Hierarchical Data Representation

Dr. Youna Jung
Northeastern University
yo.jung@northeastern.edu

Trees

- ❑ **Nonlinear** and **Hierarchical** data structure
  - ✓ Tree nodes can have **multiple successors**, but only **one predecessor**
    - – E.g.) class hierarchy, disk directory and subdirectories, family tree

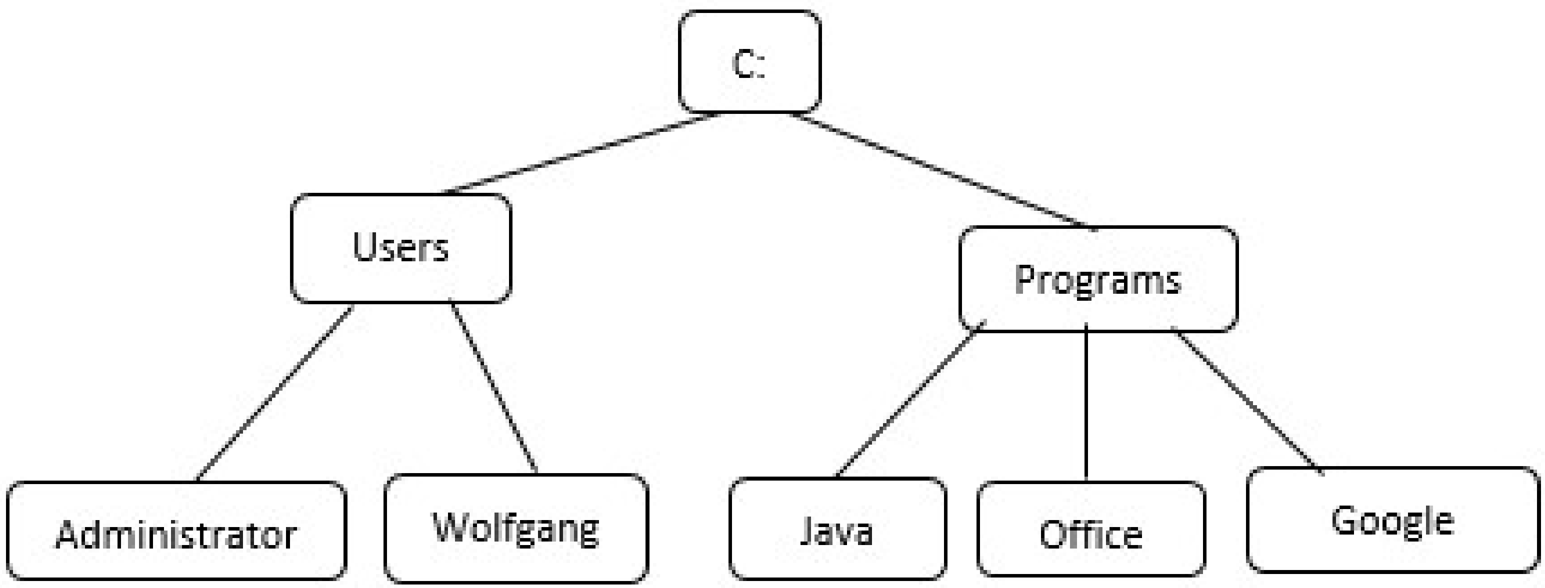
↕

- ❑ Linear data structure
  - ✓ Each element can have **only one** predecessor and successor
  - ✓ Accessing all elements in a linear sequence is **O(*n*)**

- ❑ **Recursive** data structures because they can be defined recursively

List and Tree Form of a Directory

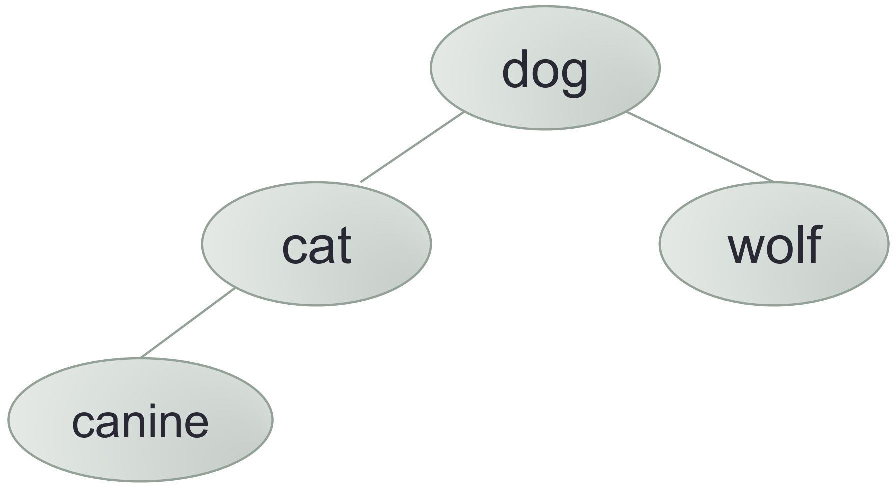
- C:
  - Users
    - Administrator
    - Wolfgang
  - Programs
    - Java
    - Office
    - Google



TREE TERMINOLOGY AND APPLICATIONS

Tree Terminology

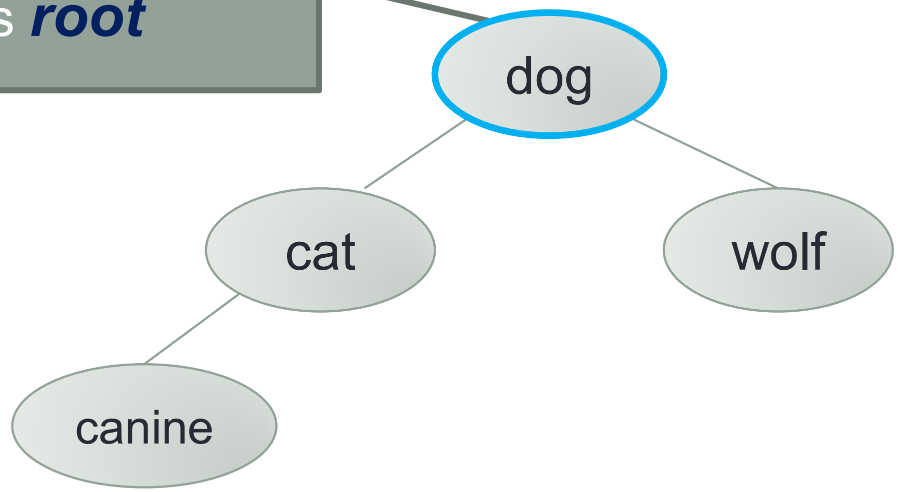
A **tree** consists of a **collection** of **elements** or **nodes**, with each node **linked** to its **successors**



Tree Terminology (cont.)

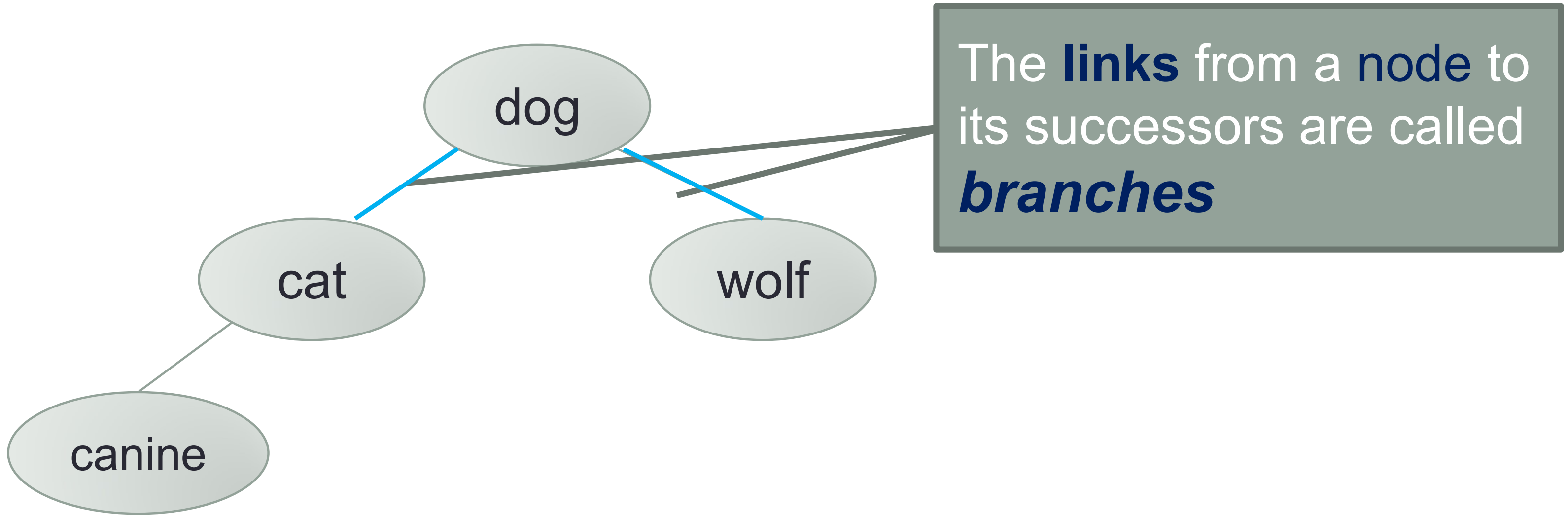
A tree consists of a collection of elements or nodes, with each node linked to its successors

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



Tree Terminology (cont.)

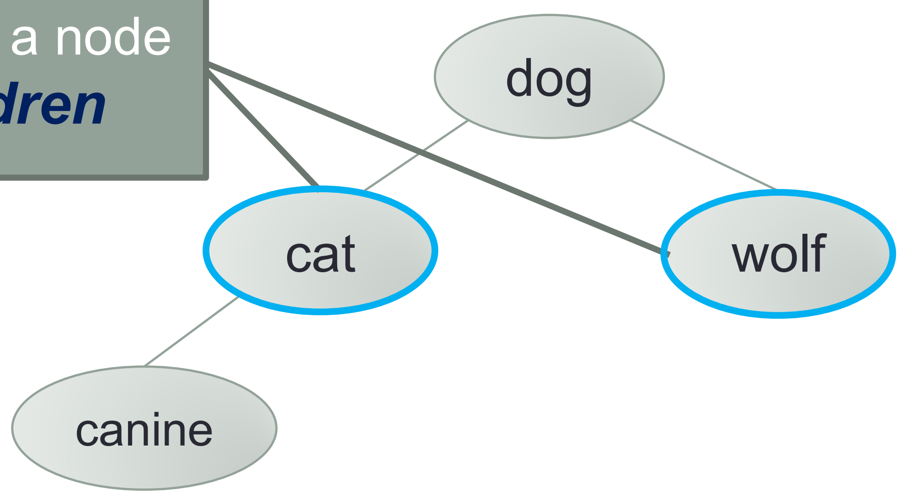
A tree consists of a collection of elements or nodes, with each node linked to its successors



Tree Terminology (cont.)

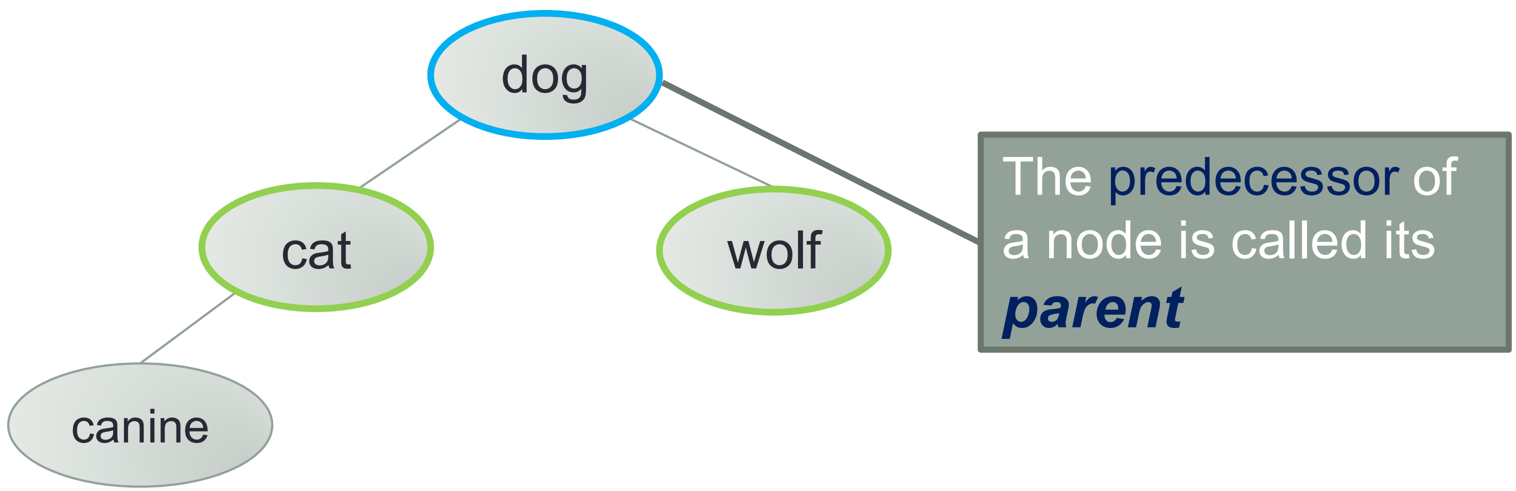
A tree consists of a collection of elements or nodes, with each node linked to its successors

The successors of a node are called its ***children***



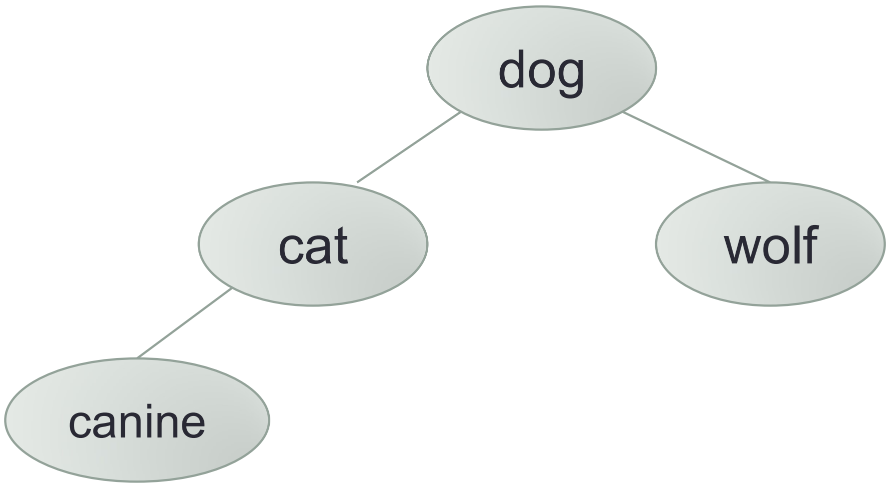
Tree Terminology (cont.)

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



Tree Terminology (cont.)

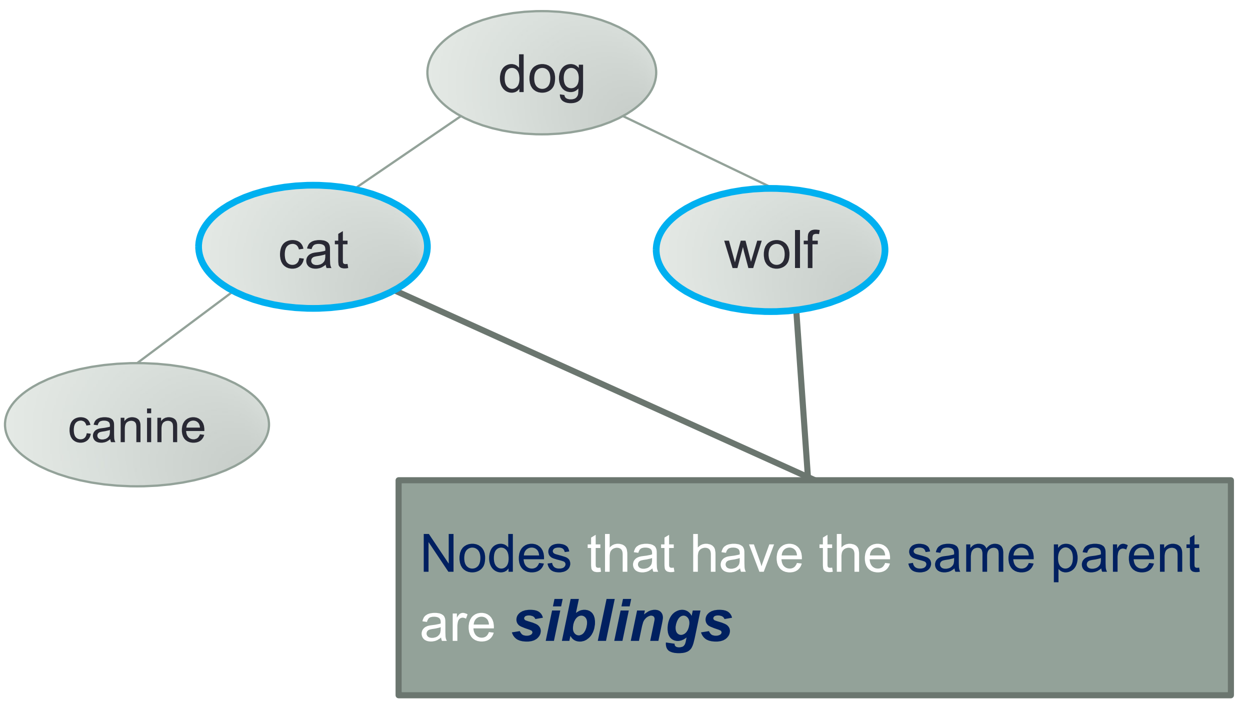
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

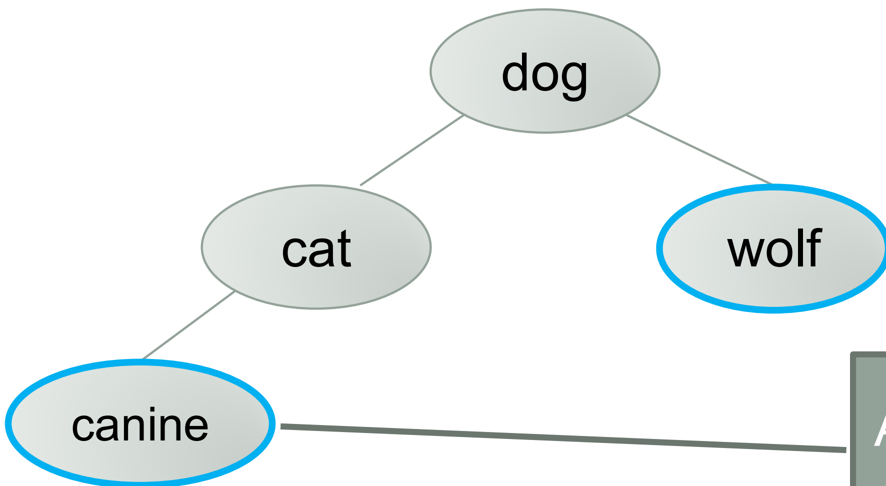
Tree Terminology (cont.)

