

TREES

Chapter 6

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 differences 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

Chapter Objectives (cont.)

- To learn how to use a binary search tree to store information so that it can be retrieved in an efficient manner
- To learn how to use a Huffman tree to encode characters using fewer bytes than ASCII or Unicode, resulting in smaller files and reduced storage requirements

Trees - Introduction

- All previous data organizations we've studied are linear—each element can have only one predecessor and 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 by linked data structures
- Searching a binary search tree, an ordered tree, is generally more efficient than searching an ordered list— $O(\log n)$ versus $O(n)$

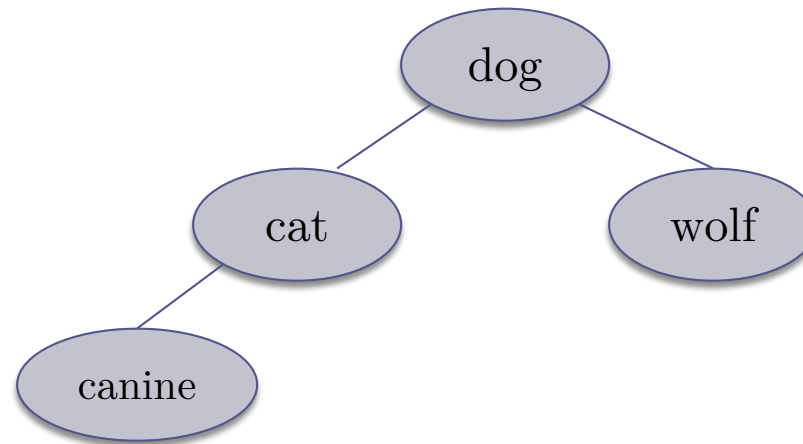


Tree Terminology and Applications

Section 6.1

Tree Terminology

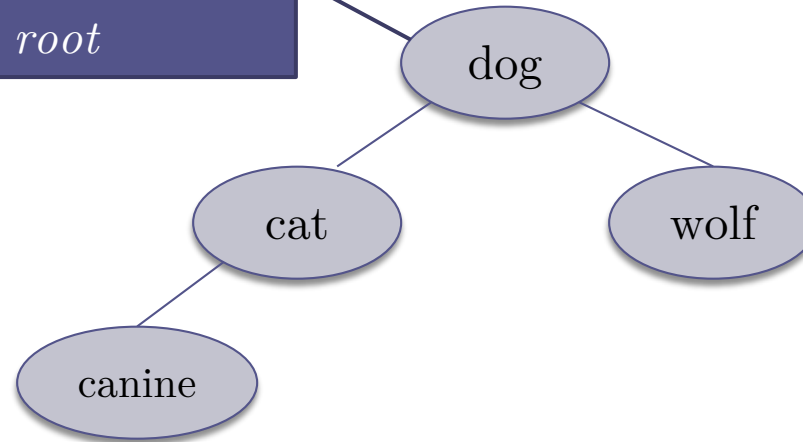
A tree consists of a collection of elements or nodes, with each node linked to its successors



Tree Terminology (cont.)

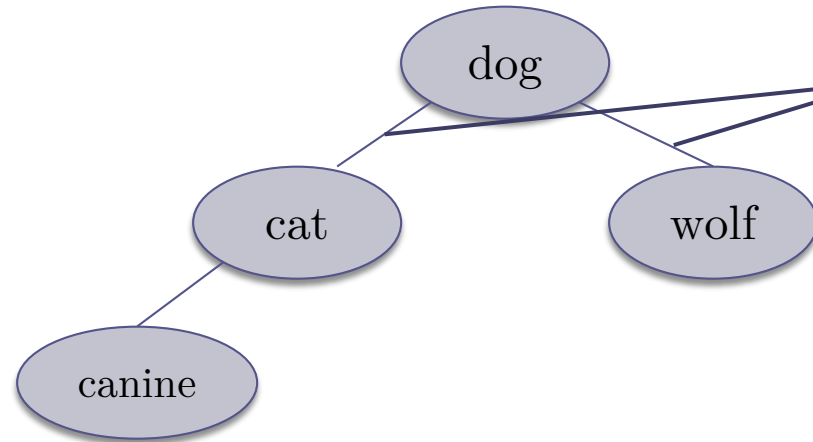
A tree consists of a collection of elements or nodes, with each node linked to its successors

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



Tree Terminology (cont.)

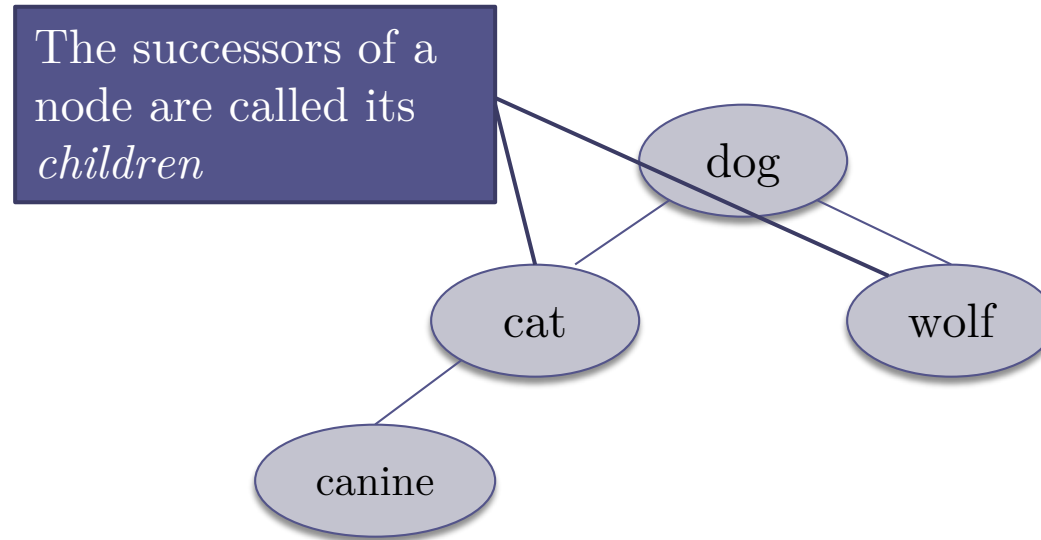
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

