ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of March 24, 2013

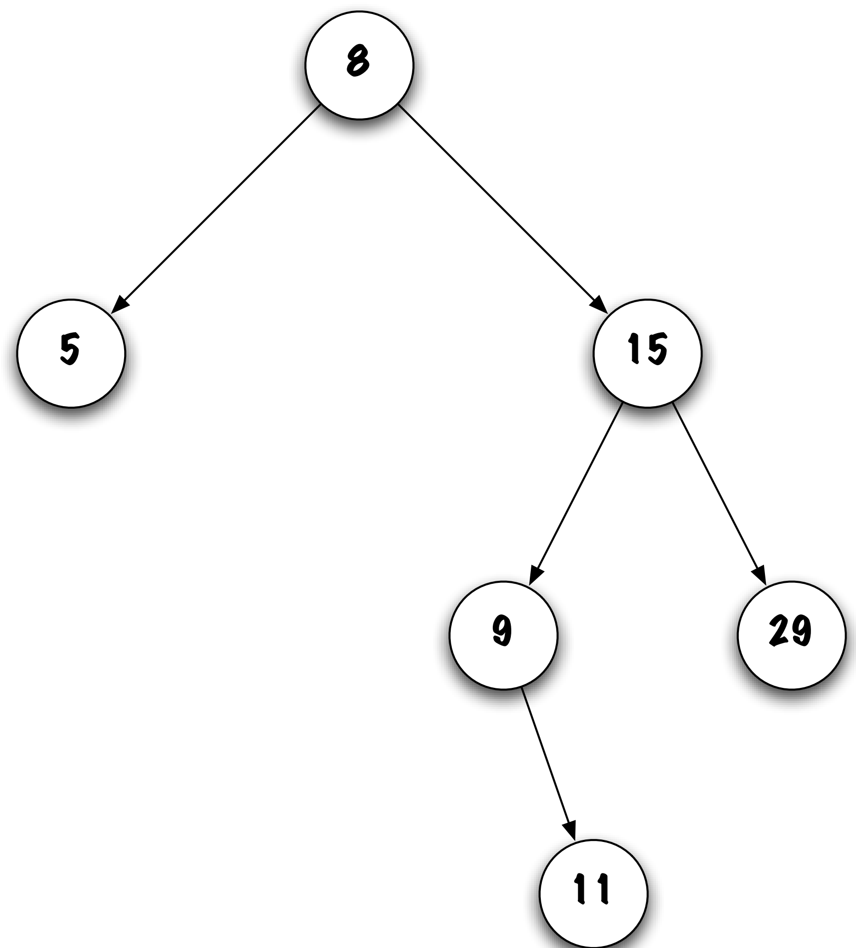
Abstract

- Binary search tree (part I)

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

Binary tree

A **binary tree** is a tree-like (hierarchical) data structure such that each **node** stores a **value** and has at most two children, which are called **left** and **right**.



Applications (general trees)

Applications (general trees)

- Representing hierarchical information such as hierarchical file systems (directories have sub-directories), programs (parse trees);

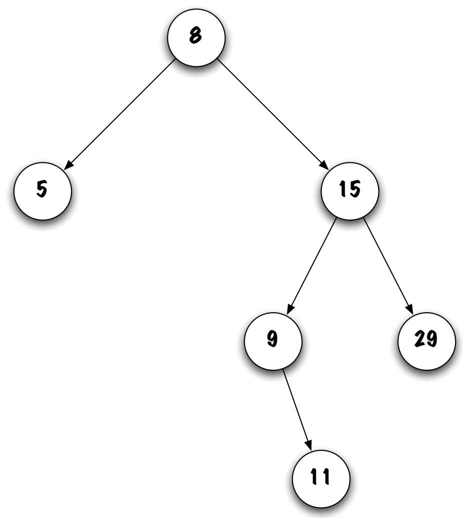
Applications (general trees)

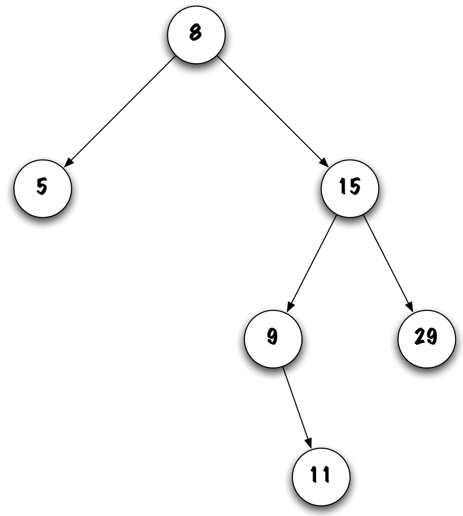
- Representing hierarchical information such as hierarchical file systems (directories have sub-directories), programs (parse trees);

- Huffman trees are used for (de-)compressing information (files);

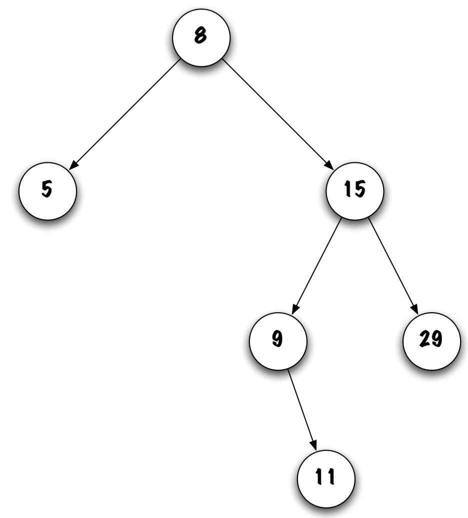
Applications (general trees)

- Representing hierarchical information such as hierarchical file systems (directories have sub-directories), programs (parse trees);
- Huffman trees are used for (de-)compressing information (files);
- An efficient data structure to implement abstract data types such as heaps, priority queues and sets.


8
5
15
9
29
11

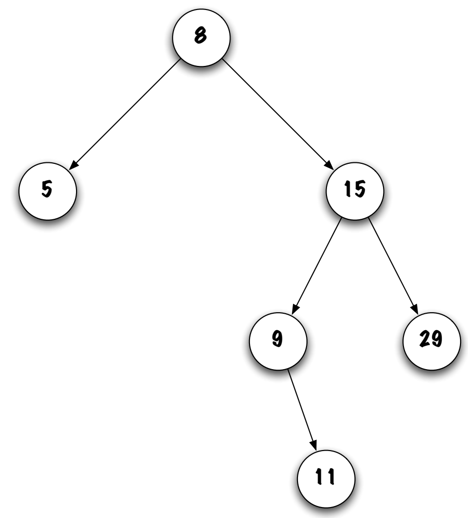


All the nodes except one have exactly one parent.



All the nodes except one have exactly one parent.

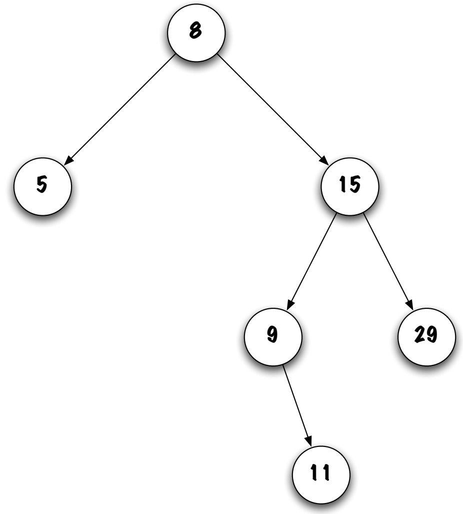
That node that has no parent is called the **root** (which is drawn at the top of the diagram).



