CS 570: Data Structures

Trees

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

CHAPTER 6
TREES

Chapter Objectives

- To learn how to use a tree to represent a hierarchical organization of information
- To learn how to use recursion to process trees
- To understand the different ways of traversing a tree
- To understand the difference between binary trees, binary search trees, and heaps
- To learn how to implement binary trees, binary search trees, and heaps using linked data structures and arrays

Week 10

- Reading Assignment: Koffman and Wolfgang, Sections 6.1-6.3, 6.5

Trees - Introduction

- All previous data organizations we've learned are linear—each element can have only one predecessor or successor
- Accessing all elements in a linear sequence is O(*n*)
- Trees are nonlinear and hierarchical
- Tree nodes can have multiple successors (but only one predecessor)

Trees - Introduction (cont.)

- Trees can represent hierarchical organizations of information:
  - class hierarchy
  - disk directory and subdirectories
  - family tree
- Trees are recursive data structures because they can be defined recursively
- Many methods to process trees are written recursively

Trees - Introduction (cont.)

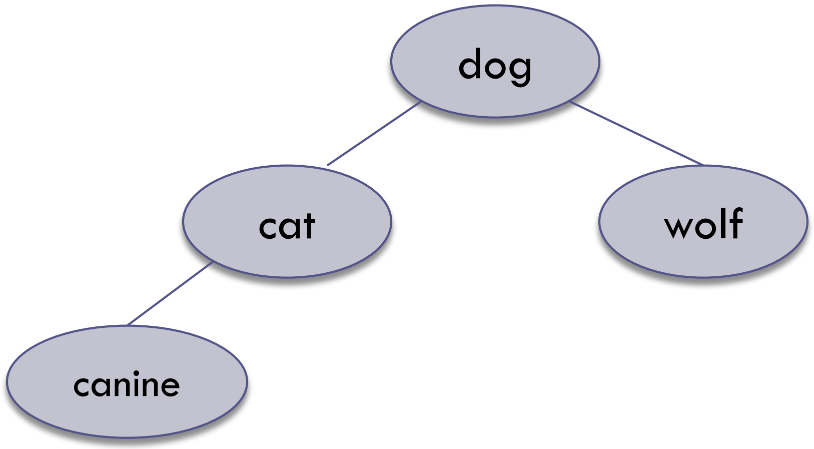
- This chapter focuses on the *binary tree*
- In a binary tree each element has two successors
- Binary trees can be represented by arrays and linked data structures
- Searching in a binary search tree, an ordered tree, is generally more efficient than searching in an ordered list—O(log *n*) versus O(n)

Tree Terminology and Applications

Section 6.1

Tree Terminology

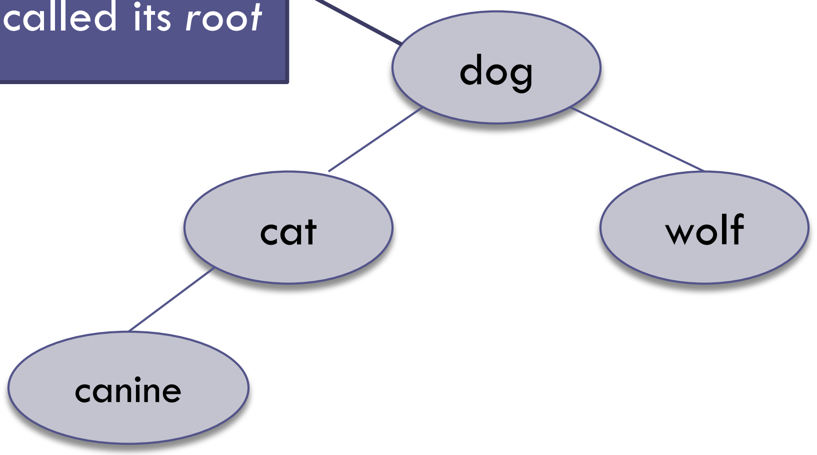
A tree consists of a collection of elements or nodes, with each node linked to its successors



Tree Terminology (cont.)

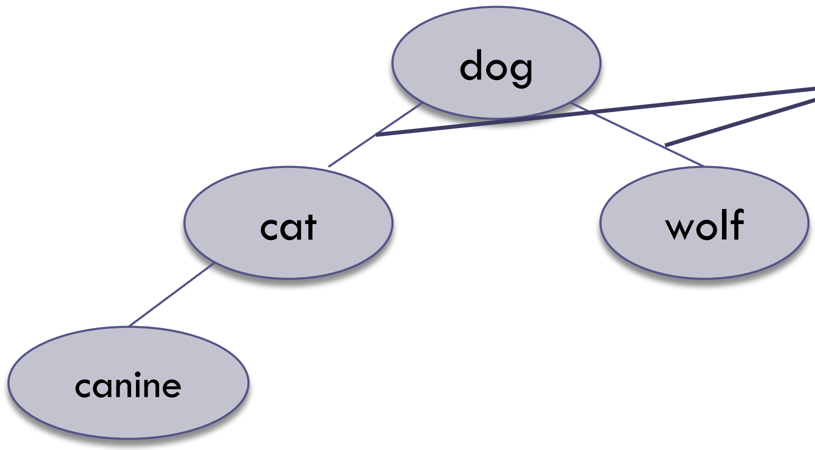
A tree consists of a collection of elements or nodes, with each node linked to its successors

The node at the top of a tree is called its *root*



Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors

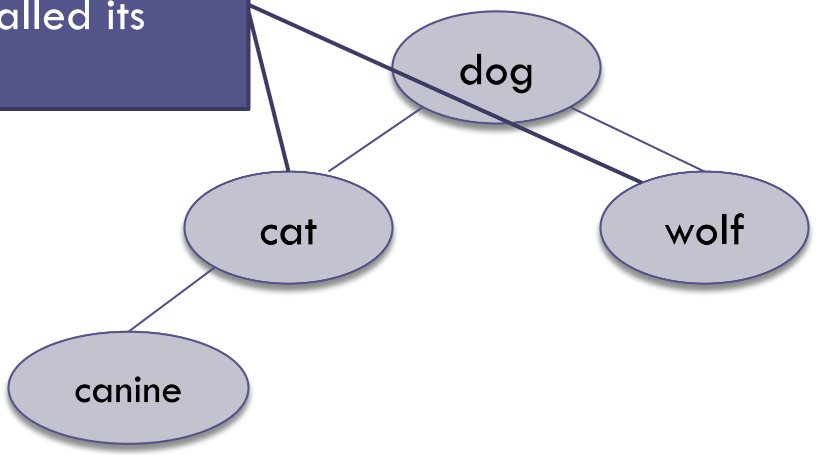


The links from a node to its successors are called *branches*

Tree Terminology (cont.)

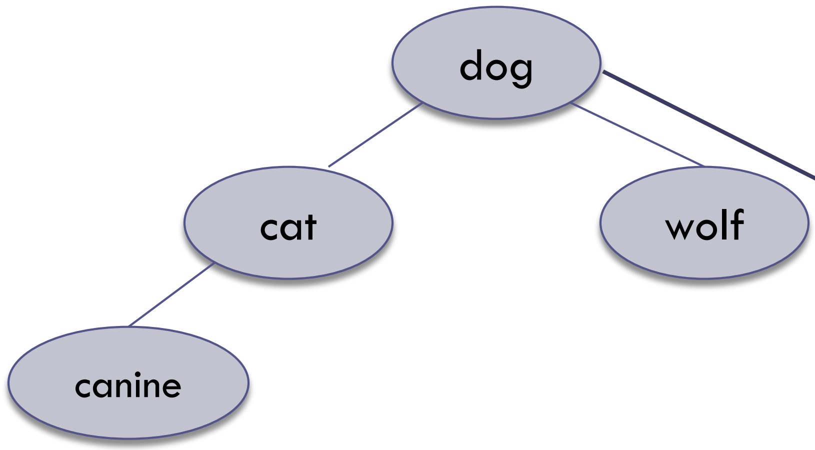
A tree consists of a collection of elements or nodes, with each node linked to its successors

The successors of a node are called its *children*



Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors

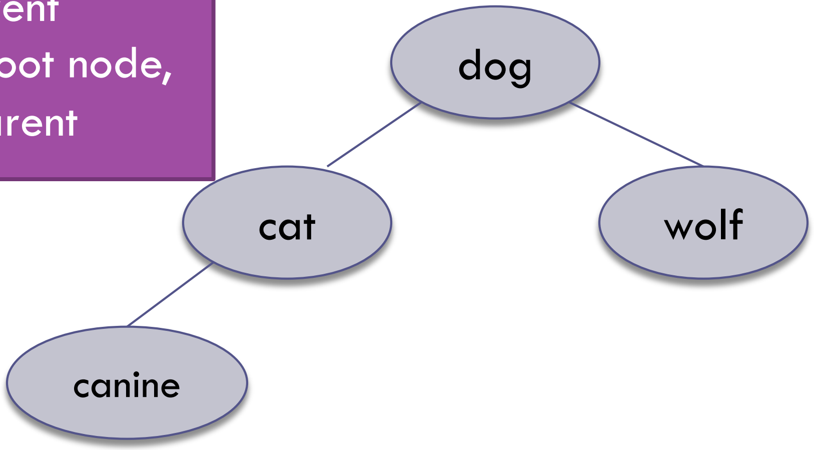


The predecessor of a node is called its *parent*

Tree Terminology (cont.)

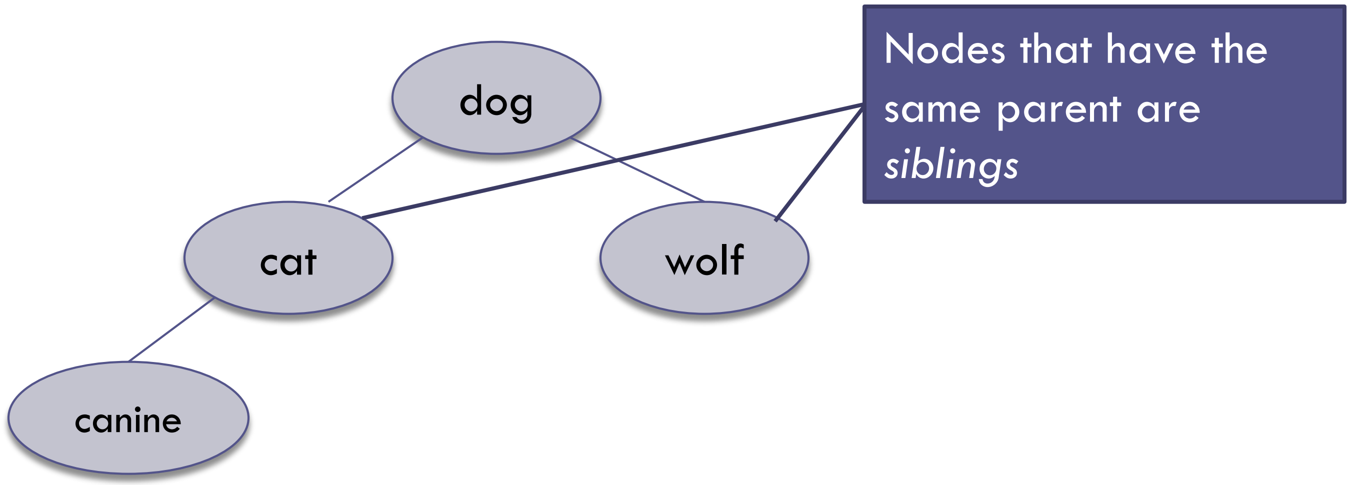
A tree consists of a collection of elements or nodes, with each node linked to its successors

