ITI 1121. Introduction to Computing II [*]

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of March 24, 2013

Abstract

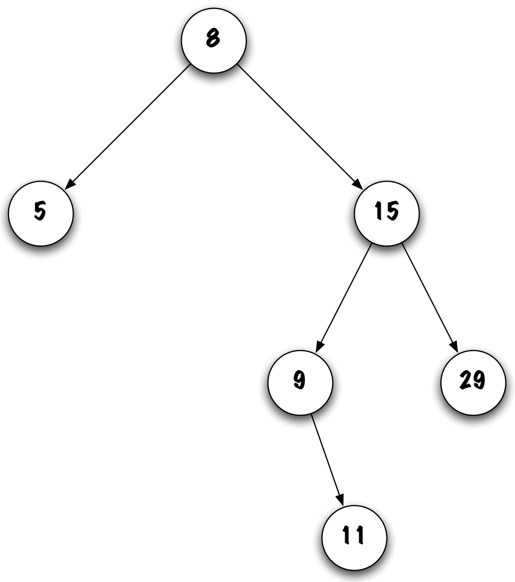
• Binary search tree (part II)

[*] These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Binary search tree

A **binary search tree** is a **binary tree** such that each node verifies the following properties:

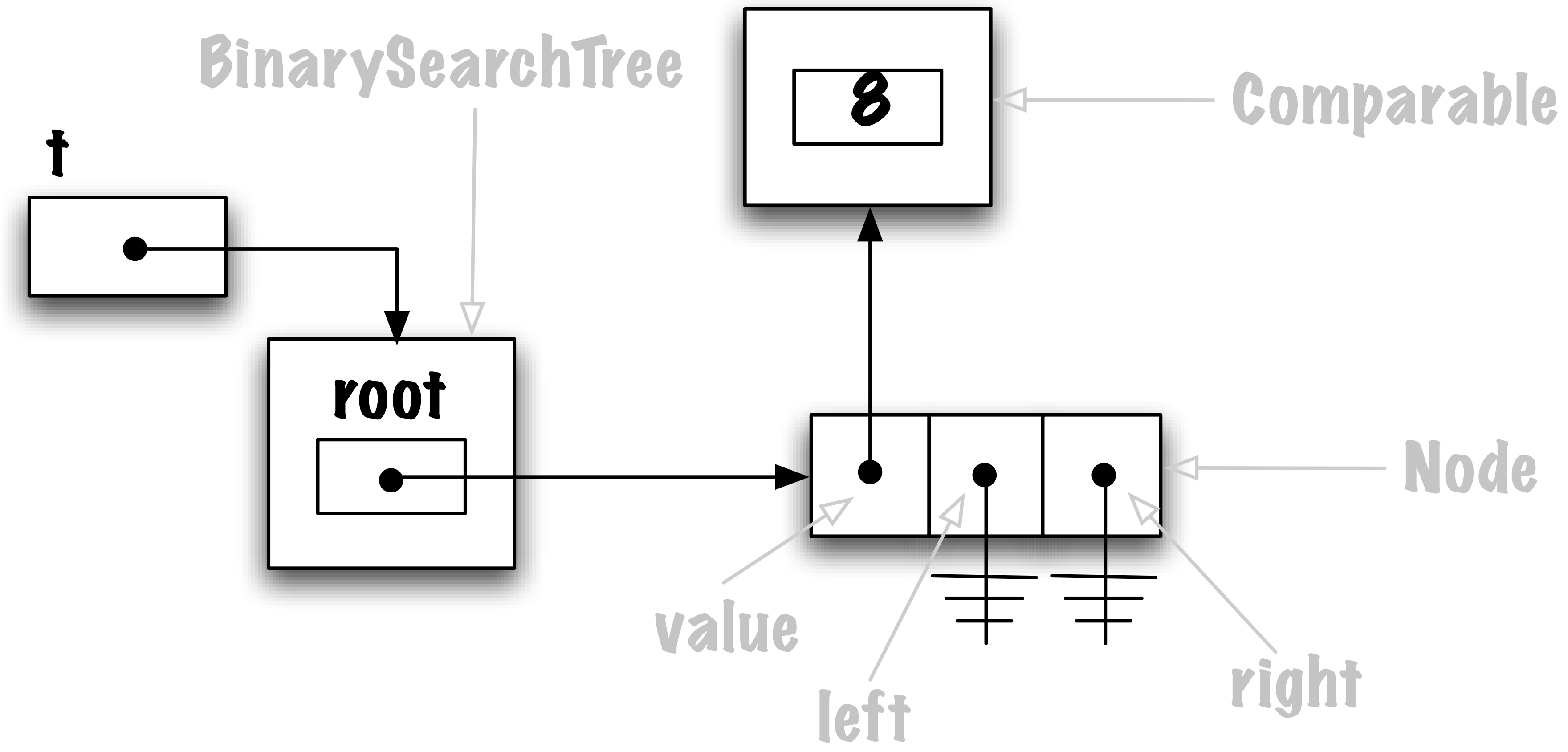
- All the nodes in its left sub-tree have values that are less than the value of this node or the left sub-tree is empty;
- All the nodes in its right sub-tree have values that are greater than the value of this node or the right sub-tree is empty.

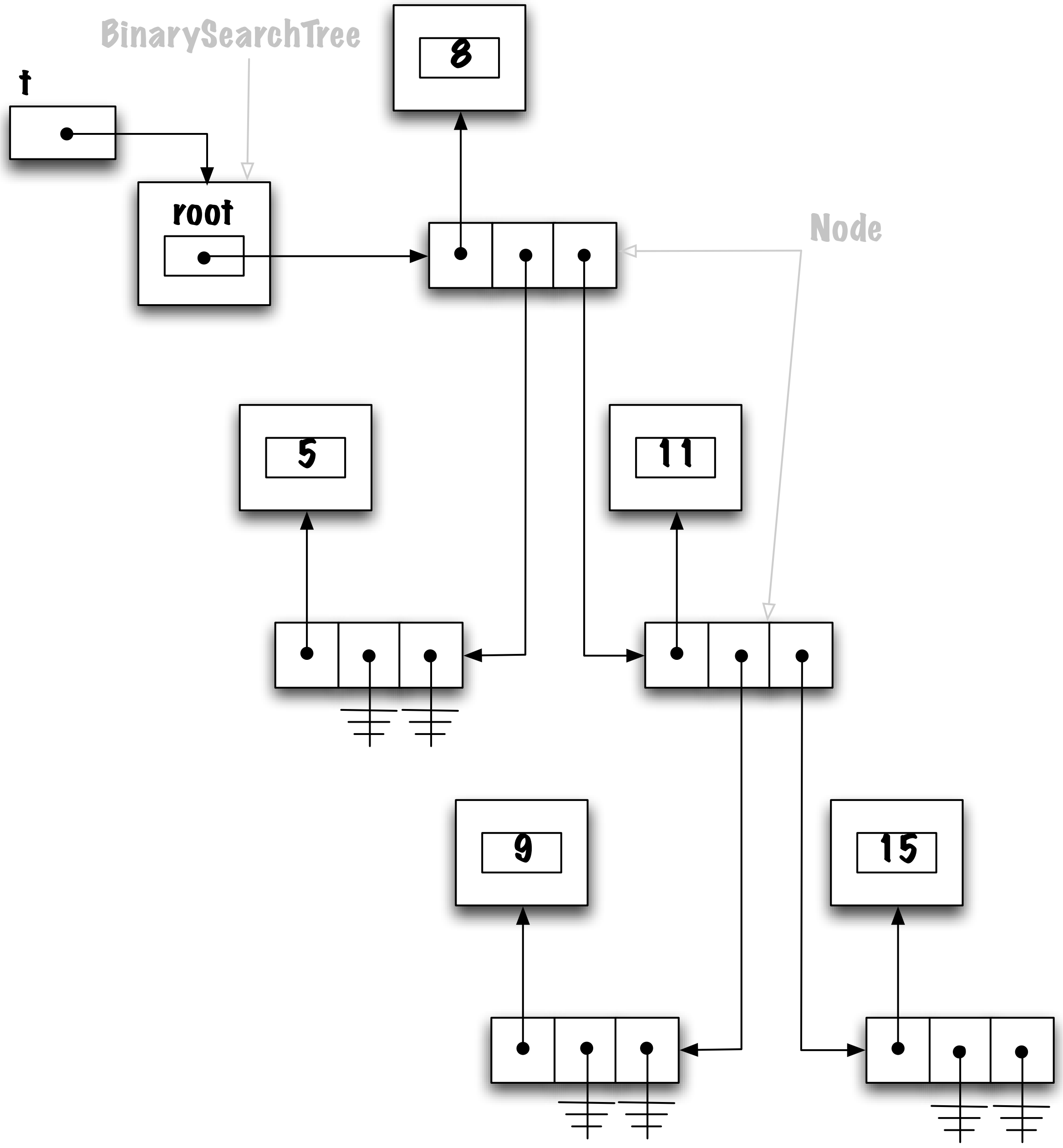


Implementing a binary search tree

```
public class BinarySearchTree< E extends Comparable<E> > {

    private static class Node<E> {
        private E value;
        private Node<E> left;
        private Node<E> right;
    }

    private Node<E> root;
```

Memory diagram




BinarySearchTree
t
root
8
Node
5
11
9
15

Traversing a tree

Pre-order: current, left, right;

In-order: left, current, right;

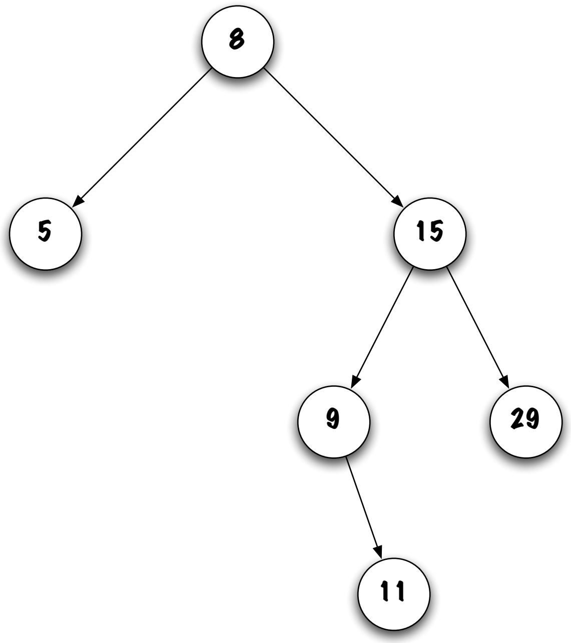
Post-order: left, right, current.

