LECTURE 23

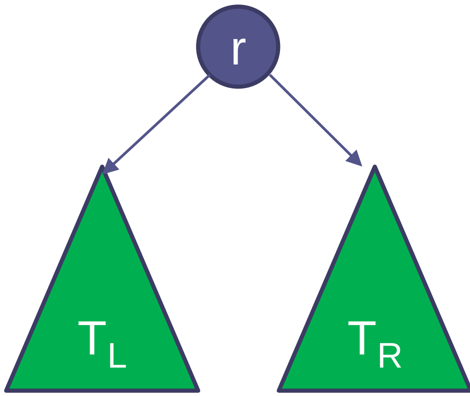
Using binary search trees

Assignment

- Read section 6.4
- Do all self-check exercises (use the code developed at recitations)
- Make sure that you understand every detail of the code presented in the book (and these slides)
- Make sure that you can prove that the number of nodes in a *perfect binary tree* of height n is $2^n - 1$.

Binary Search Tree

- Binary search trees
  - All elements in the left subtree precede those in the right subtree
- A formal definition:



A set of nodes T is a binary search tree if either of the following is true:

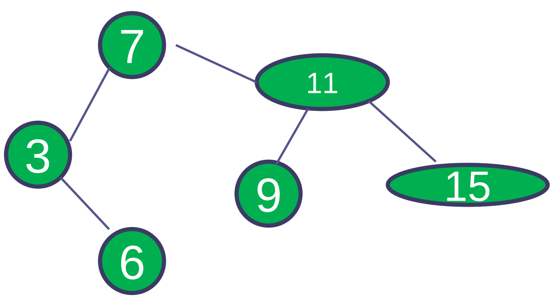
- T is a single node; or
- The root of T has two subtrees, T_L and T_R, such that T_L and T_R are binary search trees and the values in the root node of T is not less than any value in T_L and not greater than any value in T_R

$$\forall x \in T_L, \forall y \in T_R: x \leq r \leq y$$

Note: The definition on p. 300 is inaccurate!

Using binary search trees

- A binary search tree presents a sorted sequence
- New elements must be inserted (or removed) so that the order is maintained



1) insert 5

2) remove 7

Binary Search Tree (BST)

- Searching a balanced BST can be done in O(log *n*) time
- The same algorithm used on an unbalanced tree may take O(*n*) time (Why?)

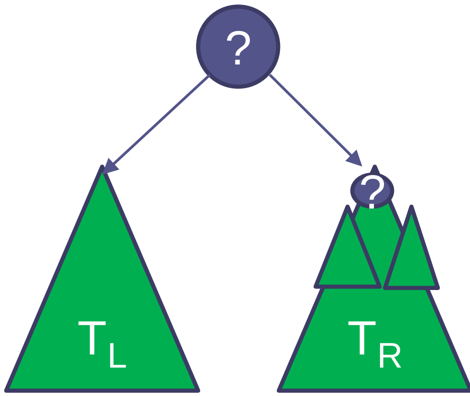
Recursive Algorithm for Searching a Binary Tree

1. **`if`** the tree is empty
2. return null *(target is not found)*

   `else if` the target matches the root node's data
3. return the data stored at the root node

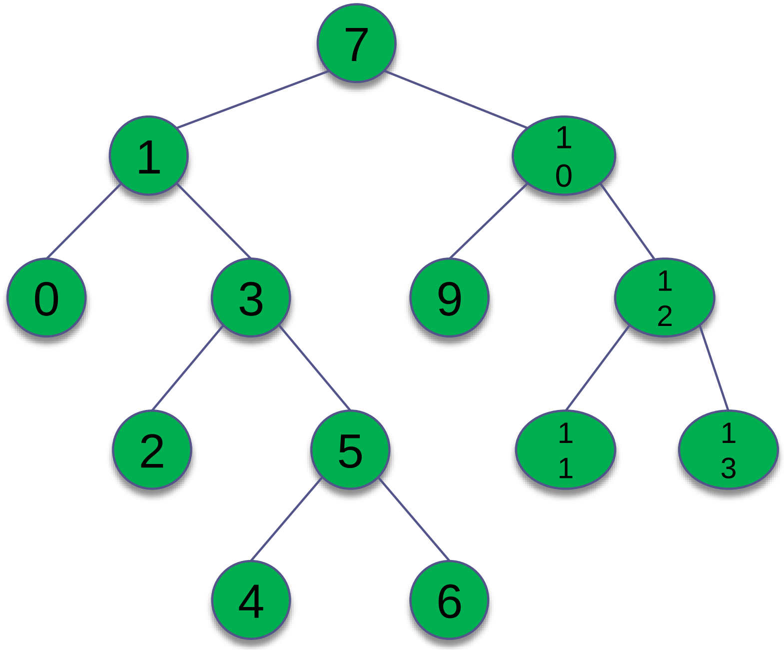
   `else if` the target is less than the root node's data
4. return the result of searching the left subtree of the root

   `else`
5. return the result of searching the right subtree of the root



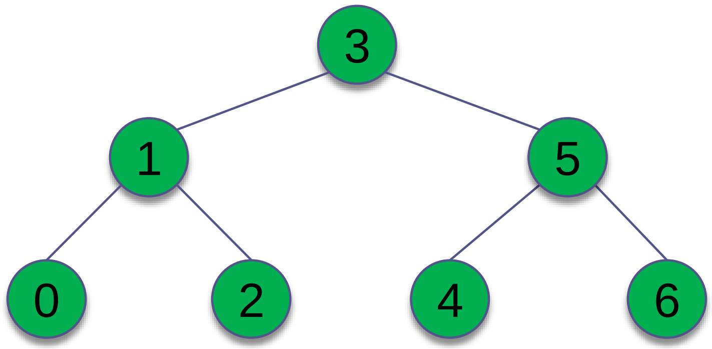
Full binaryTrees

- A full binary tree is a binary tree where each node nodes has either two children or or no children



PerfectBinary Trees

- A *perfect binary tree* is a full binary tree of height n with exactly $2^n - 1$ nodes