Each node in a tree has exactly one parent except for the root node, which has no parent



Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors

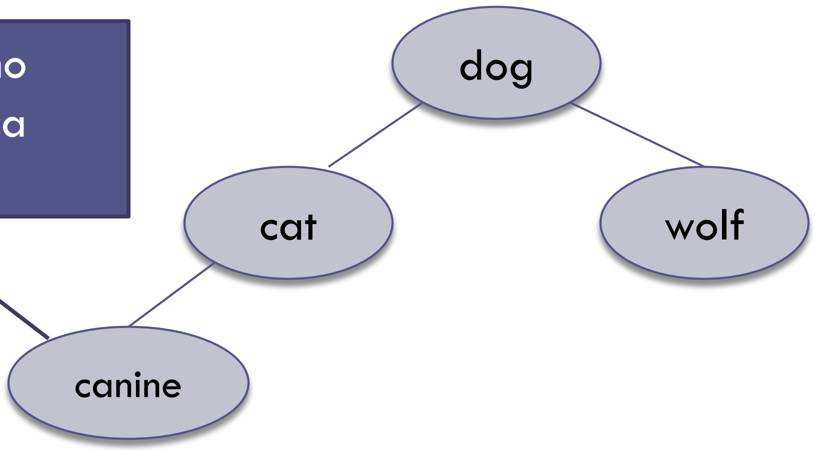


Nodes that have the same parent are *siblings*

Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors

A node that has no children is called a *leaf node*

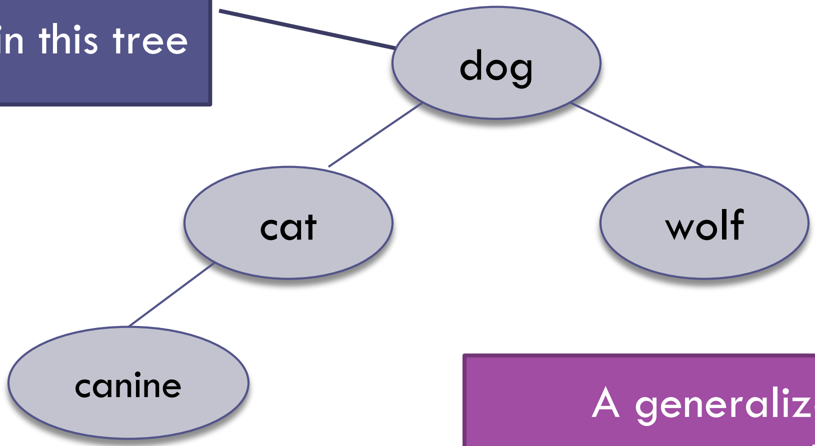


Leaf nodes also are known as *external nodes*, and nonleaf nodes are known as *internal nodes*

Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors

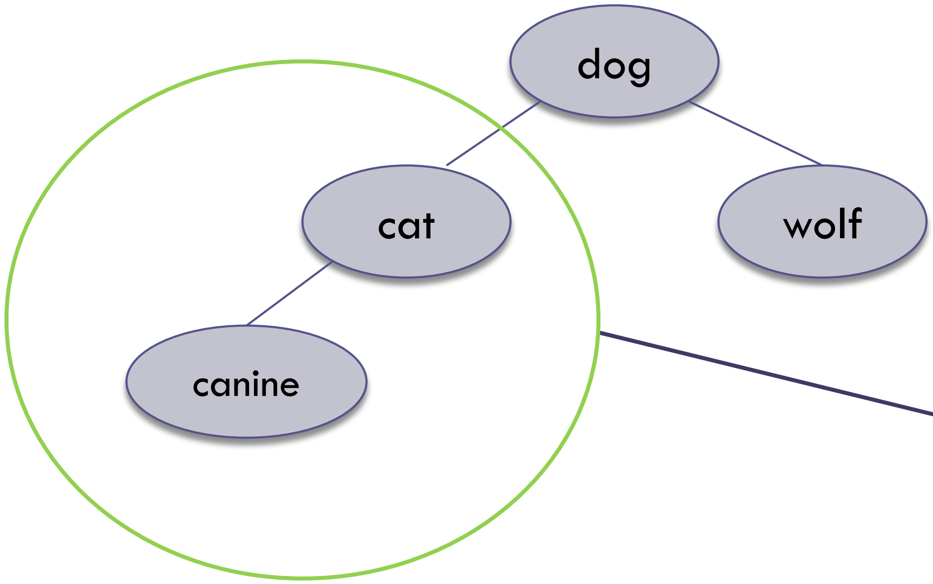
dog is an *ancestor* of canine in this tree



A generalization of the parent-child relationship is the *ancestor-descendant relationship*

Tree Terminology (cont.)

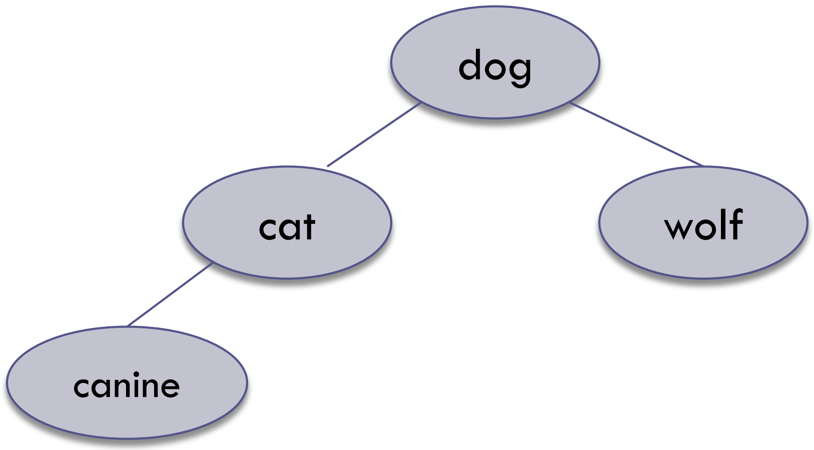
A tree consists of a collection of elements or nodes, with each node linked to its successors



A *subtree* of a node is a tree whose root is a child of that node

Tree Terminology (cont.)

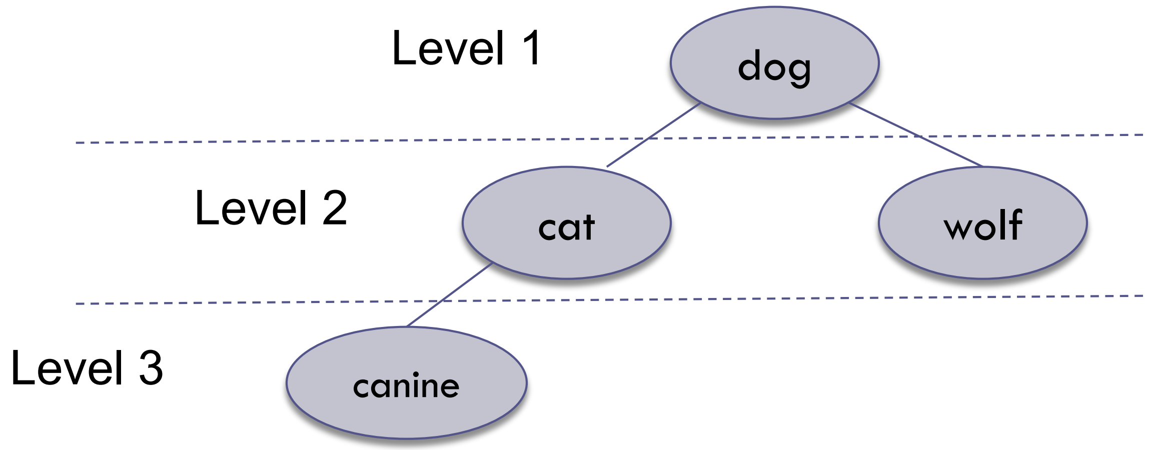
A tree consists of a collection of elements or nodes, with each node linked to its successors



The *level of a node* is a measure of its distance from the root

Tree Terminology (cont.)

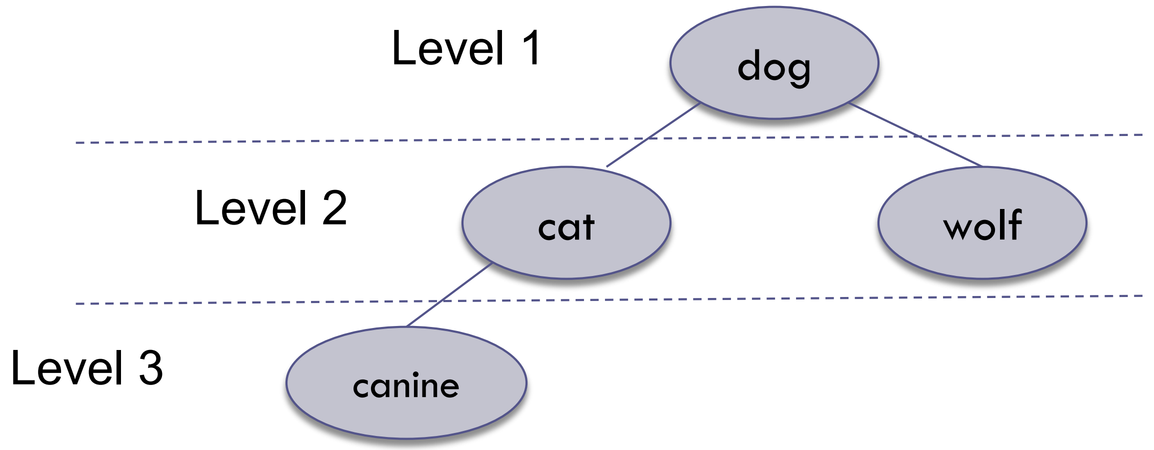
A tree consists of a collection of elements or nodes, with each node linked to its successors



The *level of a node* is a measure of its distance from the root plus 1
and is defined recursively

Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors



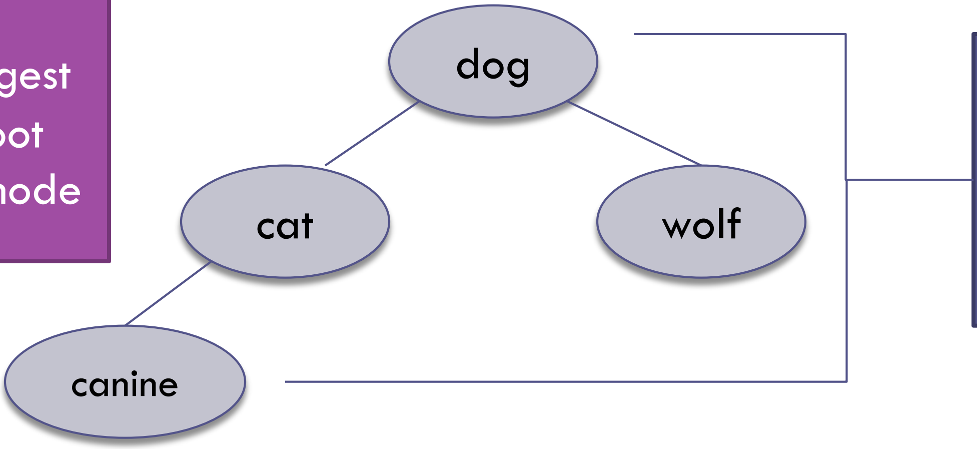
The *level of a node* is defined recursively

- If node *n* is the root of tree T, its level is 1
- If node *n* is not the root of tree T, its level is 1 + the level of its parent

Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors

The *height of a tree* is the number of nodes in the longest path from the root node to a leaf node