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



Tree Terminology (cont.)

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

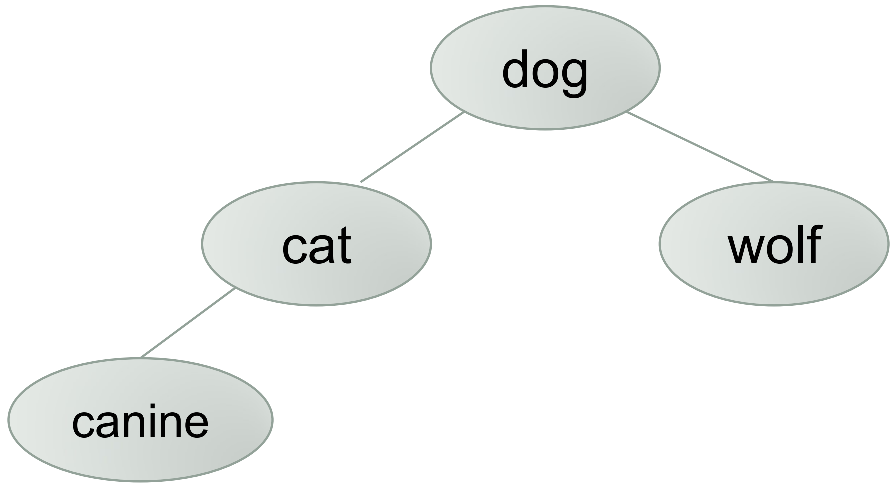


A node that has no children is called a ***leaf node***

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

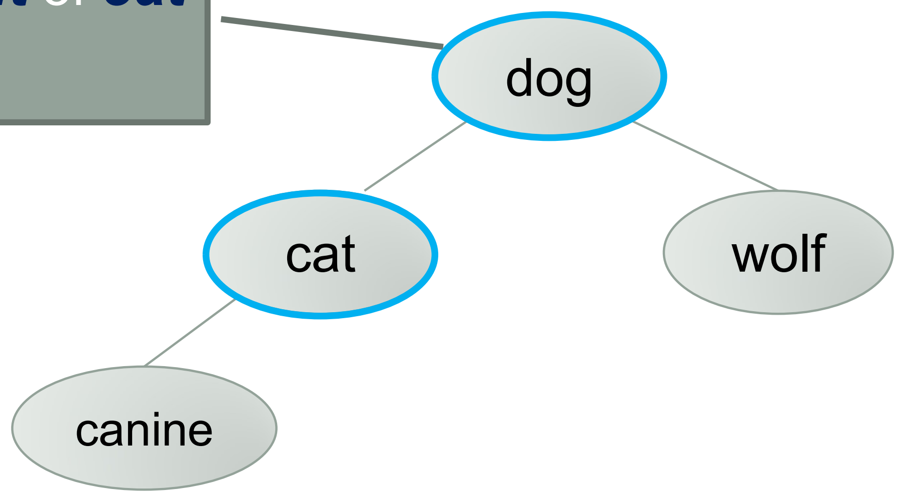


A **generalization** of the **parent-child relationship** is the **ancestor-descendant relationship**

Tree Terminology (cont.)

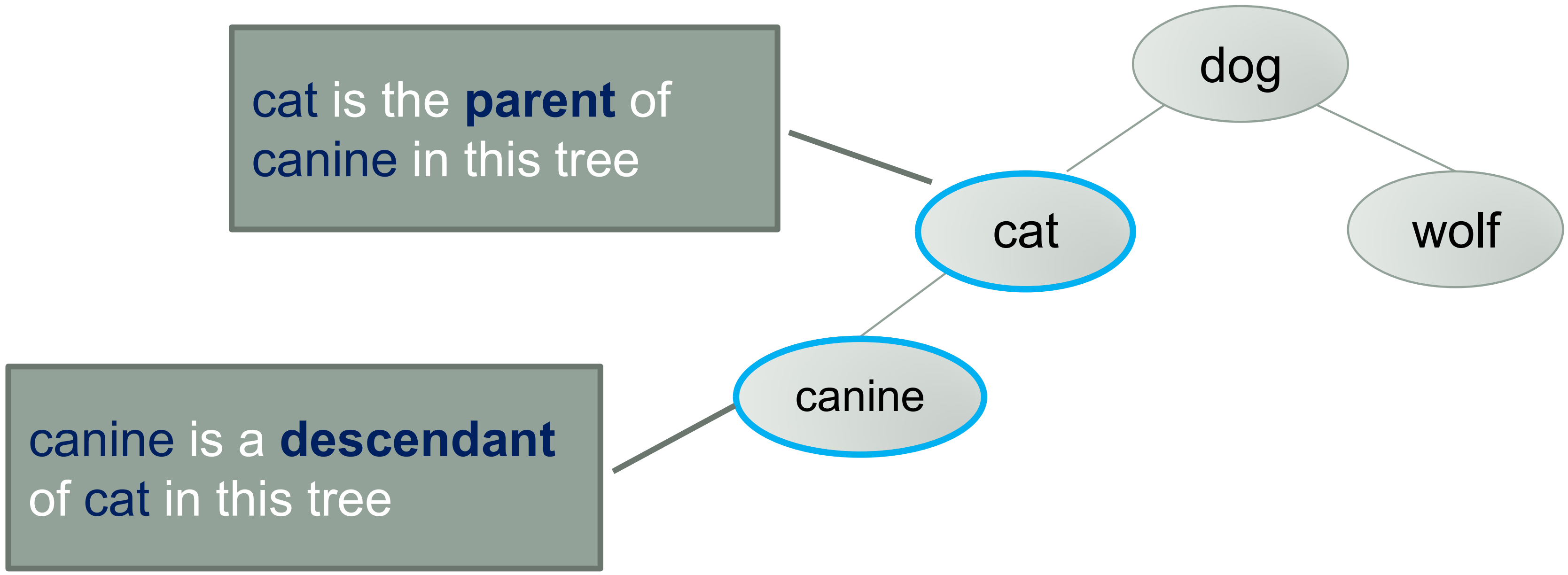
A tree consists of a collection of elements or nodes, with each node linked to its successors

dog is the ***parent*** of ***cat*** in this tree



Tree Terminology (cont.)

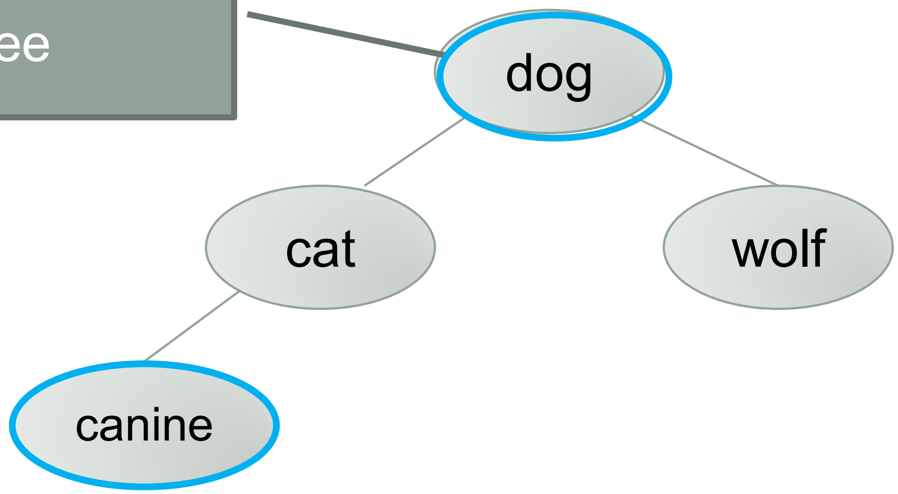
A tree consists of a collection of elements or nodes, with each node linked to its successors



Tree Terminology (cont.)

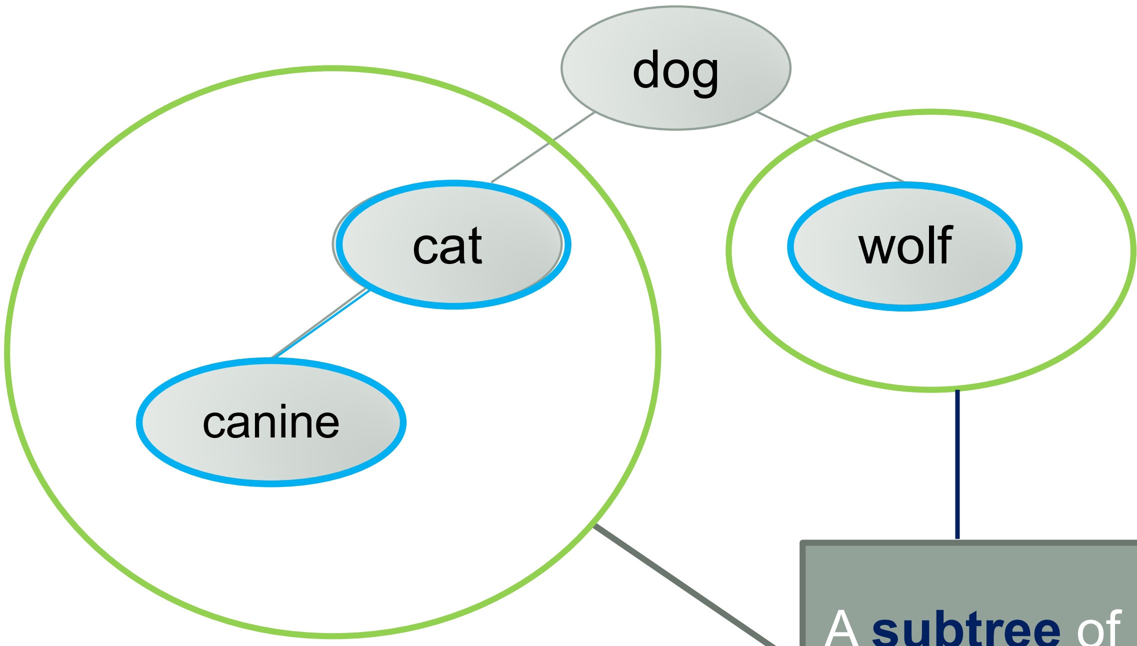
A tree consists of a collection of elements or nodes, with each node linked to its successors

dog is an **ancestor** of canine in this tree



Tree Terminology (cont.)

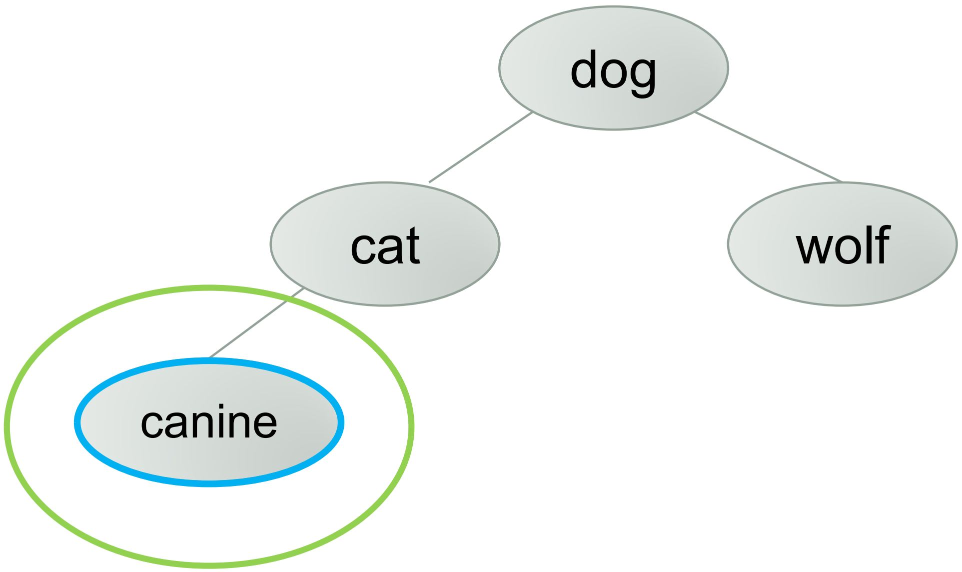
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

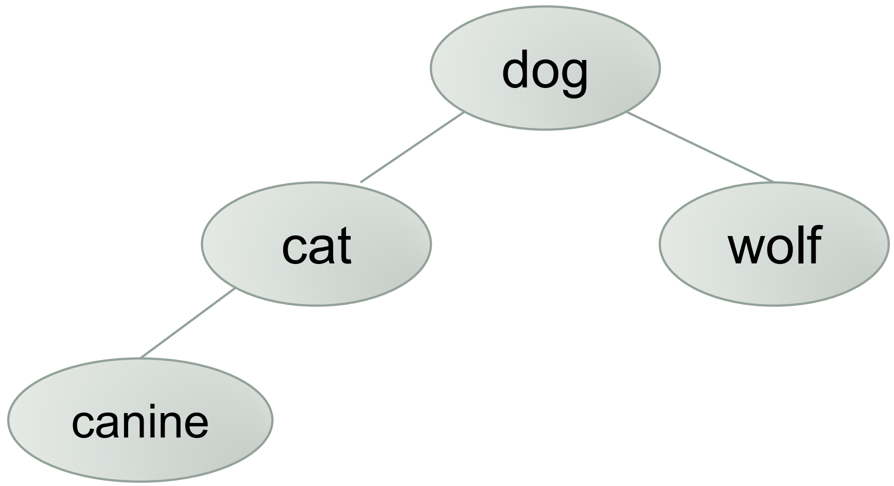
Tree Terminology (cont.)

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



Tree Terminology (cont.)

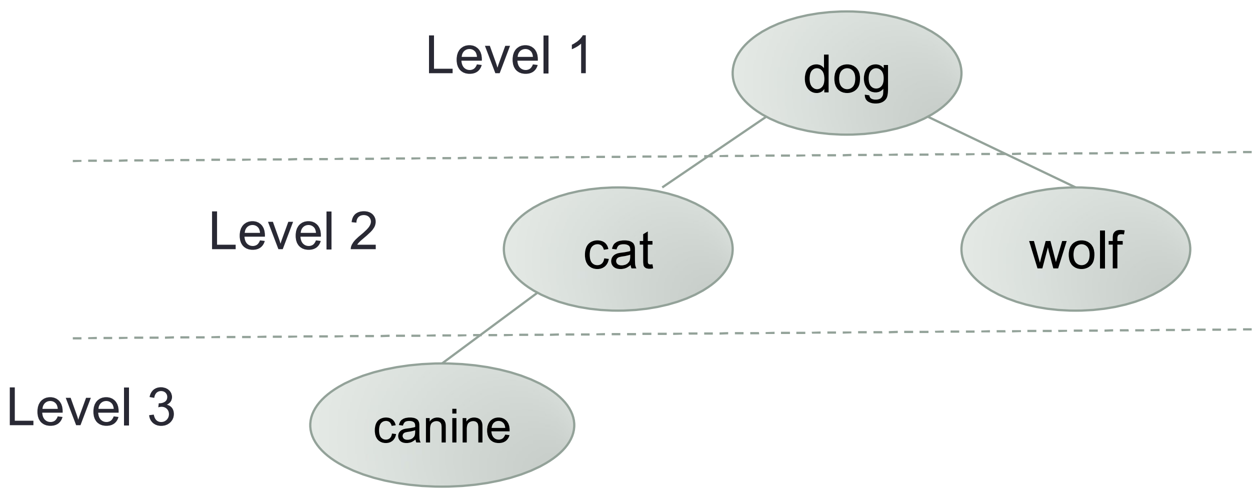
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

Tree Terminology (cont.)

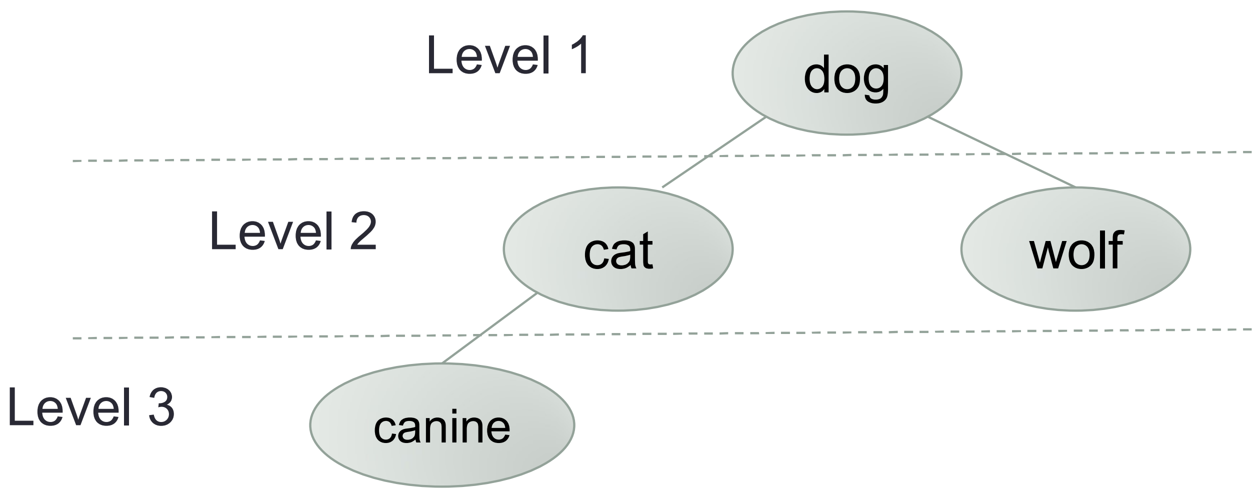
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

Tree Terminology (cont.)