All the nodes except one have exactly one parent.

That node that has no parent is called the **root** (which is drawn at the top of the diagram).

Each node has 0, 1 or 2 children.

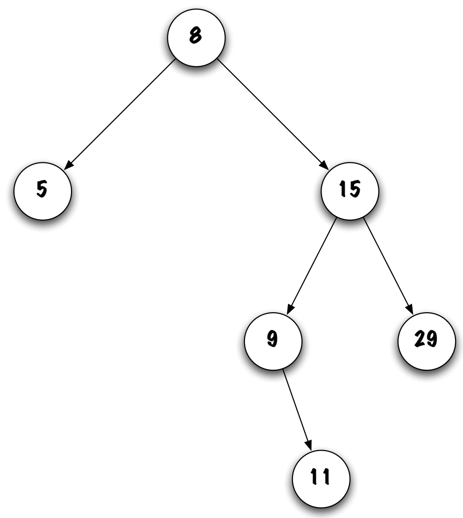


All the nodes except one have exactly one parent.

That node that has no parent is called the **root** (which is drawn at the top of the diagram).

Each node has 0, 1 or 2 children.

Nodes that have no children are called **leaves** (or external nodes).



All the nodes except one have exactly one parent.

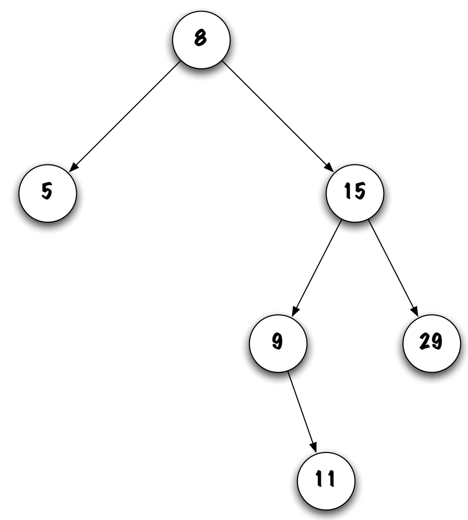
That node that has no parent is called the **root** (which is drawn at the top of the diagram).

Each node has 0, 1 or 2 children.

Nodes that have no children are called **leaves** (or external nodes).

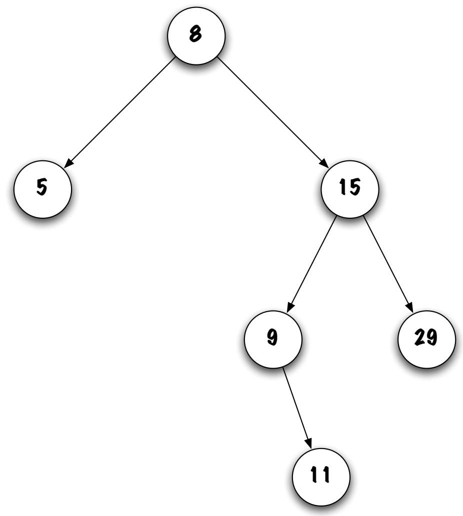
Links between nodes are called **branches**.

Binary tree



A node and its descendants is called a **subtree**.

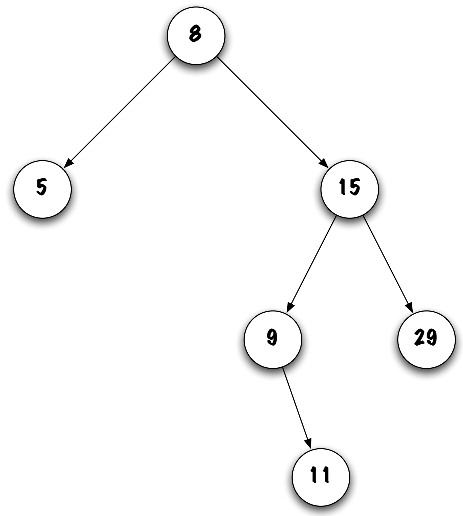
Binary tree



A node and its descendants is called a **subtree**.

The **size** of tree is the number of nodes in the tree.

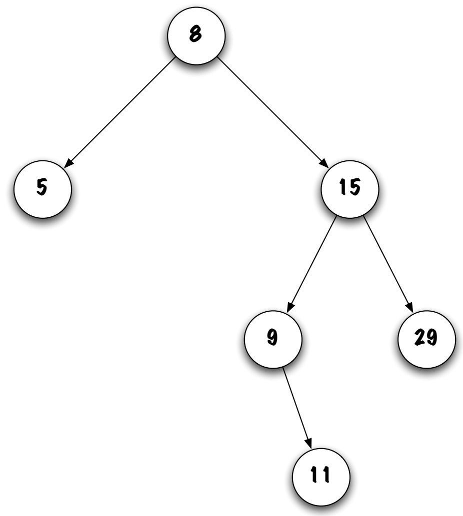
Binary tree



A node and its descendants is called a **subtree**.

The **size** of tree is the number of nodes in the tree. An **empty** tree has size 0.

Binary tree



A node and its descendants is called a **subtree**.

The **size** of tree is the number of nodes in the tree. An **empty** tree has size 0.

Since the discussion is restricted to binary trees, we will sometimes use the word tree to mean a binary tree.

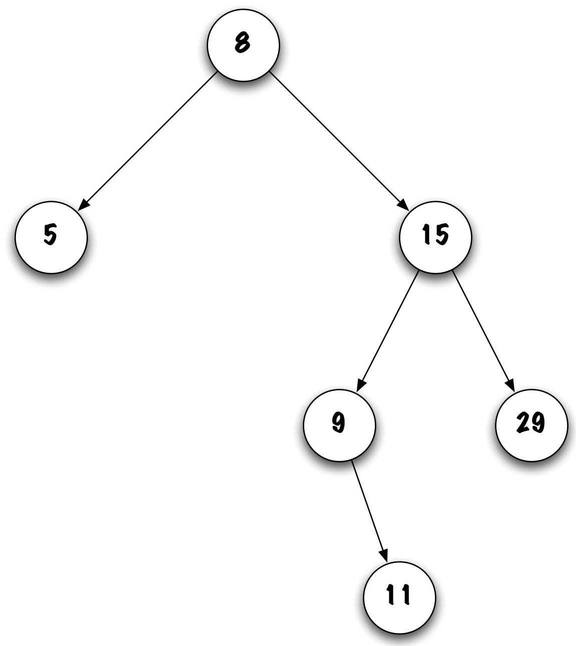
Binary tree

Binary trees can be defined recursively,

- A binary tree is empty, or;
- A binary tree consists of a value as well as two sub-trees;

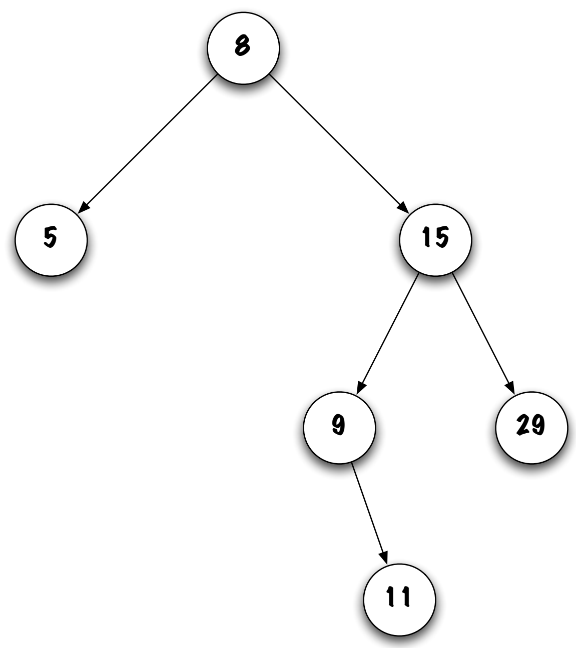
Binary tree

The **depth of a node** is the number of links starting from the root that must be followed to reach that node. The root is the most accessible node.