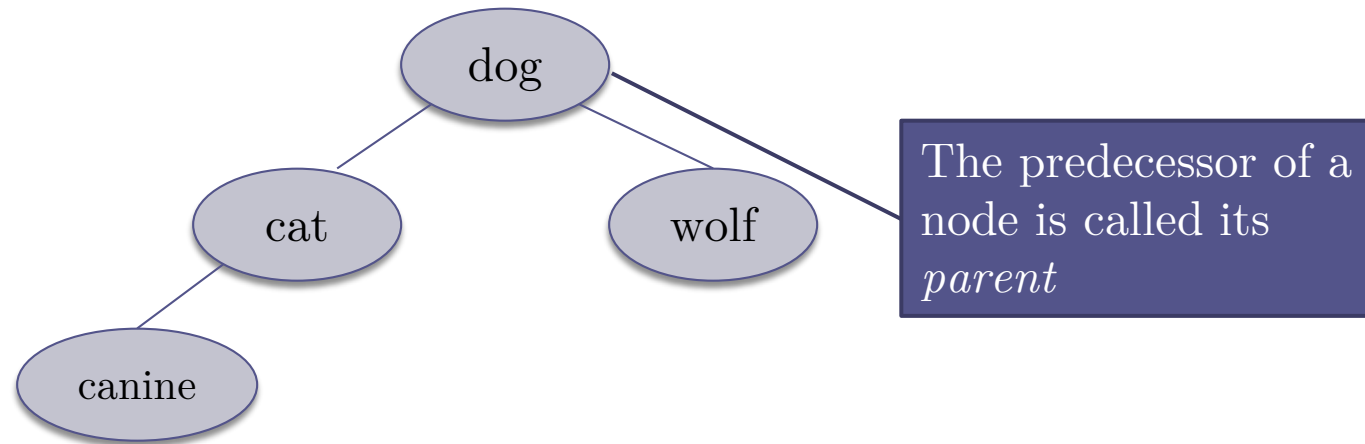
Tree Terminology (cont.)

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



Tree Terminology (cont.)

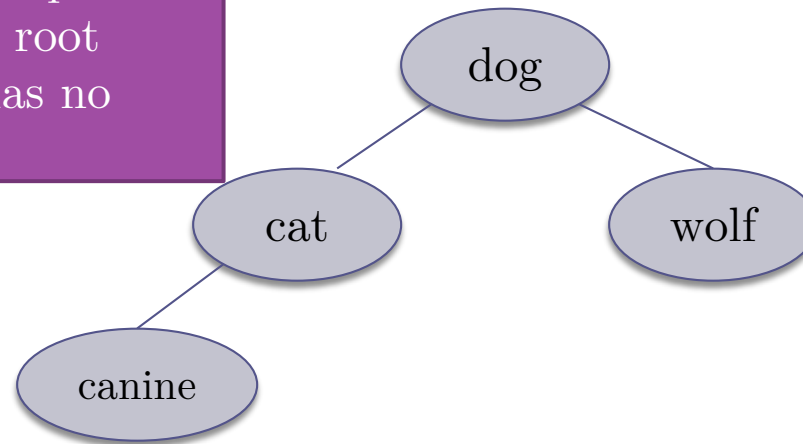
A tree consists of a collection of elements or nodes, with each node linked to its successors



Tree Terminology (cont.)

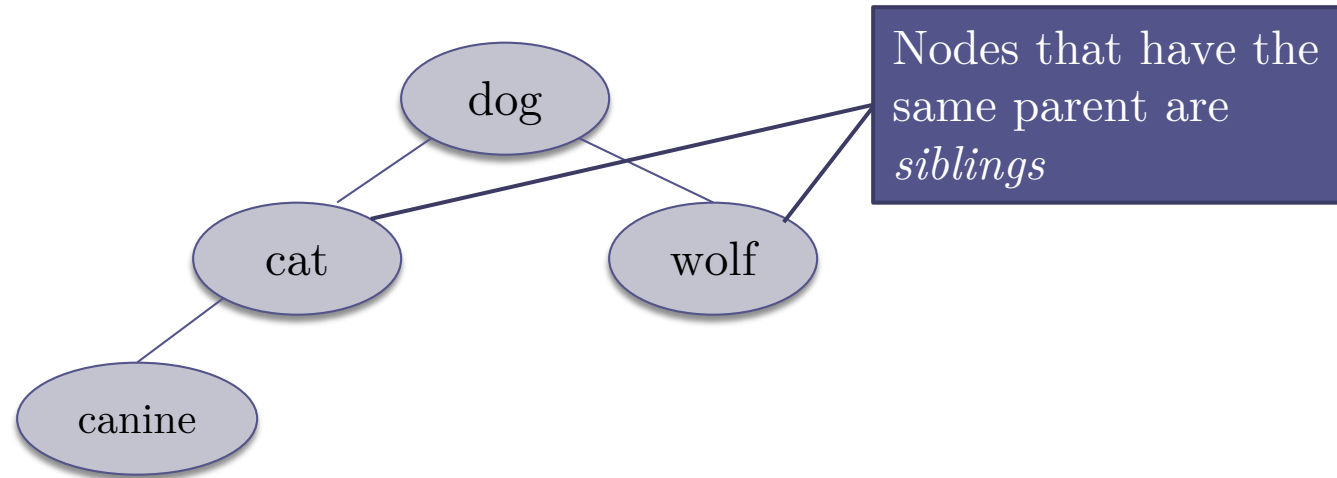
A tree consists of a collection of elements or nodes, with each node linked to its successors