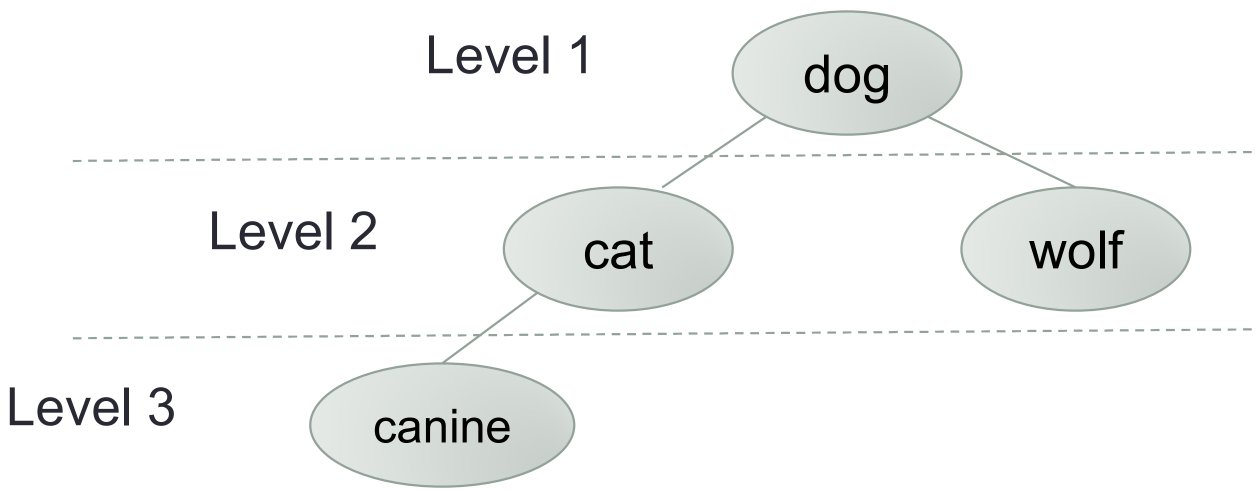
A tree consists of a collection of elements or nodes, with each node linked to its successors



The **level** of a node is defined **recursively**

Tree Terminology (cont.)

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

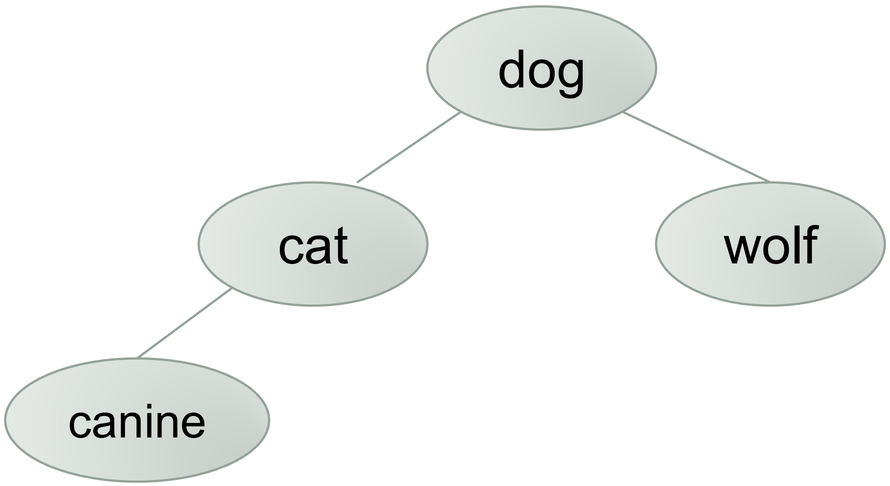


The **level** of a node is defined recursively

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

Tree Terminology (cont.)

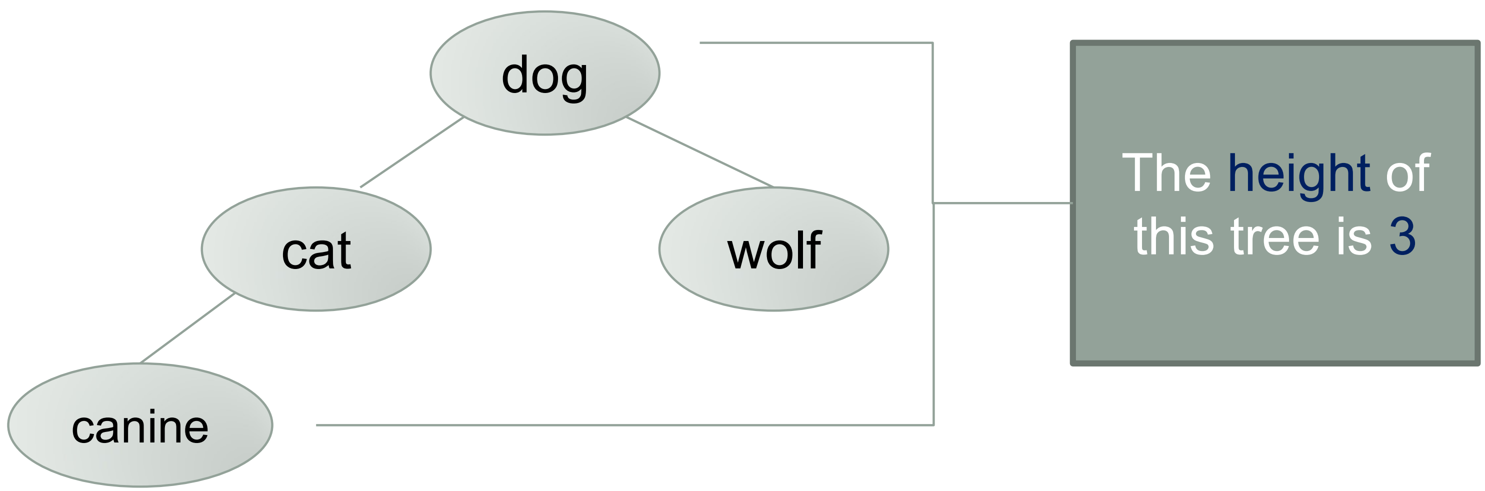
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

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

Binary Trees

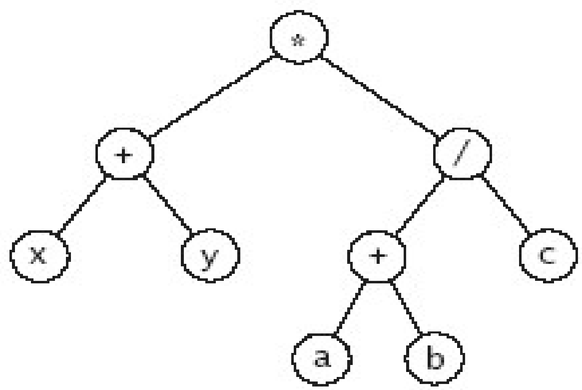
- **Each node** has **2 subtrees**

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, $\mathbf{T_L}$ (left subtree) and $\mathbf{T_R}$ (right subtree), such that $\mathbf{T_L}$ and $\mathbf{T_R}$ are **binary** trees

An Infix Expression as a Tree

- Each **node** contains an **operator** or an **operand**
  - ✓ **Operands** are stored in **leaf** nodes
- Parentheses are not stored
  - ✓ Tree structure dictates the **order** of **operand evaluation**
    - – **Operators** in nodes at **higher** tree **levels** are **evaluated after** operators in nodes at **lower** tree **levels**



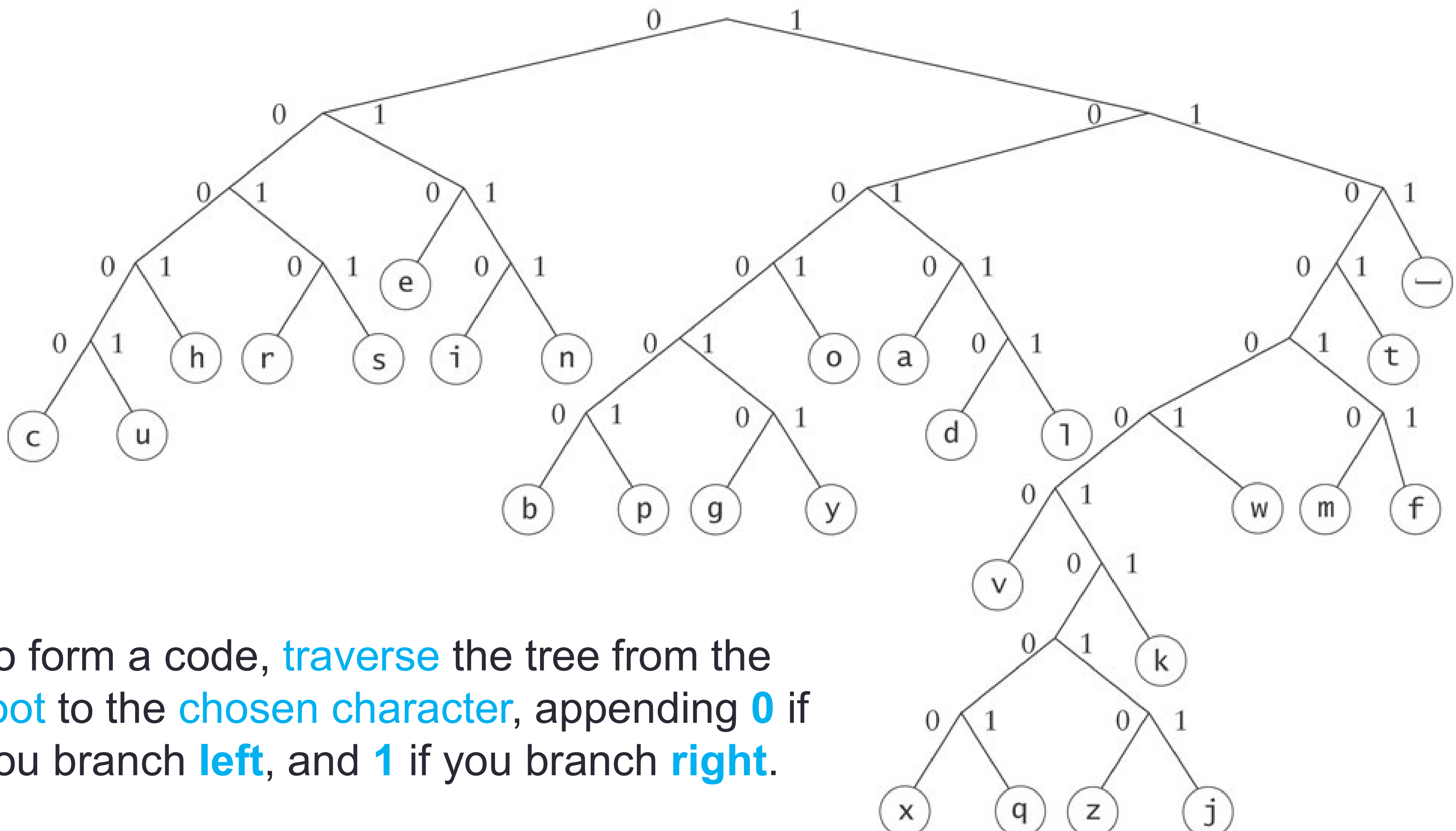
(x + y) * ((a + b) / c)

Huffman Tree

- represents **Huffman codes** for **characters**
  - ✓ uses **different numbers of bits** to encode **letters**
    - – As opposed to **ASCII** or **Unicode** (**Same numbers** of **bits**)
  - ✓ **more common** characters use **fewer** bits

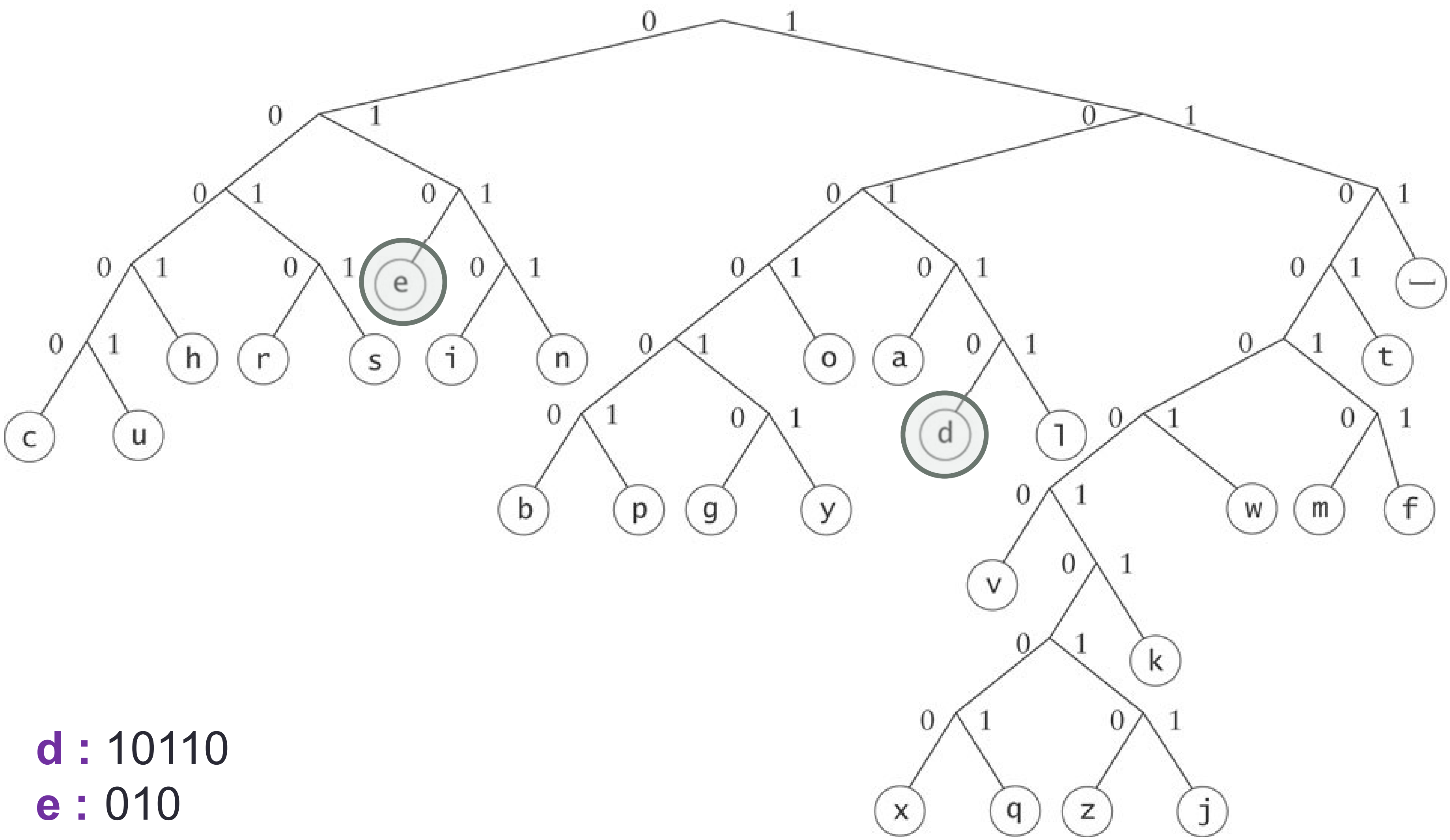
- Many programs that **compress files** use **Huffman** codes

Huffman Tree (cont.)