Traversing a tree

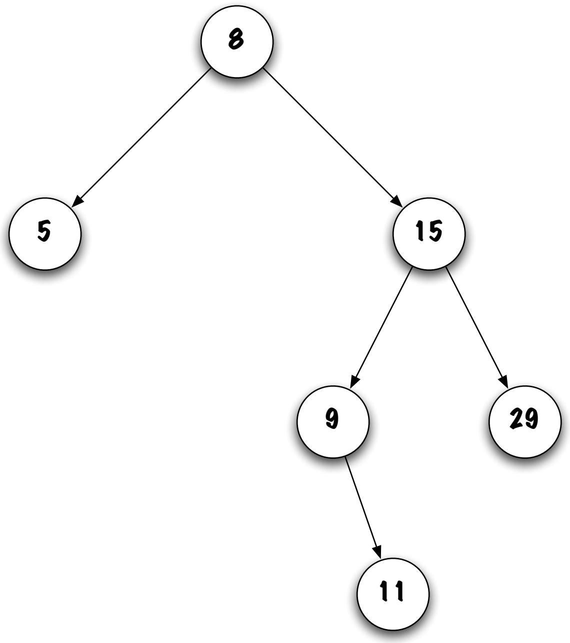
```
private void visit( Node<E> current ) {
    System.out.print( " " + current.value );
}


public void preOrder() {
    preOrder( root );
}

public void inOrder() {
    inOrder( root );
}

public void postOrder() {
    postOrder( root );
}
```

Pre-order

```
private void preOrder( Node<E> current ) {

    if ( current != null ) {

        visit( current );
        preOrder( current.left );
        preOrder( current.right );

    }

}

preOrder -> 8 5 15 9 11 29
```

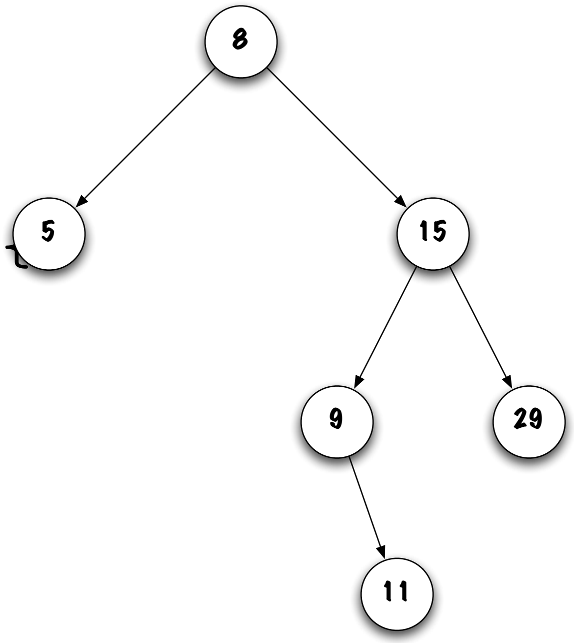


In-order

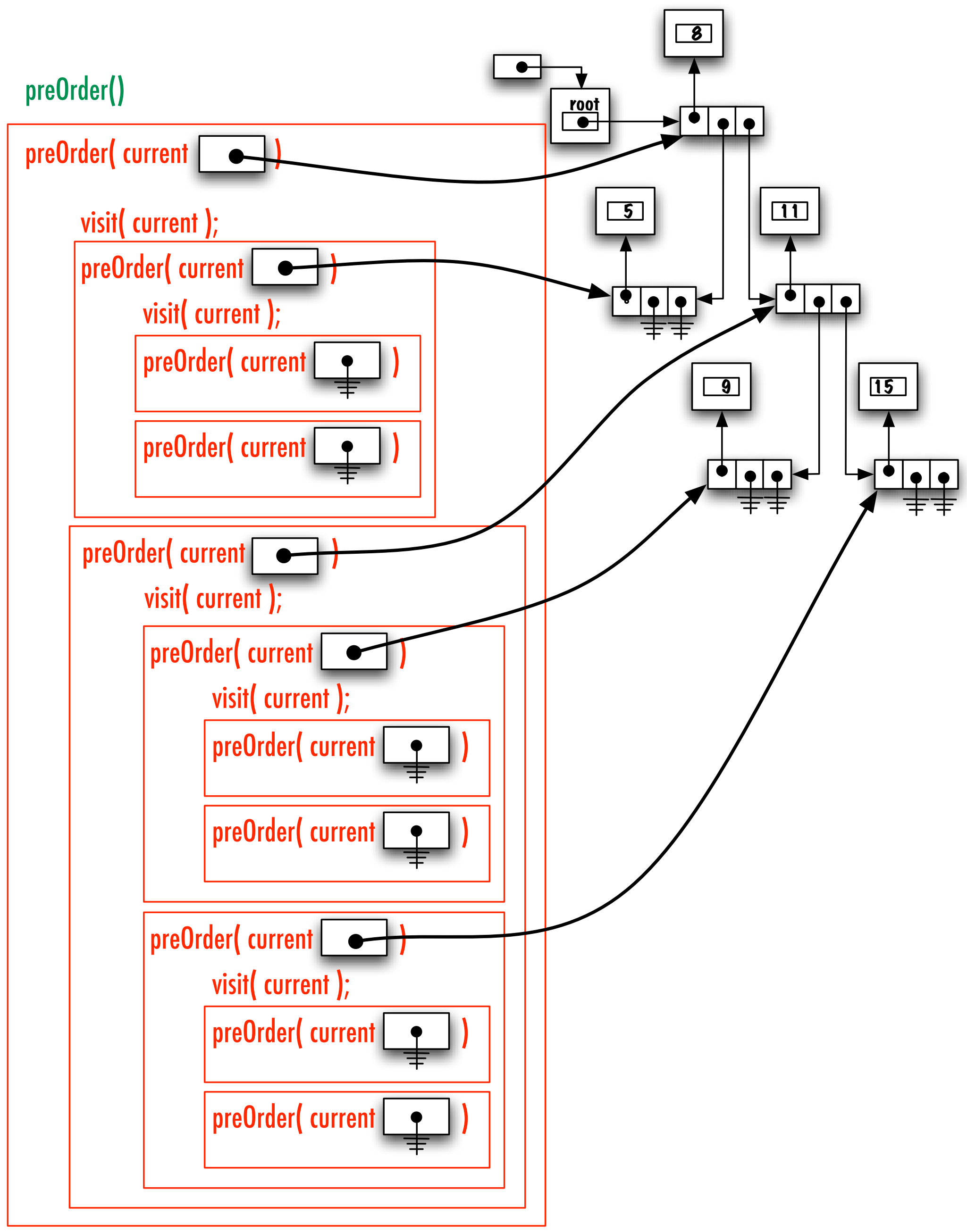


```
private void inOrder( Node<E> current ) {

    if ( current != null ) {

        inOrder( current.left );
        visit( current );
        inOrder( current.right );

    }

}

inOrder -> 5 8 9 11 15 29
```