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



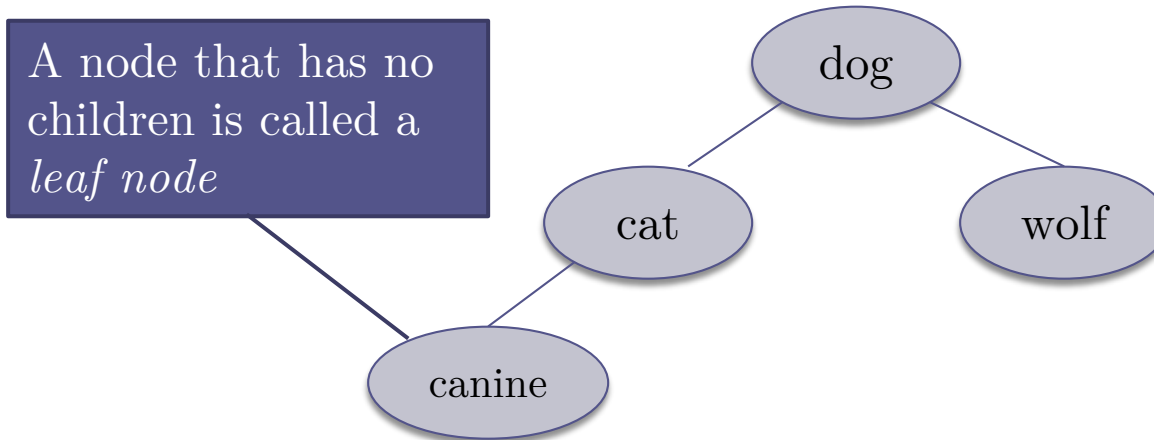
Tree Terminology (cont.)

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



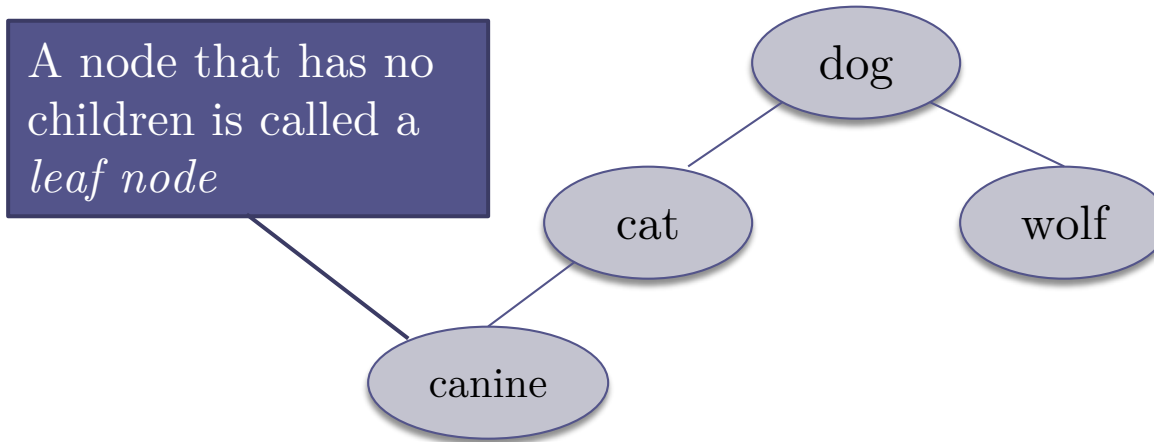
Tree Terminology (cont.)

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



Tree Terminology (cont.)

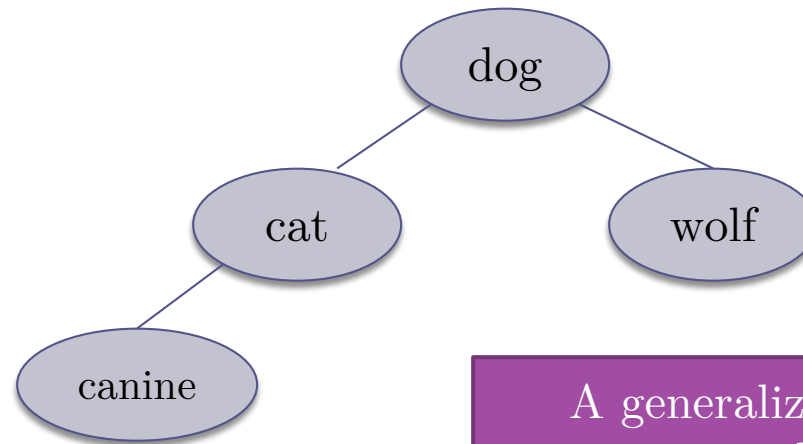
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

Tree Terminology (cont.)

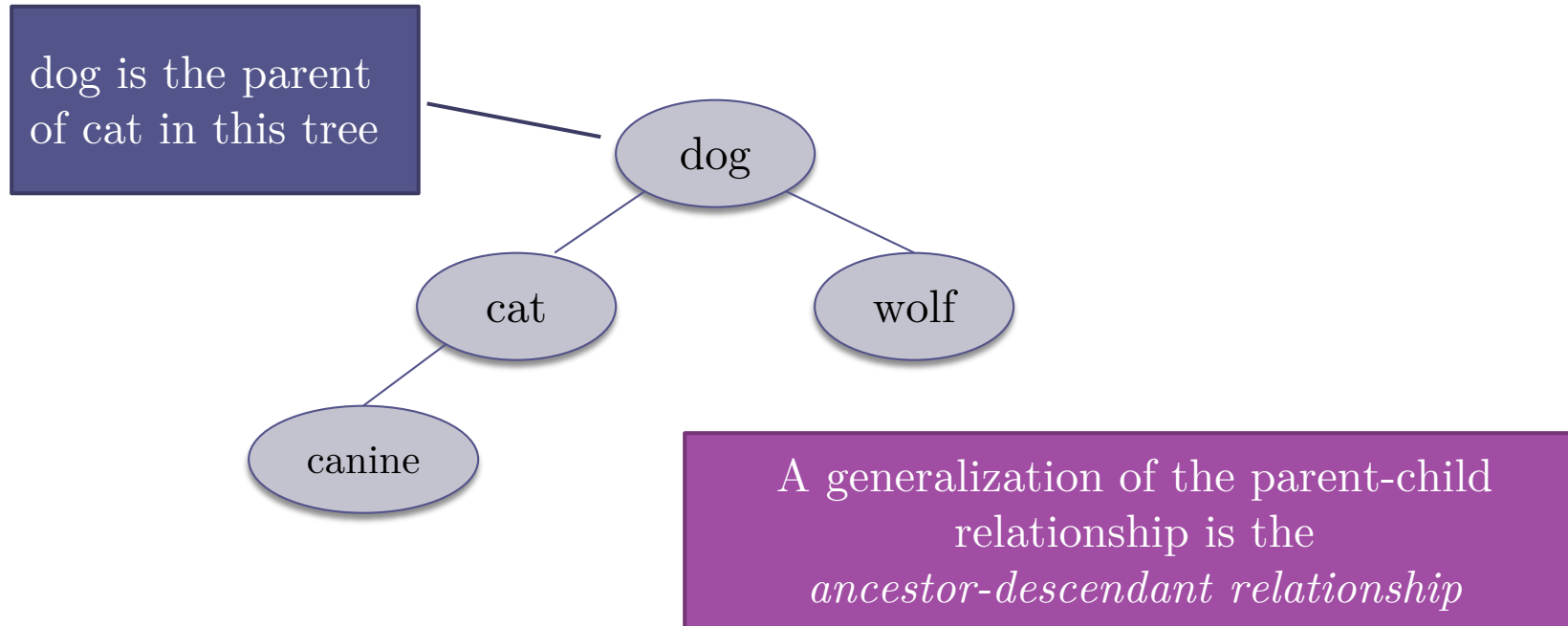
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