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

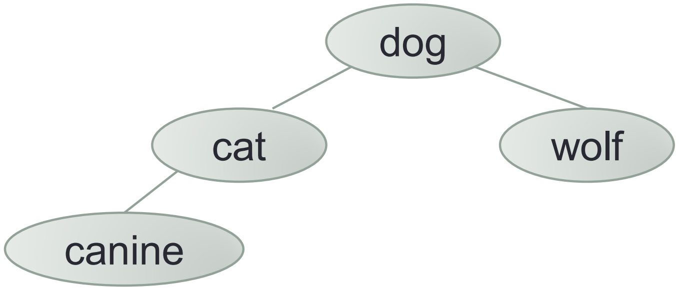
Huffman Tree (cont.)



d : 10110

e : 010

Binary Search Tree



- A **Binary Tree** AND all **elements** in the **left** subtree **precede** those in the **right** subtree
  - ✓ ➔ **Binary Tree** AND **T_L < Middle < T_R**

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 is **less than** all values in T_R

Binary Search Tree

- When new elements are **inserted** (or **removed**) properly, the BST **maintains** its **order**
  - ✓ In contrast, a **sorted array** must be **expanded** whenever new elements are added, and **compacted** whenever elements are removed—expanding and contracting are both **O(*n*)**
- 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*)**

*What would be the **worst case** of **searching** in **BST**?*

Recursive Algorithm for Searching a BT

Base 1 1. `if` the tree is empty

2. return null *(target is not found)*

Base 2 `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

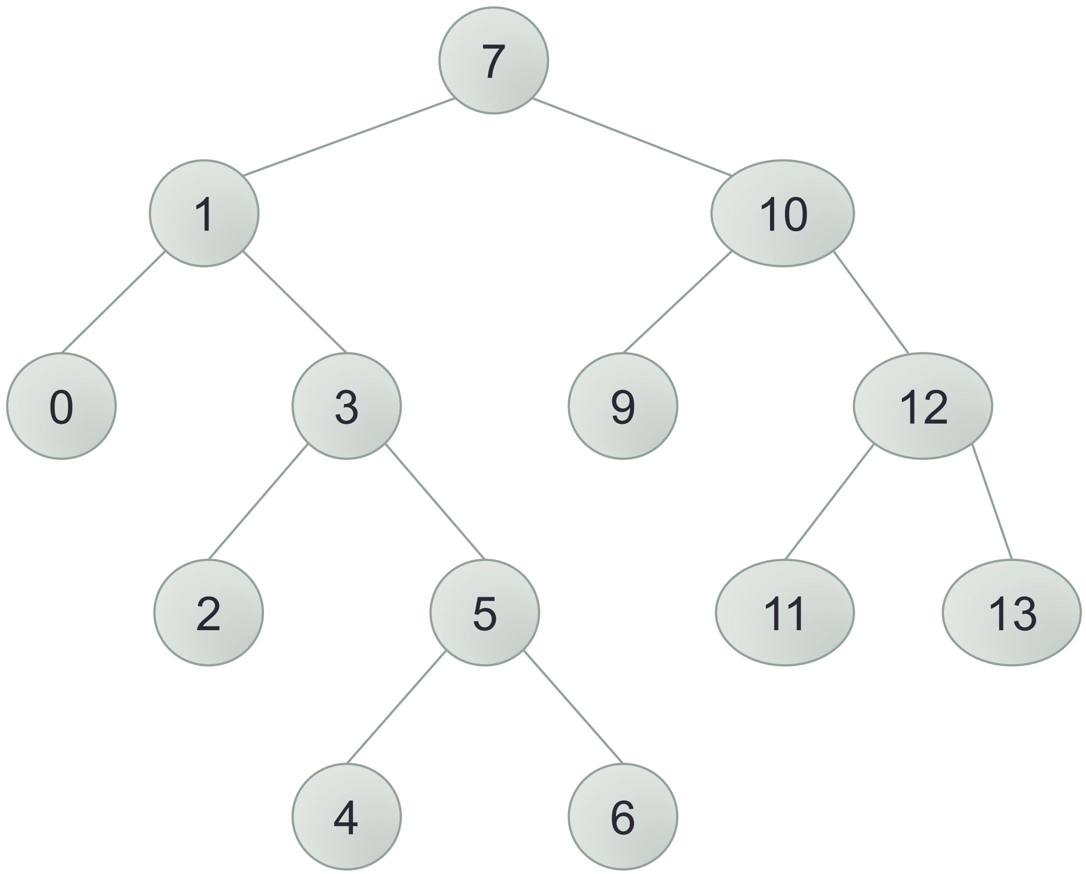
Recursive 1 4. return the result of **searching** the **left subtree** of the root

`else`

Recursive 2 5. return the result of **searching** the **right subtree** of the root

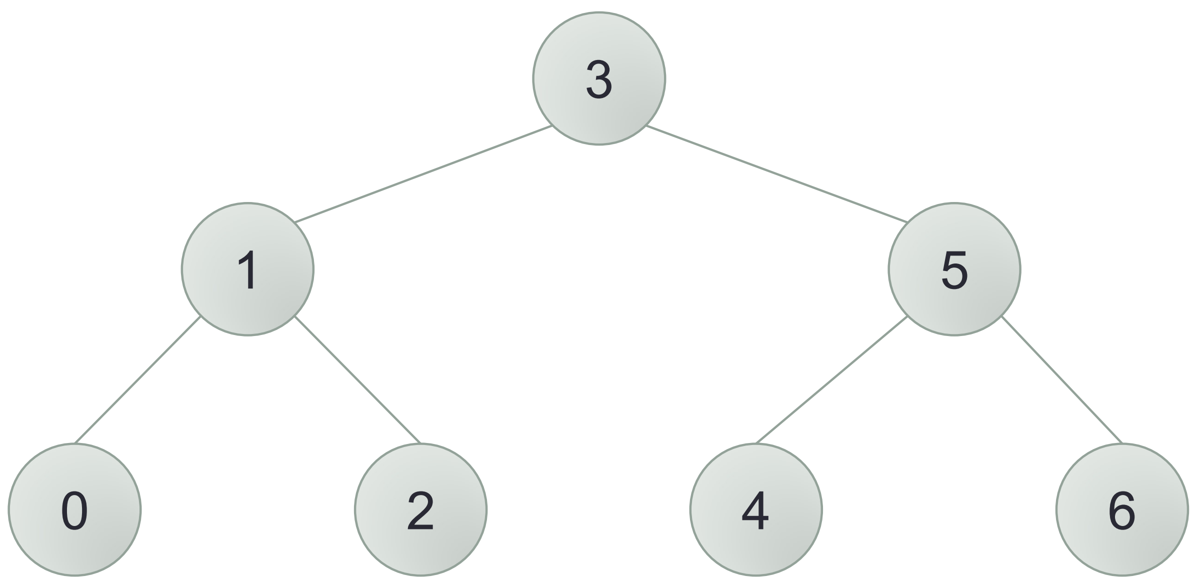
Full BT

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



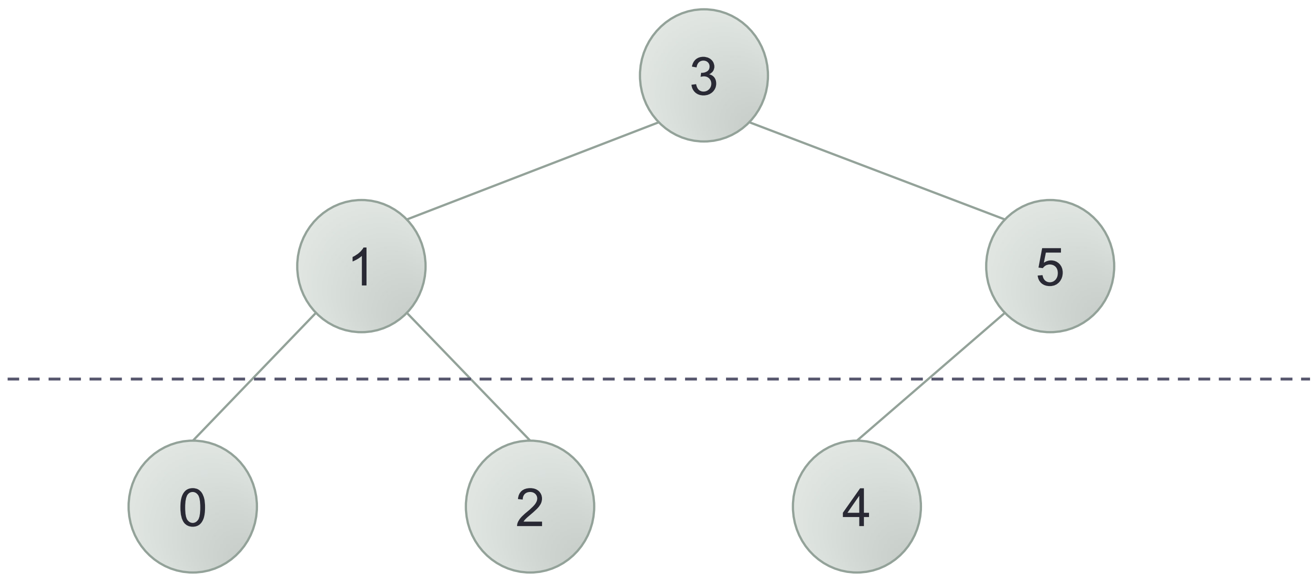
Perfect BT

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



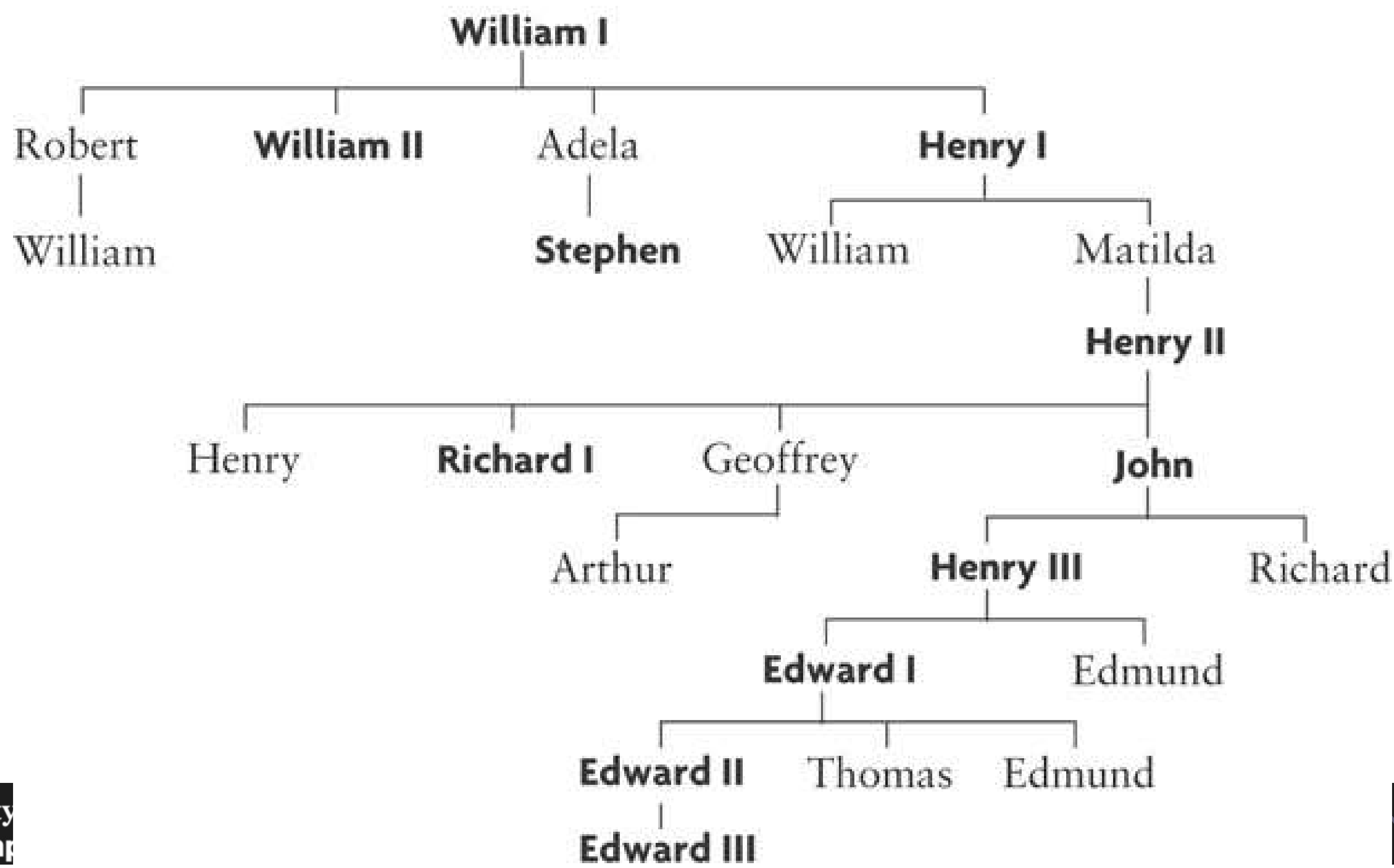
Complete BT

- 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**



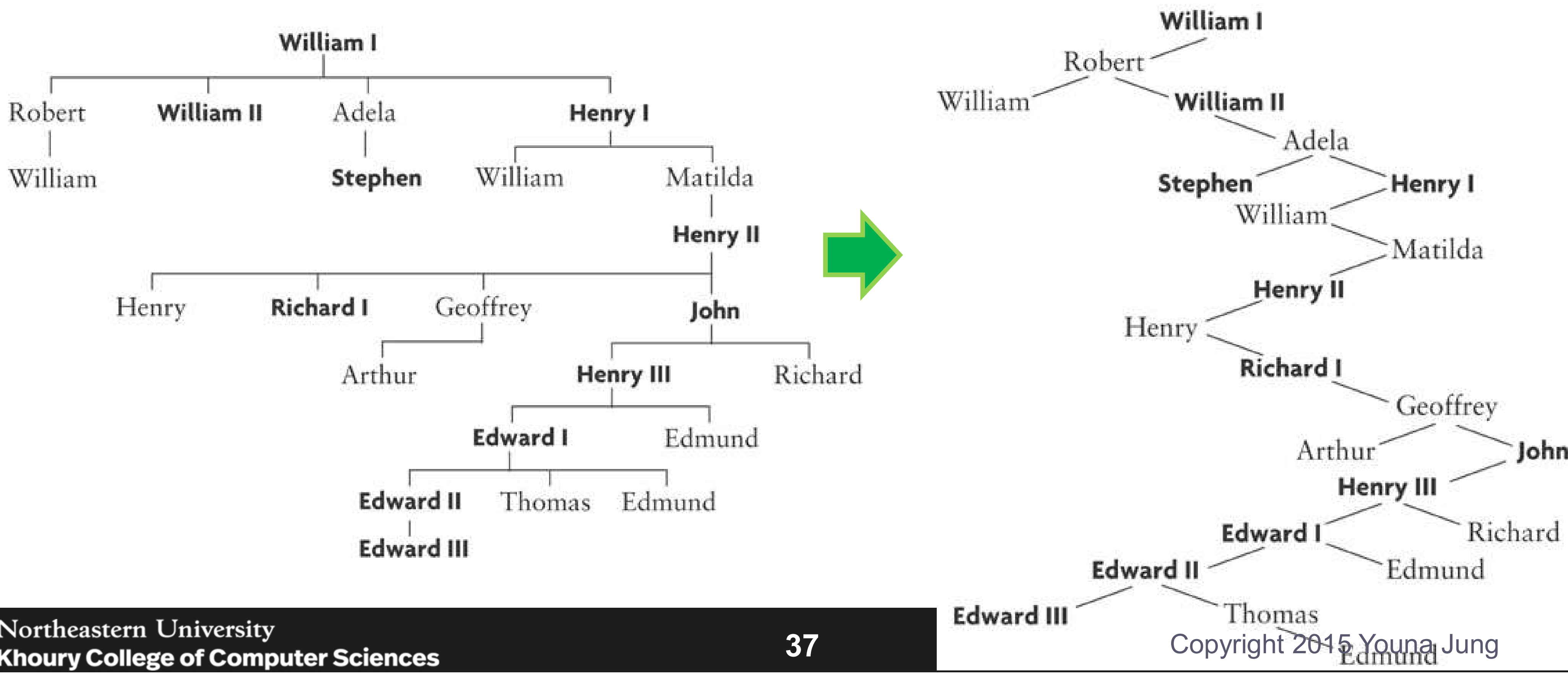
General Trees

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



General Trees

- A general tree can be represented using a **BT**
  - ✓ 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

Tree Traversals

- **Walking through** the tree in a prescribed **order** and **visiting** the **nodes**

- 3 **Types** of Tree Traversal
  - **Inorder**
    - traverse $T_L \rightarrow$ Root $\rightarrow T_R$
  - **Preorder**
    - traverse Root $\rightarrow T_L \rightarrow T_R$
  - **Postorder**
    - traverse $T_L \rightarrow T_R \rightarrow$ Root

Tree Traversals (cont.)

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.

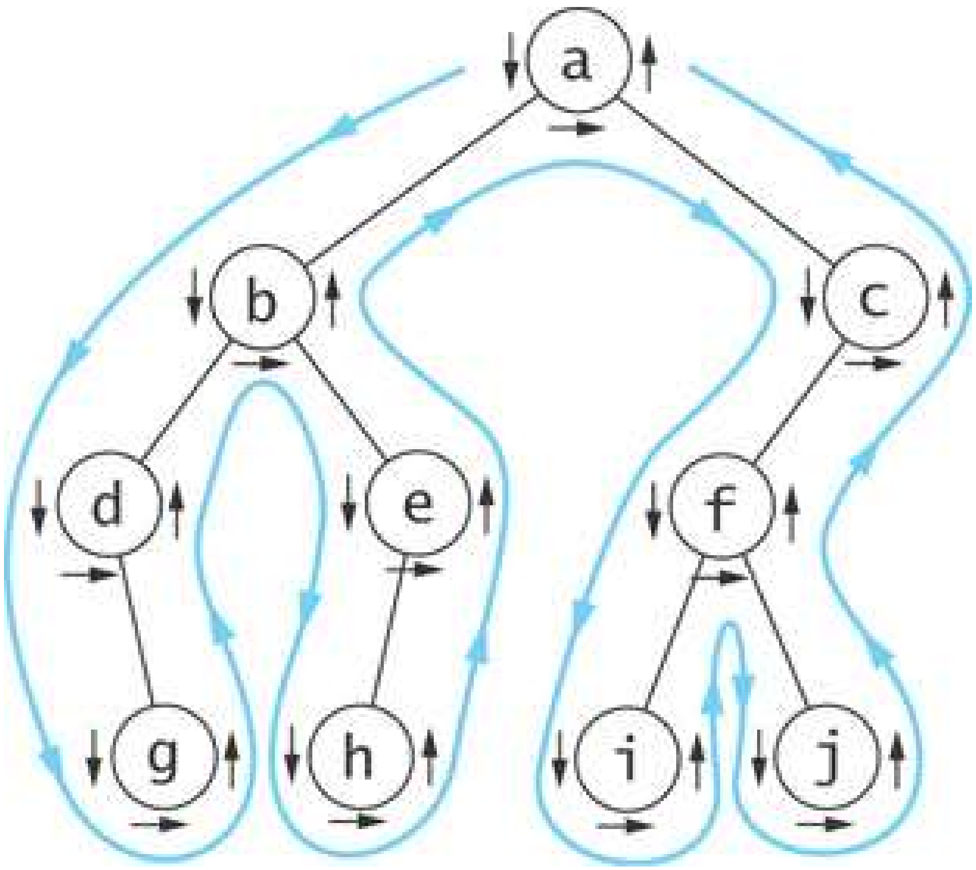
Algorithm for Inorder Traversal

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.