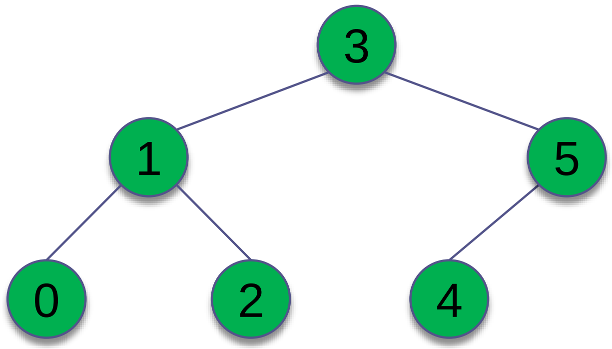


- $1 + 2 + \ldots + 2^{n-1} = 2^n - 1$

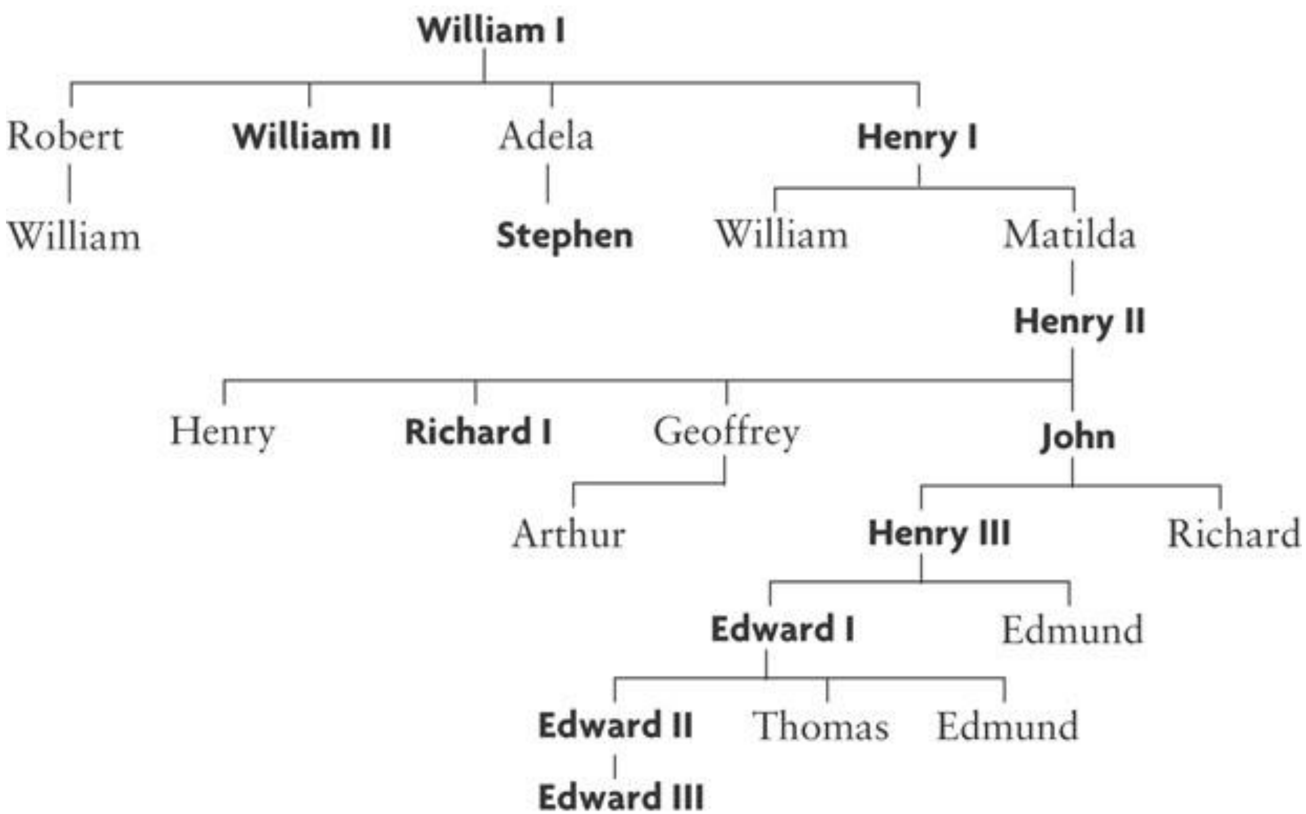
(cf. *Towers of Hanoi*)

Complete Binary Trees

- A *complete binary tree* is a perfect binary tree through level *n* - 1 with some extra leaf nodes at level *n* (the tree height), all toward the left

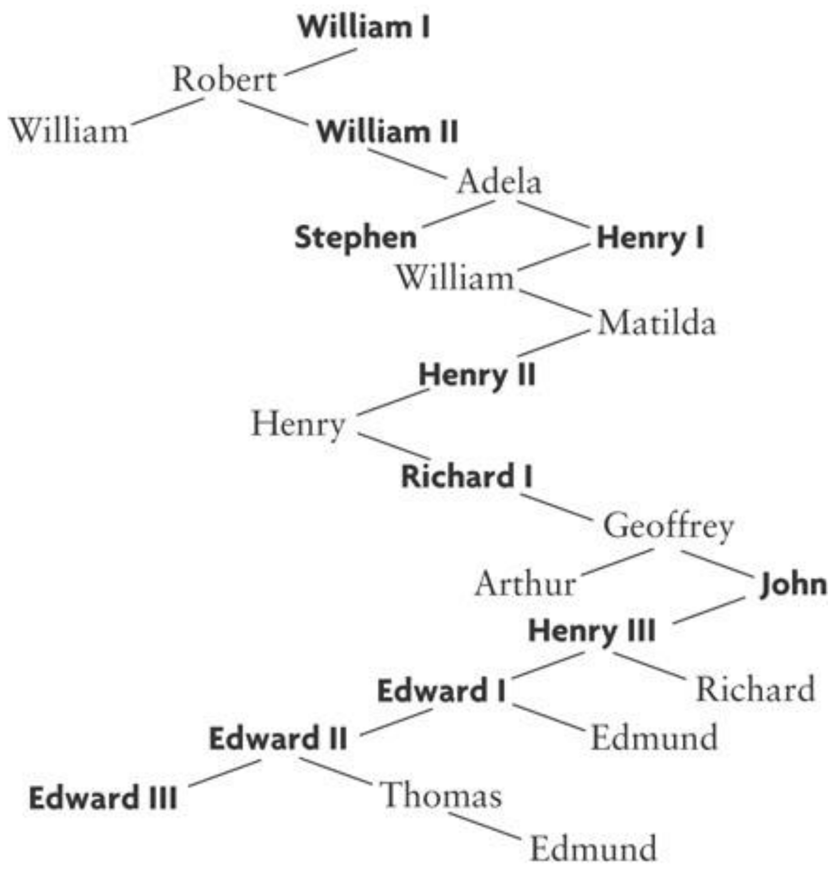


General Trees



General Trees

- A general tree can be converted into a binary tree
- The left branch of a node is the oldest child, and each right branch is connected to the next younger sibling (if any)