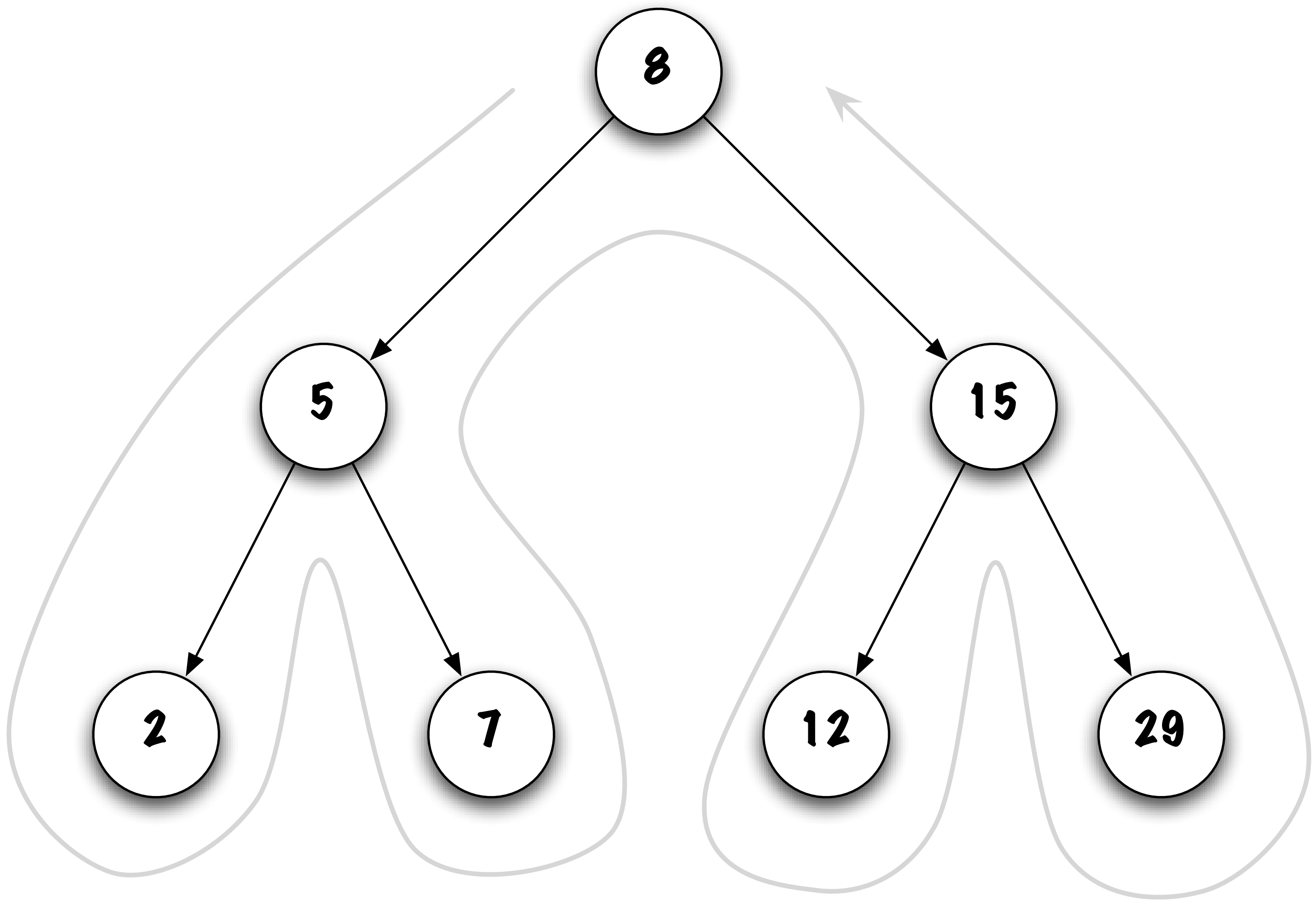
Post-order



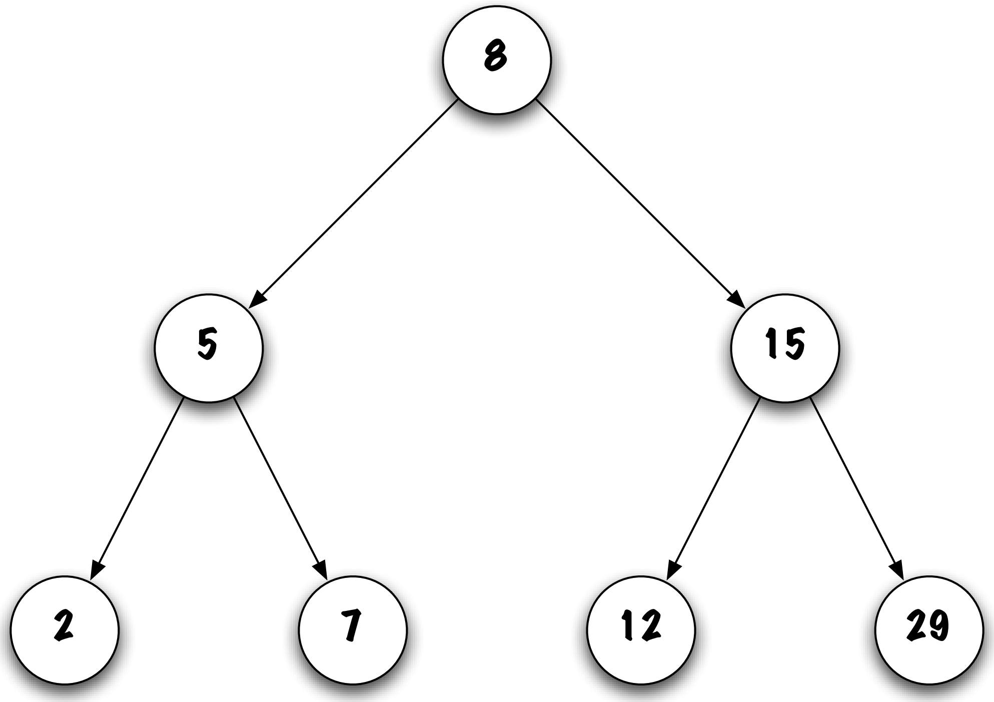
```
private void postOrder( Node<E> current ) {

    if ( current != null ) {

        postOrder( current.left );
        postOrder( current.right );
        visit( current );

    }

}

postOrder -> 5 11 9 29 15 8
```


preOrder()
preOrder(current)
visit(current);
preOrder(current)
visit(current);
preOrder(current)
preOrder(current)
preOrder(current)
visit(current);
preOrder(current)
visit(current);
preOrder(current)
preOrder(current)
preOrder(current)
visit(current);
preOrder(current)
preOrder(current)
root
8
5
11
9
15

Euler tour



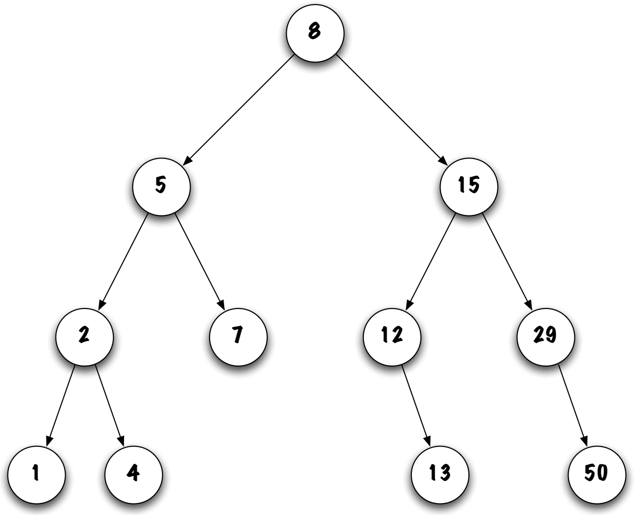
Full binary tree



A binary tree is a **full binary tree** if all its nodes have exactly two children, except for the leaves.

Binary tree

A binary tree of depth d is **balanced** if all the nodes at depth less than $d-1$ (i.e. $[0, 1 \ldots d-2]$) have exactly two children.

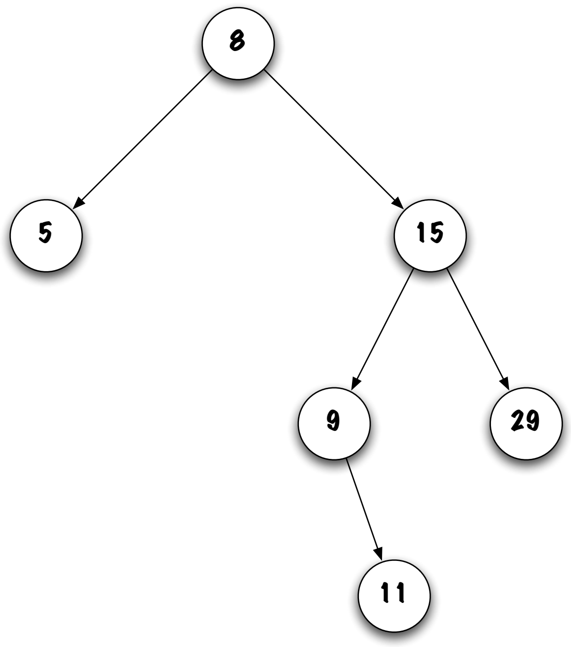


Is this a balanced tree?

Yes, the depth of the tree is $d = 3$, all the nodes at depth 0 and 1 ($\leq d-2$) have exactly two children. Nodes at depth 2 have 0, 1 or 2 children. Nodes at depth 3 have no children.

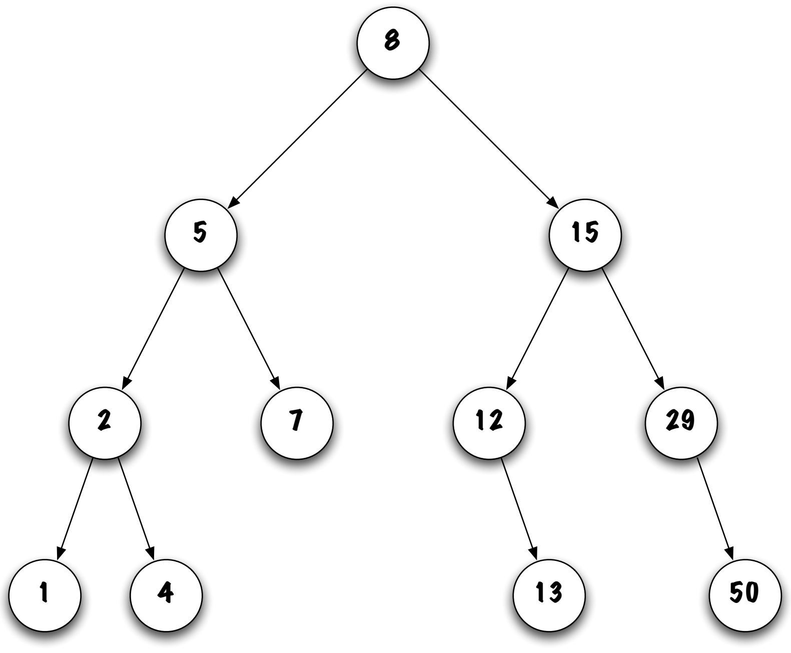
Binary tree

Is this a balanced tree?