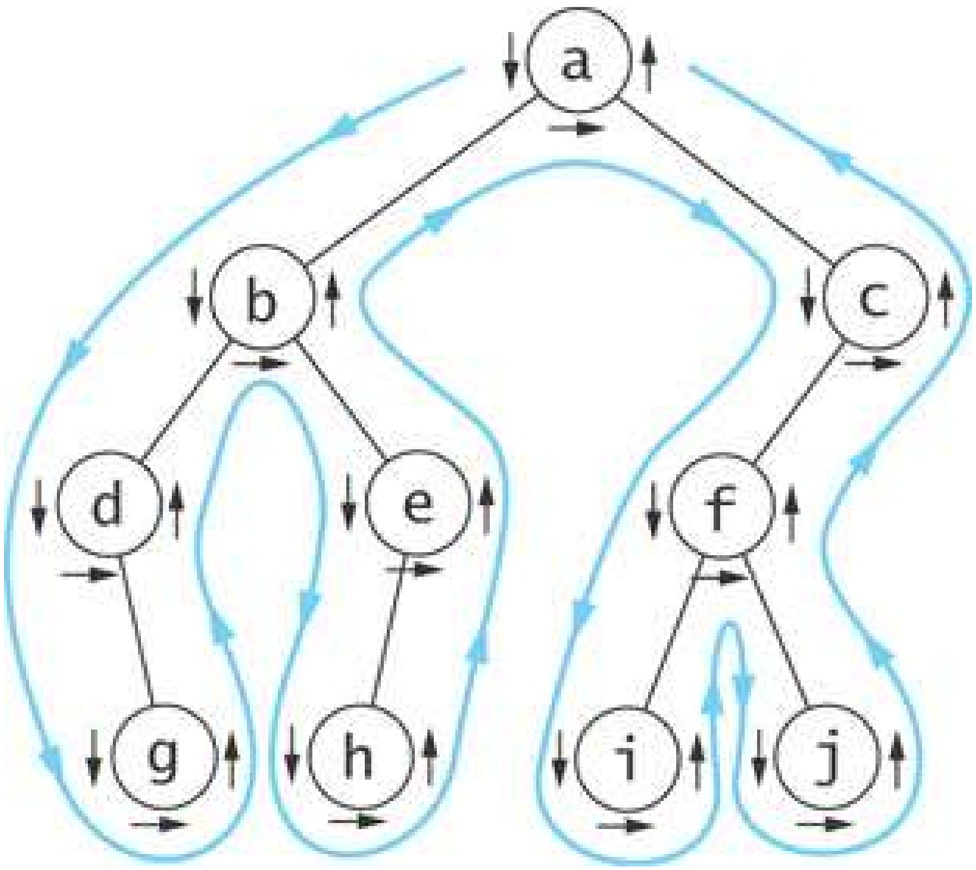
Visualizing Tree Traversals



Inorder? Preorder? Postorder?

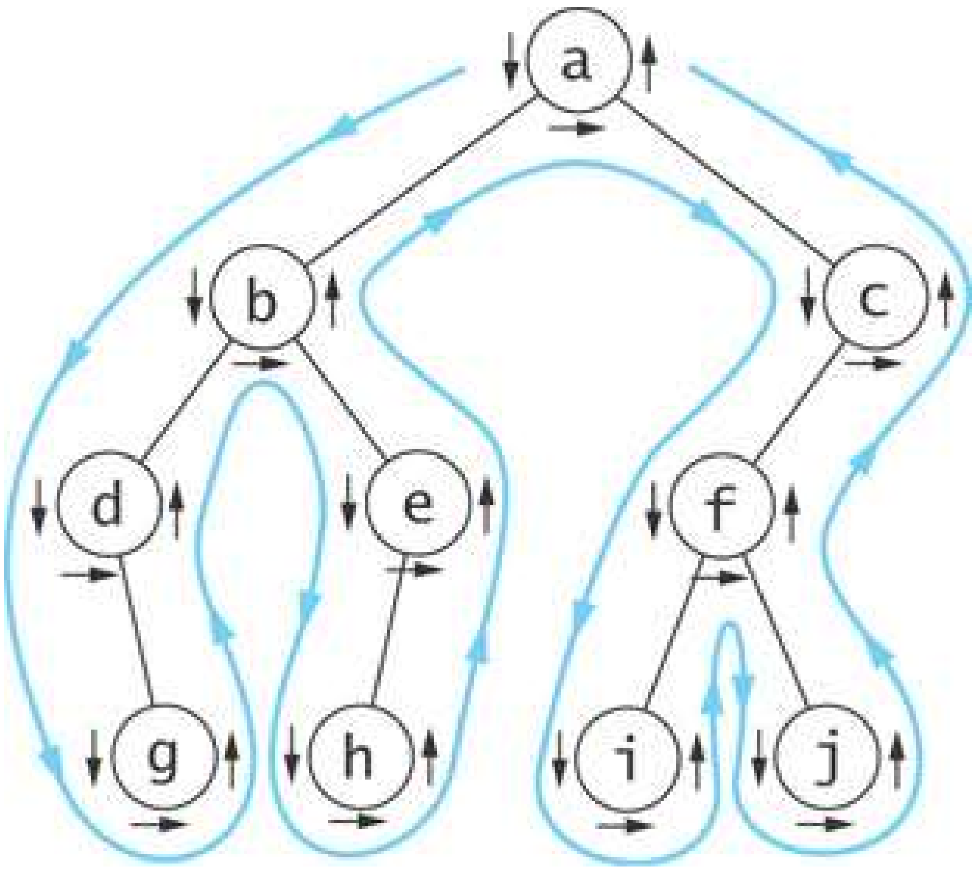
Preorder traversal : a→ b→ d→ g → e → h → c → f → i → j

Visualizing Inorder Tree Traversals



Inorder traversal : d→ g → b → h → e → a → i → f → j → c

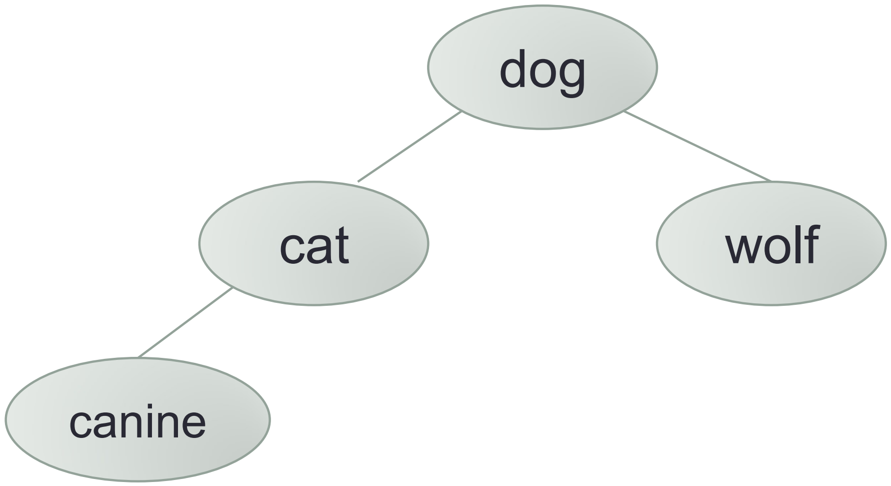
Visualizing Postorder Tree Traversals



Postorder traversal : g→ d → h → e → b → i → j → f → c → a

Traversals of BST and Expression Trees

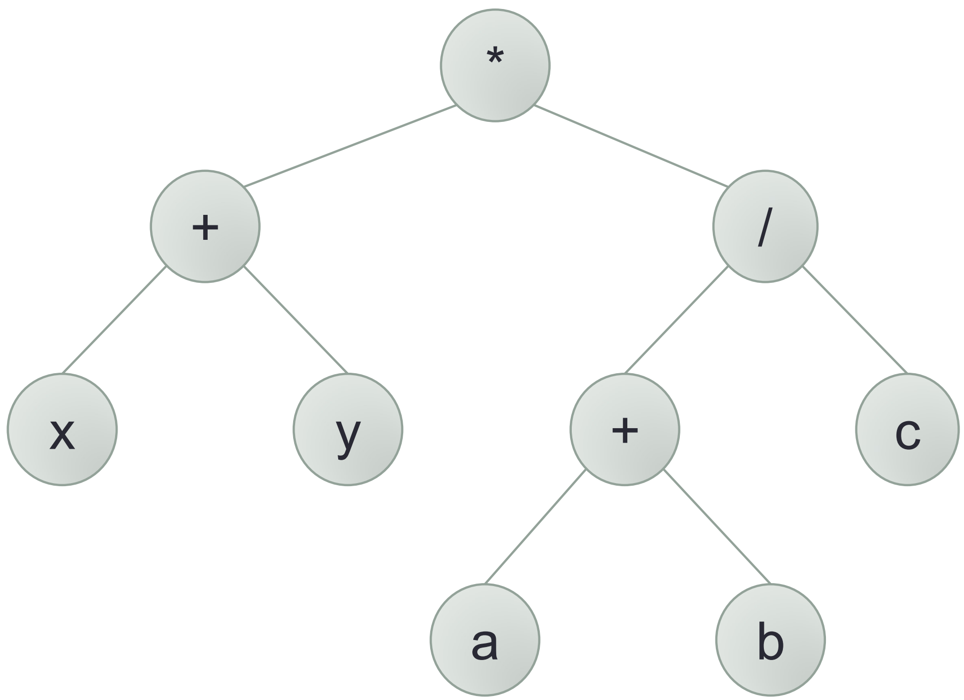
- An **inorder** traversal of a **BST** results in the nodes being visited in **sequence** by **increasing data value**



canine → cat → dog → wolf

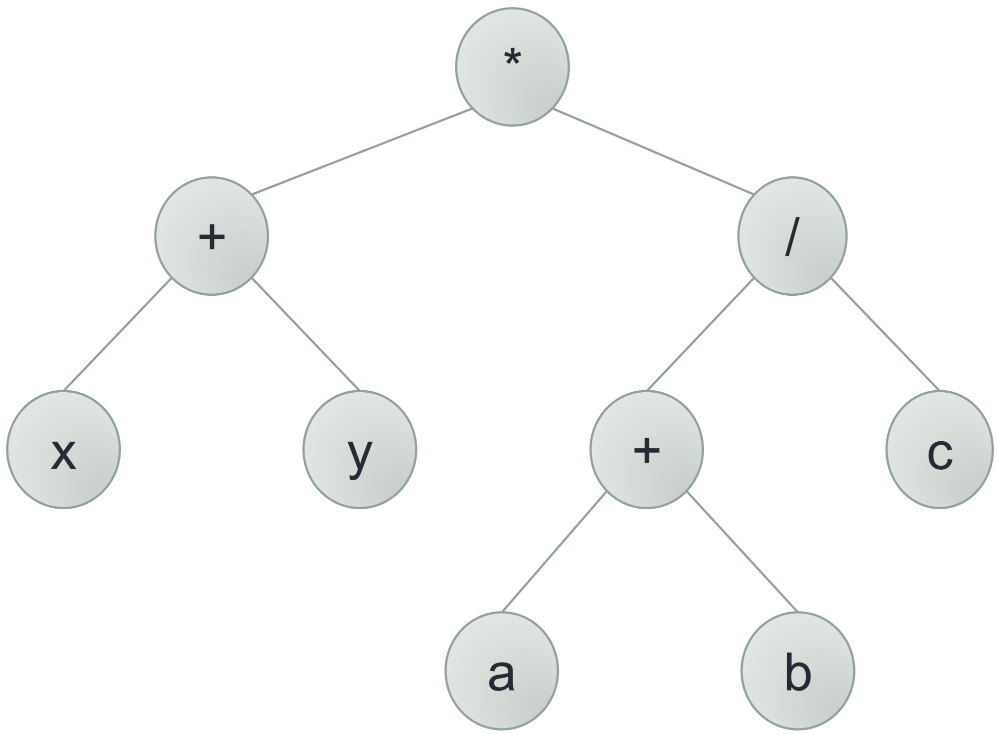
Traversals of BST and Expression Trees

- An **inorder** traversal of this 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)`



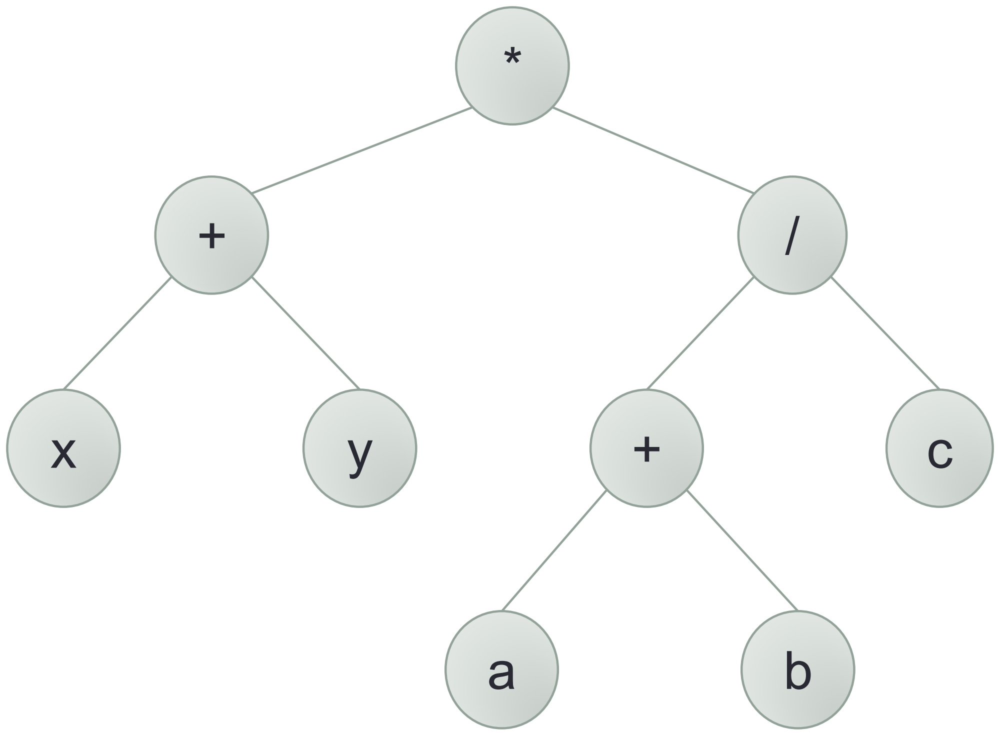
Traversals of BST and Expression Trees

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



Traversals of BST and Expression Trees

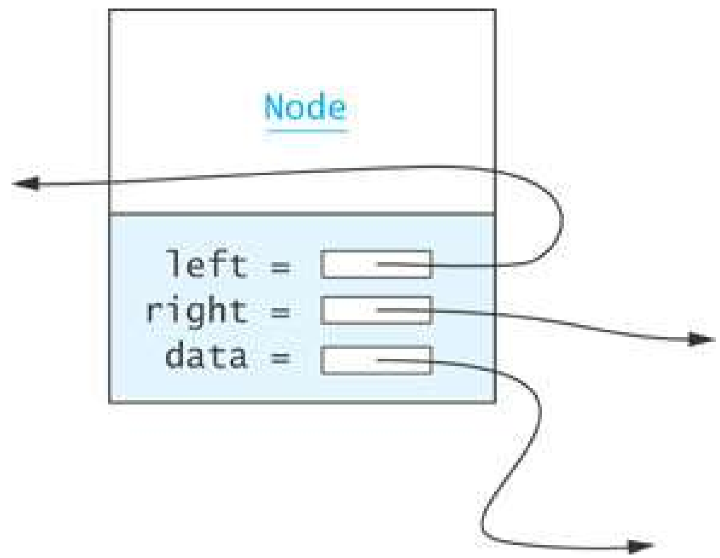
- A **preorder** traversal of this expression tree results in the sequence: *** + x y / + a b c**
- This is the **prefix** or **forward** form of the **expression**
  - ✓ **Operators** precede **operands**



IMPLEMENTING `BINARYTREE` CLASS

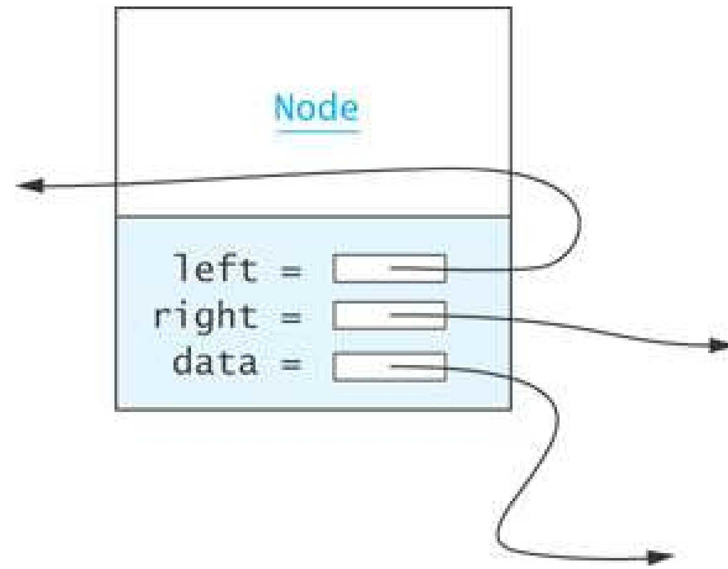
`Node<E>` Class

- 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.)