Tree Traversals

- How can we list all nodes in a tree?
  - We can do this by walking through the tree in a prescribed order and visiting the nodes as they are encountered
  - This process is called *tree traversal*
- Three common kinds of tree traversal are
  - *Inorder*
  - *Preorder*
  - *Postorder*

Tree Traversals

- *Preorder*: visit root node, traverse T_L, traverse T_R
- *Inorder*: traverse T_L, visit root node, traverse T_R
- *Postorder*: traverse T_L, traverse T_R, visit root node

Algorithm for Preorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree.

Algorithm for Inorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.

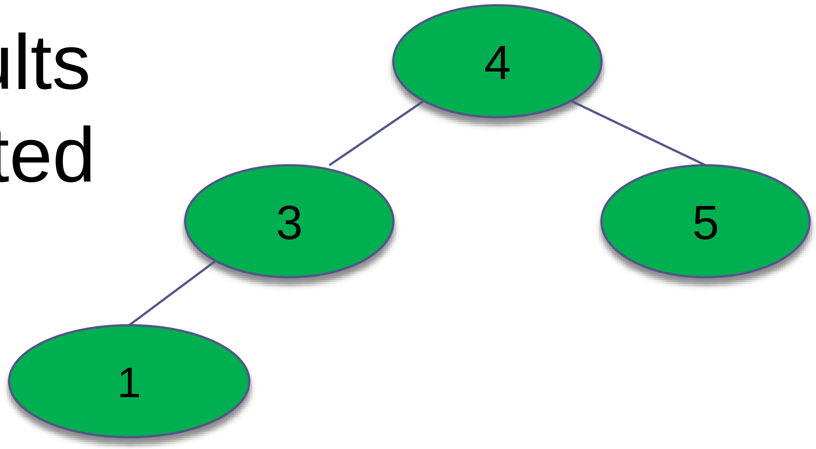
Algorithm for Postorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

Traversals of Binary Search Trees and Expression Trees

- An inorder traversal of a binary search tree results in the nodes being visited in sequence by increasing data value

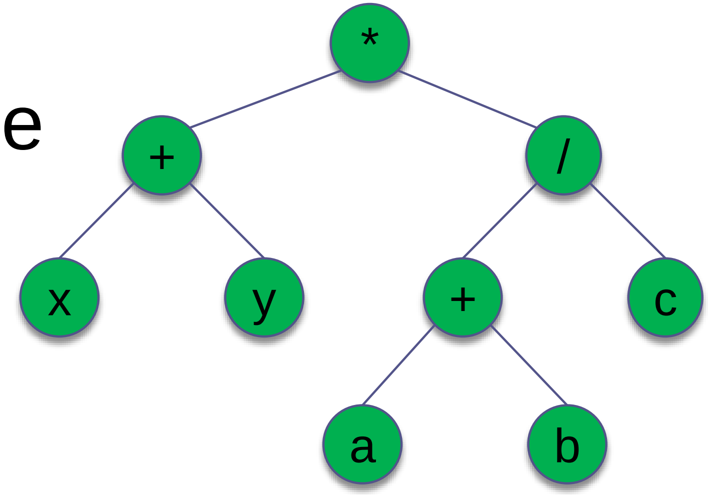


1, 3, 4, 5

Traversals of Binary Search Trees and Expression Trees (cont.)

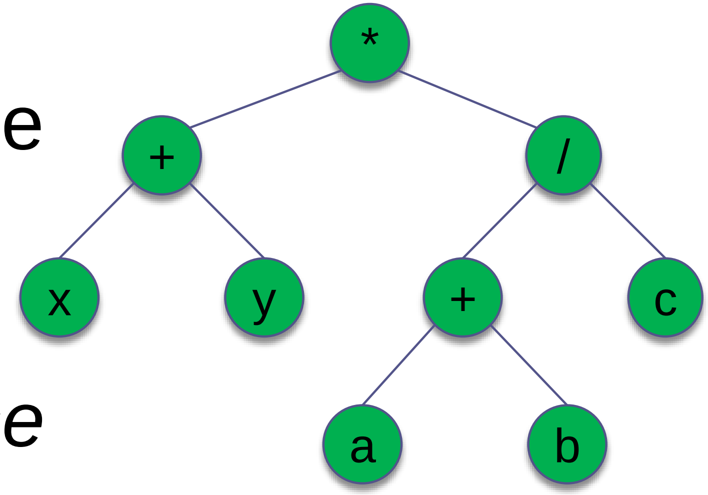
- An inorder traversal of an expression tree results in the sequence
  x + y * a + b / c
- If we insert parentheses where they belong, we get the infix form:

```
(x + y) * ((a + b)) / c)
```



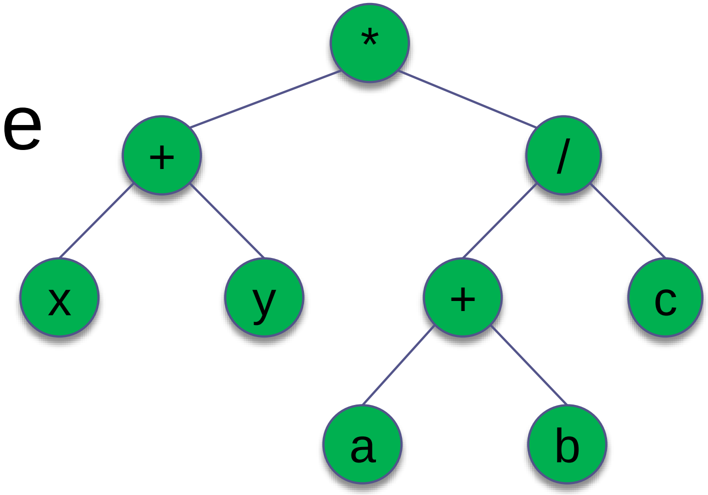
Traversals of Binary Search Trees and Expression Trees (cont.)