Binary tree

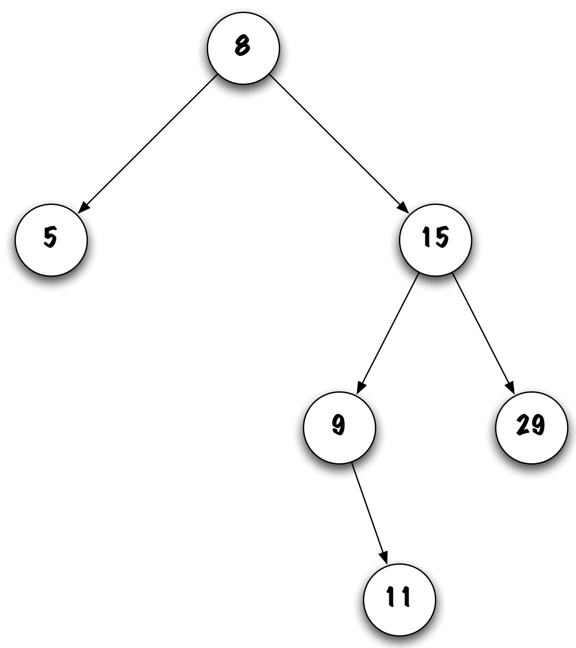
The **depth of a node** is the number of links starting from the root that must be followed to reach that node. The root is the most accessible node.



What is the depth of the root?

Binary tree

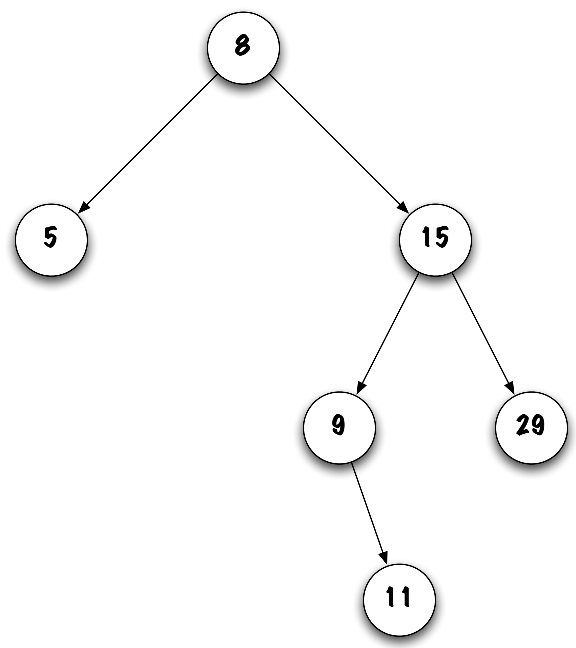
The **depth of a node** is the number of links starting from the root that must be followed to reach that node. The root is the most accessible node.



What is the depth of the root? The root always has a depth of 0.

Binary tree

The **depth of a node** is the number of links starting from the root that must be followed to reach that node. The root is the most accessible node.

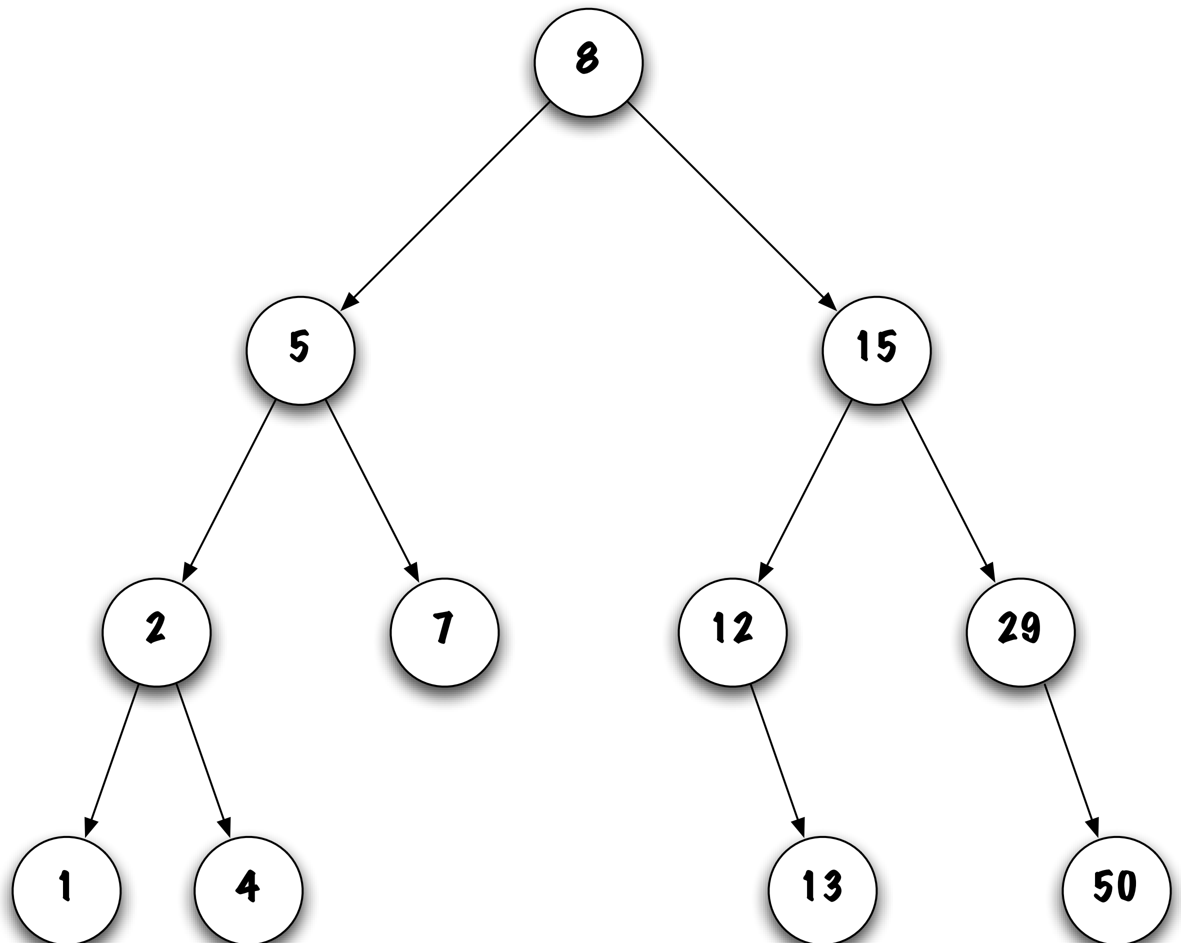


What is the depth of the root? The root always has a depth of 0.

The **depth of a tree** is the depth of the deepest node.

Binary tree

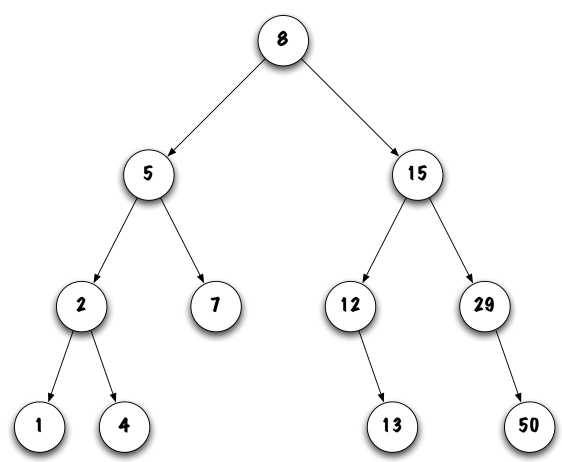
All the trees presented thus far exhibit a certain property, what is it?