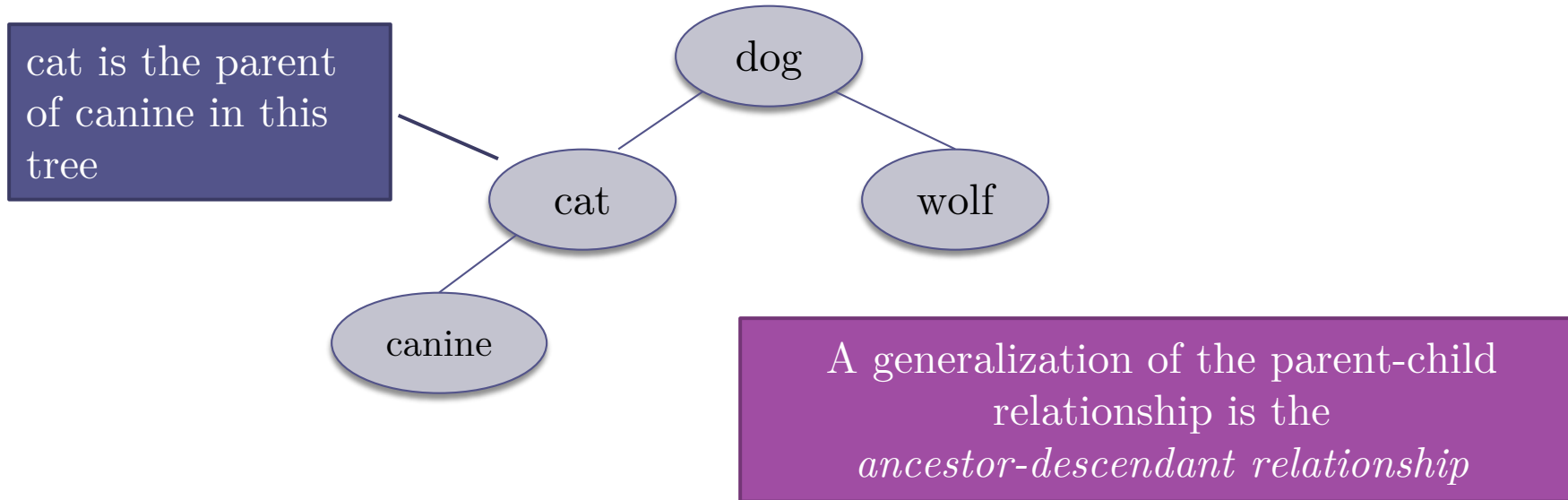
Tree Terminology (cont.)

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



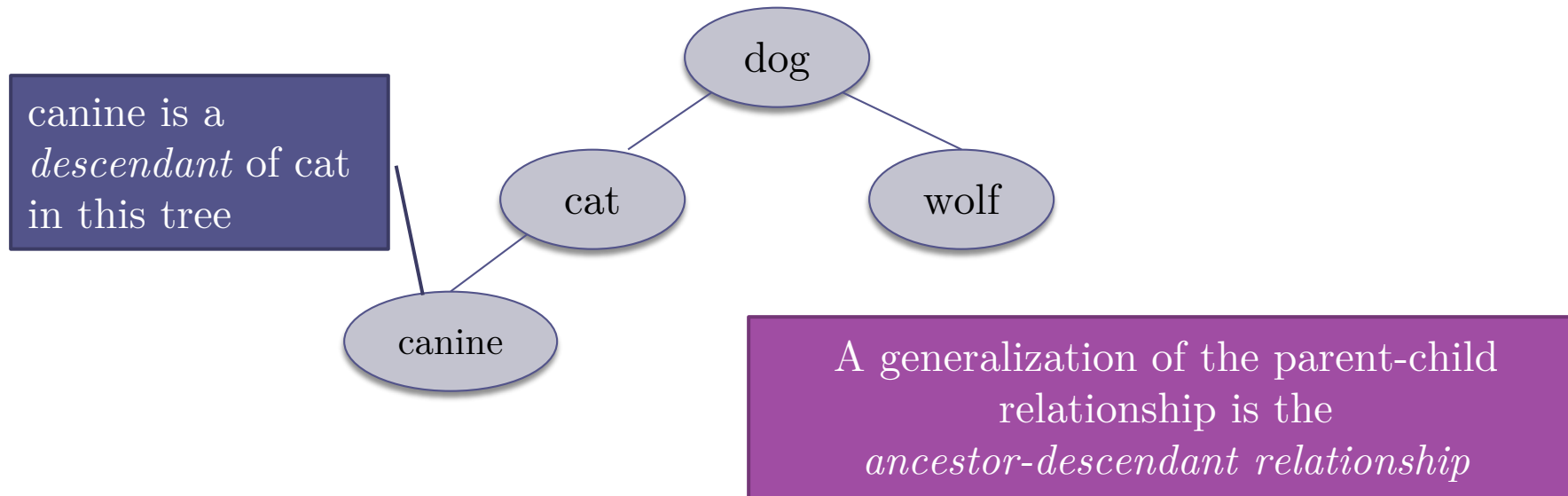
Tree Terminology (cont.)