- A postorder traversal of an expression tree results in the sequence
  x y + a b + c / *
- This is the *postfix* or *reverse polish* form of the expression
- Operators follow operands



Traversals of Binary Search Trees and Expression Trees (cont.)

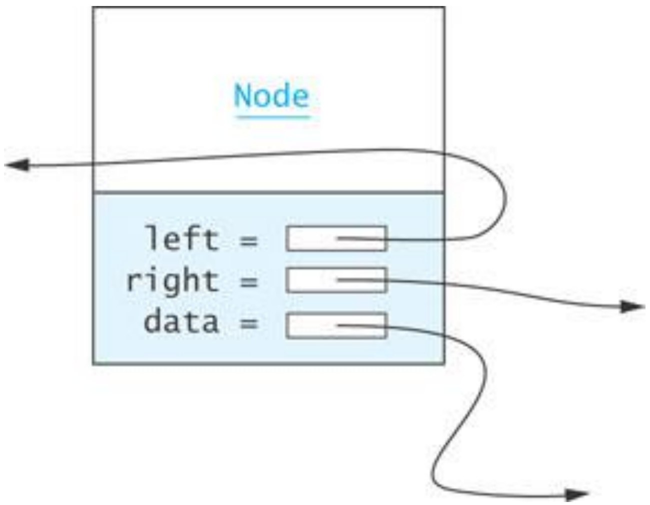
- A preorder traversal of an expression tree results in the sequence
  * + x y / + a b c
- This is the *prefix* or *forward polish* form of the expression
- Operators precede operands



Implementing a `BinaryTree` Class

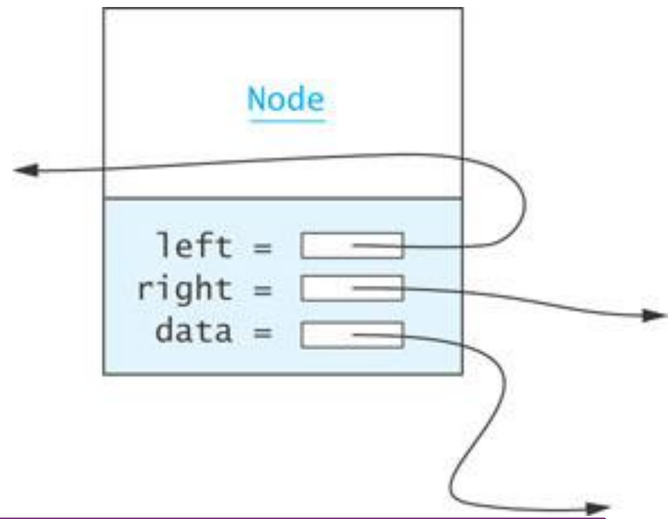
Node<E> **Class**

- Just as in a linked list, a node consists of 1) a data part and 2) the links to successor nodes
- The data part is a reference to type E
- A binary tree node must have links to both its left and right subtrees



Node<E> **Class** (cont.)

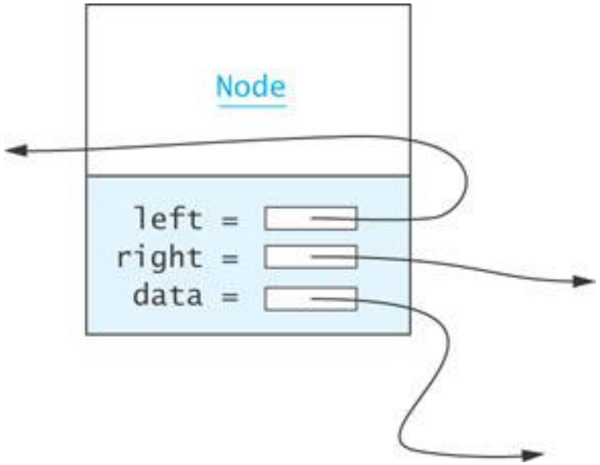
```
protected static class Node<E>
         implements Serializable {
   protected E data;
   protected Node<E> left;
   protected Node<E> right;

   public Node(E data) {
     this.data = data;
     left = null;
     right = null;
   }

   public String toString() {
      return data.toString();
   }
}
```