The height of this tree is 3

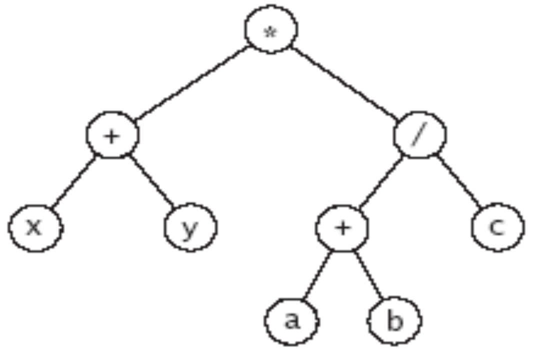
Binary Trees

- In a binary tree, each node has two subtrees
- A set of nodes T is a binary tree if either of the following is true
  - T is empty
  - Its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary trees

(T_L = left subtree; T_R = right subtree)

Expression Tree

- Each node contains an operator or an operand
- Operands are stored in leaf nodes



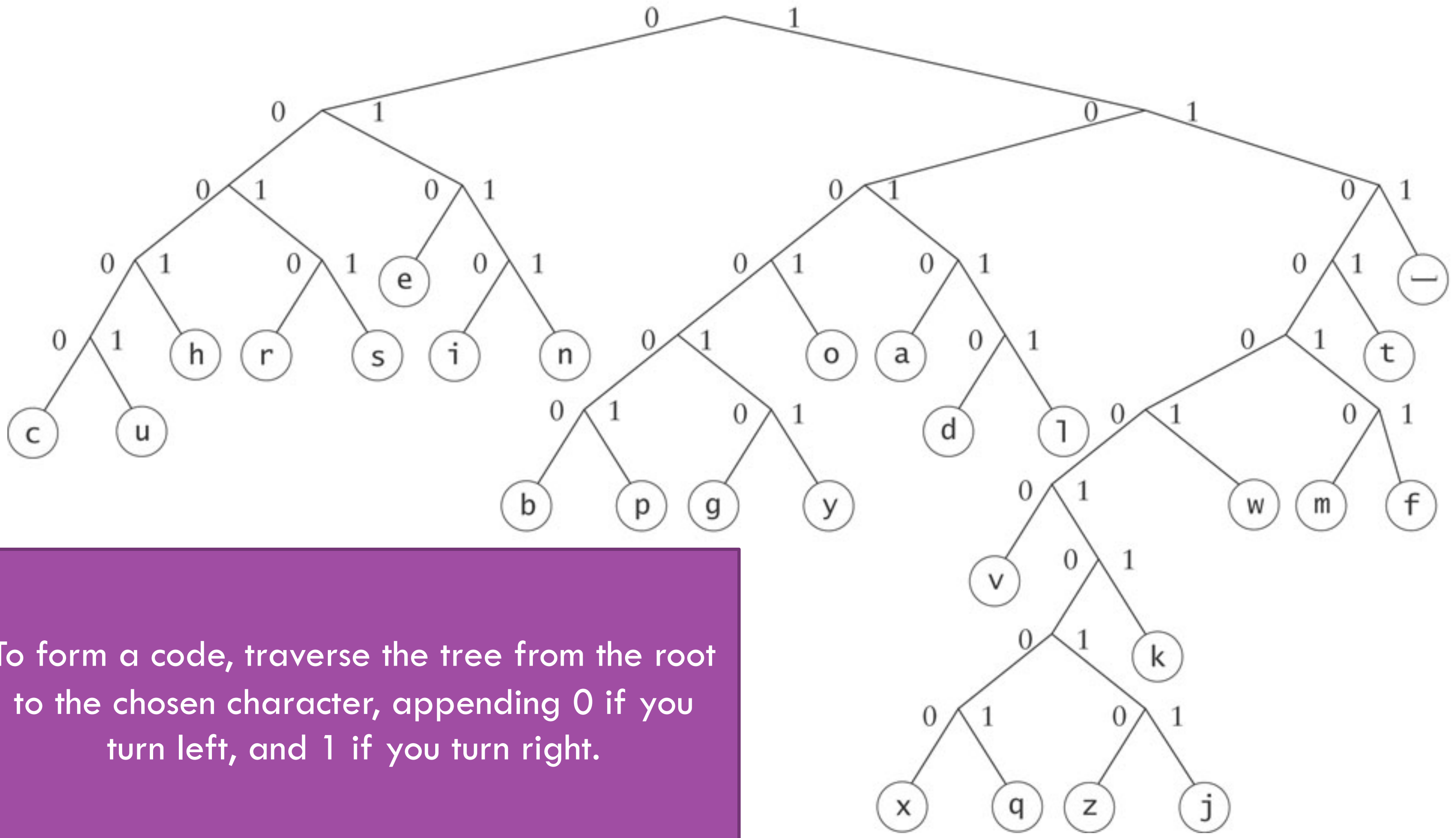
```
(x + y) * ((a + b) / c)
```

- Parentheses are not stored in the tree because the tree structure dictates the order of operand evaluation
- Operators in nodes at higher levels are evaluated after operators in nodes at lower levels

Huffman Tree

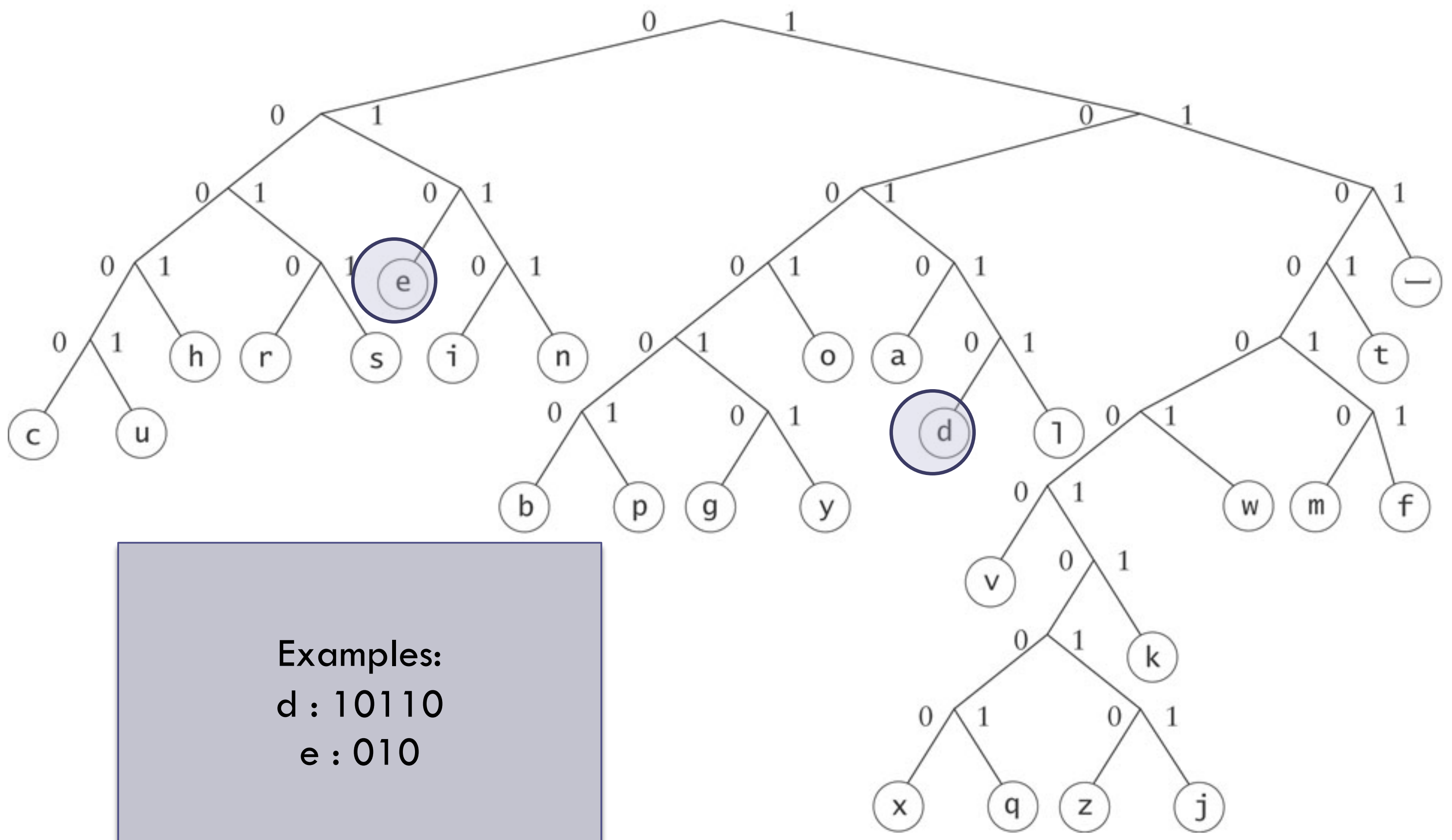
- A *Huffman tree* represents *Huffman codes* for characters that might appear in a text file
- As opposed to ASCII or Unicode, Huffman code uses different numbers of bits to encode letters; more common characters use fewer bits
- Many programs that compress files use Huffman codes

Huffman Tree (cont.)



To form a code, traverse the tree from the root to the chosen character, appending 0 if you turn left, and 1 if you turn right.

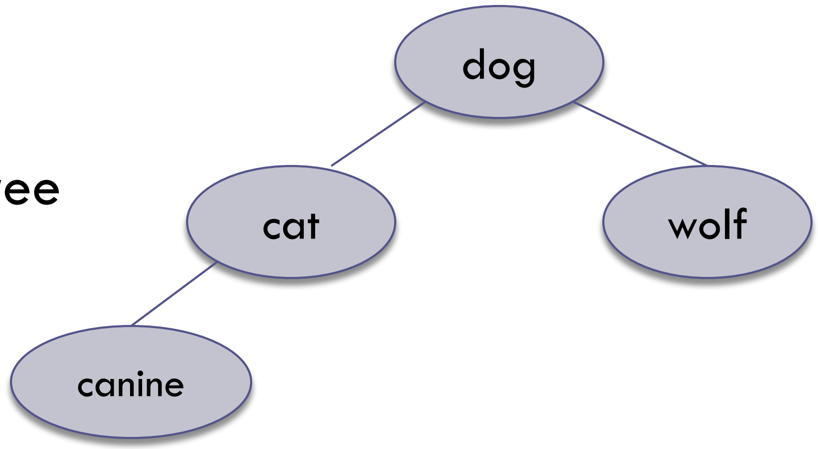
Huffman Tree (cont.)



Examples:
d : 10110
e : 010

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 empty
- If T is not empty, its root node 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 greater than all values in T_L and is less than all values in T_R

Binary Search Tree (cont.)

- A binary search tree never has to be sorted because its elements always satisfy the required order relations
- When new elements are inserted (or removed) properly, the binary search tree maintains its order
- In contrast, an array must be expanded whenever new elements are added, and compacted when elements are removed—expanding and contracting are both O(*n*)

Binary Search Tree (cont.)