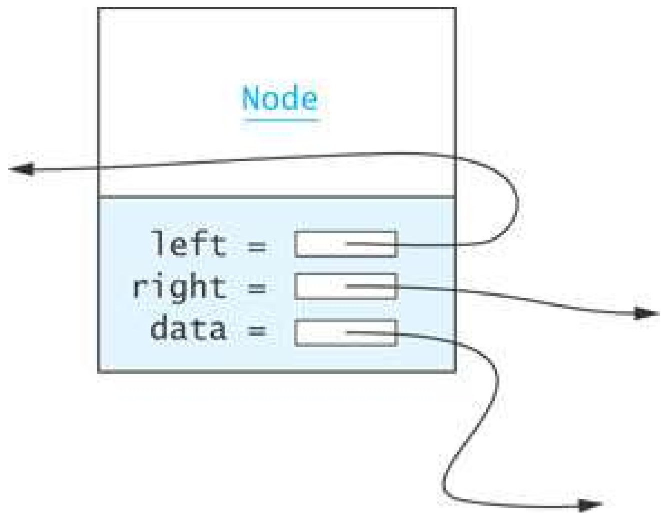
```java
protected static class Node<E> {
   protected E data;
   protected Node<E> left;
   protected Node<E> right;

   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>**

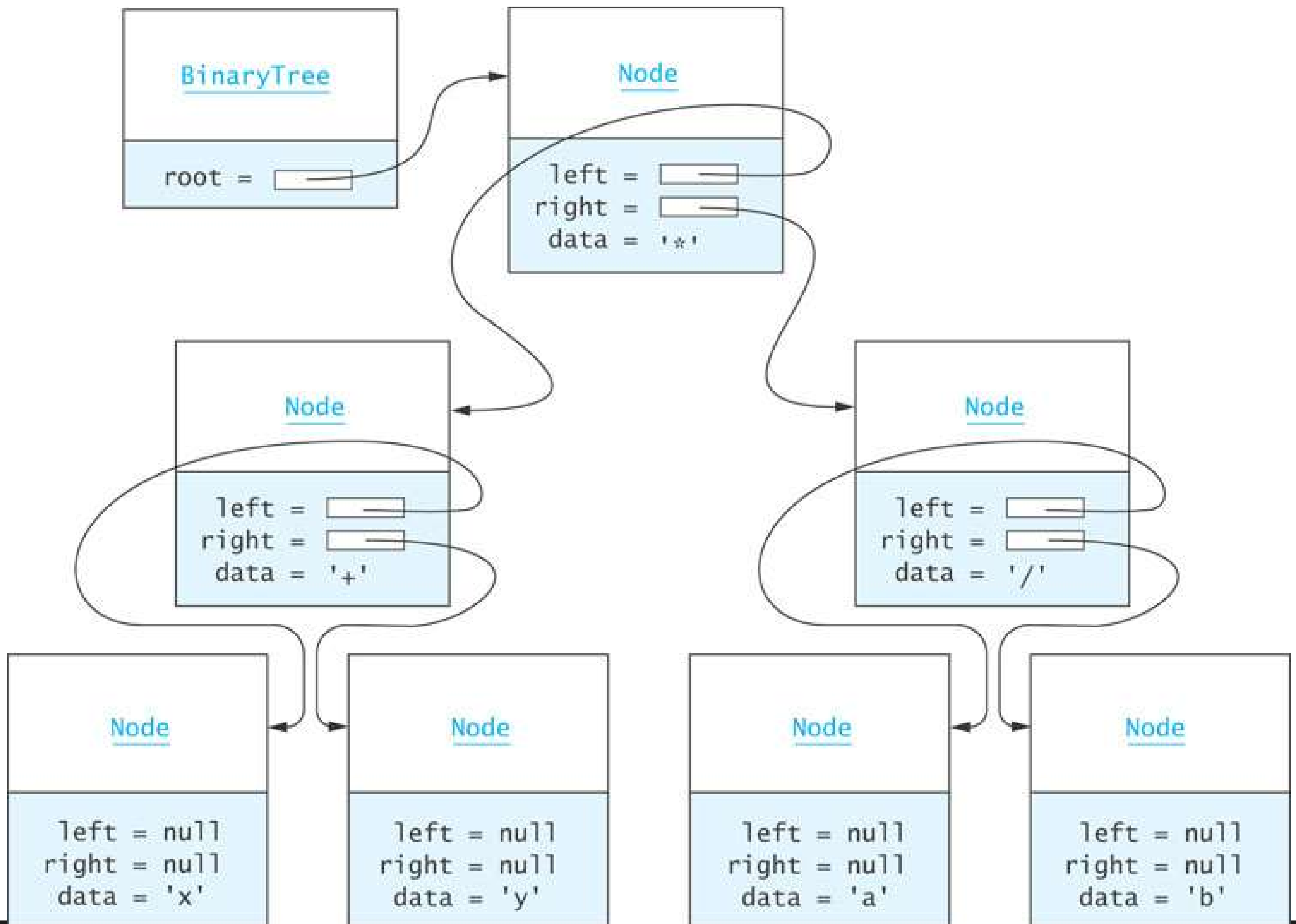
Node<E> Class (cont.)

```
protected static class Node<E> {
 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();
 }
}
```

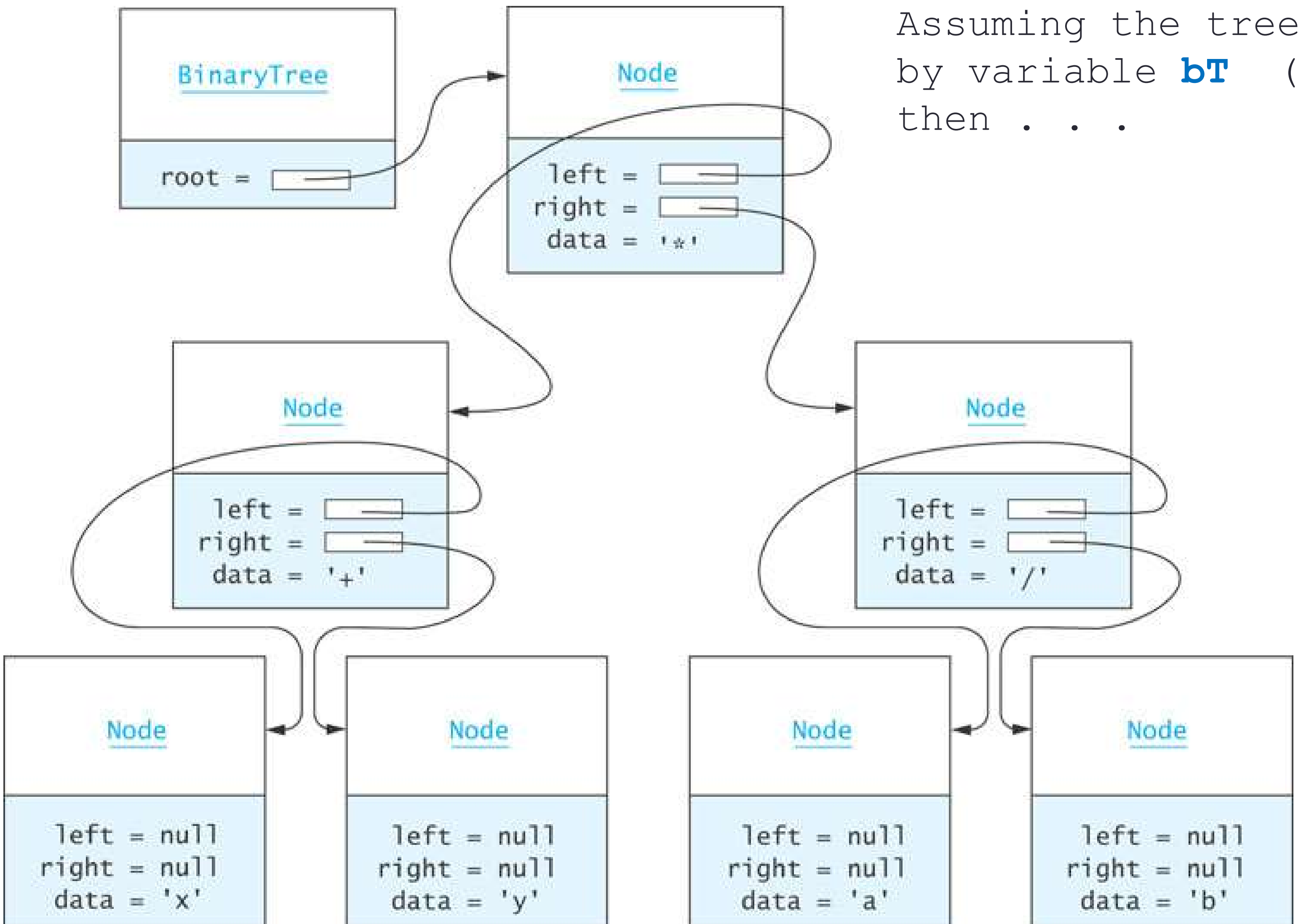


Node<E> is declared **protected.** This way we can use it as a superclass.

BinaryTree<E> Class



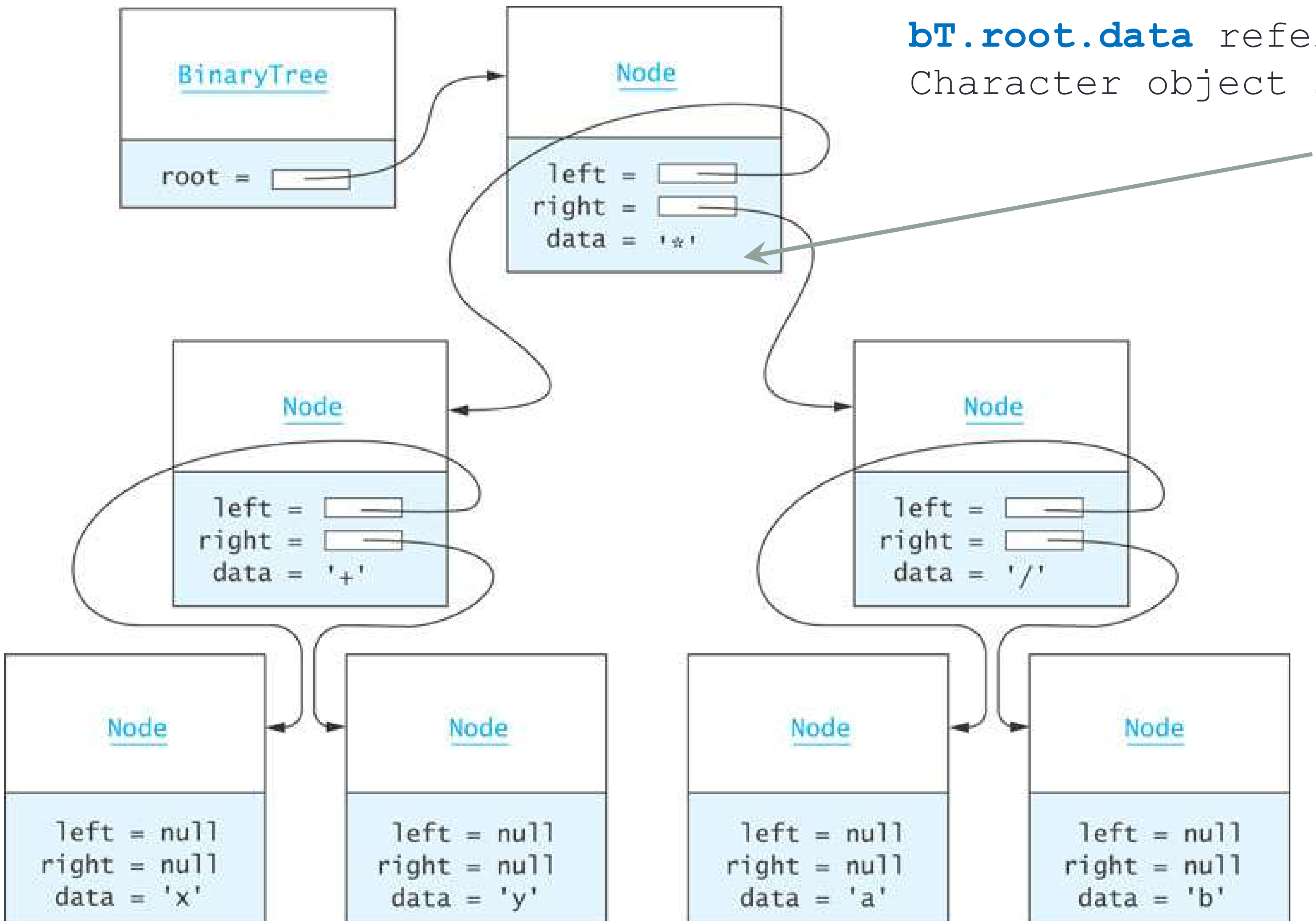
BinaryTree<E> Class (cont.)



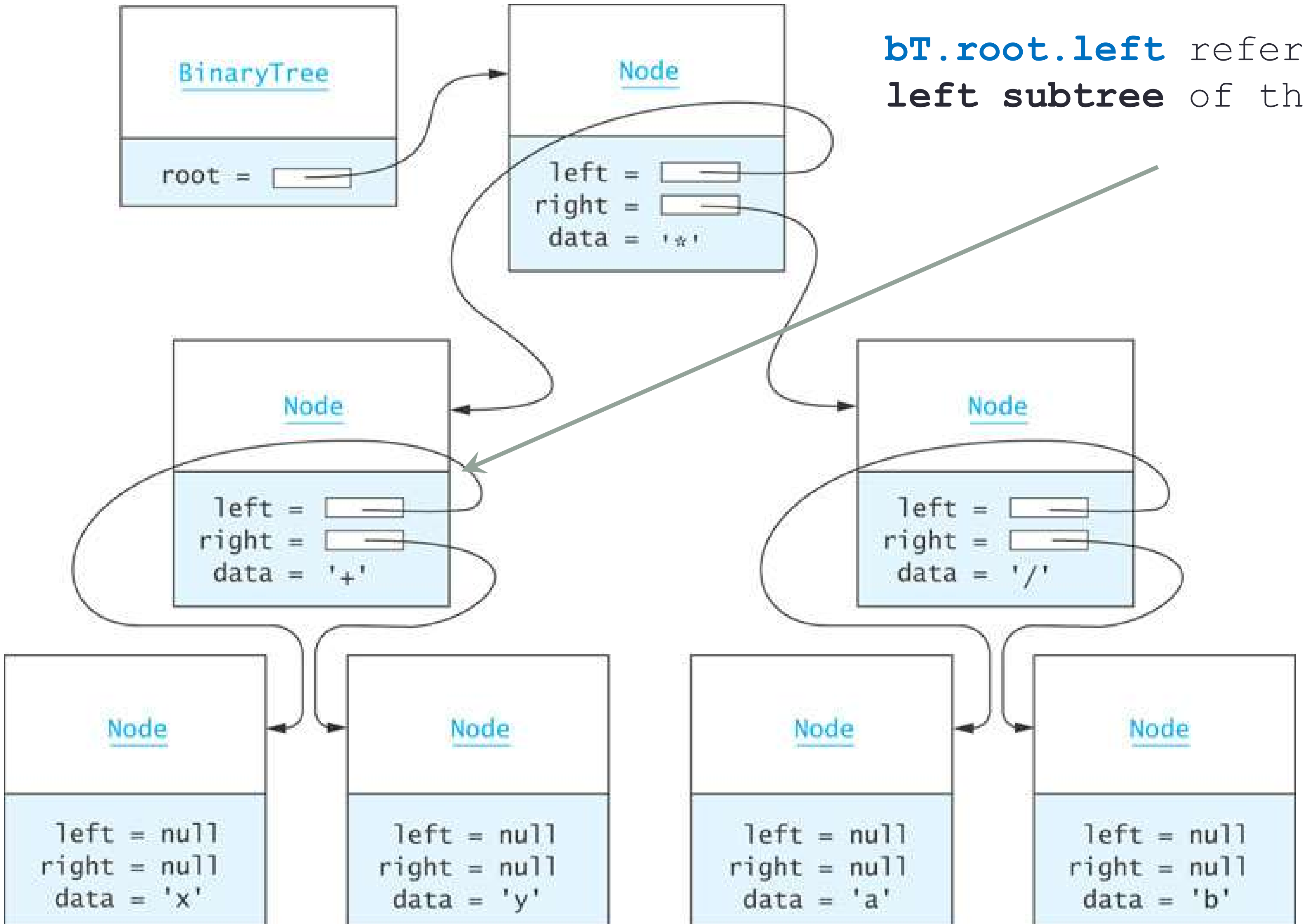
Assuming the tree is referenced by variable **bT** (type **BinaryTree**) then . . .

`BinaryTree<E>` Class (cont.)

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

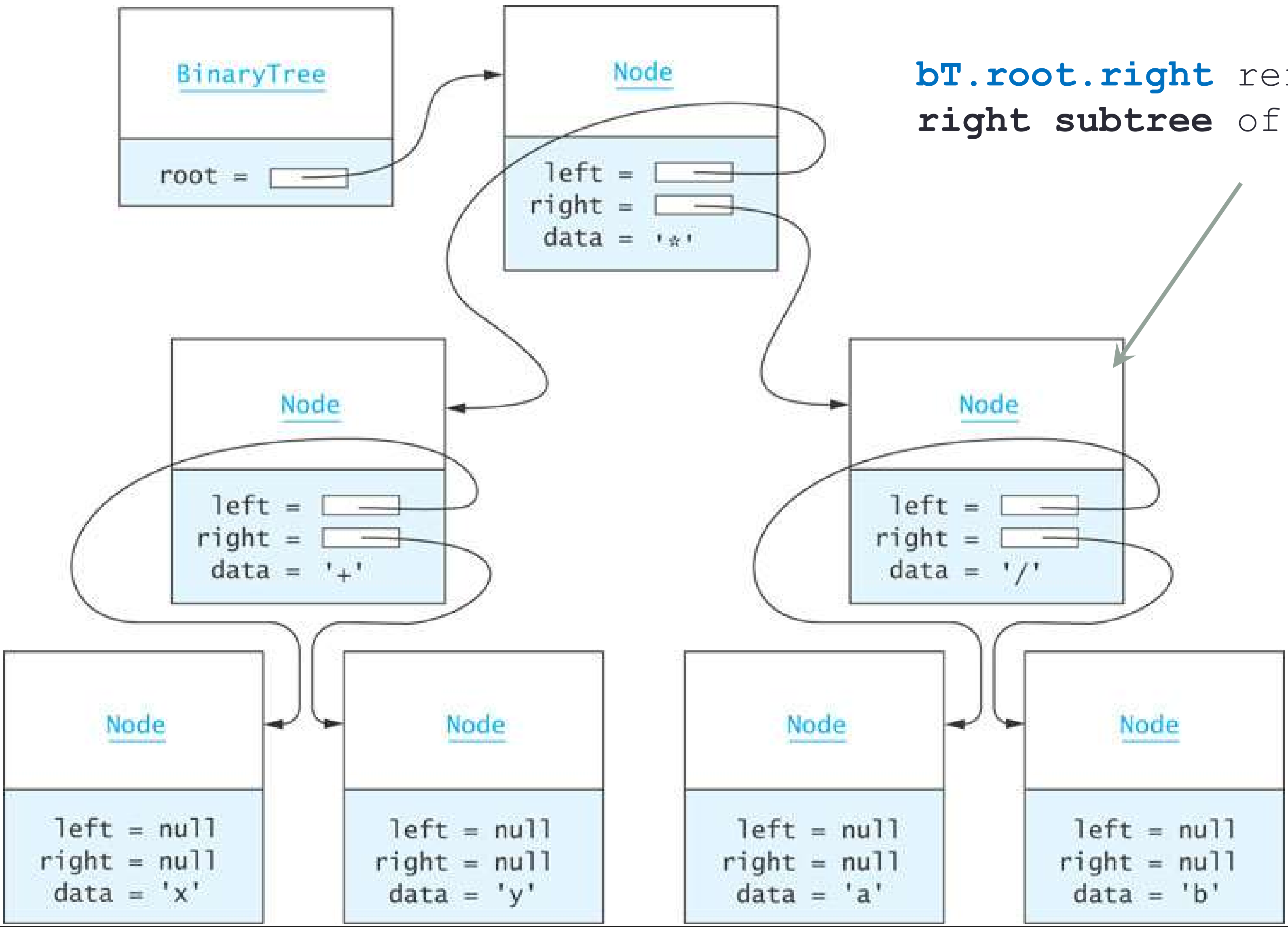


BinaryTree<E> Class (cont.)



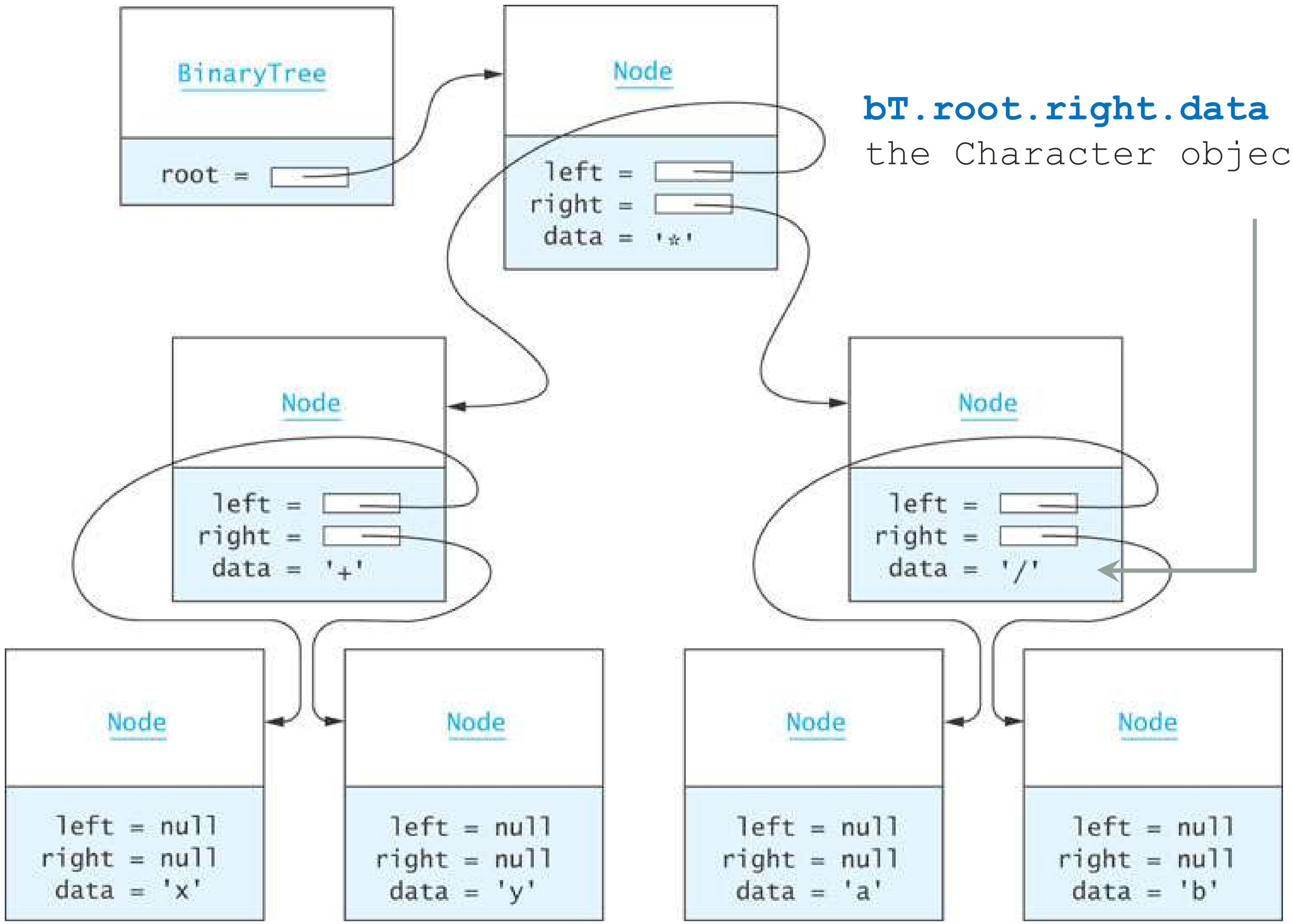
bT.root.left references the **left subtree** of the root

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 '/'

Design of `BinaryTree<E>` Class

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(BiConsumer<E, Integer> consumer)`	Performs a preorder traversal of the tree. Each node and its depth are passed to the `consumer` function. `BiConsumer` will be discussed in the next section.
`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> {
   // Insert inner class Node<E> here

   // The root of the tree
   protected Node<E> root;

   . . .
}
```