Binary search tree

A **binary search tree** is a binary tree such that,

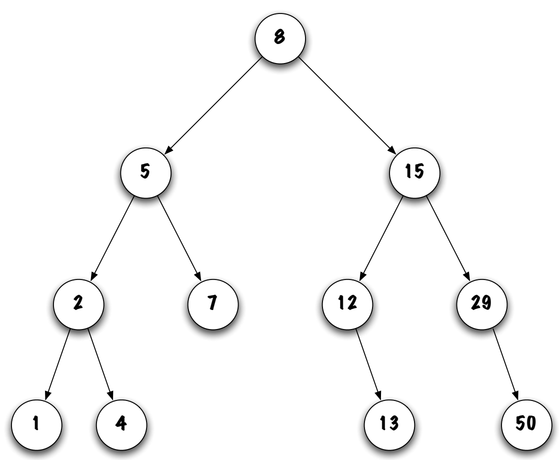
- the nodes of a left sub-tree contain elements that are less than the element stored at the local root (or is empty);
- the nodes of a right sub-tree contain elements that are greater than the element stored at the local root (or is empty).



Binary search tree

A **binary search tree** is a binary tree such that,

- the nodes of a left sub-tree contain elements that are less than the element stored at the local root (or is empty);
- the nodes of a right sub-tree contain elements that are greater than the element stored at the local root (or is empty).



The definition precludes duplicate values.

Binary search tree

Implementing a binary search tree, what is needed?

Binary search tree

Implementing a binary search tree, what is needed?

That's right, we need a class **Node**.

Binary search tree

Implementing a binary search tree, what is needed?

That's right, we need a class **Node**. What are its instance variables?

Binary search tree

Implementing a binary search tree, what is needed?

That's right, we need a class **Node**. What are its instance variables?

Its instance variables are **value**, **left** and **right**.

Binary search tree

Implementing a binary search tree, what is needed?

That's right, we need a class **Node**. What are its instance variables?

Its instance variables are **value**, **left** and **right**.

What are the types of these variables?

Binary search tree

Implementing a binary search tree, what is needed?

That's right, we need a class **Node**. What are its instance variables?

Its instance variables are **value**, **left** and **right**.

What are the types of these variables? **value** should be **Comparable**, **left** and **right** should be of type **Node**.

Binary search tree

A static nested class to store the elements of the tree.

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

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

Binary search tree

Instance variable(s) of the class **BinarySearchTree**?