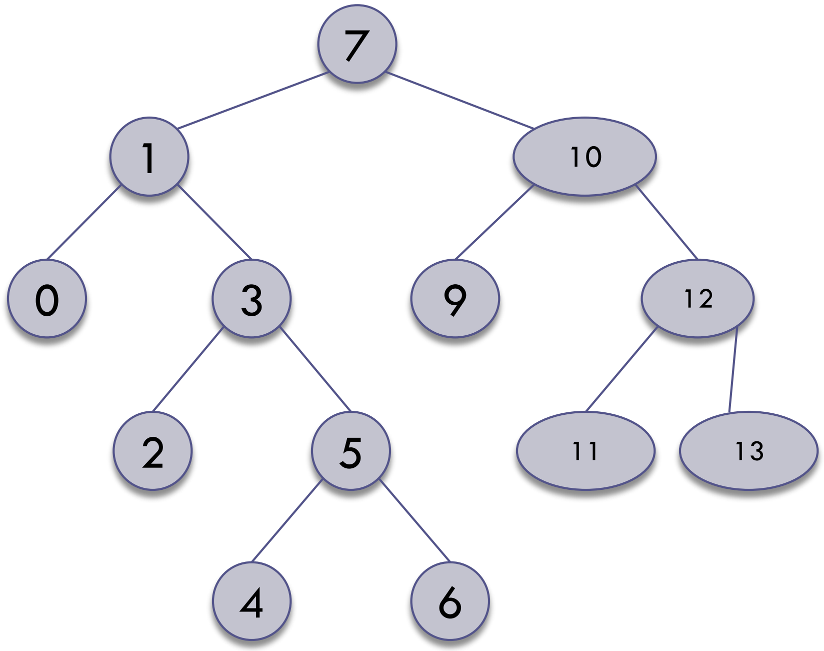
- When searching a BST, each probe has the potential to eliminate half the elements in the tree, so searching can be O(log n)
- In the worst case, searching is O(n)

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
```

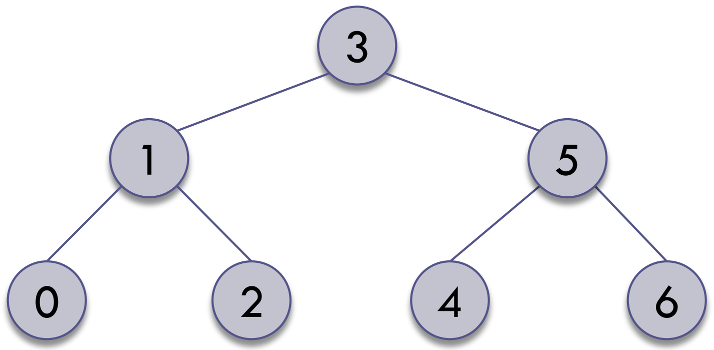
Full, Perfect, and Complete Binary Trees

- A full binary tree is a binary tree where all nodes have either 2 children or 0 children (the leaf nodes)



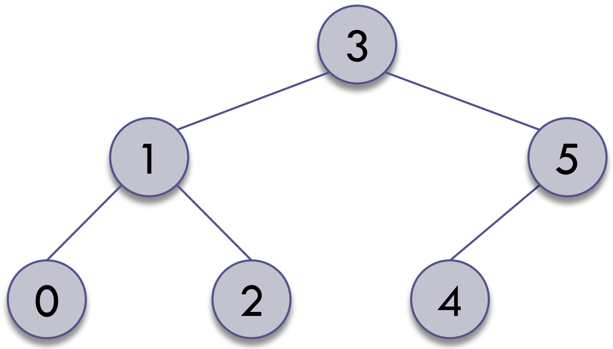
Full, Perfect, and Complete Binary Trees (cont.)

- A *perfect binary tree* is a full binary tree of height *n* with exactly $2^n - 1$ nodes
- In this case, $n = 3$ and $2^n - 1 = 7$



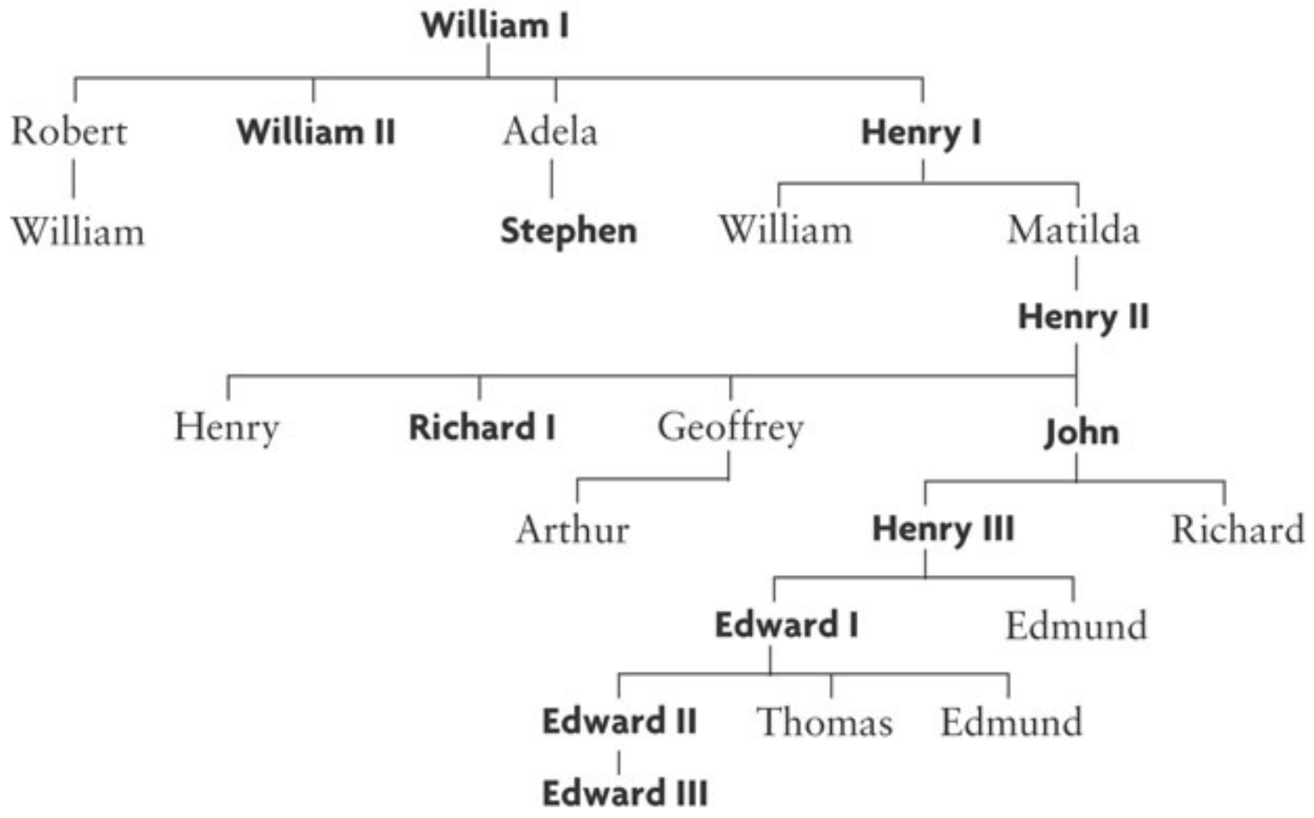
Full, Perfect, and Complete Binary Trees (cont.)

- 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

- We do not discuss general trees in this chapter, but nodes of a general tree can have any number of subtrees



Tree Traversals

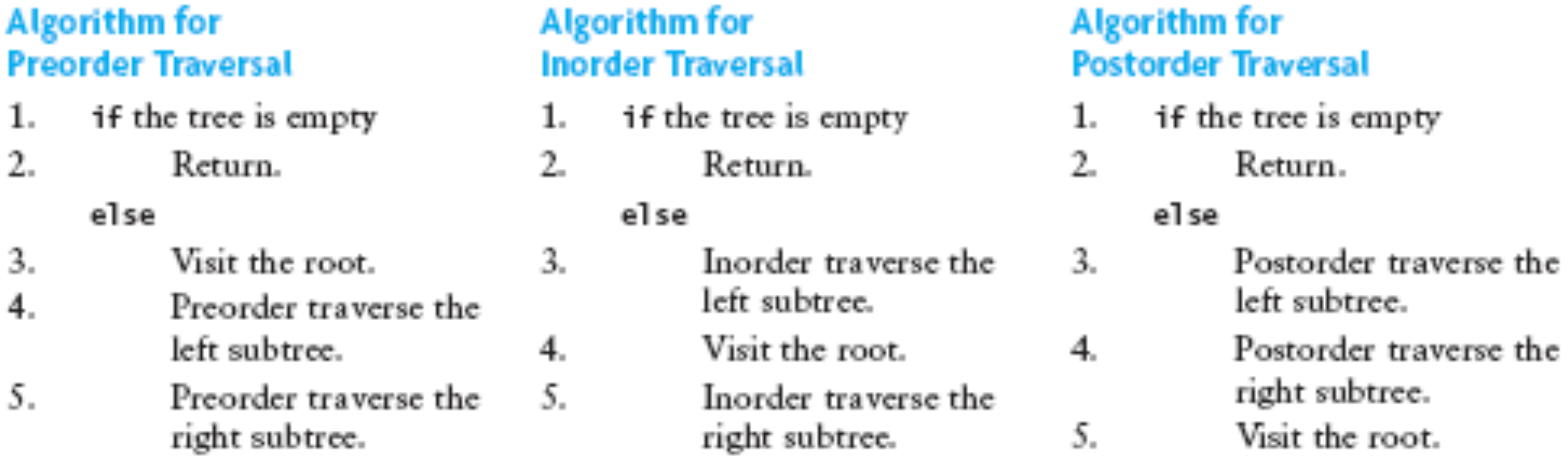
Section 6.2

Tree Traversals

- Often we want to determine the nodes of a tree and their relationship
  - 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
  - Inorder
  - Preorder
  - Postorder

Tree Traversals (cont.)

- 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

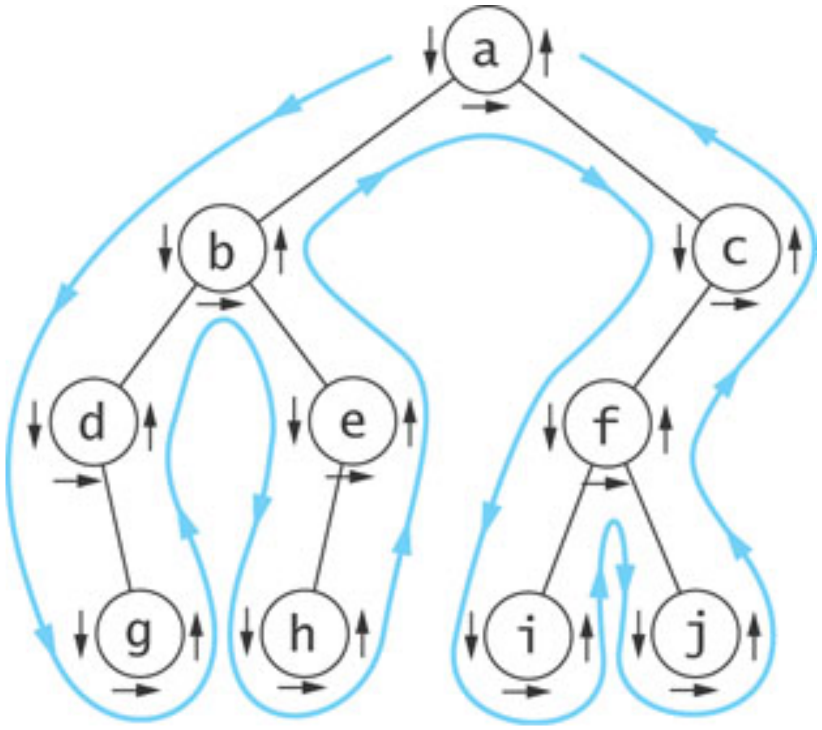
```
1.    if the tree is empty
2.        Return.
      else
3.        Inorder traverse the
          left subtree.
4.        Visit the root.
5.        Inorder traverse the
          right subtree.
```

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.
```