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



Tree Terminology (cont.)

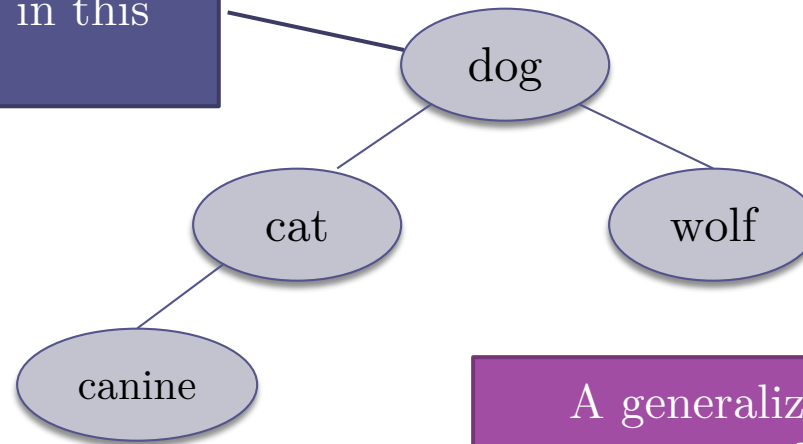
A tree consists of a collection of elements or nodes, with each node linked to its successors



Tree Terminology (cont.)

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

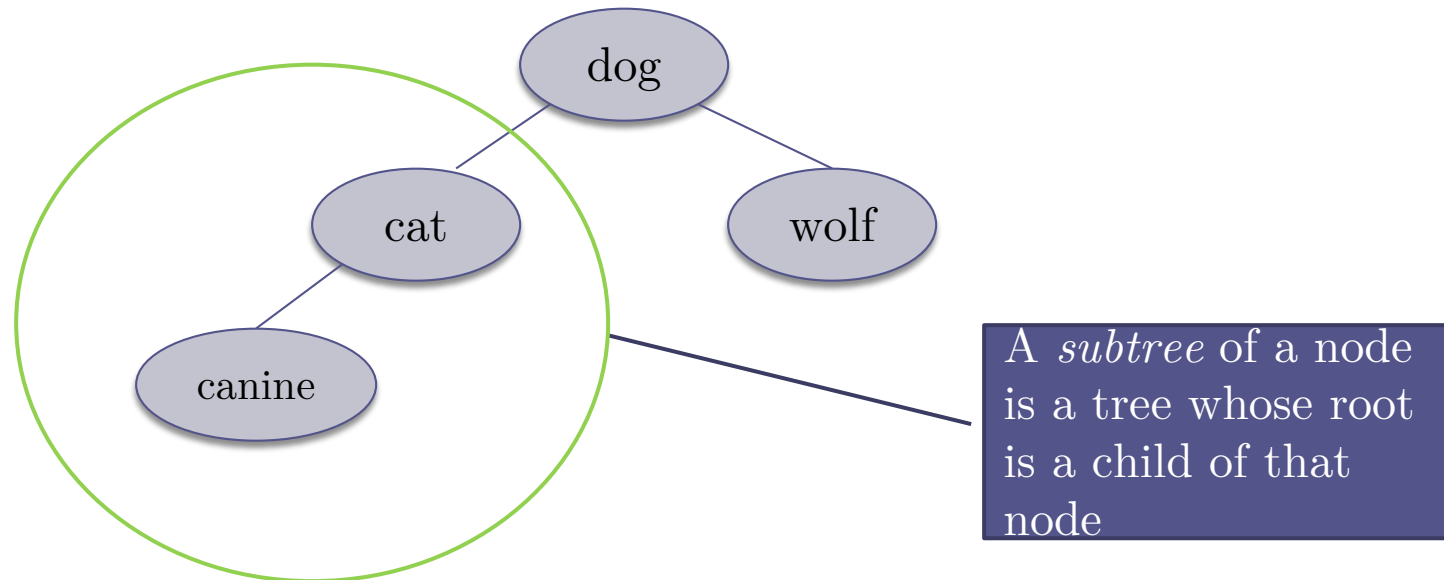
dog is an *ancestor*
of canine in this
tree



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

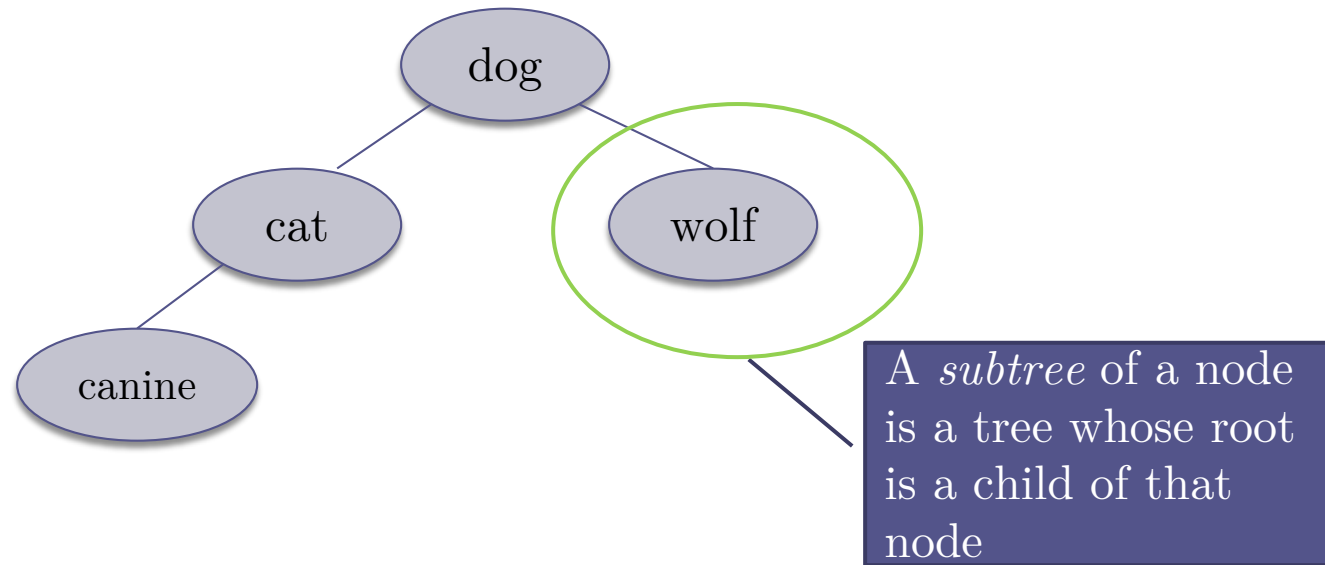
Tree Terminology (cont.)