No, the depth of the tree is $d = 3$, node 5 at depth 1 ($\leq d - 2$) does have two children.

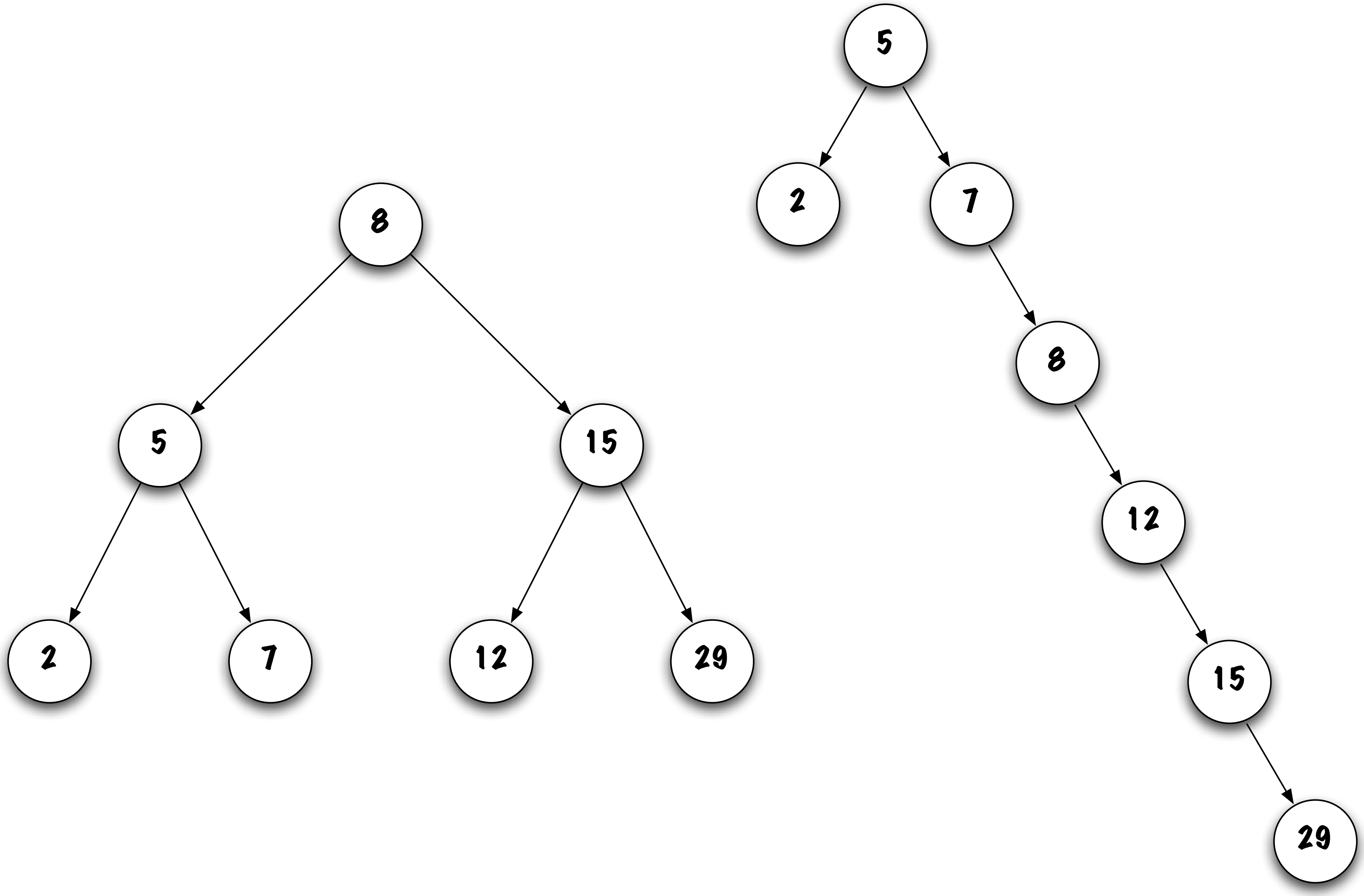
Binary tree

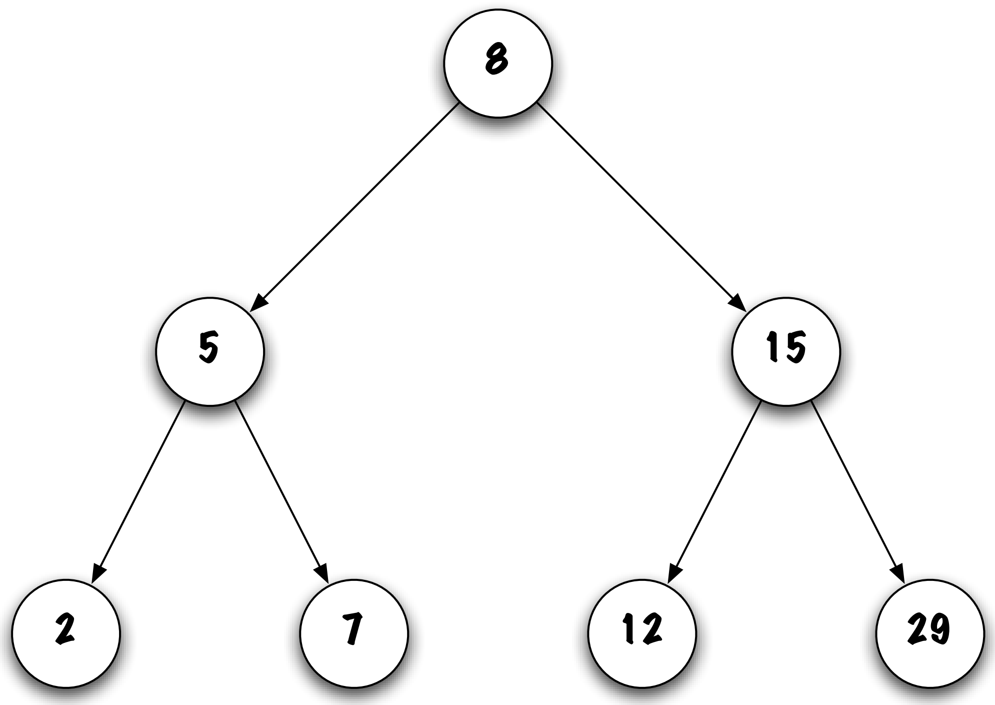


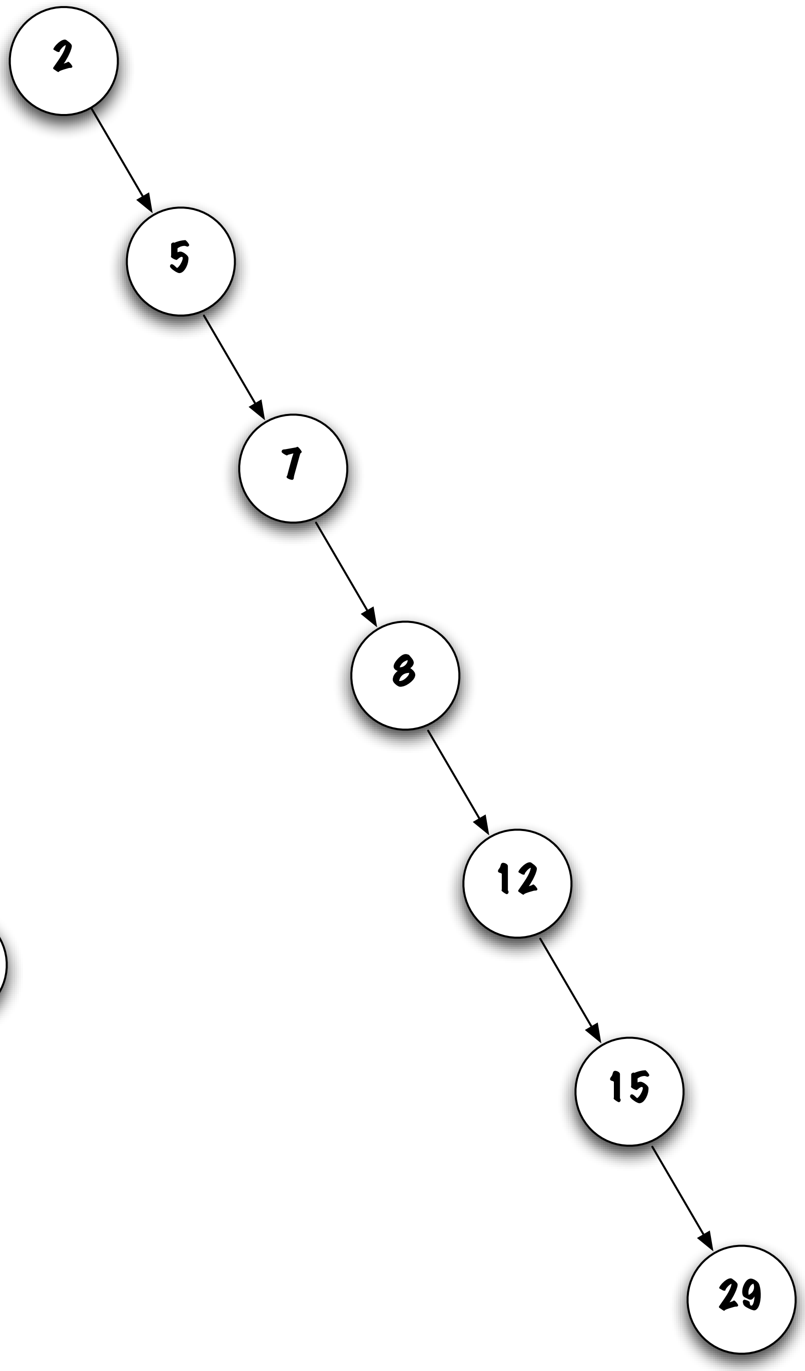
- A balanced binary tree of depth d has 2^d to $2^{d+1} - 1$ nodes;
- The depth of a balanced binary tree of size n is $\lfloor \log_2 n \rfloor$.