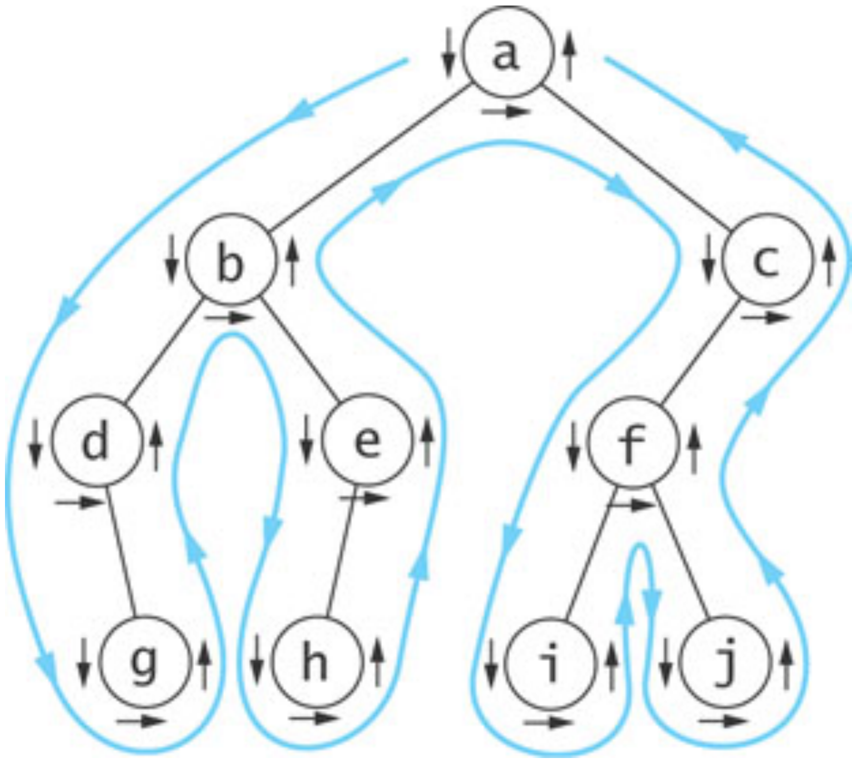
Visualizing Tree Traversals

- You can visualize a tree traversal by imagining a mouse that walks along the edge of the tree
- If the mouse always keeps the tree to the left, it will trace a route known as the *Euler tour*
- The Euler tour is the path traced in blue in the figure on the right



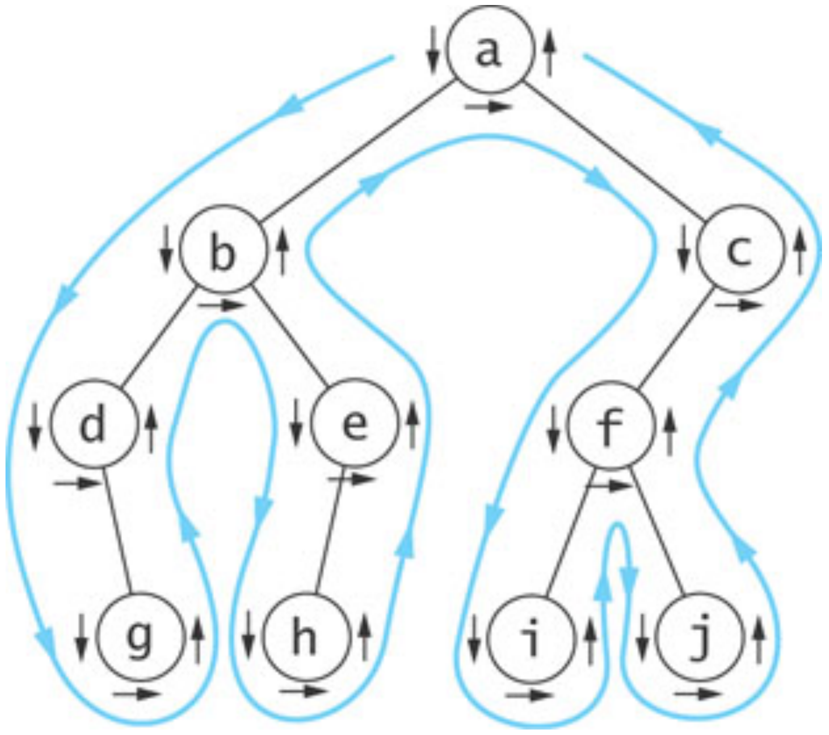
Visualizing Tree Traversals (cont.)

- An Euler tour (blue path) is a preorder traversal
- The sequence in this example is
  a b d g e h c f i j
- The mouse visits each node before traversing its subtrees (shown by the downward pointing arrows)



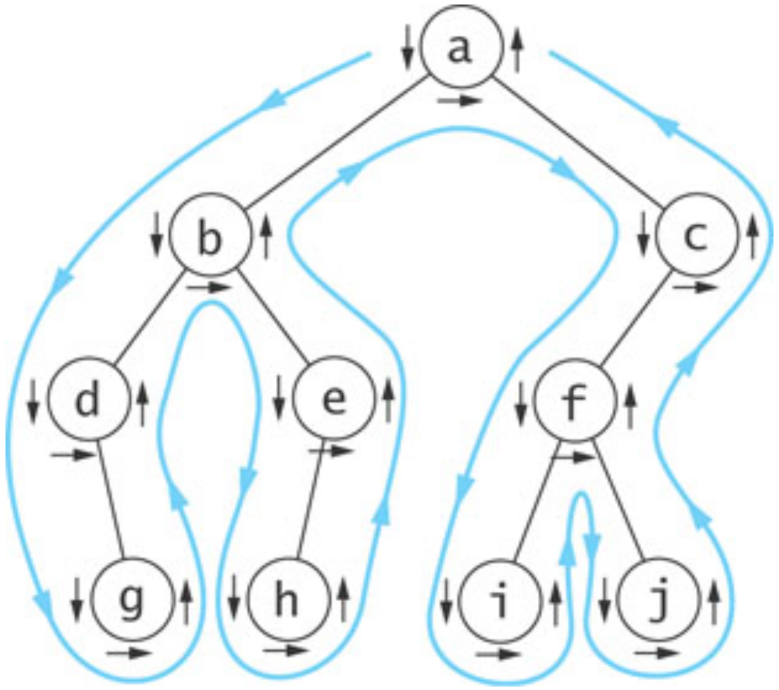
Visualizing Tree Traversals (cont.)

- If we record a node as the mouse returns from traversing its left subtree (horizontal black arrows in the figure) we get an inorder traversal
- The sequence is
  d g b h e a i f j c



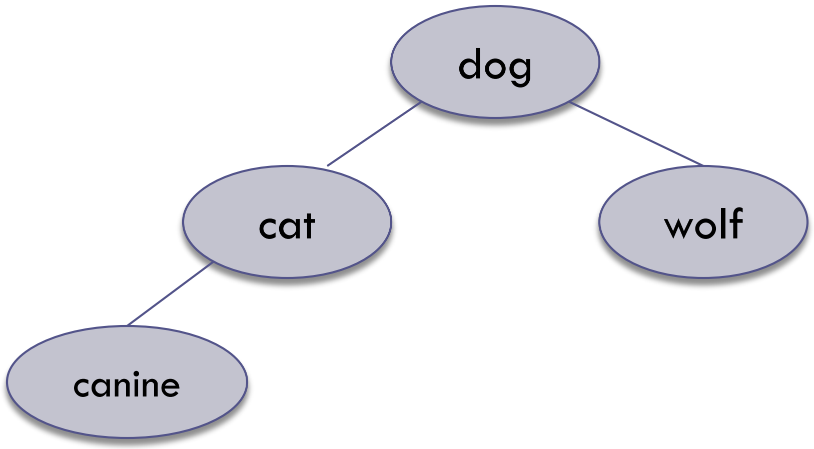
Visualizing Tree Traversals (cont.)

- If we record each node as the mouse last encounters it, we get a postorder traversal (shown by the upward pointing arrows)
- The sequence is
  g d h e b i j f c a



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



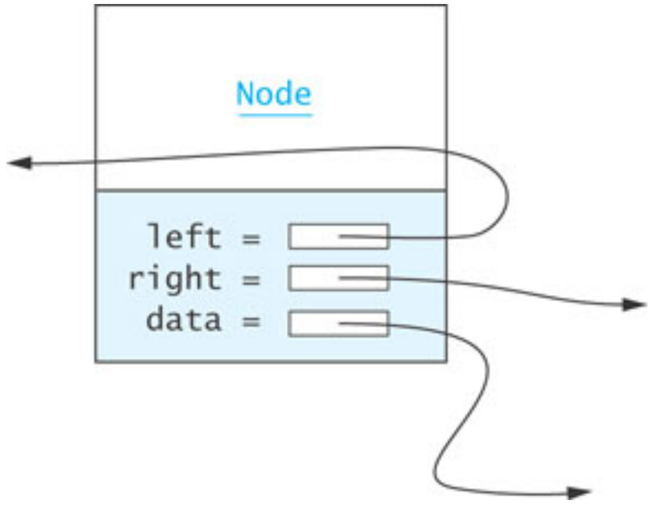
canine, cat, dog, wolf

Implementing a `BinaryTree` Class

Section 6.3

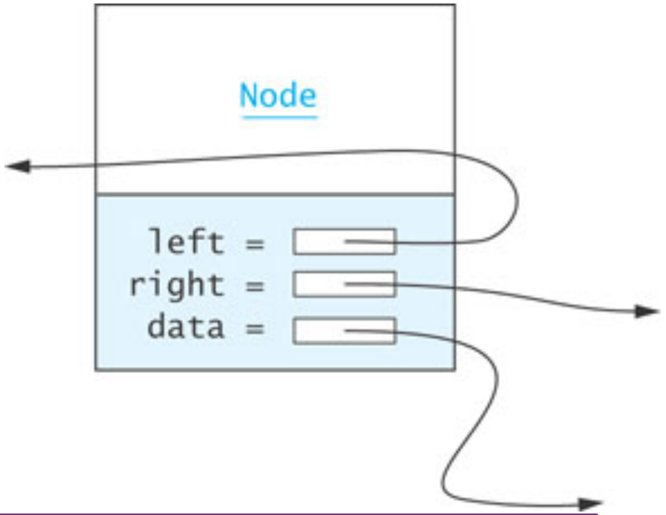
Node<E> Class

- Just as for a linked list, a node consists of a data part and 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