Constructors

- The no-parameter constructor creates a null tree.

```
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

```
/** Return the left subtree.
    @return The left subtree or null if either the root or
                   the left subtree is null
*/
public BinaryTree<E> getLeftSubtree() {
    if (root != null && root.left != null) {
        return new BinaryTree<E>(root.left);
    }   else {
        return null;
    }
}
```

- **`getRightSubtree`** method is **symmetric**

toString() Method

- Generates a **string** representing a **preorder traversal** in which **each local root** is **indented** a distance proportional to its **depth**.

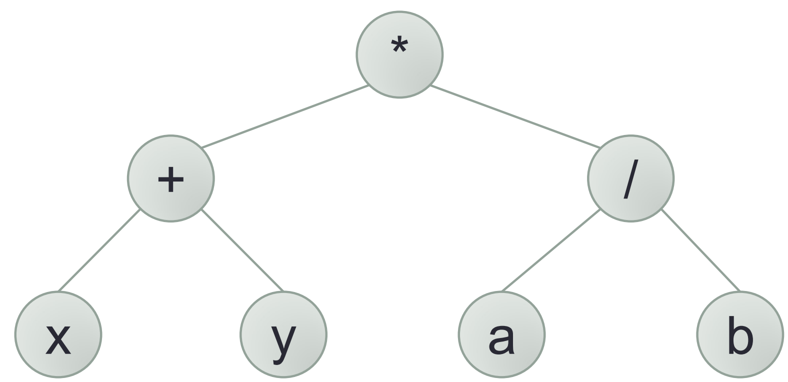
```
public String toString() {
    var sb = new StringBuilder();
    toString(root, 1, sb);  // call recursive toString
    return sb.toString();
}
```

Recursive **toString()**

```java
/** Converts a sub-tree to a string with a preorder traversal.
    @param node The local root
    @param depth The depth
    @param sb The StringBuilder to save the output
*/
private void toString(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");
           toString(node.left, depth + 1, sb);
           toString(node.right, depth + 1, sb);
      }
 }
```

Method `toString` Output

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



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

BINARY SEARCH TREES

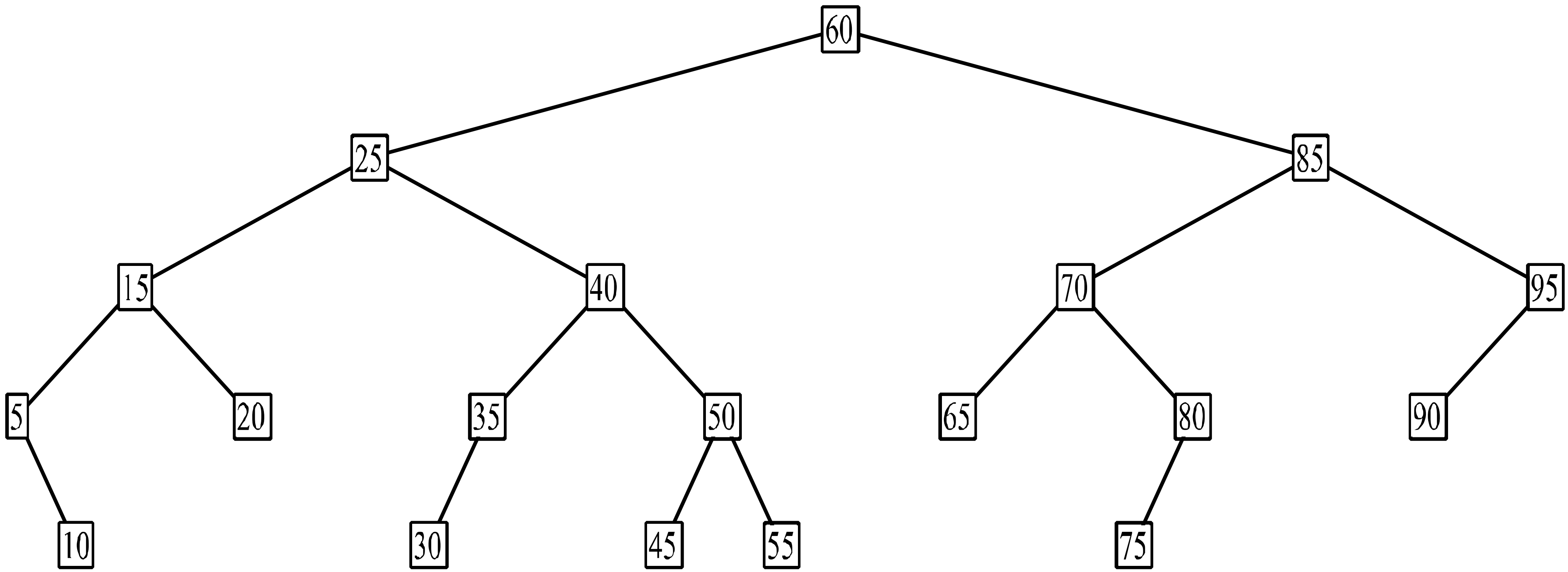
Overview of a Binary Search Tree

- A Binary Tree and all elements in the **left** subtree **precede** those in the **right** subtree
  - **Binary Tree** and **T_L < Middle < T_R**

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 is less than all values in T_R

Overview of a Binary Search Tree (cont.)



Recursive Algorithm for Searching a BST

1. `if` the root is `null` *Base case 1*
2.   the item is not in the tree; return `null`
3. Compare the value of `target` with `root.data`
4. `if` they are equal *Base case 2*
5.   the target has been found; return the data at the root

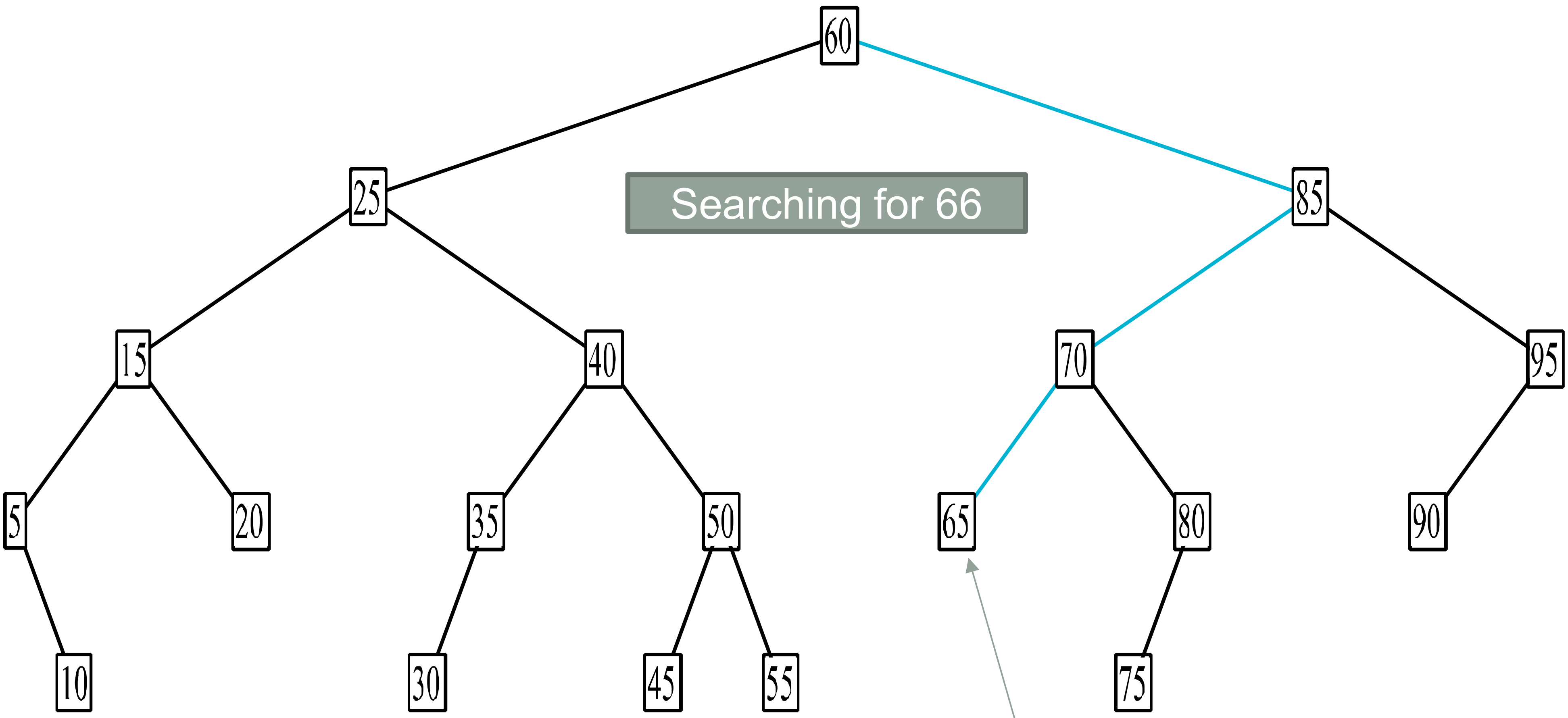
   `else if` the target is less than `root.data`

6.   return the result of **searching** the **left subtree** *Recursive case 1*

   `else`

7.   return the result of **searching** the **right subtree** *Recursive case 2*

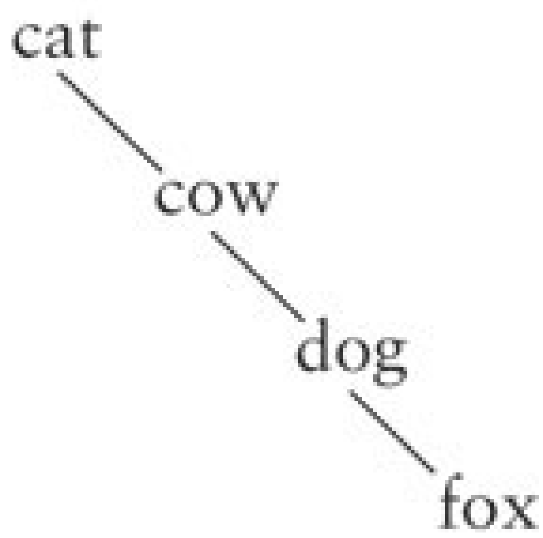
Searching for 66 in a Binary Search Tree



root of right subtree is **null**—**66** is **not in** the **tree**

Performance

- Searching 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*)**

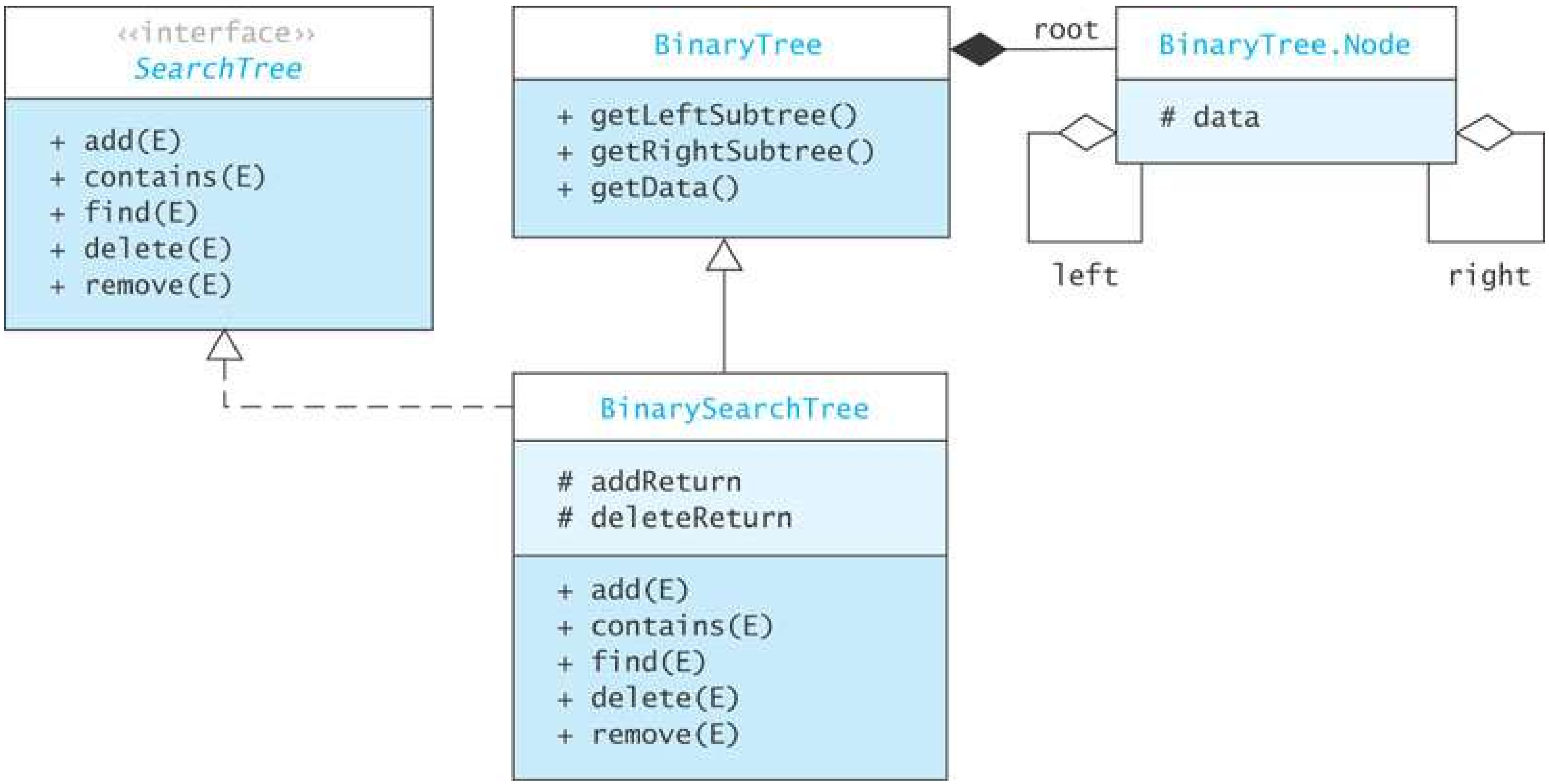


SearchTree<E> Interface

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`**.

`BinarySearchTree<E>` Class

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.



Recursive `find()` Methods

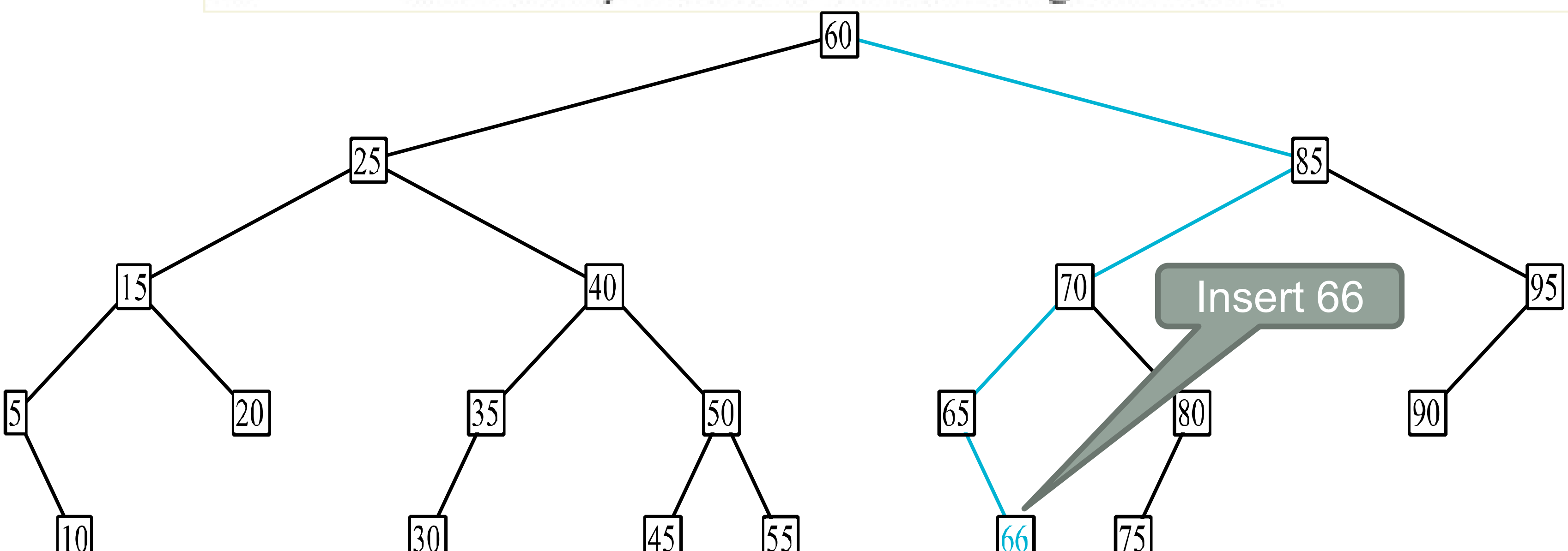
BinarySearchTree find Method