Discussion

Discuss the relationship between the efficiency of the method **contains** and the topology of the tree (balanced or not).


8
5
15
2
7
12
29
5
2
7
8
12
15
29


8
5
15
2
7
12
29


2
5
7
8
12
15
29

Remarks

- When looking up for a value, each comparison eliminates a sub-tree;
- The maximum number of nodes visited depends on the depth of the tree;
- Hence, balanced trees are advantageous (since the depth of a balanced tree is $\lfloor \log_2 n \rfloor$ compared to n for an ill-balanced tree).

Remarks

n	$\lfloor \log_2 n \rfloor$
10	3
100	6
1,000	9
10,000	13
100,000	16
1,000,000	19
10,000,000	23
100,000,000	26
1,000,000,000	29
10,000,000,000	33
100,000,000,000	36
1,000,000,000,000	39
10,000,000,000,000	43
100,000,000,000,000	46
1,000,000,000,000,000	49

Remarks

- Methods that follow a single path along the tree, from the **root** to some node, can easily be implemented without recursion, see **contains**;
- Methods that visit more than one sub-tree are often simpler to implement with help of recursion, see **toString**.

boolean add(E obj)

Exercise. Starting with an empty tree, add all the following elements: “Lion”, “Fox”, “Rat”, “Cat”, “Pig”, “Dog”, “Tiger”.

What are your conclusions?

In order to add an element, the insertion point must be located. What method is it most similar to? It is similar to the method **contains**.

What are the necessary changes? Here is the method **contains**:

```
public boolean contains( E obj ) {
    boolean found = false;
    Node<E> current = root;
    while ( ! found && current != null ) {
        int test = obj.compareTo( current.value );
        if ( test == 0 ) {
            found = true;
        } else if ( test < 0 ) {
            current = current.left;
        } else {
            current = current.right;
        }
    }
    return found;
}
```

boolean add(E obj)

Special cases?

Operations that involve change the variable **root** are special cases, just like operations that are changing the variable **head** are special cases in the case of a linked list.

```
if ( current == null ) {
    root = new Node<E>( obj );
}
```

General case:

```
boolean done = false;
while ( ! done ) {
    int test = obj.compareTo( current.value );
    if ( test == 0 ) {
        done = true;
    } else if ( test < 0 ) {
        if ( current.left == null ) {
            current.left = new Node<E>( obj );
            done = true;
        } else {
            current = current.left;
        }
    } else {
        if ( current.right == null ) {
            current.right = new Node<E>( obj );
            done = true;
        } else {
            current = current.right;
        }
    }
}
```

boolean add(E obj)

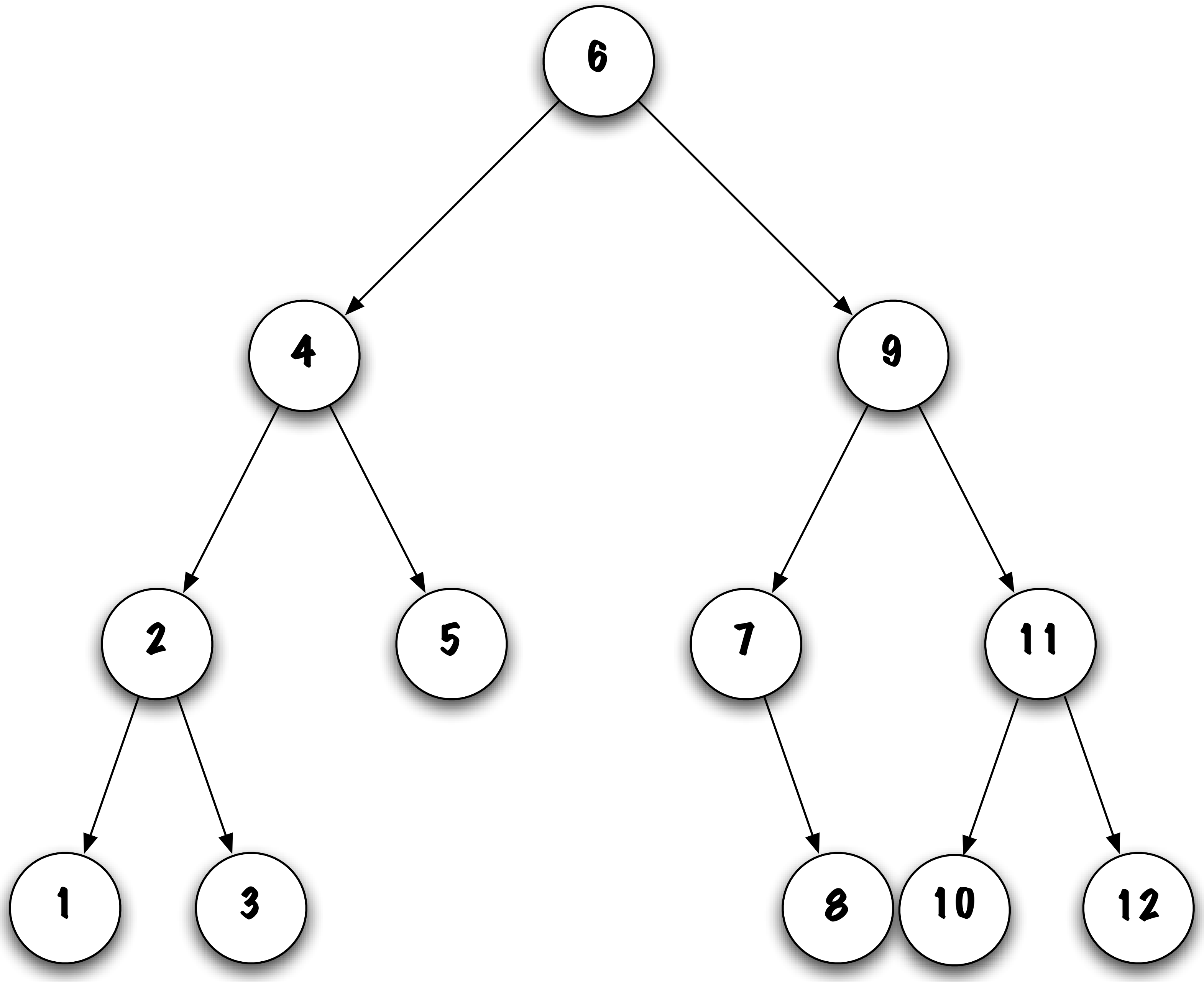
- It always replaces a **null** value by a new node;
- The existing structure of the tree is not changed;
- The topology of the tree depends largely on the order of insertion.

boolean remove(E obj)

Removing elements implies changing the structure of the tree (except for leaves).

Explore various cases using the tree on the next page.

Eliminate each of the 12 nodes, one by one.


6
4
9
2
5
7
11
1
3
8
10
12

boolean remove(E obj)

Consider the following cases:

- Removing the leftmost node.
  - How many sub-cases there are?
  - There are two sub-cases:
    1. The node has no sub-trees;
       Node **1** of the sub-tree **6** is an example;
       What should be done? **parent.left = null**;
    2. The node has a right sub-tree;
       Node **7** of the sub-tree **9** is an example;
       What should be done? **parent.~~right~~** left **= "right sub tree"**;
    3. The node cannot have a left sub-tree, because it would not be the leftmost one!

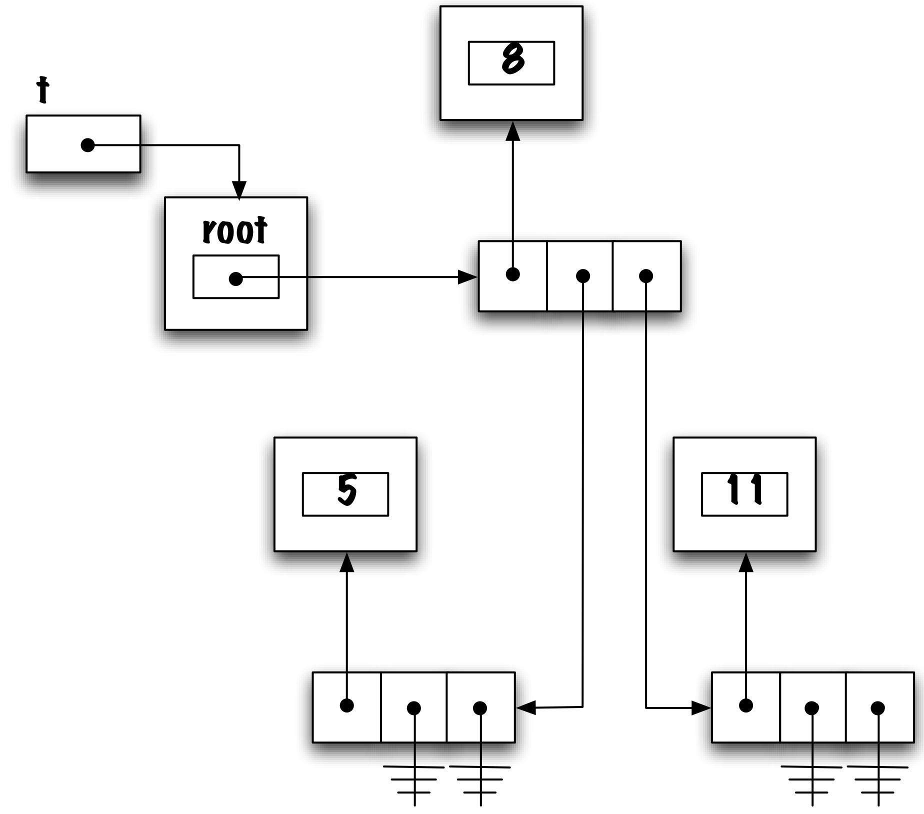
boolean remove(E obj)