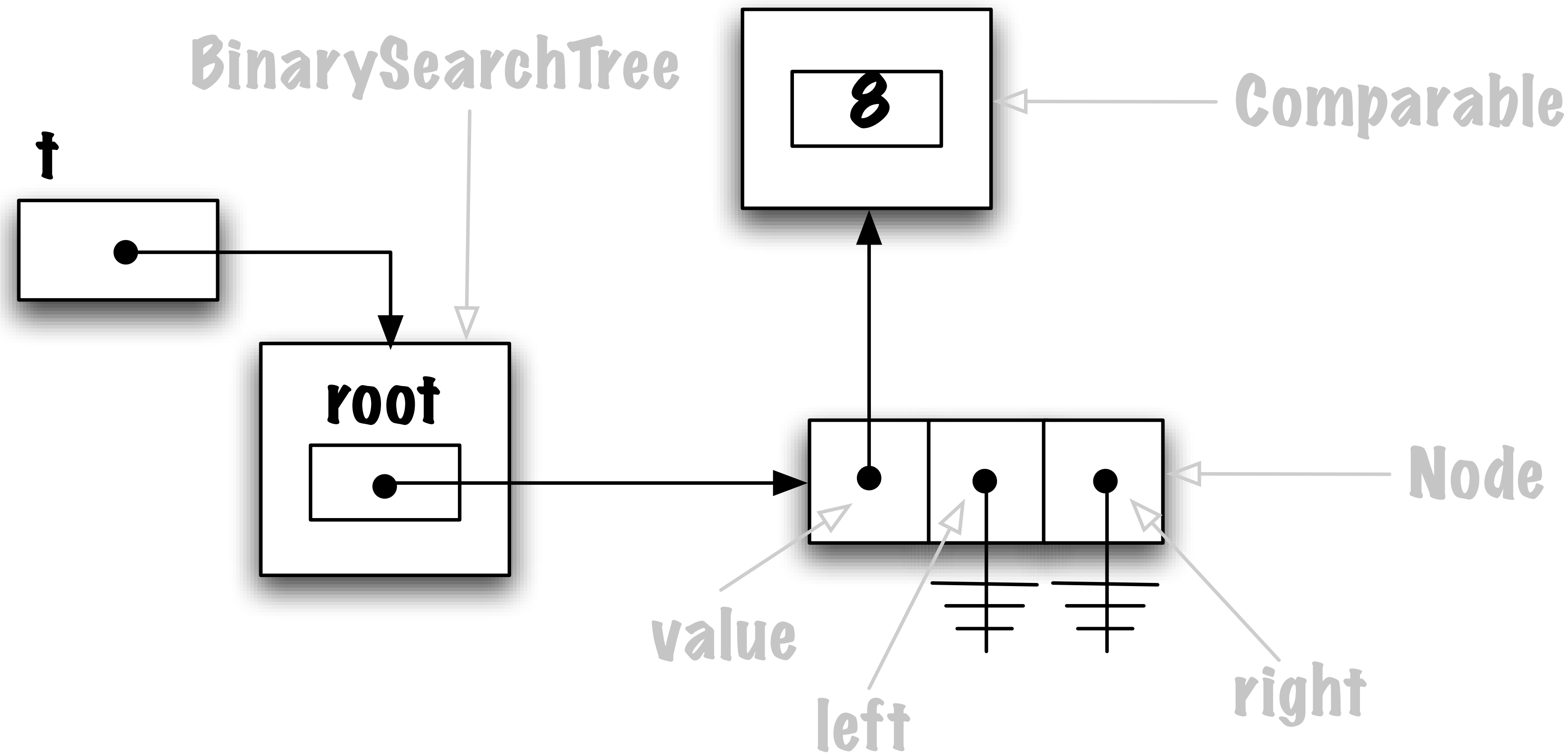
```
public class BinarySearchTree< E extends Comparable<E> > {

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

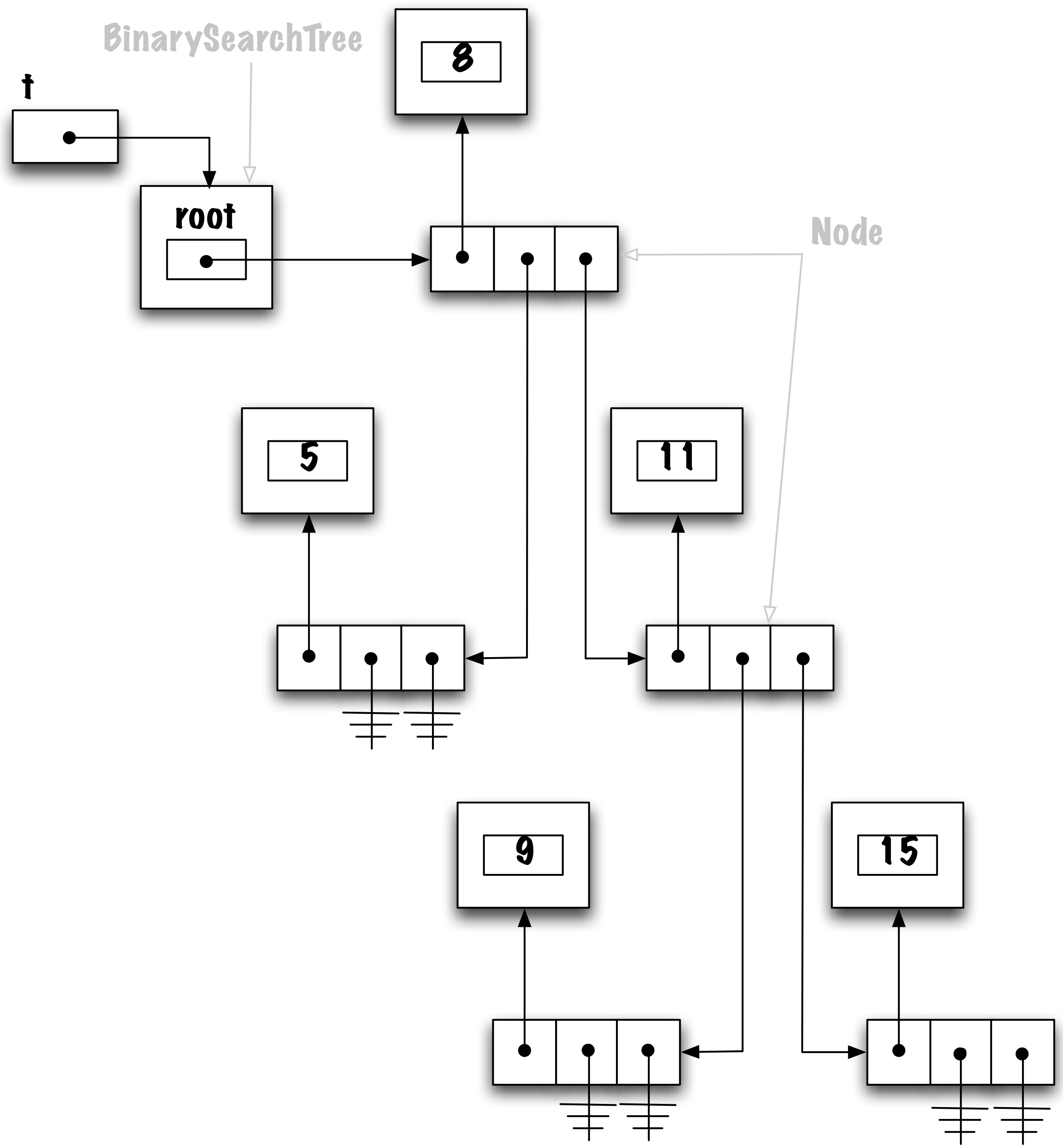
Binary search tree

Instance variable(s) of the class **BinarySearchTree**?

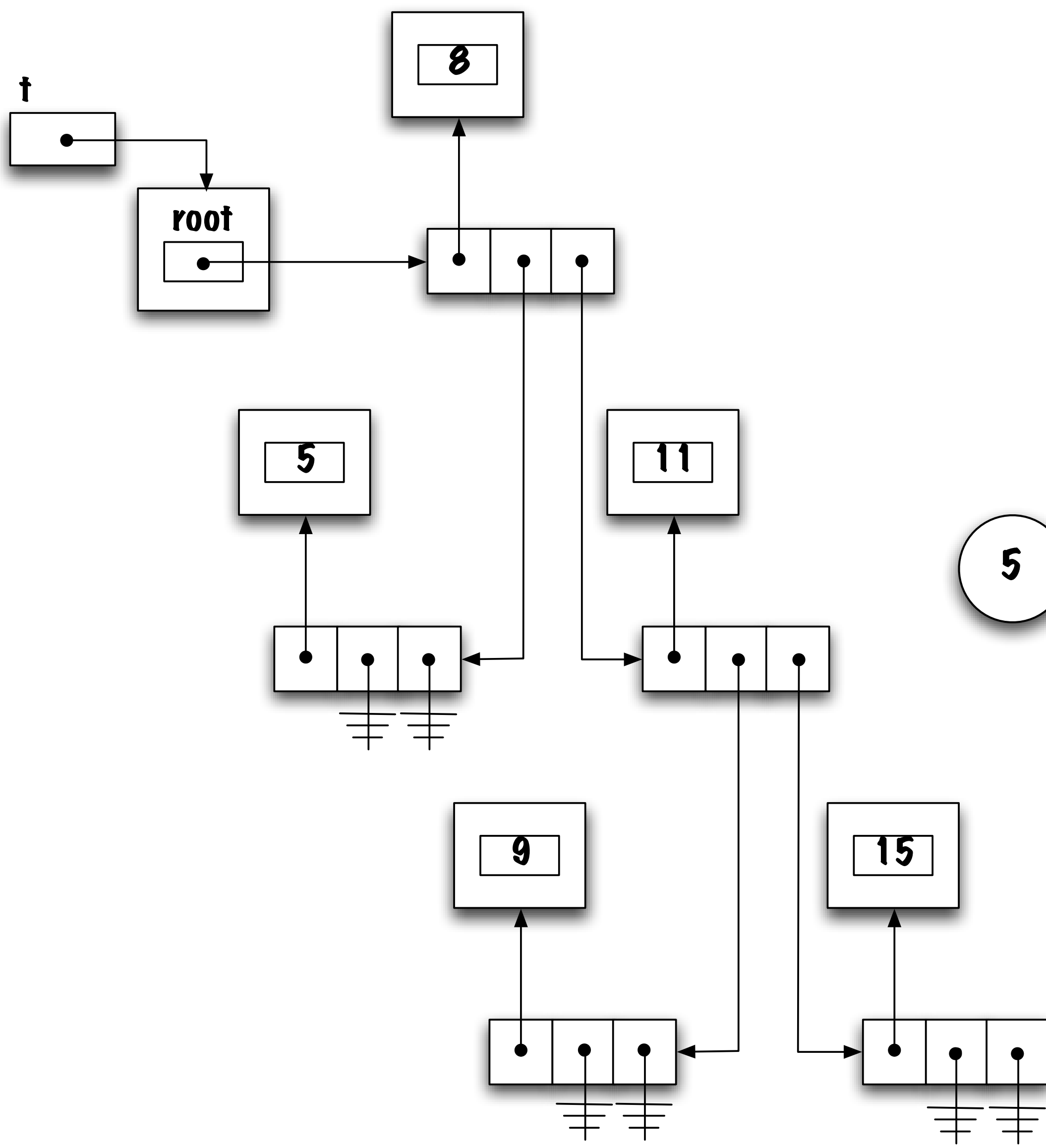
```
public class BinarySearchTree< E extends Comparable<E> > {