```
/** 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);
}
```

Recursive Insertion into a BST

Recursive Algorithm for Insertion in a Binary Search Tree

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



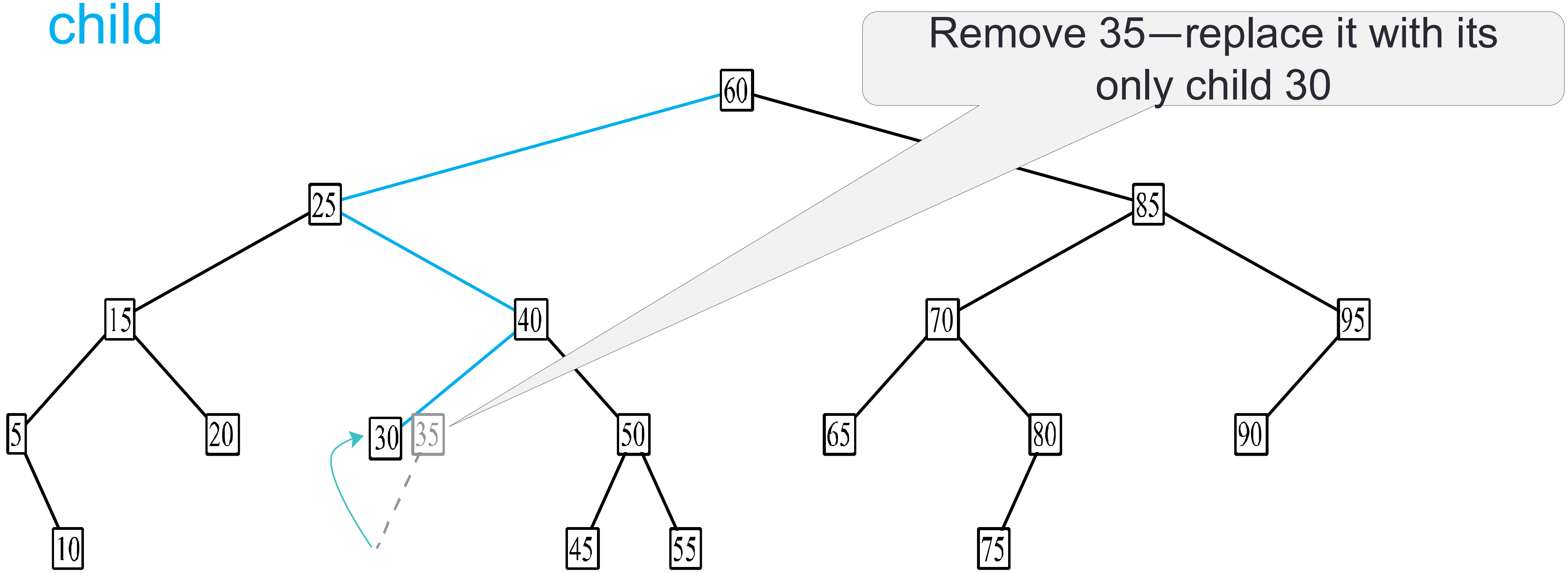
Recursive `add()`

```
/** 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;
}
```

```java
/** 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) {
      if (localRoot == null) {
         // item is not in the tree - insert it.
         addReturn = true;
         return new Node<>(item);
      } else if (item.compareTo(localRoot.data) == 0) {
         addReturn = false;
         return localRoot;
      } else if (item.compareTo(localRoot.data) < 0) {
         // item is less than localRoot.data
         localRoot.left = add(localRoot.left, item);
         return localRoot;
      } else {
         // item is greater than localRoot.data
         localRoot.right = add(localRoot.right, item);
         return localRoot;}
   }
```

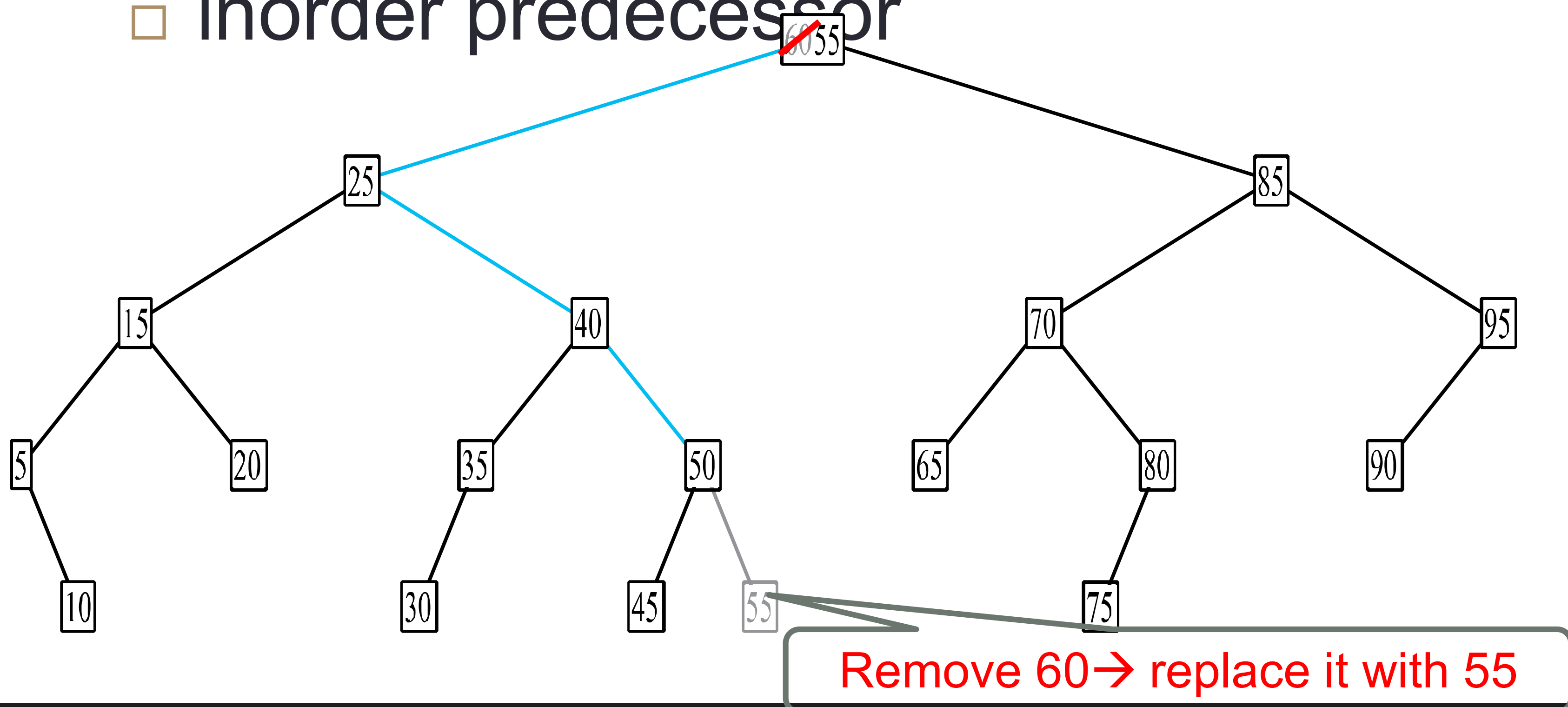
Removal from a BST

- If the item to be removed has no children, simply delete the reference to the item
- If the item to be removed has only one child, change the reference to the item so that it references the item's only child



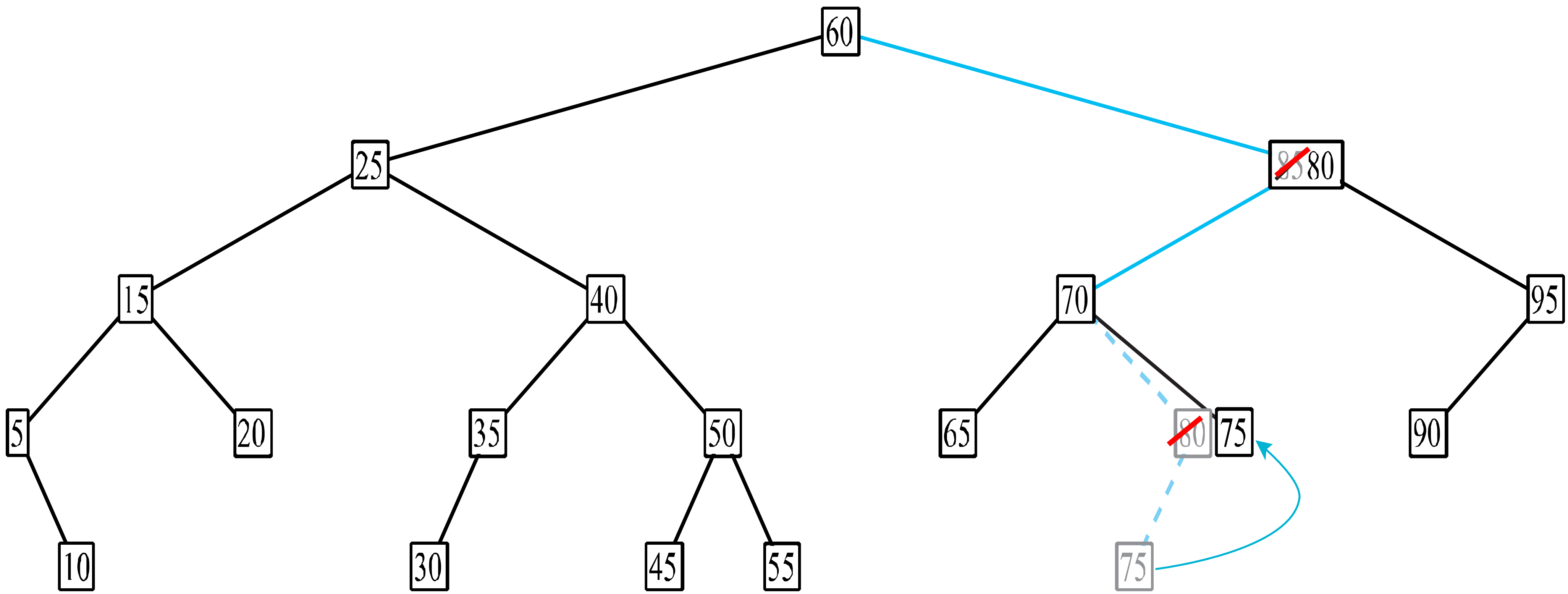
Removal from a BST

- If the item to be removed has 2 children, replace it with the largest item in its left subtree
  - inorder predecessor



Removal from a BST

- Remove 85
  - Its inorder predecessor 80 has a child 75. Replace 85 with 80 and reset the right subtree of 80's parent to reference 75.



Algorithm for Binary Search Tree Removal

```
if the root is null
    The item is not in tree—return null.
Compare the item to the data at the local root.
if the item is less than the data at the local root
    Return the result of deleting from the left subtree.
else if the item is greater than the local root
    Return the result of deleting from the right subtree.
else // The item is in the local root
    Store the data in the local root in deleteReturn.
    if the local root has no children
        Set the parent of the local root to reference null.
    else if the local root has one child
        Set the parent of the local root to reference that child.
    else // Find the inorder predecessor
        if the left child has no right child it is the inorder predecessor
            Set the parent of the local root to reference the left child.
        else
            Find the rightmost node in the right subtree of the left child.
            Copy its data into the local root's data—remove it by setting its
                parent to reference its left child.
```

Method `findLargestChild()`

BinarySearchTree findLargestChild Method

```
/** Find the node that is the
    inorder predecessor and replace it
    with its left child (if any).
    post: The inorder predecessor is removed from the tree.
    @param parent The parent of possible inorder
                  predecessor (ip)
    @return The data in the ip
*/
private E findLargestChild(Node<E> parent) {
    // If the right child has no right child, it is
    // the inorder predecessor.
    if (parent.right.right == null) {
        E returnValue = parent.right.data;
        parent.right = parent.right.left;
        return returnValue;
    } else {
        return findLargestChild(parent.right);
    }
}
```

Testing a Binary Search Tree

- To test a binary search tree, verify that an inorder traversal will display the tree contents in ascending order after a series of insertions and deletions are performed