Consider the following cases:

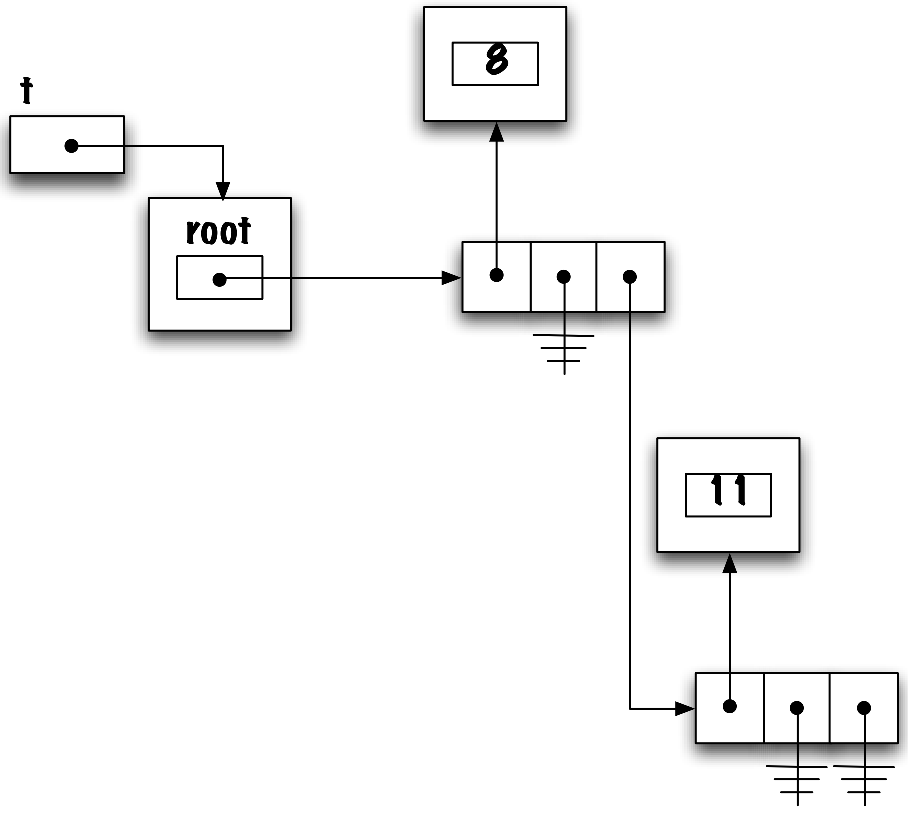
- Removing the root of a sub-tree.
  - How many cases there are?
  - There are 4 cases:
    1. The node has no sub-trees;
       What should be done? Remove this node;
    2. The node has a left sub-tree only;
       What should be done? Replace the node by that sub-tree;
    3. The node has a right sub-tree only;
       What should be done? Replace the node by that sub-tree;
    4. The node has two sub-trees that are not null;
       What should be done? There are two strategies: 1) replacing the value of that node with the one that immediately precedes, hence, the rightmost value of its left sub-tree, or, 2) replace the value of this node with the one that immediately follows, hence, the leftmost value of its right sub-tree.

Case 1: removing a leaf

Before:

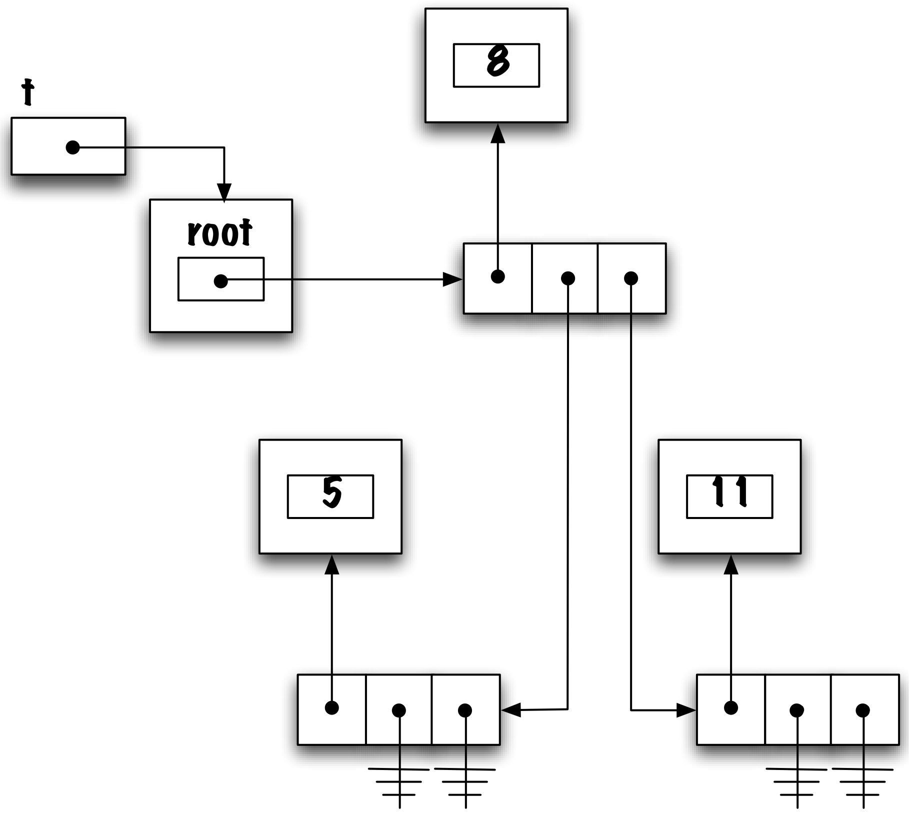


After:

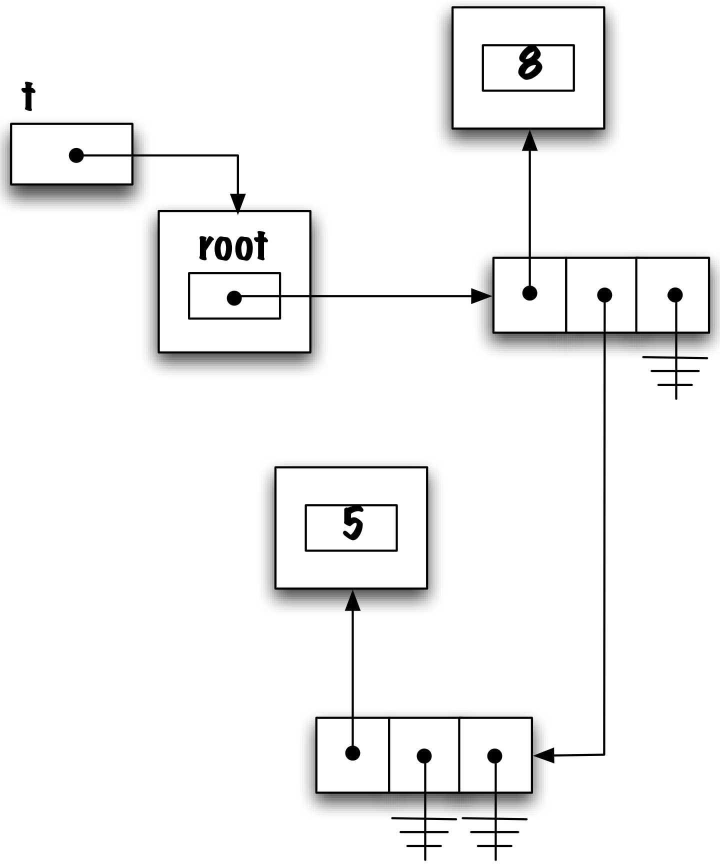


Case 1: removing a leaf

Before:

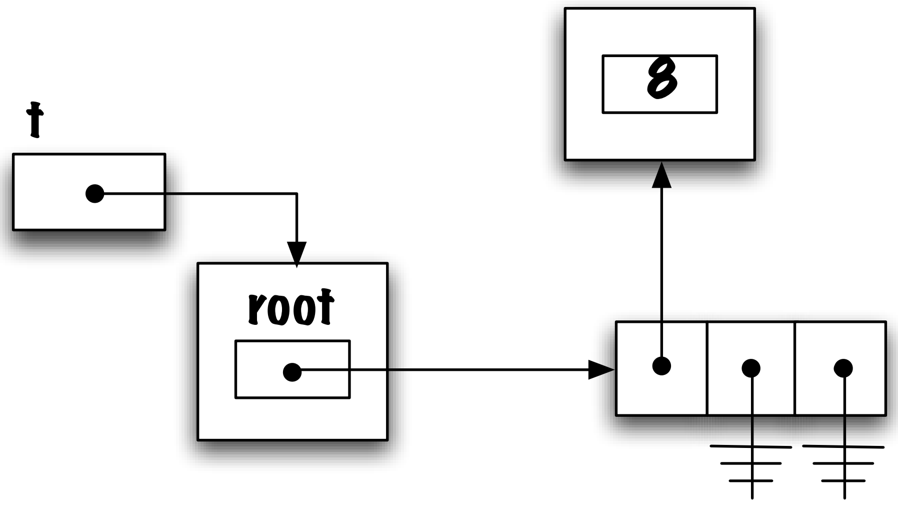


After:

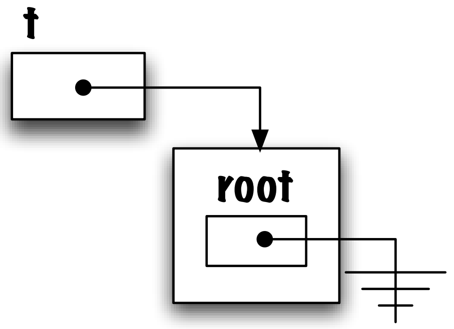


Case 1: removing a leaf

Before:

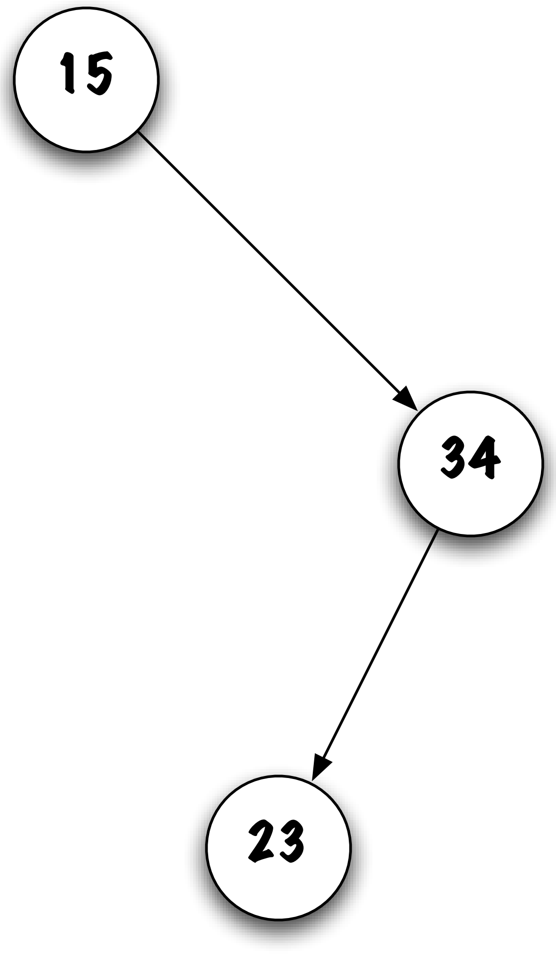


After:

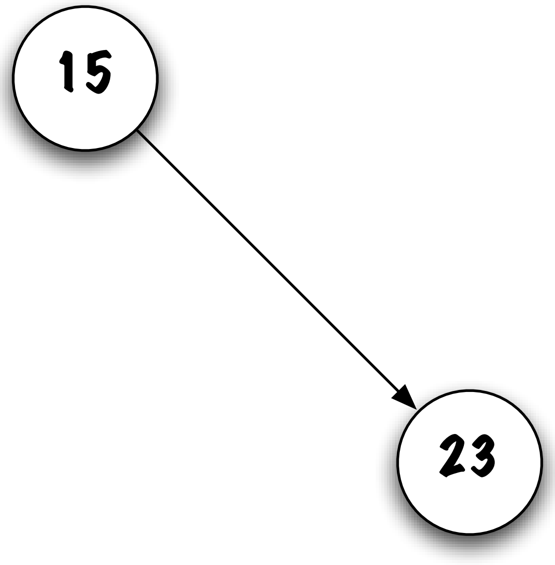


Case 2: t.remove(new Integer(34))

Before:

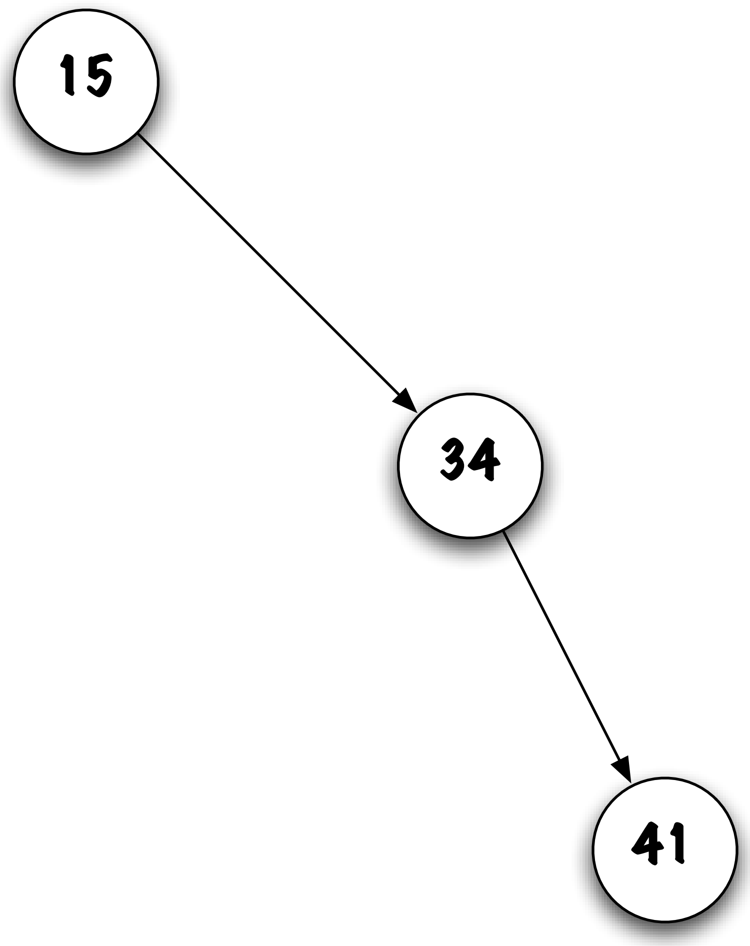


After:

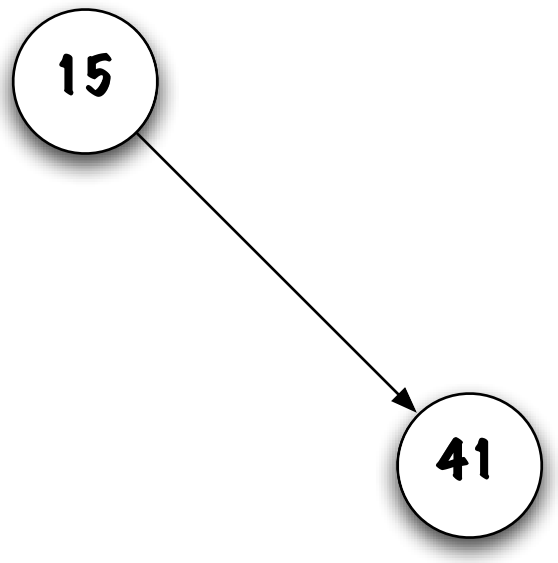


Case 3: t.remove(new Integer(34))

Before:

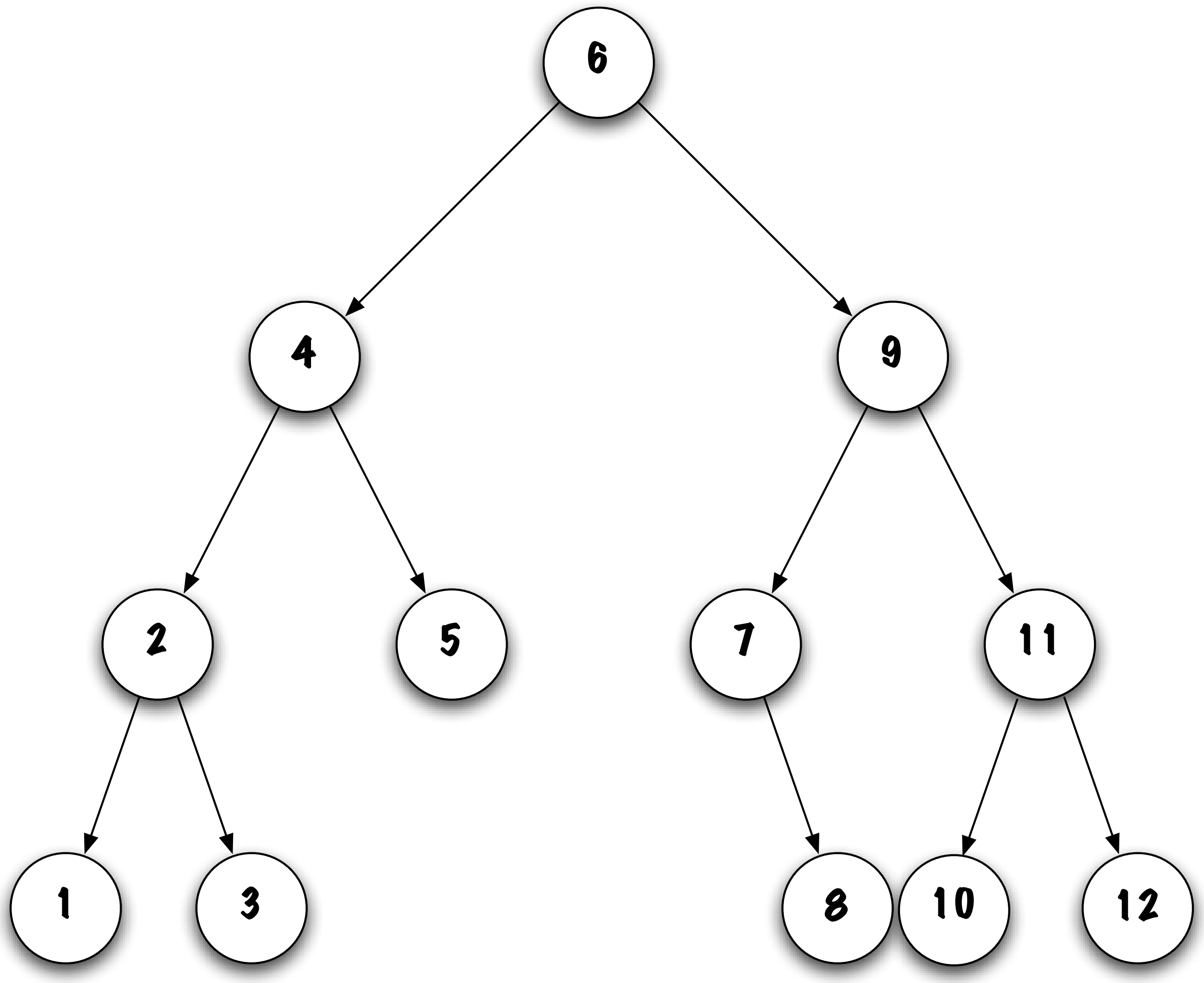


After:



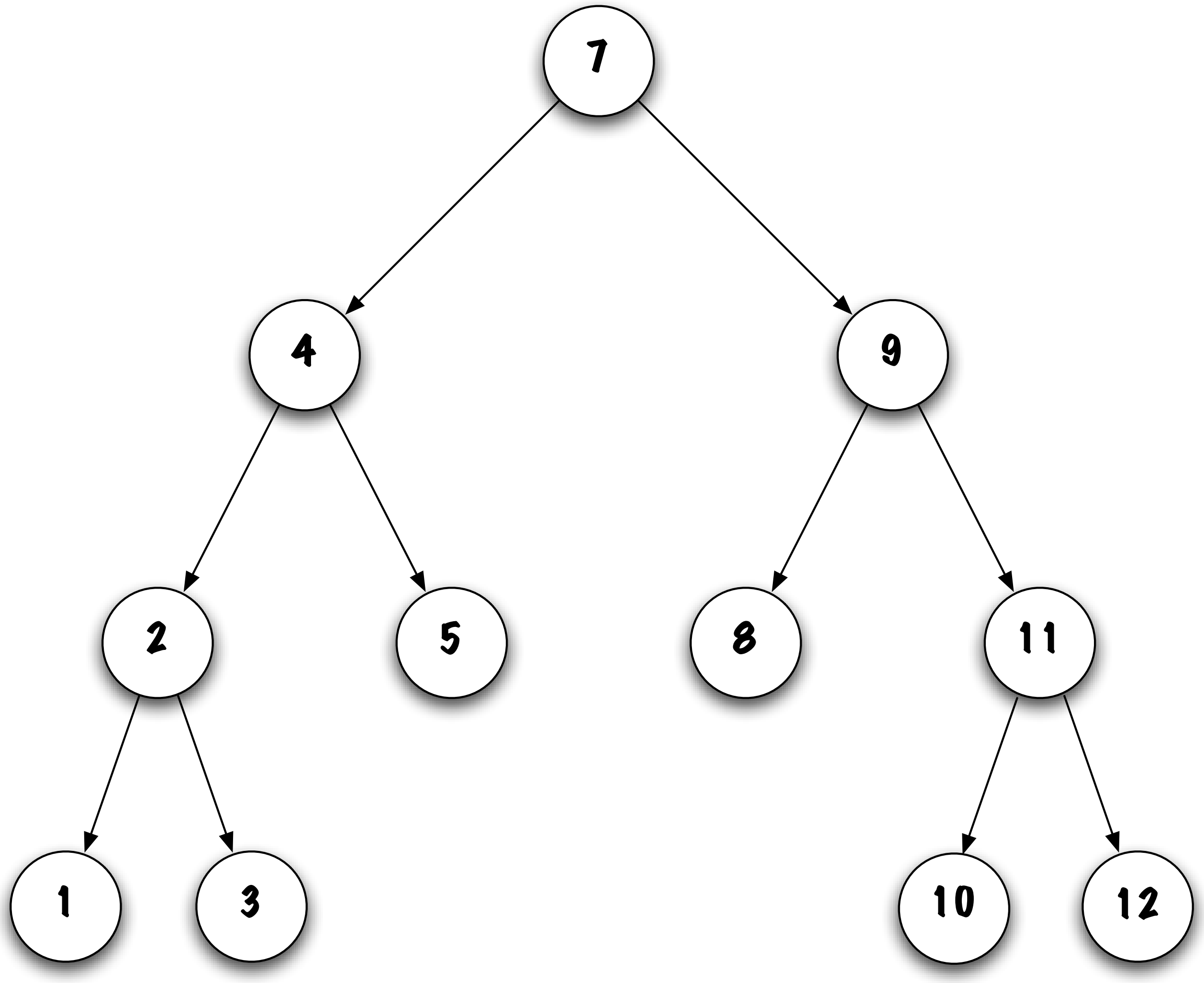
Case 4: t.remove(new Integer(6))

Before:



Case 4: t.remove(new Integer(6))

After:



Node<E> remove(E obj)

```
// pre-condtion:

if ( obj == null ) {
    throw new IllegalArgumentException( "null" );
}

if ( root == null ) {
    throw new NoSuchElementException( );
}
```

Node<E> remove(E obj)

```
// Replacing the root element (special case)

if ( obj.compareTo( root.value ) == 0 ) {

    root = removeTopMost( root );

}
```

Node<E> remove(E obj)

```
} else { // obj is not found at the root

    Node<E> current, parent = root;

    if ( obj.compareTo( root.value ) < 0 ) {
        current = root.left;
    } else {
        current = root.right;
    }

    // ...
```

Node<E> remove(E obj)

```
// ...
while ( current != null ) {
    int test = obj.compareTo( current.value );
    if ( test == 0 ) {
        if ( current == parent.left ) {
            parent.left = removeTopMost( current );
        } else {
            parent.right = removeTopMost( current );
        }
        current = null; // stopping criteria
    } else {
        parent = current;
        if ( test < 0 ) {
            current = parent.left;
        } else {
            current = parent.right;
        }
    }
}
```

Node<E> removeTopMost(Node<E> current)

```
private Node<E> removeTopMost( Node<E> current ) {

    Node<E> top;

    if ( current.left == null ) {
        top = current.right;
    } else if ( current.right == null ) {
        top = current.left;
    } else {
        current.value = getLeftMost( current.right );
        current.right = removeLeftMost( current.right );
        top = current;
    }
    return top;
}
```

E getLeftMost(Node<E> current)

```
private E getLeftMost( Node<E> current ) {

    if ( current == null ) {
        throw new IllegalArgumentException( "null" );
    }

    if ( current.left == null ) {
        return current.value;
    }

    return getLeftMost( current.left );
}
```

Node<E> removeLeftMost(Node<E> current)

```
private Node<E> removeLeftMost( Node<E> current ) {

    if ( current.left == null ) {
        return current.right;
    }
    Node<E> top = current, parent = current;
    current = current.left;

    while ( current.left != null ) {
        parent = current;
        current = current.left;
    }

    parent.left = current.right;
    return top;
}
```

Alternative implementation

```
public void remove( E obj ) {
    Node<E> parent = null, current = root; boolean done = false;
    while ( current != null ) {
        int test = obj.compareTo( current.value );
        if ( test == 0 ) {
            Node<E> newTop = removeTopMost( current );
            if ( current == root ) {
                root = newTop;
            } else if ( current == parent.left ) {
                parent.left = newTop;
            } else {
                parent.right = newTop;
            }
            current = null;
        } else {
            parent = current;
            if ( test < 0 ) {
                current = parent.left;
            } else {
                current = parent.right;
            }
        }
    }
}
```

Remove (recursive implementation)

```
public void remove( E obj ) {

    // pre-condtion:
    if ( obj == null ) {
        throw new IllegalArgumentException( "null" );
    }

    root = remove( root, obj );
}
```

```java
private Node<E> remove( Node<E> current, E obj ) {
    Node<E> result = current;
    int test = obj.compareTo( current.value );
    if ( test == 0 ) {
        if ( current.left == null ) {
            result = current.right;
        } else if ( current.right == null ) {
            result = current.left;
        } else {
            current.value = getLeftMost( current.right );
            current.right = remove( current.right, current.value );
        }
    } else if ( test < 0 ) {
        current.left = remove( current.left, obj );
    } else {
        current.right = remove( current.right, obj );
    }
    return result;
}
```

Iterators

```
java.util.Iterator<E> i = t.iterator();

while ( i.hasNext() ) {
    System.out.println( i.next() );
}
```

But which order?

Iterators

```
Iterator<E> i = t.preOrderIterator();

while ( i.hasNext() ) {
    System.out.println( i.next() );
}
```

Iterators

Strategy?

Iterators

```
private class PreOrderIterator implements Iterator<E> {

    private Stack<E> trail;

    private PreOrderIterator() {
        trail = new LinkedStack<E>();
        if ( root != null ) {
           trail.push( root );
        }
    }
    // ...
```

Iterators

```
public boolean hasNext() {
    return ! trail.isEmpty();
}
```

Iterators

```
public E next() {

    if ( trail.isEmpty() ) {
        throw new NoSuchElementException();
    }

    Node<E> current = trail.pop();

    if ( current.right != null ) {
        trail.push( current.right );
    }
    if ( current.left != null ) {
        trail.push( current.left );
    }
    return current.value;
}
```

Remarks

A variety of trees exist, including self-balancing search trees (AVL, Red-Black, B).

A general tree is a tree such that its nodes are allowed to have more than two children.