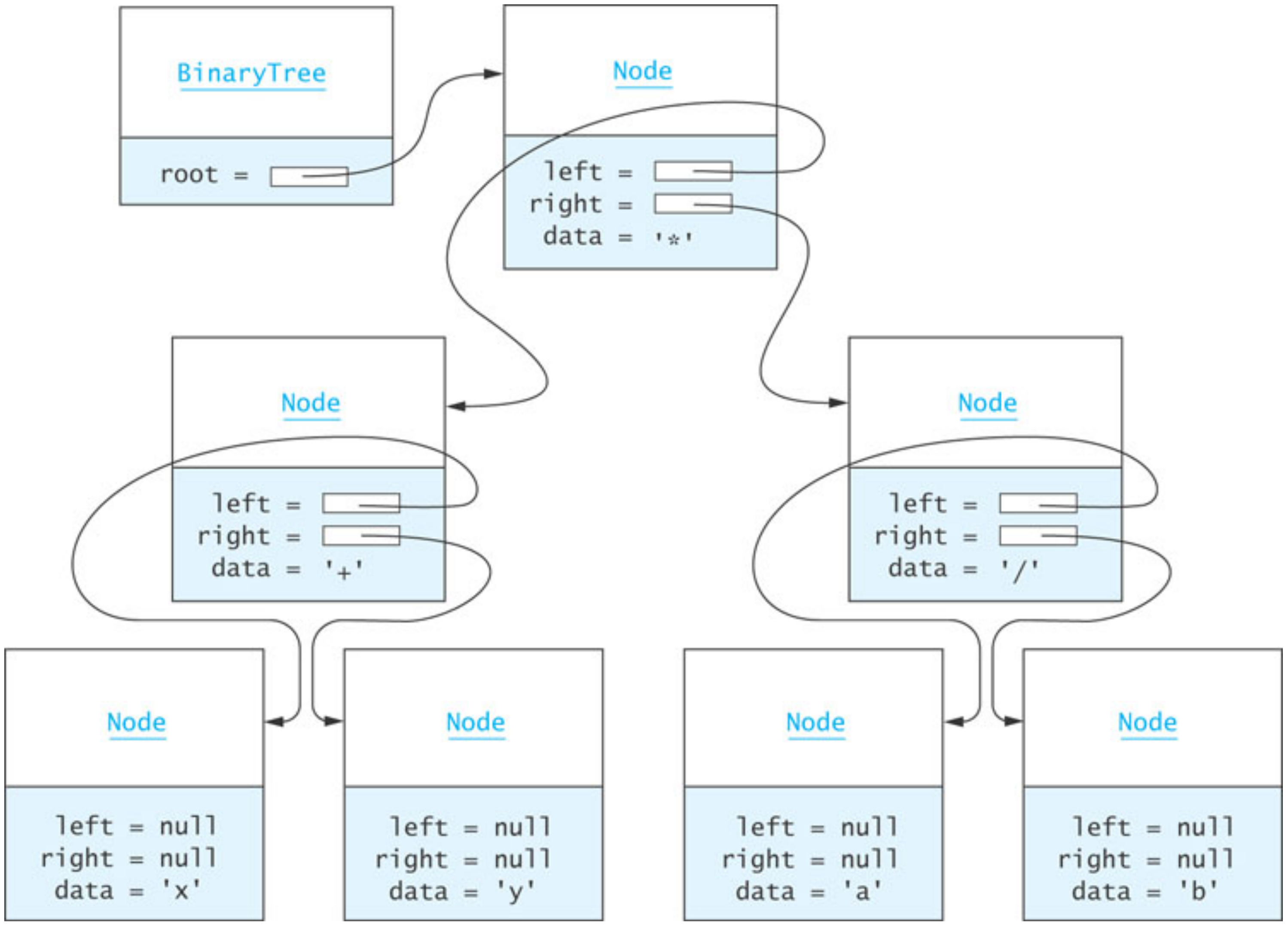
Node<E> Class (cont.)

```
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();
   }
}
```

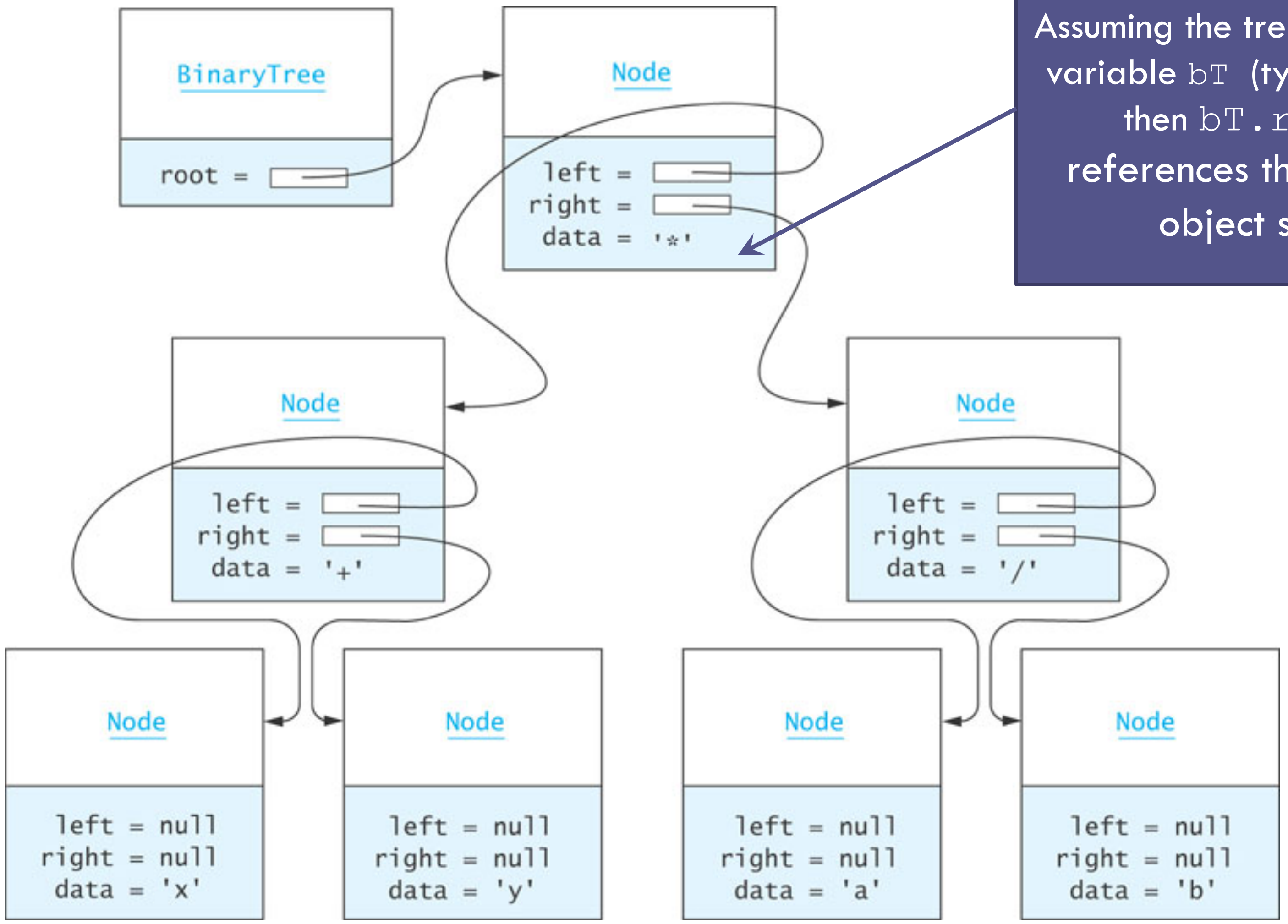


Node<E> is declared as an inner class within BinaryTree<E>

BinaryTree<E> Class (cont.)



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



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

Constructors

- The no-parameter constructor:

```
public BinaryTree() {

    root = null;
}
```

- The constructor that creates a tree with a given node at the root:

```
protected BinaryTree(Node<E> root) {

    this.root = root;
}
```

Constructors (cont.)

- The constructor that builds a tree from a data value and two trees:

```
public BinaryTree(E data, BinaryTree<E> leftTree, BinaryTree<E>
       rightTree) {
   root = new Node<E>(data);
   if (leftTree != null) {
      root.left = leftTree.root;
   } else {
      root.left = null;
   }
   if (rightTree != null) {
      root.right = rightTree.root;
   } else {
      root.right = null;
   }
}
```

getLeftSubtree and getRightSubtree Methods

```
public BinaryTree<E> getLeftSubtree() {
    if (root != null && root.left != null) {
        return new BinaryTree<E>(root.left);
  }   else {
        return null;
  }
}
```

- getRightSubtree method is symmetric

`isLeaf` Method

```
public boolean isLeaf() {

       ????

}
```

`toString` **Method**

- The toString method generates a string representing a preorder traversal in which each local root is indented a distance proportional to its depth

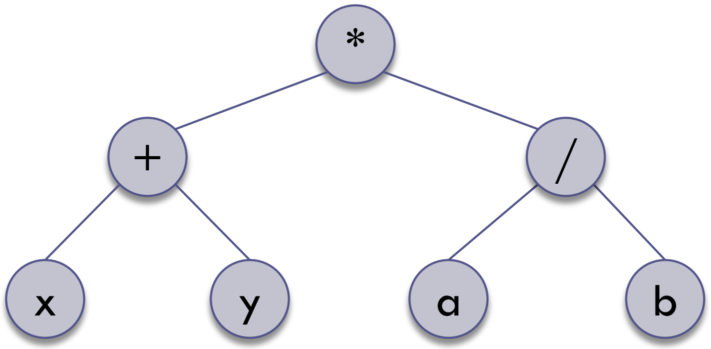
```
public String toString() {
    StringBuilder sb = new StringBuilder();
    preOrderTraverse(root, 1, sb);
    return sb.toString();
}
```

preOrderTraverse **Method**

```
private void preOrderTraverse(Node<E> node, int depth,
                              StringBuilder sb) {

    for (int i = 1; i < depth; i++) {
        sb.append("  ");
    }
    if (node == null) {
        sb.append("null\n");
    } else {
        sb.append(node.toString());
        sb.append("\n");
        preOrderTraverse(node.left, depth + 1, sb);
        preOrderTraverse(node.right, depth + 1, sb);
    }
}
```

`preOrderTraverse` **Method (cont.)**

```
*
  +
    x
      null
      null
    y
      null
      null
  /
    a
      null
      null
    b
      null
      null
```



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

Binary Search Trees

Section 6.5

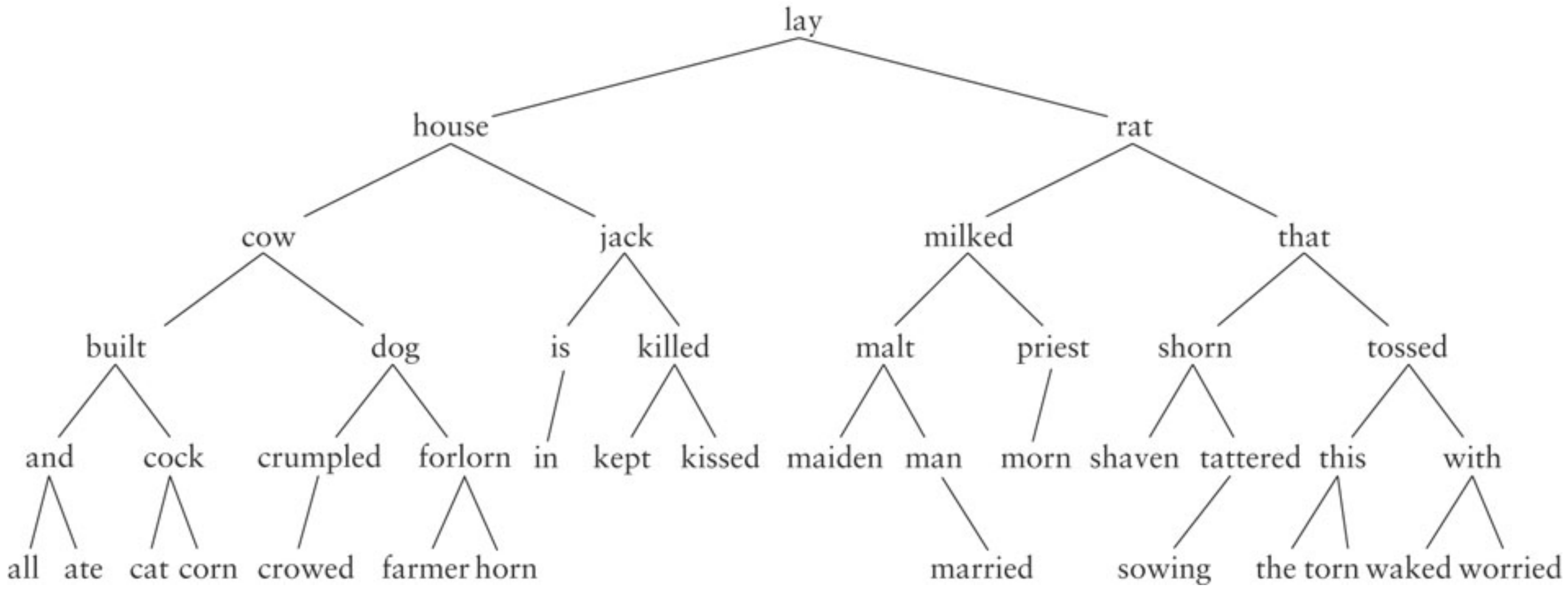
Overview of a Binary Search Tree

- Recall the definition of a binary search tree:

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

  - T is empty
  - If T is not empty, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary search trees and the value in the root node of T is greater than all values in T_L and less than all values in T_R

Overview of a Binary Search Tree (cont.)