Node<E> is declared as an inner class within BinaryTree<E>

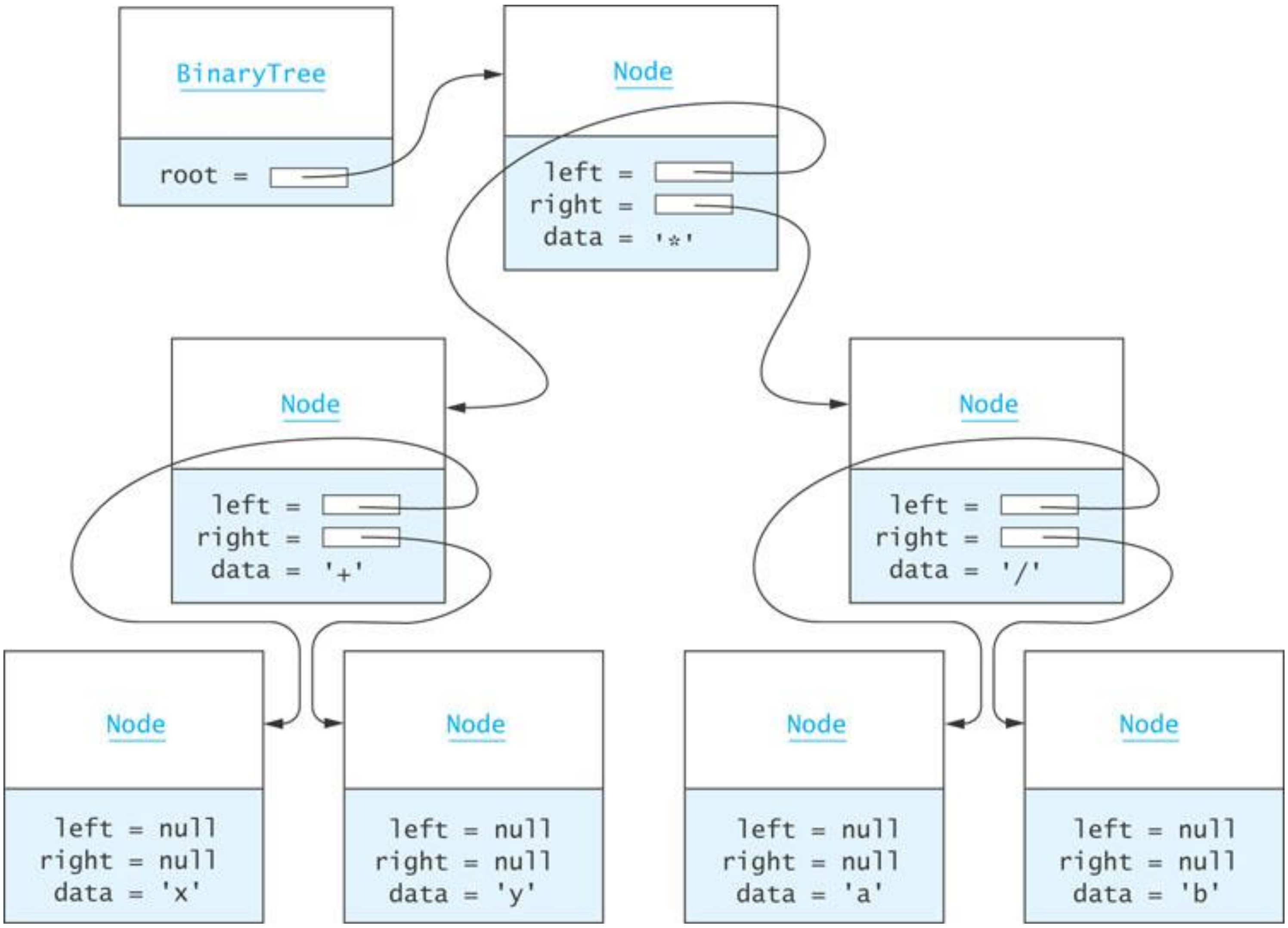
`Node<E>` **Class** (cont.)

```
protected static class Node<E>
         implements Serializable {
   protected E data;
   protected Node<E> left;
   protected Node<E> right;

   public Node(E data) {
     this.data = data;
     left = null;
     right = null;
   }

   public String toString() {
      return data.toString();
   }
}
```



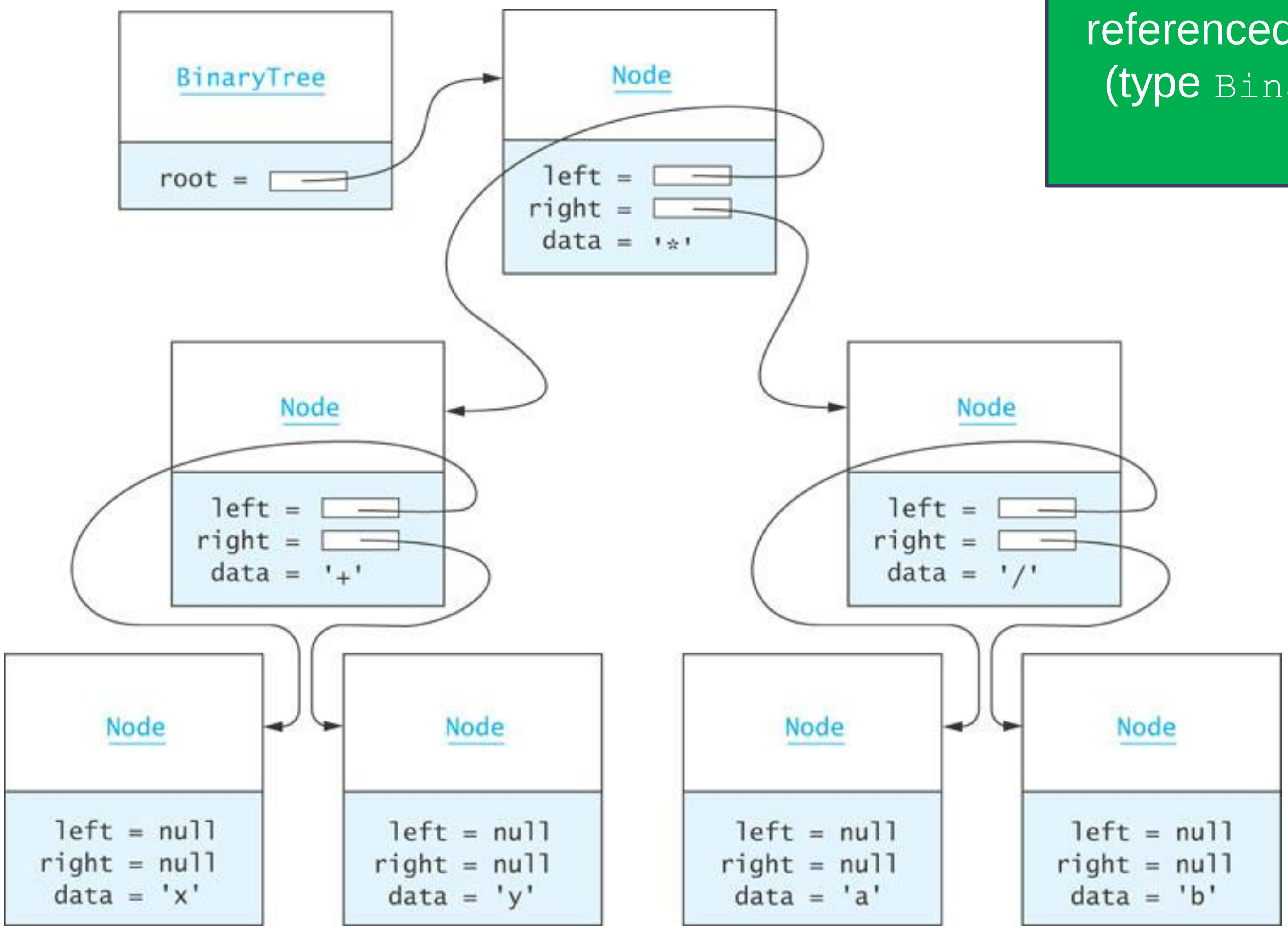
`Node<E>` is declared `protected`. This way we can use it as a superclass.