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

    private Node<E> root;
```

Memory diagram

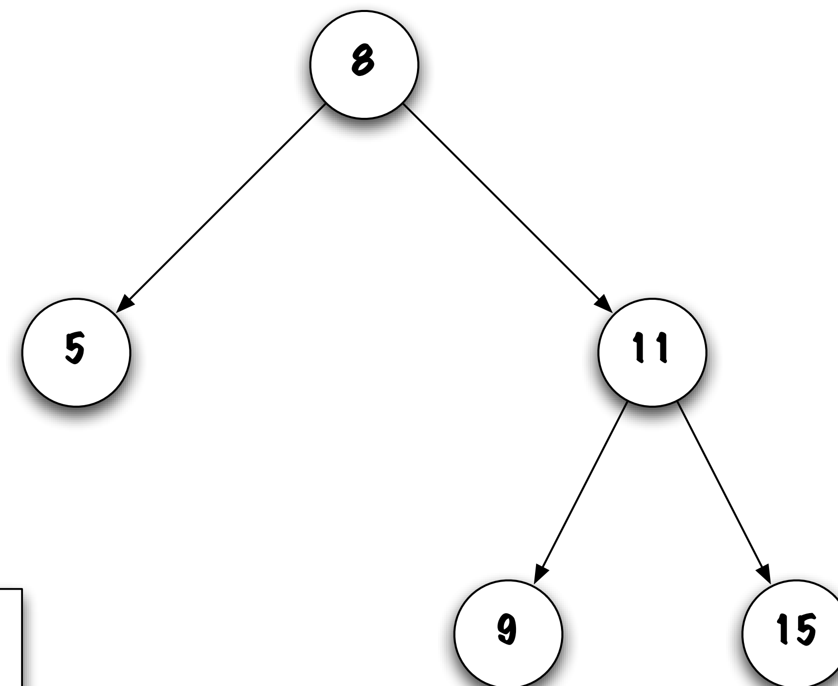




BinarySearchTree
t
root
8
Node
5
11
9
15



t
root
8
5
11
9
15



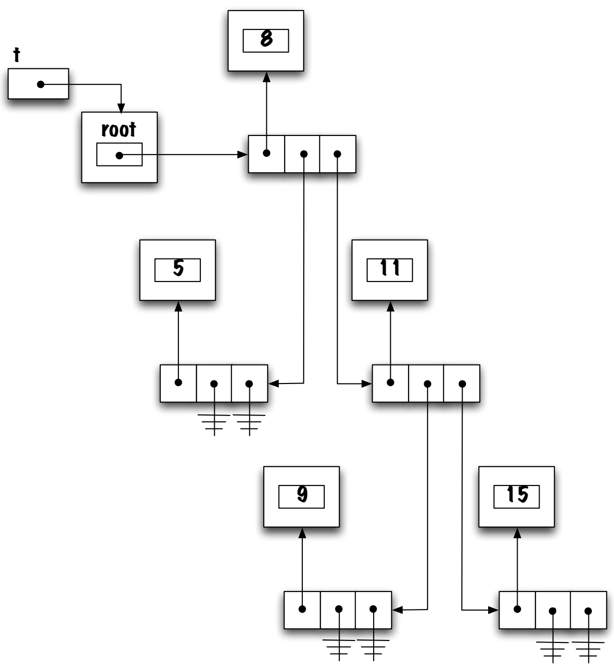
8
5
11
9
15

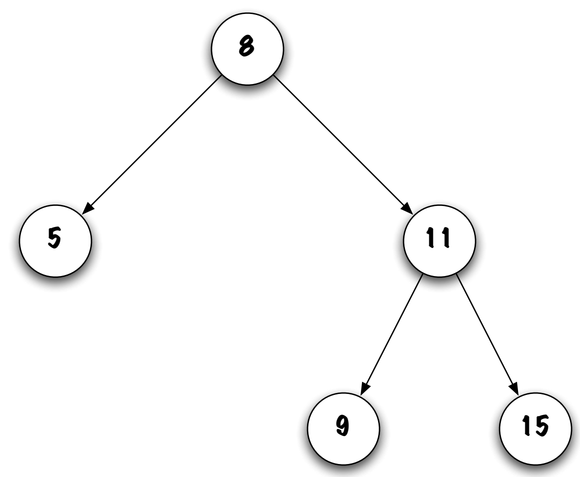
Observations

A **leaf** is a **Node** such that both its descendant reference variables (**left** and **right**) are **null**.

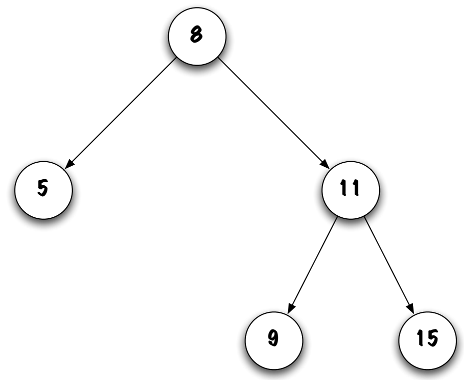
The reference **root** can be **null**, in which case the tree is empty, i.e. has size 0.

For brevity, we will often use the more abstract representation on the right.

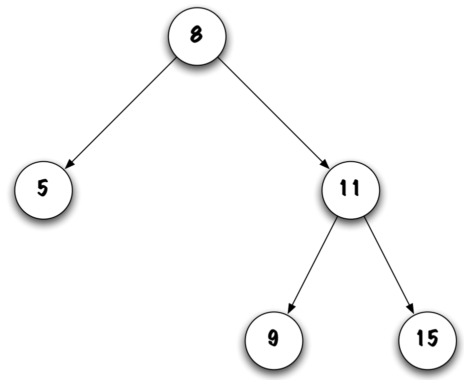




boolean contains(E obj)

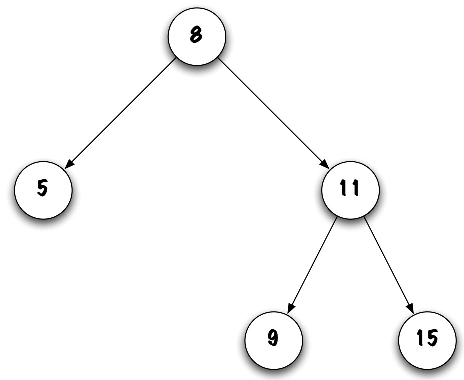


boolean contains(E obj)



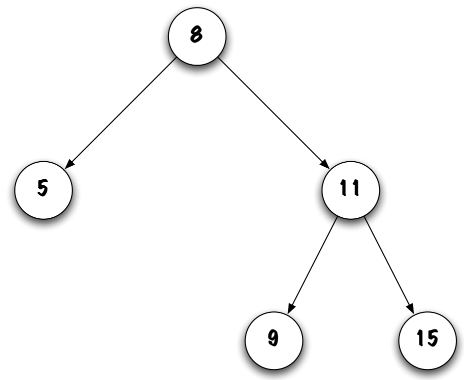
1. Empty tree?

boolean contains(E obj)



1. Empty tree? **obj** not found;

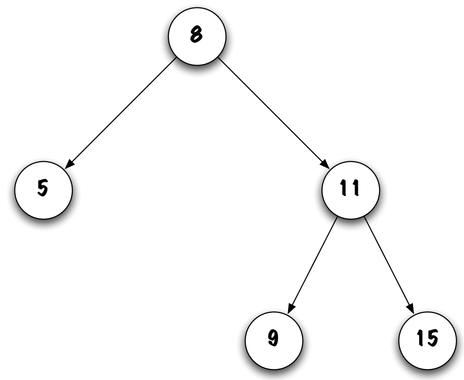
boolean contains(E obj)



1. Empty tree? **obj** not found;

2. The root contains **obj**?

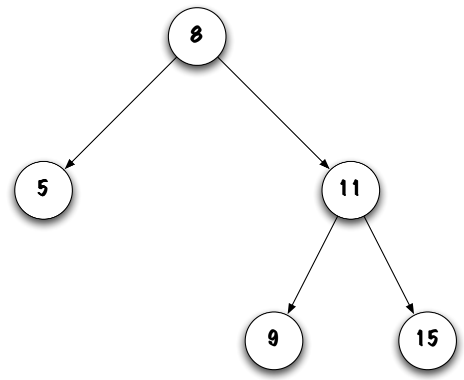
boolean contains(E obj)



1. Empty tree? **obj** not found;

2. The root contains **obj**? **obj** was found;

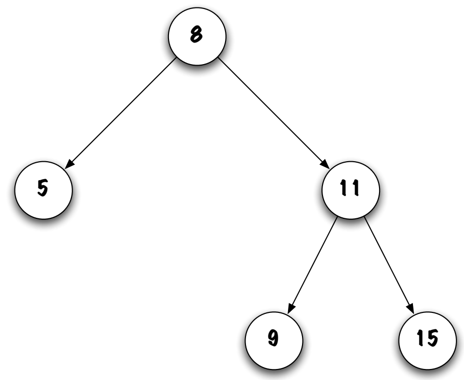
boolean contains(E obj)



1. Empty tree? **obj** not found;

2. The root contains **obj**? **obj** was found; Otherwise?

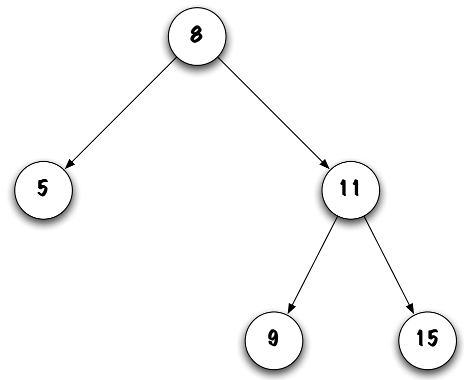
boolean contains(E obj)



1. Empty tree? **obj** not found;

2. The root contains **obj**? **obj** was found; Otherwise? Where should you start looking?

boolean contains(E obj)

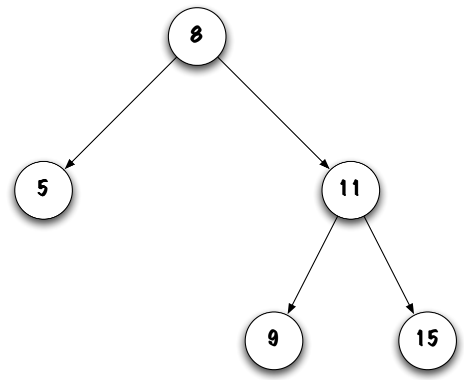


1. Empty tree? **obj** not found;