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



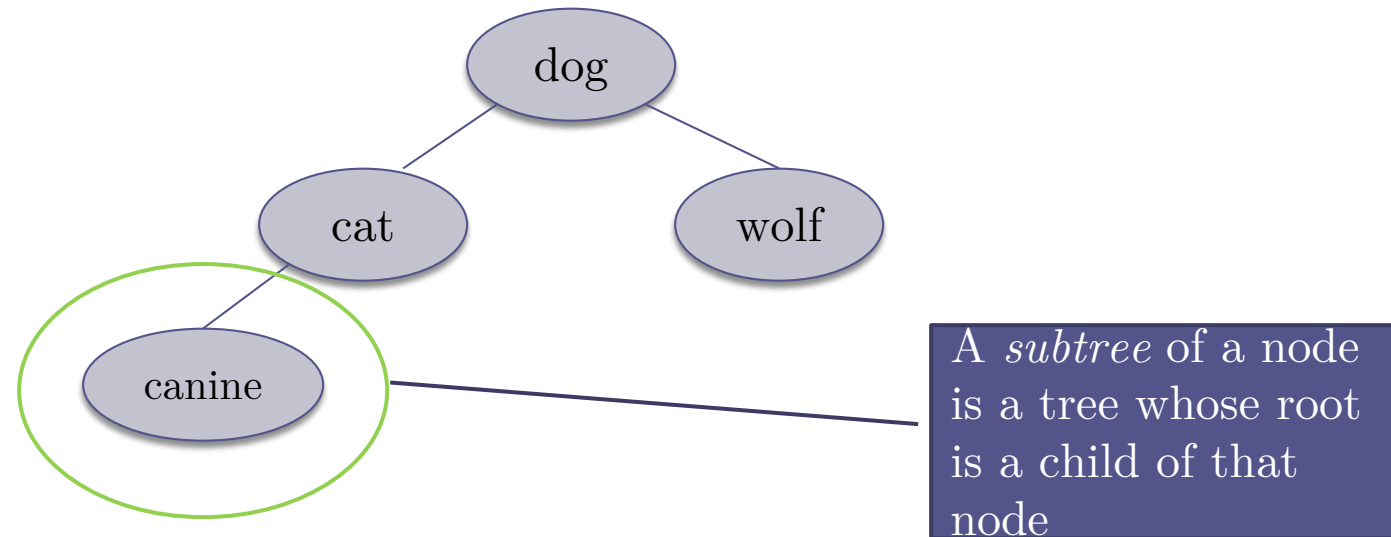
Tree Terminology (cont.)

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



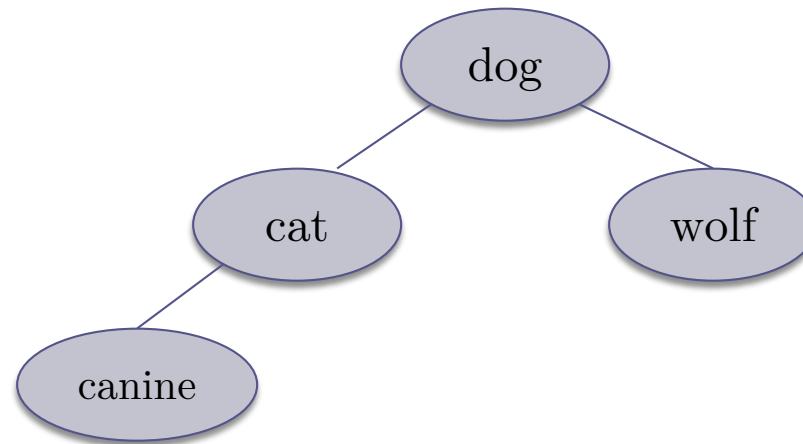
Tree Terminology (cont.)

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



Tree Terminology (cont.)

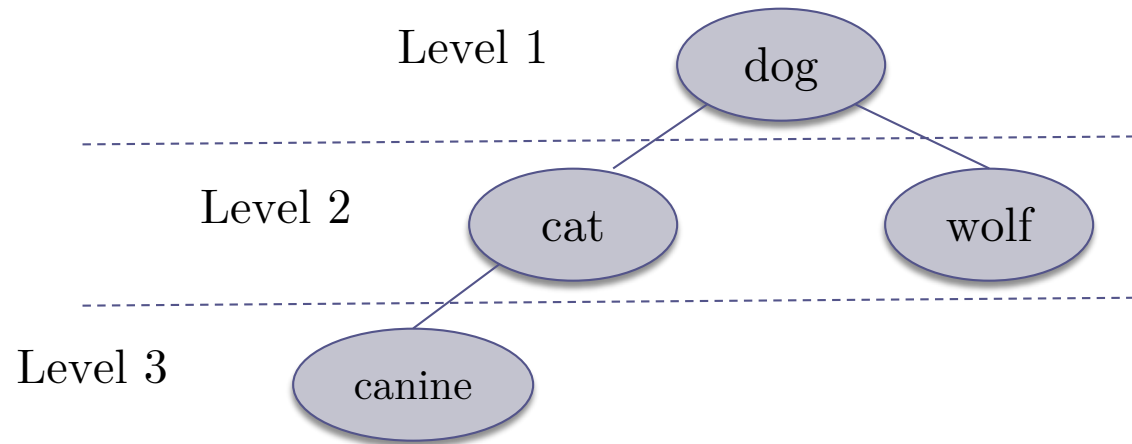
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

Tree Terminology (cont.)