`BinaryTree<E>` **Class** (cont.)

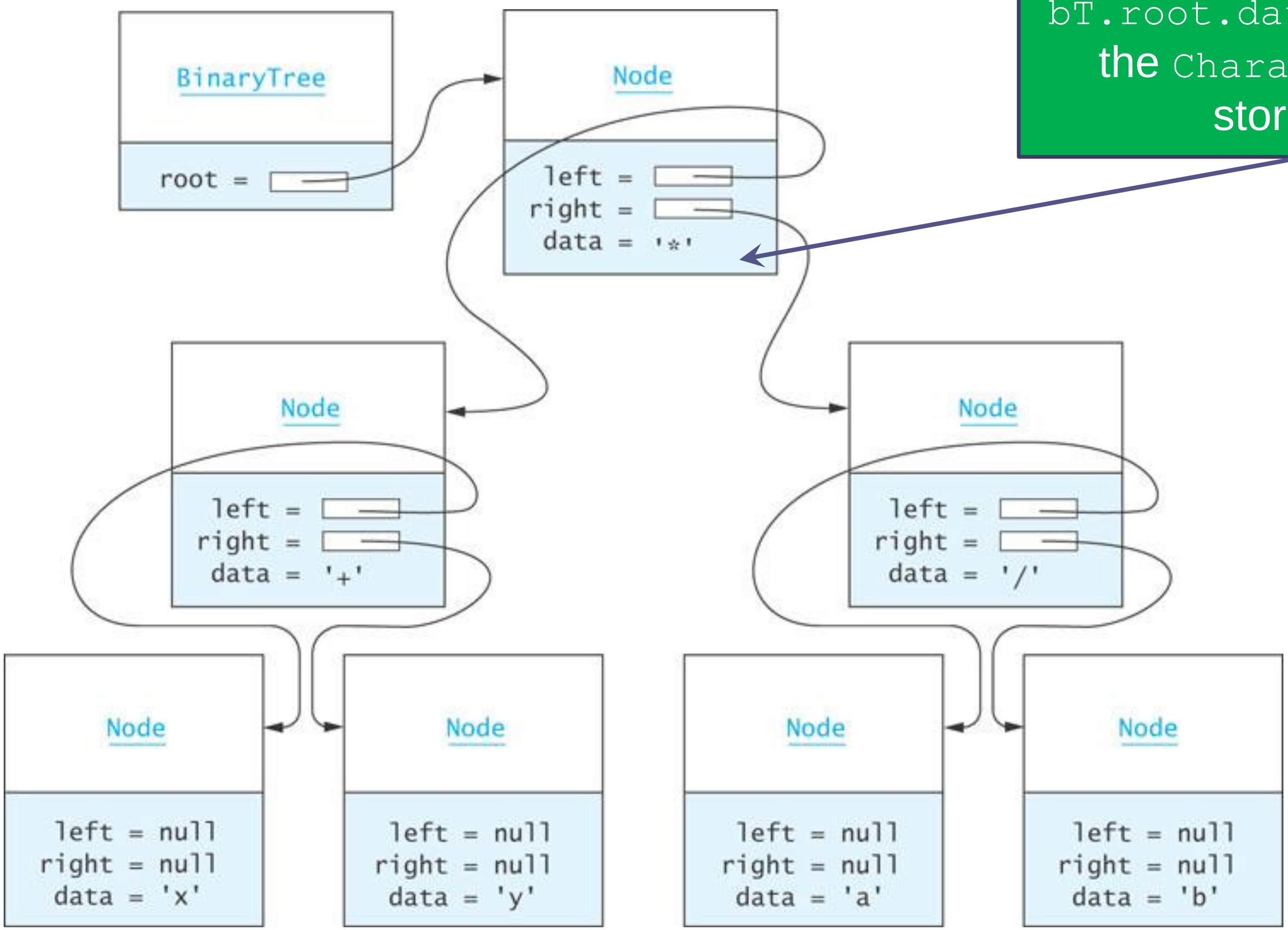


`BinaryTree<E>` Class (cont.)