2. The root contains **obj**? **obj** was found; Otherwise? Where should you start looking?

3. If **obj** is less than the value found at the root?

boolean contains(E obj)

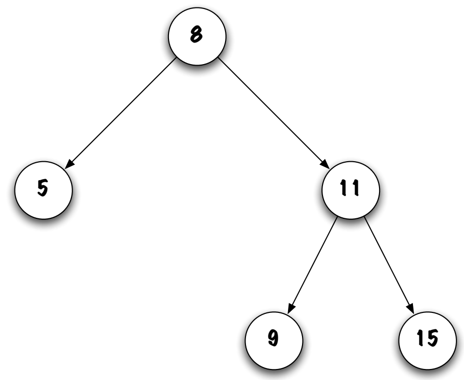


1. Empty tree? **obj** not found;

2. The root contains **obj**? **obj** was found; Otherwise? Where should you start looking?

3. If **obj** is less than the value found at the root? Look for **obj** in the left sub-tree;

boolean contains(E obj)



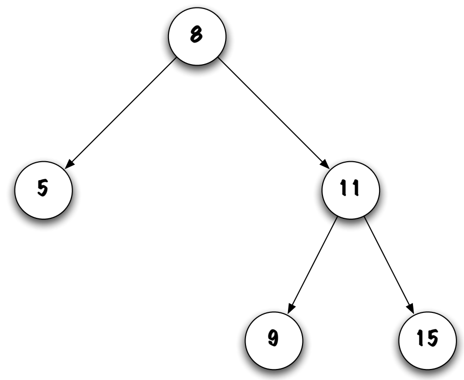
1. Empty tree? **obj** not found;

2. The root contains **obj**? **obj** was found; Otherwise? Where should you start looking?

3. If **obj** is less than the value found at the root? Look for **obj** in the left sub-tree;

4. Else (**obj** must be larger than the value stored at the root)?

boolean contains(E obj)



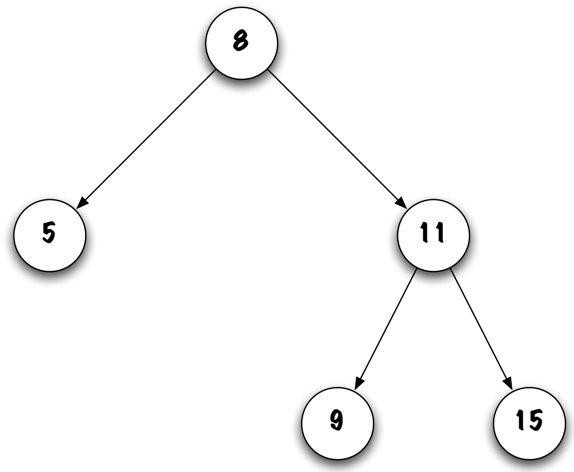
1. Empty tree? **obj** not found;

2. The root contains **obj**? **obj** was found; Otherwise? Where should you start looking?

3. If **obj** is less than the value found at the root? Look for **obj** in the left sub-tree;

4. Else (**obj** must be larger than the value stored at the root)? Look for **obj** in the right sub-tree.

boolean contains(E obj)



Exercise: apply the algorithm for finding the values 8, 9 and 7.

public boolean contains(E obj)

The above presentation suggests a recursive algorithm.

public boolean contains(E obj)

The above presentation suggests a recursive algorithm. What will be the signature of the method?

public boolean contains(E obj)

The above presentation suggests a recursive algorithm. What will be the signature of the method?