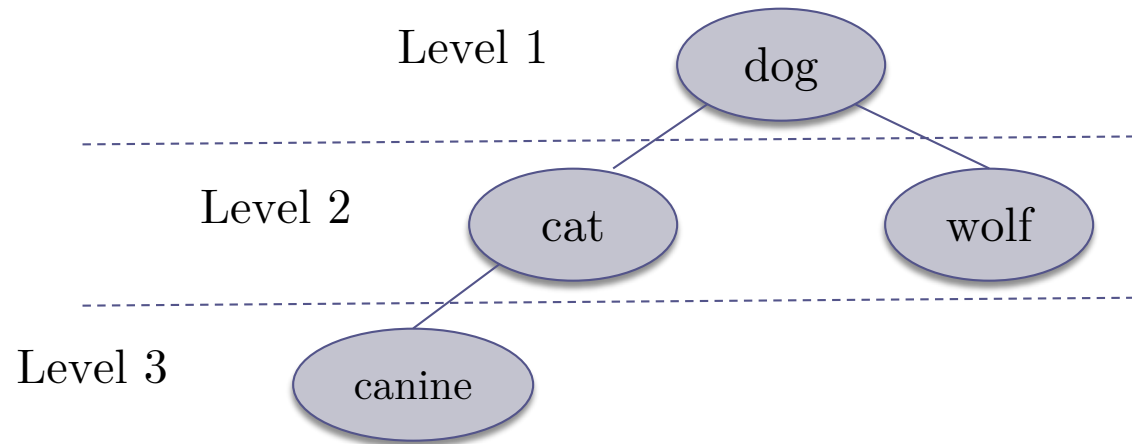
A tree consists of a collection of elements or nodes, with each node linked to its successors



The level of a node is its distance from the root plus 1

Tree Terminology (cont.)

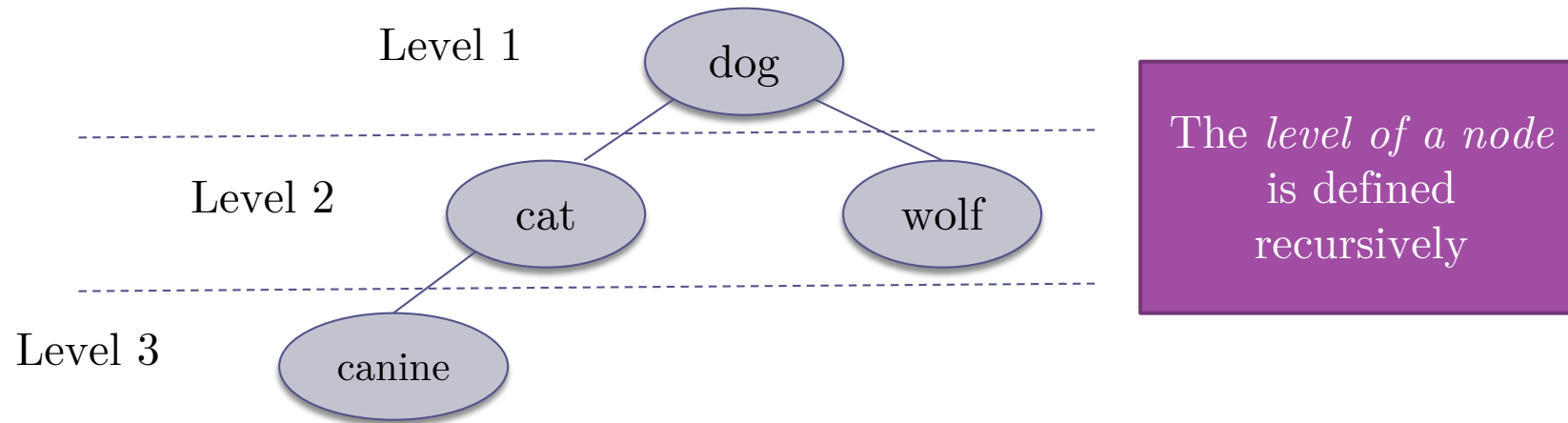
A tree consists of a collection of elements or nodes, with each node linked to its successors



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

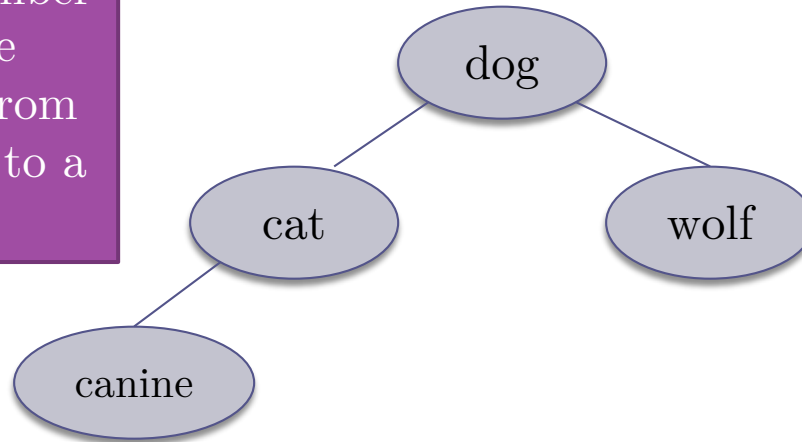


- If node n is the root of tree T , its level is 1
- If node n is not the root of tree T , its level is $1 +$ the level of its parent

Tree Terminology (cont.)