Assuming the tree is referenced by variable `bT` (type `BinaryTree`) then



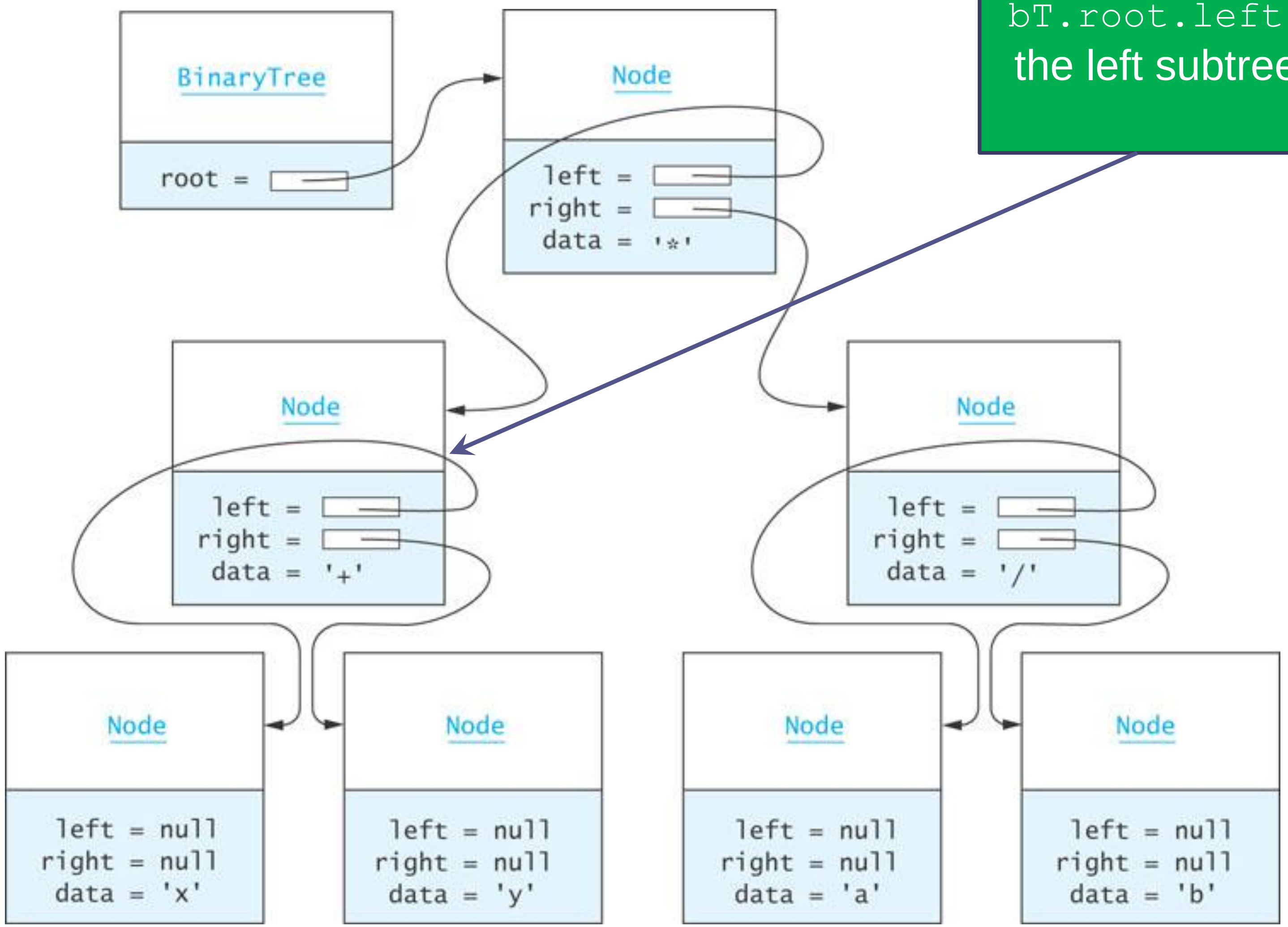
BinaryTree<E> Class (cont.)



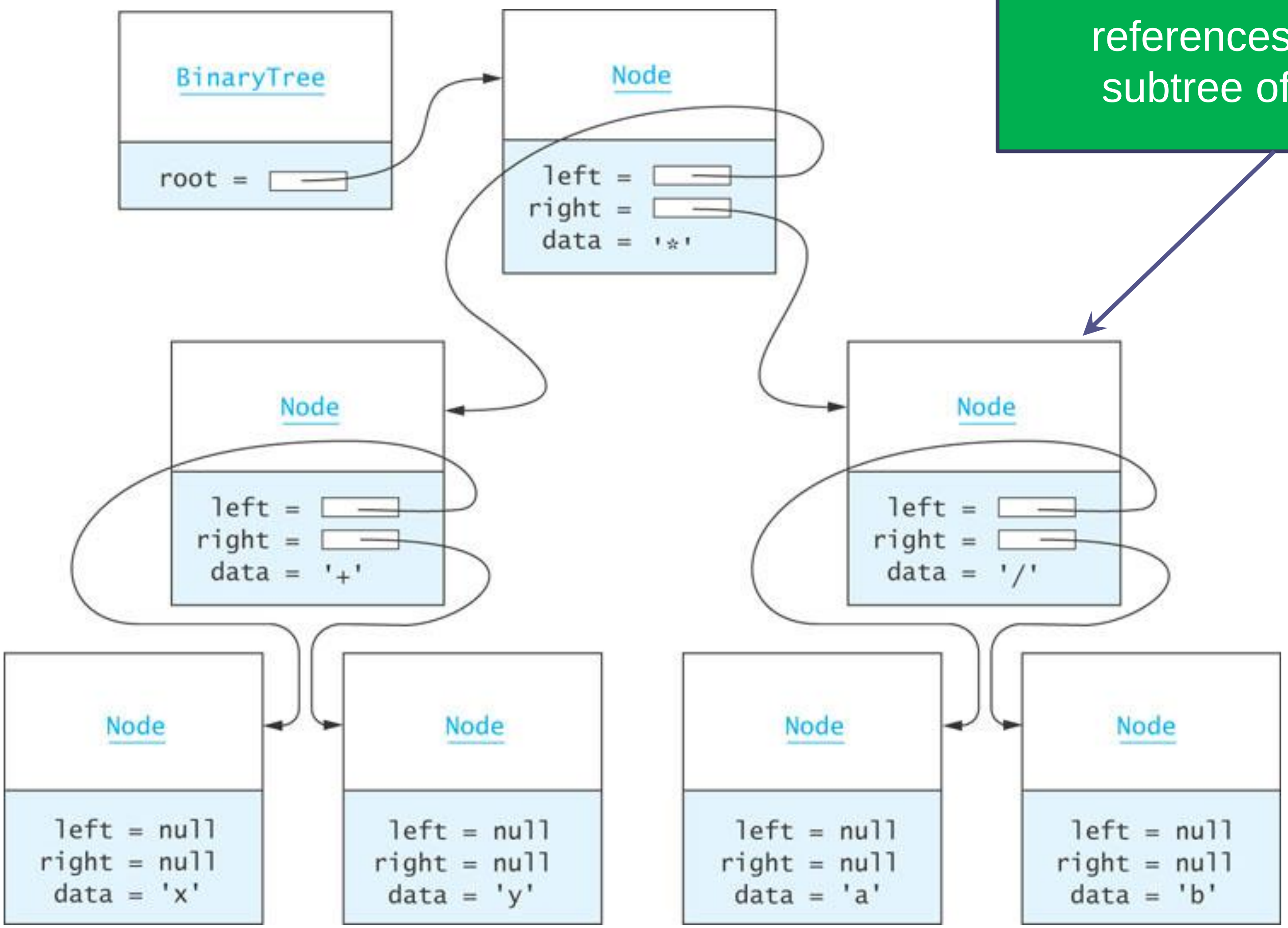
bT.root.data references the Character object storing '*'

`BinaryTree<E>` **Class** (cont.)