```
public boolean contains( E obj ) {
    // pre-condition:
    if ( obj == null ) {
        throw new IllegalArgumentException( "null" );
    }
    return contains( obj, root );
}
```

Similarly to the methods for recursive list processing, these methods will consist of two parts, a starter method as well as a **private** method whose signature is augmented with a parameter of type **Node**.

boolean contains(Node<E> current, E obj)

Base case(s):

boolean contains(Node<E> current, E obj)

Base case(s):

```
if ( current == null ) {
    result = false;
}
```

boolean contains(Node<E> current, E obj)

Base case(s):

```
if ( current == null ) {
    result = false;
}
```

but also

boolean contains(Node<E> current, E obj)

Base case(s):

```
if ( current == null ) {
    result = false;
}
```

but also

```
if ( obj.compareTo( current.value ) == 0 ) {
    result = true;
}
```

boolean contains(Node<E> current, E obj)

General case:

boolean contains(Node<E> current, E obj)

General case: Look left or right (recursively).

boolean contains(Node<E> current, E obj)

General case: Look left or right (recursively).

```
if ( obj.compareTo( current.value ) < 0 ) {
     result = contains( current.left, obj );
} else {
     result = contains( current.right, obj );
}
```

boolean contains(Node$<$E$>$ current, E obj)

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

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

Develop a strategy.

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

Develop a strategy.

1. Use a temporary variable **current** of type **Node**;

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

Develop a strategy.

1. Use a temporary variable **current** of type **Node**;
2. Initialise this variable to designate the **root** of the tree;

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

Develop a strategy.

1. Use a temporary variable **current** of type **Node**;
2. Initialise this variable to designate the **root** of the tree;
3. If **current** is **null** the **obj** was not found, end;

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

Develop a strategy.

1. Use a temporary variable **current** of type **Node**;
2. Initialise this variable to designate the **root** of the tree;
3. If **current** is **null** the **obj** was not found, end;
4. If **current.value** is the value we're looking for, end;

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

Develop a strategy.

1. Use a temporary variable **current** of type **Node**;
2. Initialise this variable to designate the **root** of the tree;
3. If **current** is **null** the **obj** was not found, end;
4. If **current.value** is the value we're looking for, end;
5. If the value is smaller than that of the current node,

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

Develop a strategy.

1. Use a temporary variable **current** of type **Node**;
2. Initialise this variable to designate the **root** of the tree;
3. If **current** is **null** the **obj** was not found, end;
4. If **current.value** is the value we're looking for, end;
5. If the value is smaller than that of the current node, **current = current.left**,

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

Develop a strategy.

1. Use a temporary variable **current** of type **Node**;
2. Initialise this variable to designate the **root** of the tree;
3. If **current** is **null** the **obj** was not found, end;
4. If **current.value** is the value we're looking for, end;
5. If the value is smaller than that of the current node, **current = current.left**, go to 3;

public boolean contains(E obj) (take 2)

Should the method **boolean contains(Comparable o)** be necessarily recursive?
No.

Develop a strategy.

1. Use a temporary variable **current** of type **Node**;
2. Initialise this variable to designate the **root** of the tree;
3. If **current** is **null** the **obj** was not found, end;
4. If **current.value** is the value we're looking for, end;
5. If the value is smaller than that of the current node, **current = current.left**, go to 3;
6. Else **current = current.right**, go to 3.

public boolean contains(E obj) (take 2)

```
public boolean contains2( 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;
}
```

Tree traversal

Similarly to lists, it is often necessary to visit all the nodes of a tree and this is called **traversing** the tree.

Tree traversal

Similarly to lists, it is often necessary to visit all the nodes of a tree and this is called **traversing** the tree.

While traversing the tree, operations are applied to each node, we call these operations **visiting** the node.

Tree traversal

Similarly to lists, it is often necessary to visit all the nodes of a tree and this is called **traversing** the tree.

While traversing the tree, operations are applied to each node, we call these operations **visiting** the node.

- $<$ $\underline{\text{Visit the root}}$, $\underline{\text{traverse left sub-tree}}$, $\underline{\text{traverse right sub-tree}}$ $>$ is called **pre-order** traversal;

Tree traversal

Similarly to lists, it is often necessary to visit all the nodes of a tree and this is called **traversing** the tree.

While traversing the tree, operations are applied to each node, we call these operations **visiting** the node.

- $<$ Visit the root, traverse left sub-tree, traverse right sub-tree $>$ is called **pre-order** traversal;

- $<$ Traverse left sub-tree, visit the root, traverse right sub-tree $>$ is called **in-order** traversal;

Tree traversal

Similarly to lists, it is often necessary to visit all the nodes of a tree and this is called **traversing** the tree.

While traversing the tree, operations are applied to each node, we call these operations **visiting** the node.

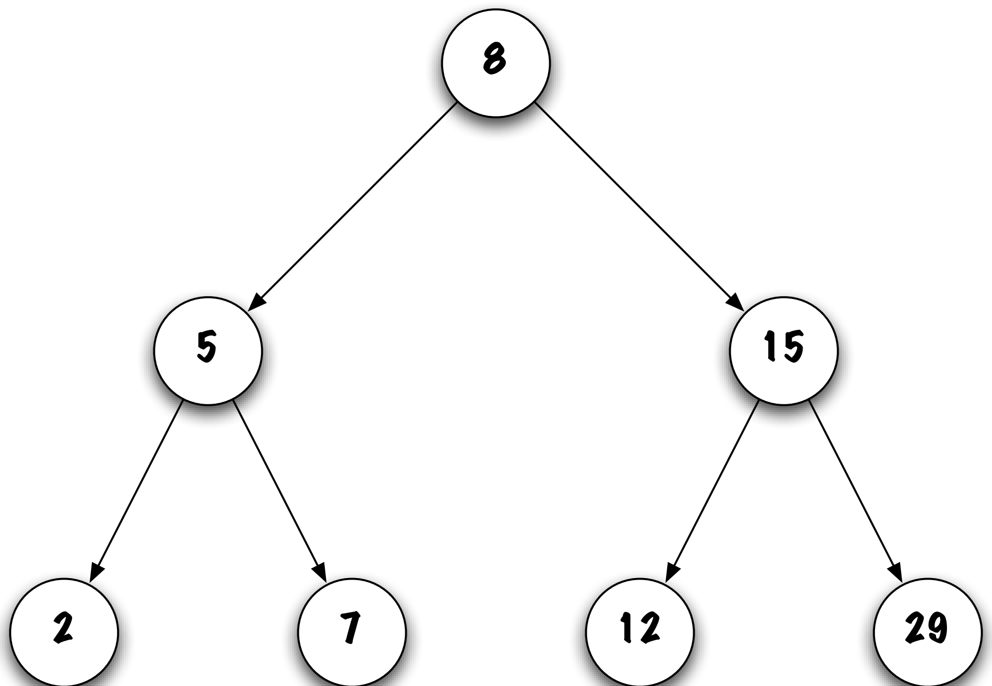
- $<$ Visit the root, traverse left sub-tree, traverse right sub-tree $>$ is called **pre-order** traversal;

- $<$ Traverse left sub-tree, visit the root, traverse right sub-tree $>$ is called **in-order** traversal;

- $<$ Traverse left sub-tree, traverse right sub-tree, visit the root $>$ is called **post-order** traversal;

Exercises

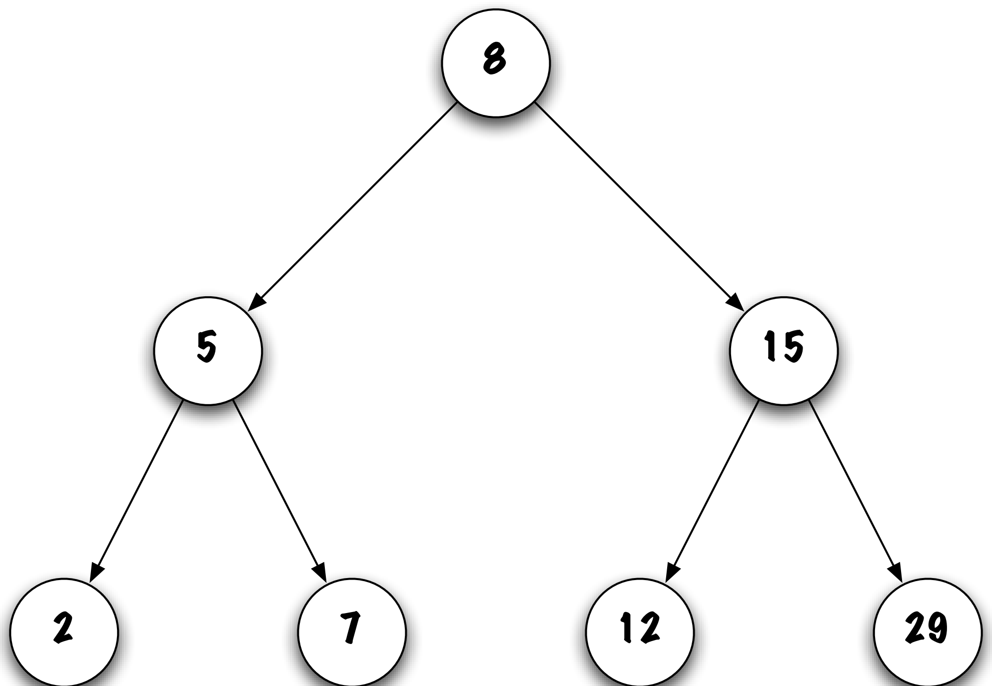
The simplest operation consists of printing the value of the node.



Show the result for each strategy: **pre-order**, **in-order** and **post-order** traversal.

Exercises

The simplest operation consists of printing the value of the node.



Show the result for each strategy: **pre-order**, **in-order** and **post-order** traversal.

Which strategy prints the values in increasing order?