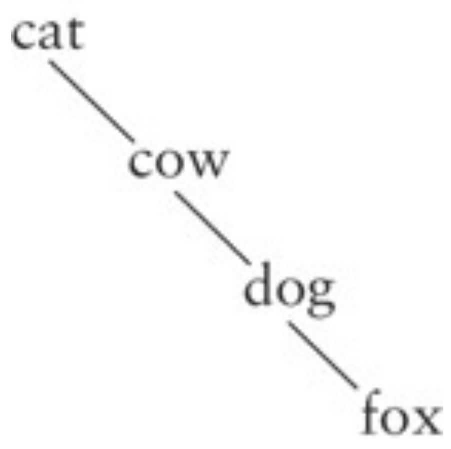


Recursive Algorithm for Searching a Binary Search Tree

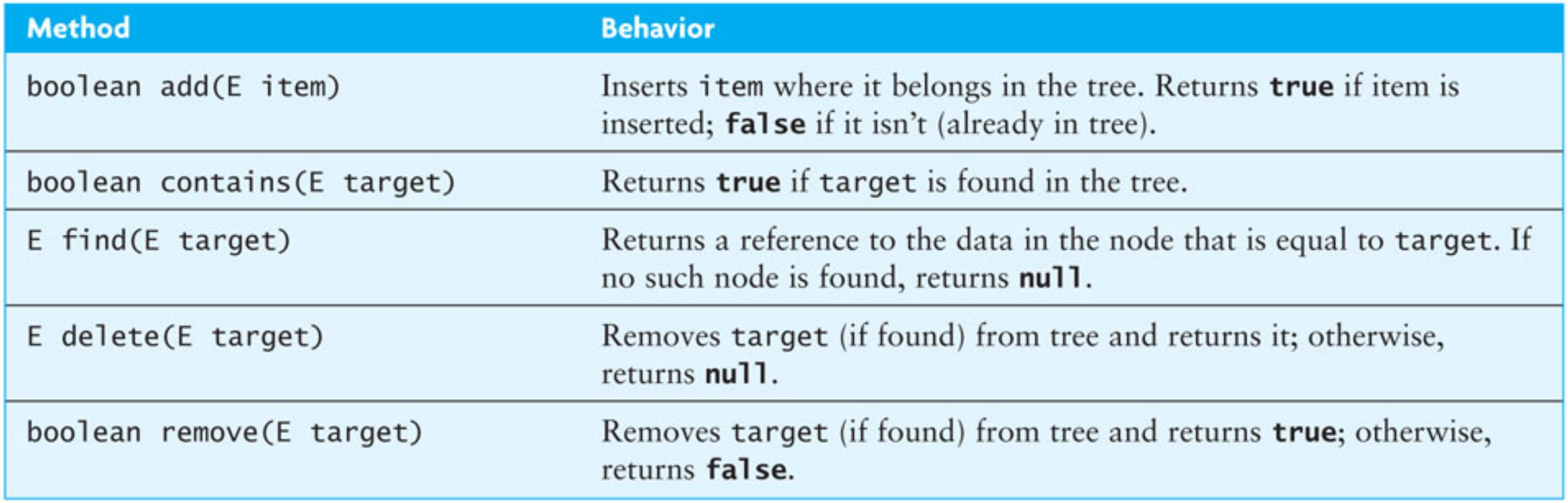
```
1.  if the root is null
2.          the item is not in the tree; return null
3.  Compare the value of target with root.data
4.  if they are equal
5.          the target has been found; return the data at the root
    else if  the target is less than root.data
6.          return the result of searching the left subtree
    else
7.          return the result of searching the right subtree
```

Performance

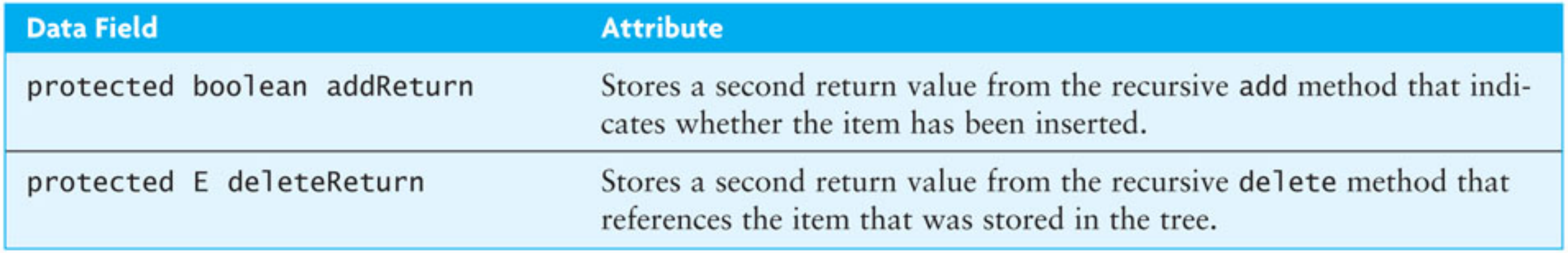
- Search a tree is generally O(log *n*)
- If a tree is not very full, performance will be worse
- Searching a tree with only right subtrees, for example, is O(*n*)



Interface `SearchTree<E>`



Method	Behavior
`boolean add(E item)`	Inserts `item` where it belongs in the tree. Returns **`true`** if item is inserted; **`false`** if it isn't (already in tree).
`boolean contains(E target)`	Returns **`true`** if `target` is found in the tree.
`E find(E target)`	Returns a reference to the data in the node that is equal to `target`. If no such node is found, returns **`null`**.
`E delete(E target)`	Removes `target` (if found) from tree and returns it; otherwise, returns **`null`**.
`boolean remove(E target)`	Removes `target` (if found) from tree and returns **`true`**; otherwise, returns **`false`**.



Data Field	Attribute
`protected boolean addReturn`	Stores a second return value from the recursive `add` method that indicates whether the item has been inserted.
`protected E deleteReturn`	Stores a second return value from the recursive `delete` method that references the item that was stored in the tree.

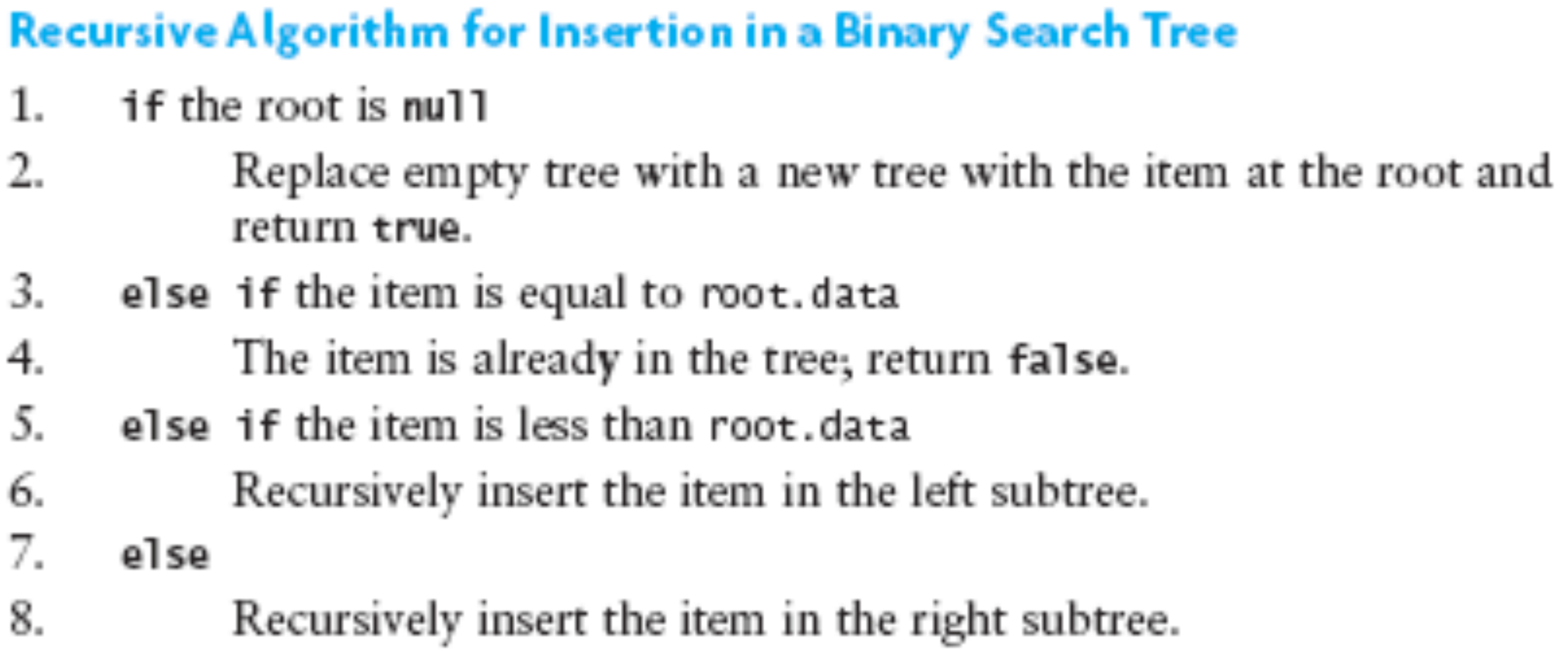
Implementing `find` Methods

LISTING 6.3

`BinarySearchTree find` Method

```
/** Starter method find.
    pre: The target object must implement
         the Comparable interface.
    @param target The Comparable object being sought
    @return The object, if found, otherwise null
*/
public E find(E target) {
    return find(root, target);
}

/** Recursive find method.
    @param localRoot The local subtree's root
    @param target The object being sought
    @return The object, if found, otherwise null
*/
private E find(Node<E> localRoot, E target) {
    if (localRoot == null)
        return null;

    // Compare the target with the data field at the root.
    int compResult = target.compareTo(localRoot.data);
    if (compResult == 0)
        return localRoot.data;
    else if (compResult < 0)
        return find(localRoot.left, target);
    else
        return find(localRoot.right, target);
}
```

Insertion into a Binary Search Tree



Recursive Algorithm for Insertion in a Binary Search Tree

```
1.  if the root is null
2.      Replace empty tree with a new tree with the item at the root and
        return true.
3.  else if the item is equal to root.data
4.      The item is already in the tree; return false.
5.  else if the item is less than root.data
6.      Recursively insert the item in the left subtree.
7.  else
8.      Recursively insert the item in the right subtree.
```