`bT.root.left` references the left subtree of the root

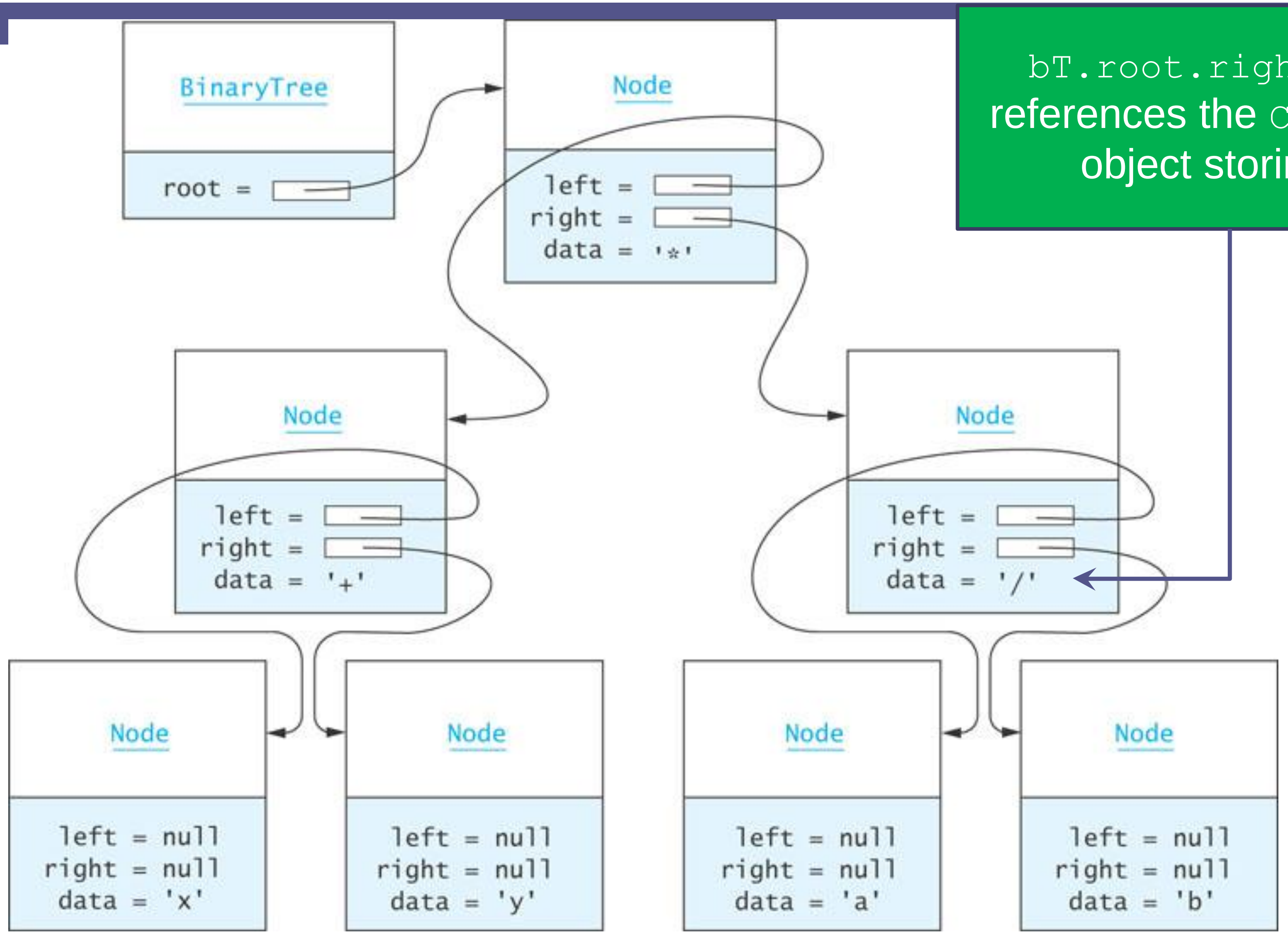


BinaryTree<E> **Class (cont.)**



`bT.root.right` references the right subtree of the root

BinaryTree<E> **Class** (cont.)



bT.root.right.data references the Character object storing '/'

BinaryTree<E> Class (cont.)

Data Field	Attribute
protected Node<E> root	Reference to the root of the tree.
Constructor	**Behavior**
public BinaryTree()	Constructs an empty binary tree.
protected BinaryTree(Node<E> root)	Constructs a binary tree with the given node as the root.
public BinaryTree(E data, BinaryTree<E> leftTree, BinaryTree<E> rightTree)	Constructs a binary tree with the given data at the root and the two given subtrees.
Method	**Behavior**
public BinaryTree<E> getLeftSubtree()	Returns the left subtree.
public BinaryTree<E> getRightSubtree()	Returns the right subtree.
public E getData()	Returns the data in the root.
public boolean isLeaf()	Returns **true** if this tree is a leaf, **false** otherwise.
public String toString()	Returns a String representation of the tree.
private void preOrderTraverse(Node<E> node, int depth, StringBuilder sb)	Performs a preorder traversal of the subtree whose root is node. Appends the representation to the StringBuilder. Increments the value of depth (the current tree level).
public static BinaryTree<E> readBinaryTree(Scanner scan)	Constructs a binary tree by reading its data using Scanner scan.

`BinaryTree<E>` **Class** (cont.)

- Class heading and data field declarations:

```
import java.io.*;

public class BinaryTree<E> implements Serializable {
   // Insert inner class Node<E> here

   protected Node<E> root;

   . . .
}
```

`BinaryTree<E>` **Class** (cont.)

- The `Serializable` interface defines no methods
- It provides a marker for classes that can be written to a binary file using the `ObjectOutputStream` and read using the `ObjectInputStream`