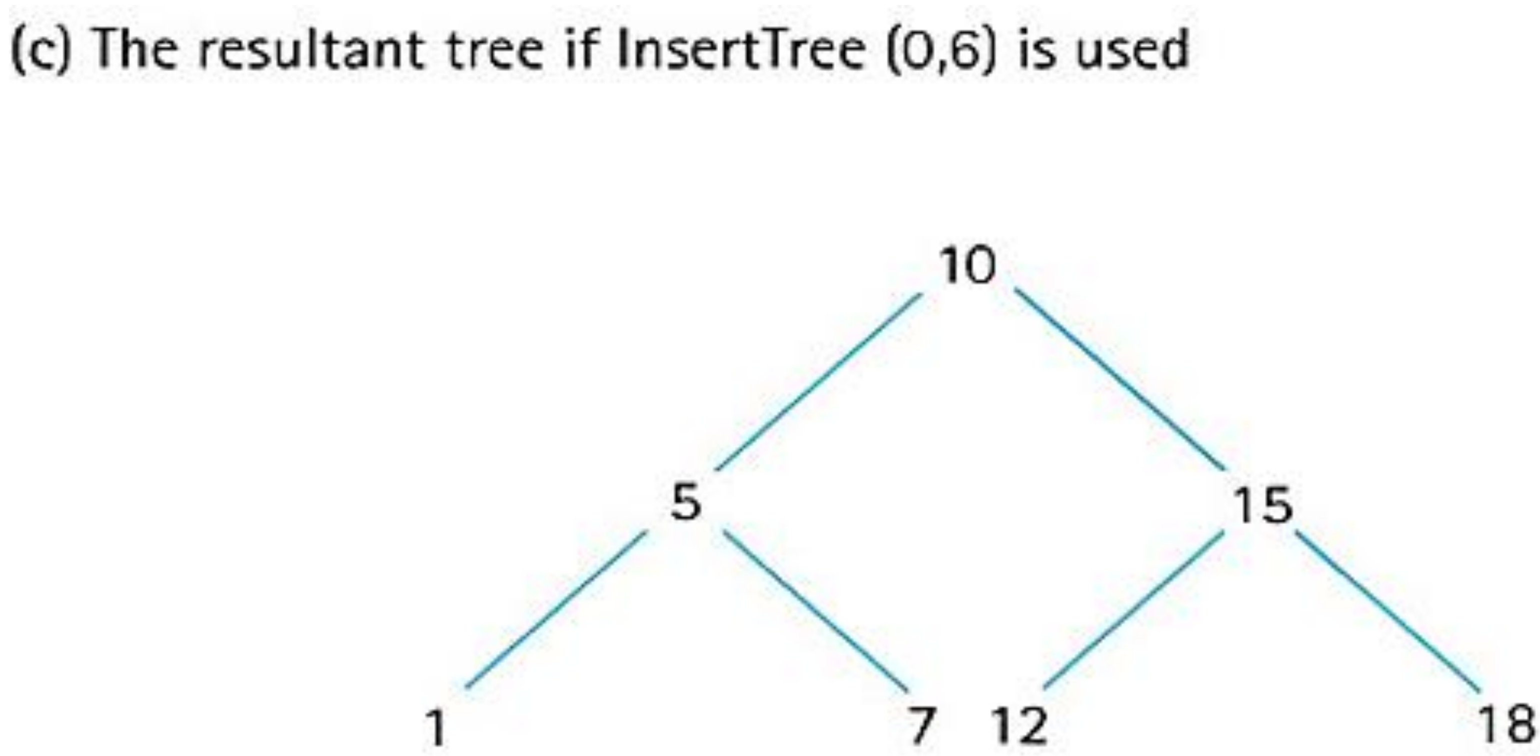


(b) The inorder traversal

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

Balancing a BST

- Using recursive **`insertTree`**

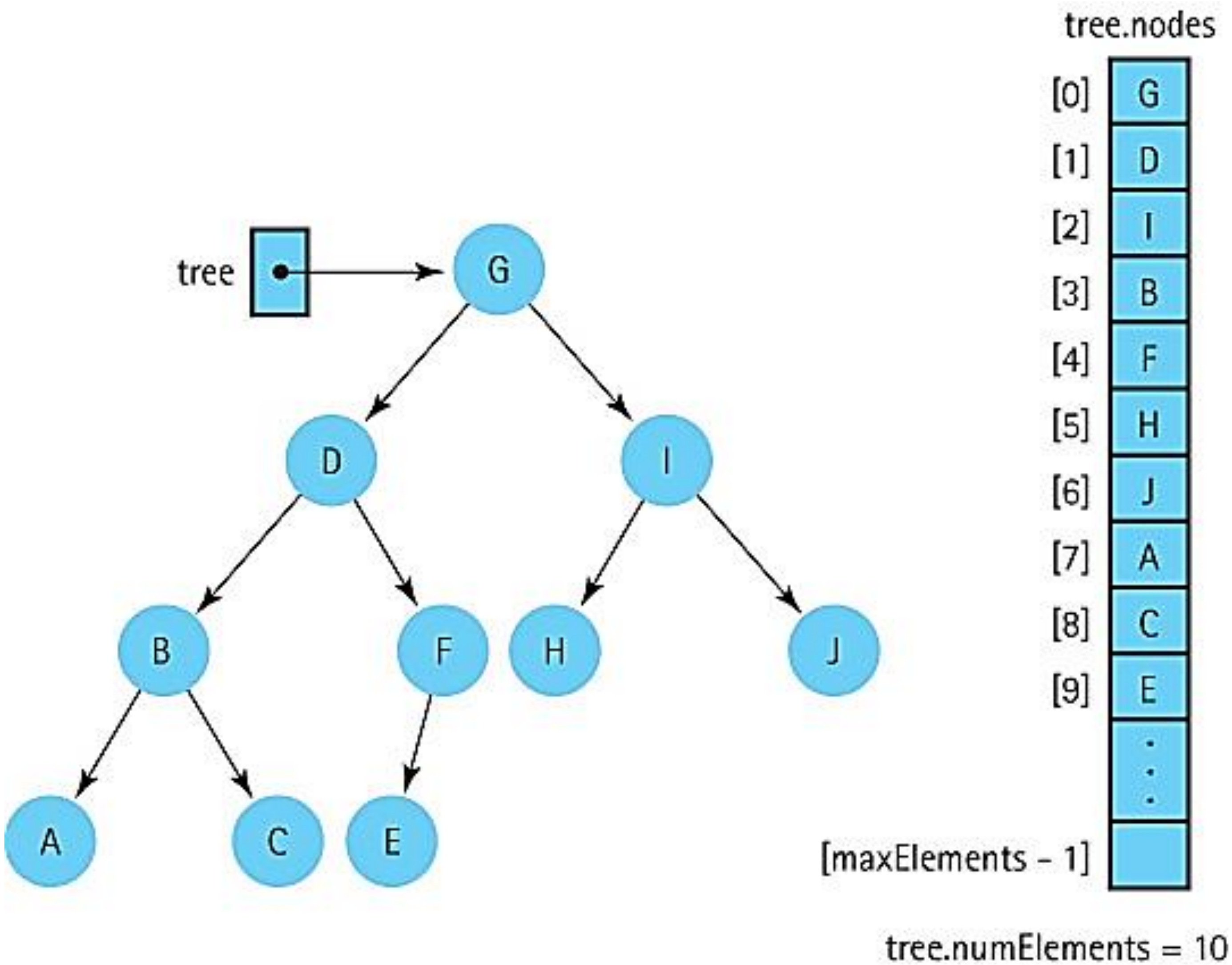


A non-linked representation of binary trees

- 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 non-linked representation of binary trees

- A binary tree and its array representation:

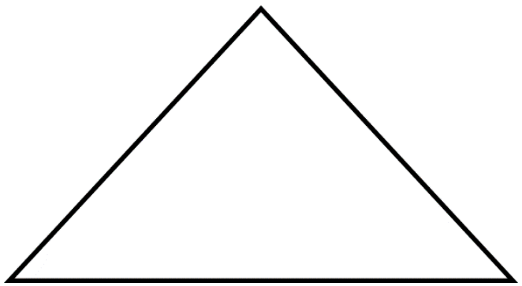


A non-linked representation of binary trees

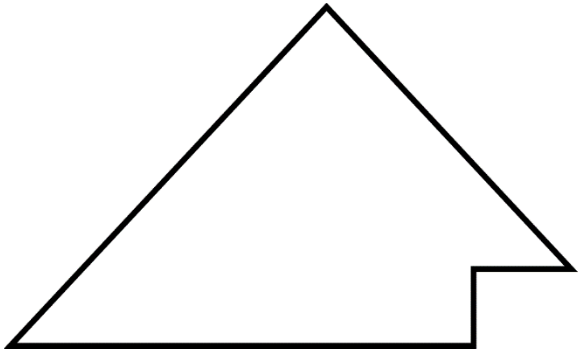
- To implement the algorithms that manipulate the tree, we must be able to find the left and right children, as well as the parent of a node (with the index `i`) in the tree:
  - Its left child has the index `i * 2 + 1`.
  - Its right child has the index `i * 2 + 2`.
  - Its parent has the index `(i – 1) / 2`.
- This representation works best, space wise, if the tree is complete.

Full binary trees and complete binary trees

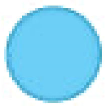
- A *full binary tree* is a binary tree in which all of the leaves are on the same level and every nonleaf node has two children.



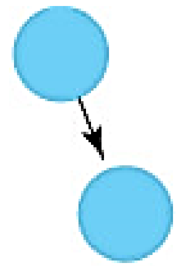
- A *complete binary tree* is a binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible.



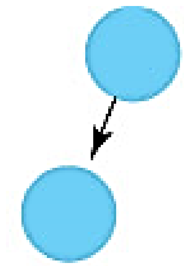
Examples of different types of binary trees



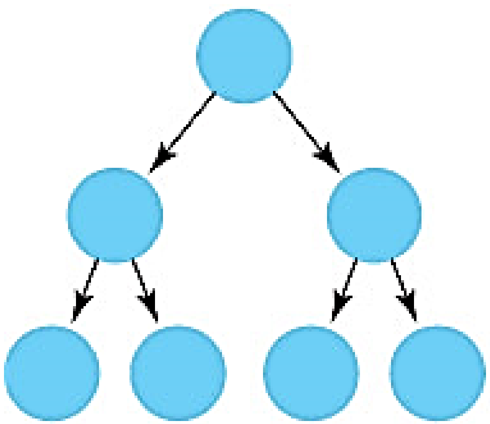
(a) Full and complete



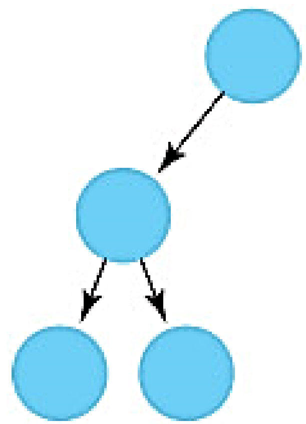
(b) Neither full nor complete



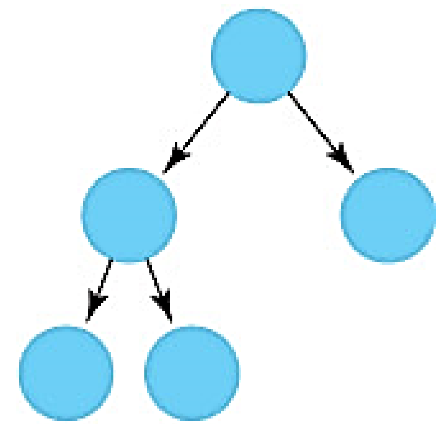
(c) Complete



(d) Full and complete



(e) Neither full nor complete



(f) Complete

Action items

- Read book chapter 8.