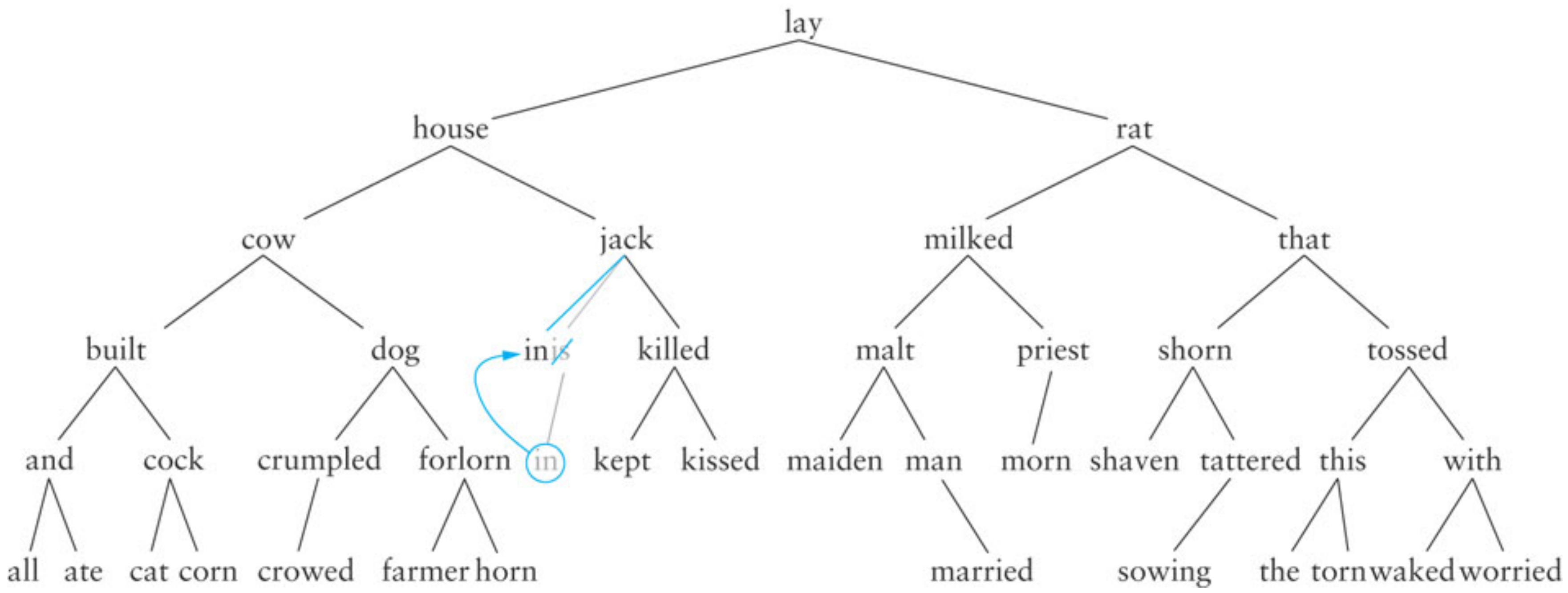
Implementing the `add` Methods

```
/** Starter method add.
    pre: The object to insert must implement the
       Comparable interface.
    @param item The object being inserted
    @return true if the object is inserted, false
          if the object already exists in the tree
*/
public boolean add(E item) {
    root = add(root, item);
    return addReturn;
}
```

Implementing the `add` Methods (cont.)

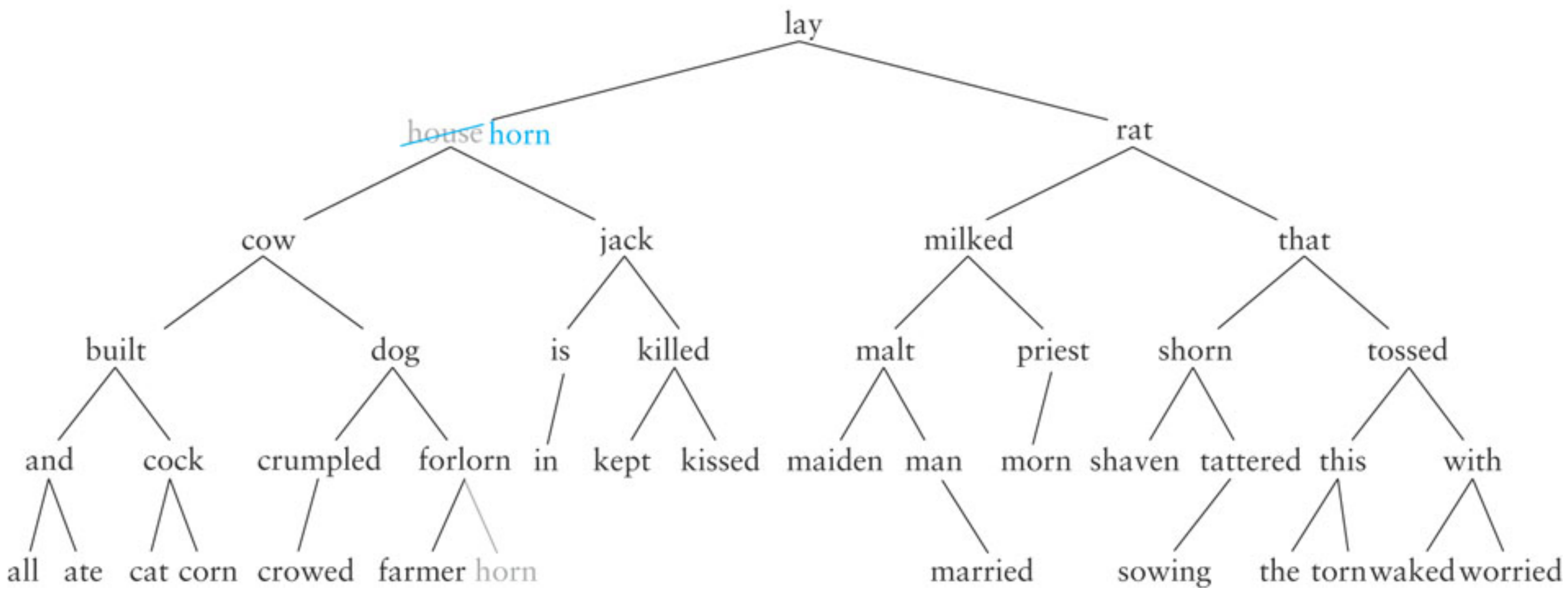
```
/** Recursive add method.
    post: The data field addReturn is set true if the item is added to
         the tree, false if the item is already in the tree.
    @param localRoot The local root of the subtree
    @param item The object to be inserted
    @return The new local root that now contains the
         inserted item
*/
private Node<E> add(Node<E> localRoot, E item) {
    ?????????????????
```

Removal from a Binary Search Tree

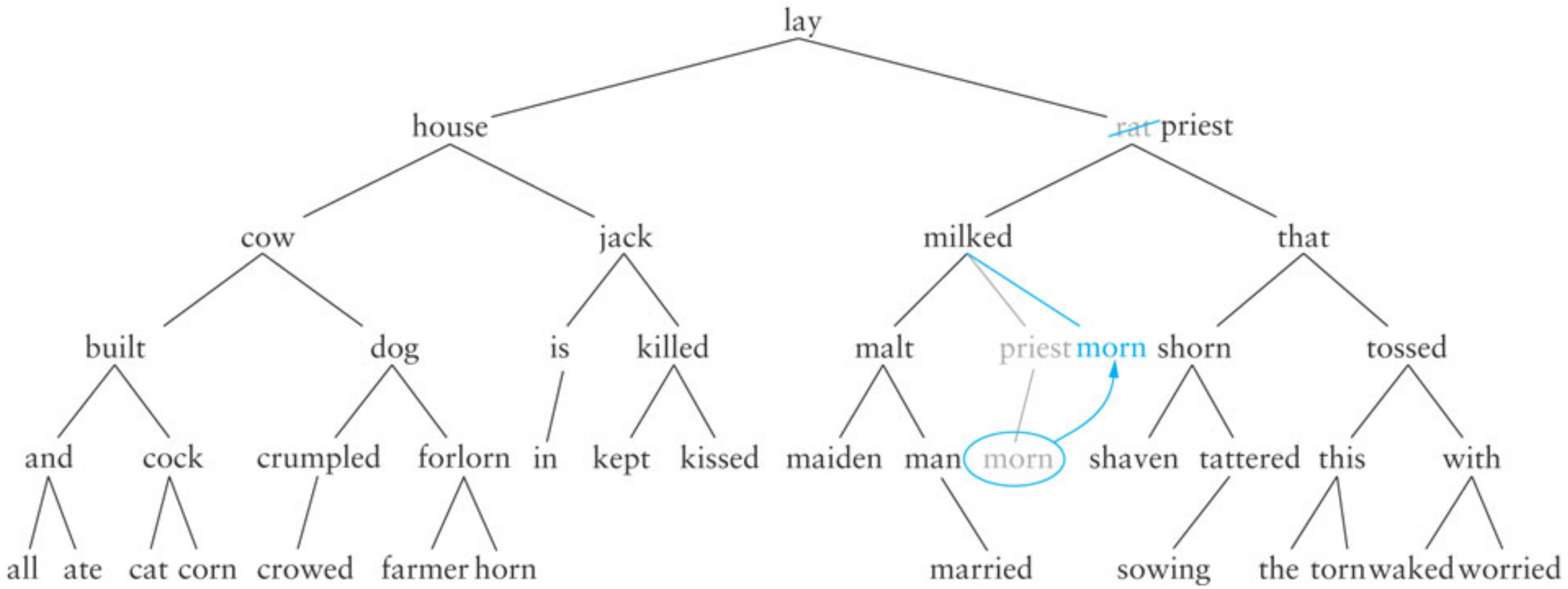


- If the item to be removed has two children, replace it with the largest item in its left subtree – the *inorder predecessor*

Removing from a Binary Search Tree (cont.)



Removing from a Binary Search Tree (cont.)



Algorithm for Removing from a Binary Search Tree

Recursive Algorithm for Removal from a Binary Search Tree

```
 1.  if the root is null
 2.      The item is not in tree – return null.
 3.  Compare the item to the data at the local root.
 4.  if the item is less than the data at the local root
 5.      Return the result of deleting from the left subtree.
 6.  else if the item is greater than the local root
 7.      Return the result of deleting from the right subtree.
 8.  else // The item is in the local root
 9.      Store the data in the local root in deletedReturn.
10.      if the local root has no children
11.          Set the parent of the local root to reference null.
12.      else if the local root has one child
13.          Set the parent of the local root to reference that child.
14.      else // Find the inorder predecessor
15.          if the left child has no right child it is the inorder predecessor
16.              Set the parent of the local root to reference the left child.
17.          else
18.              Find the rightmost node in the right subtree of the left
                 child.
19.              Copy its data into the local root's data and remove it by
                 setting its parent to reference its left child.
```

Implementing the `delete` Method

```
/** Starter method delete.
      post: The object is not in the tree.
      @param target The object to be deleted
      @return The object deleted from the tree
              or null if the object was not in the tree
      @throws ClassCastException if target does not implement
              Comparable
   */
  public E delete(E target) {
    root = delete(root, target);
    return deleteReturn;
  }
```

Implementing the `delete` Method

```
/** Recursive delete method.
    post: The item is not in the tree;
          deleteReturn is equal to the deleted item
          as it was stored in the tree or null
          if the item was not found.
    @param localRoot The root of the current subtree
    @param item The item to be deleted
    @return The modified local root that does not contain
            the item
 */
private Node < E > delete(Node < E > localRoot, E item) {
```

?????????????????