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

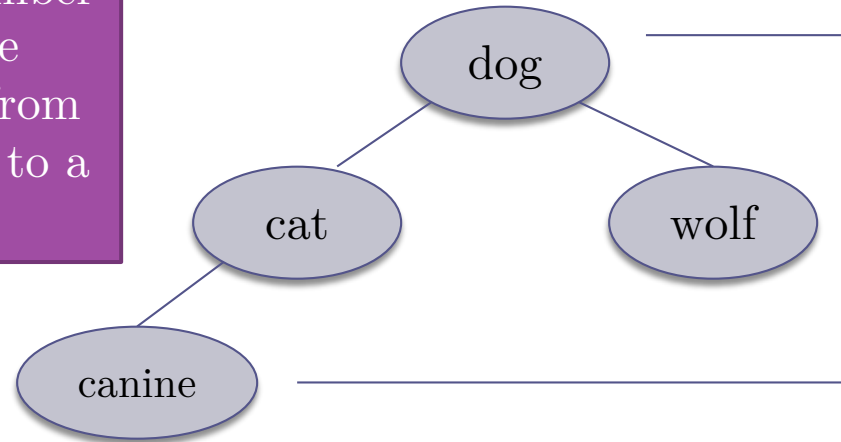
The *height* of a tree is the number of nodes in the longest path from the root node to a leaf node



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



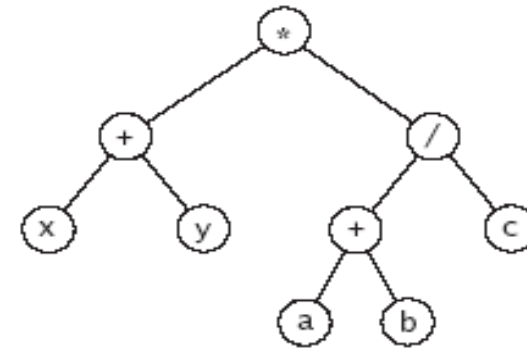
The height of this tree is 3

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
- ❑ Parentheses are not stored in the tree because the 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

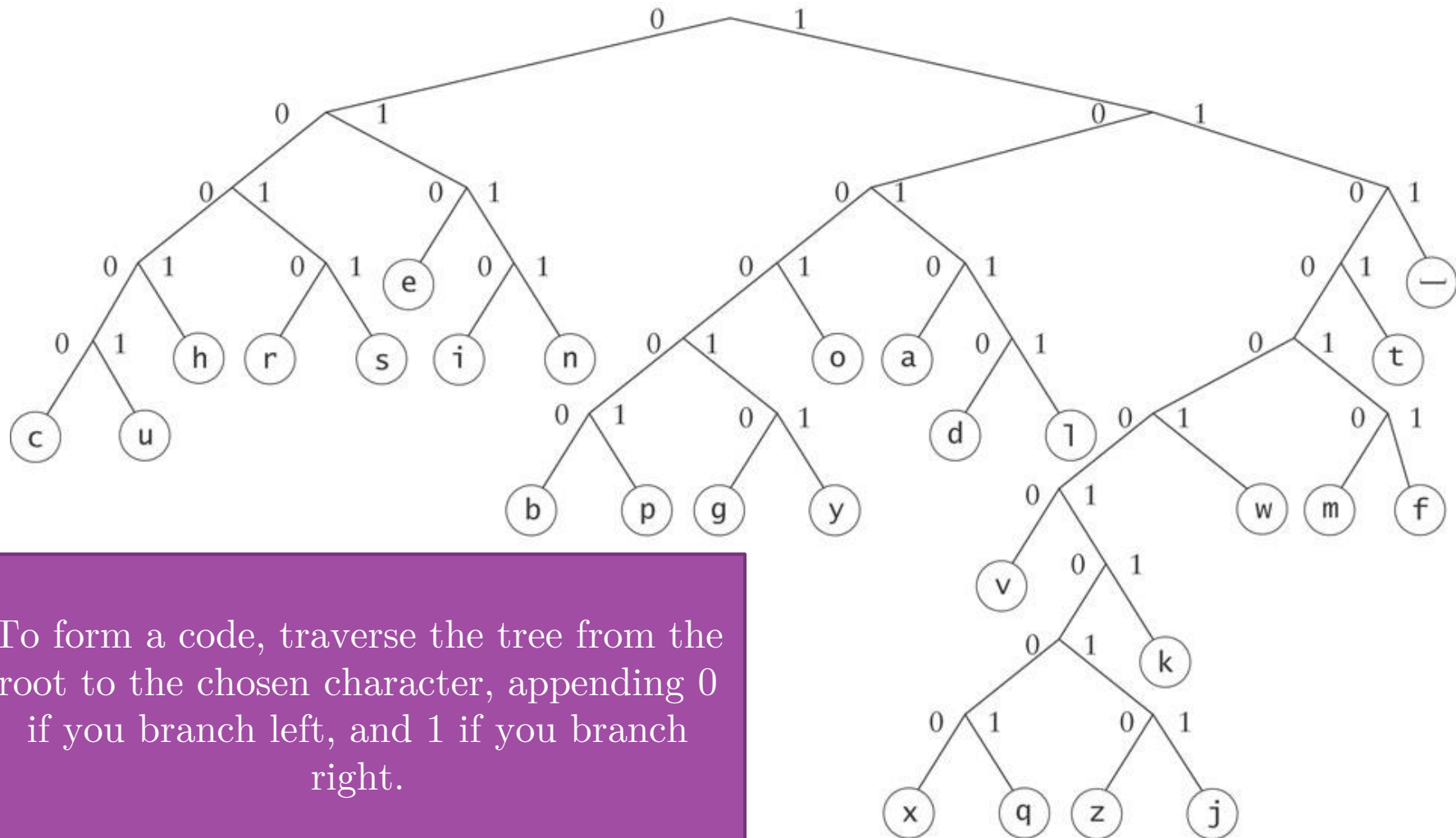


$(x + y) * ((a + b) / c)$

Huffman Tree

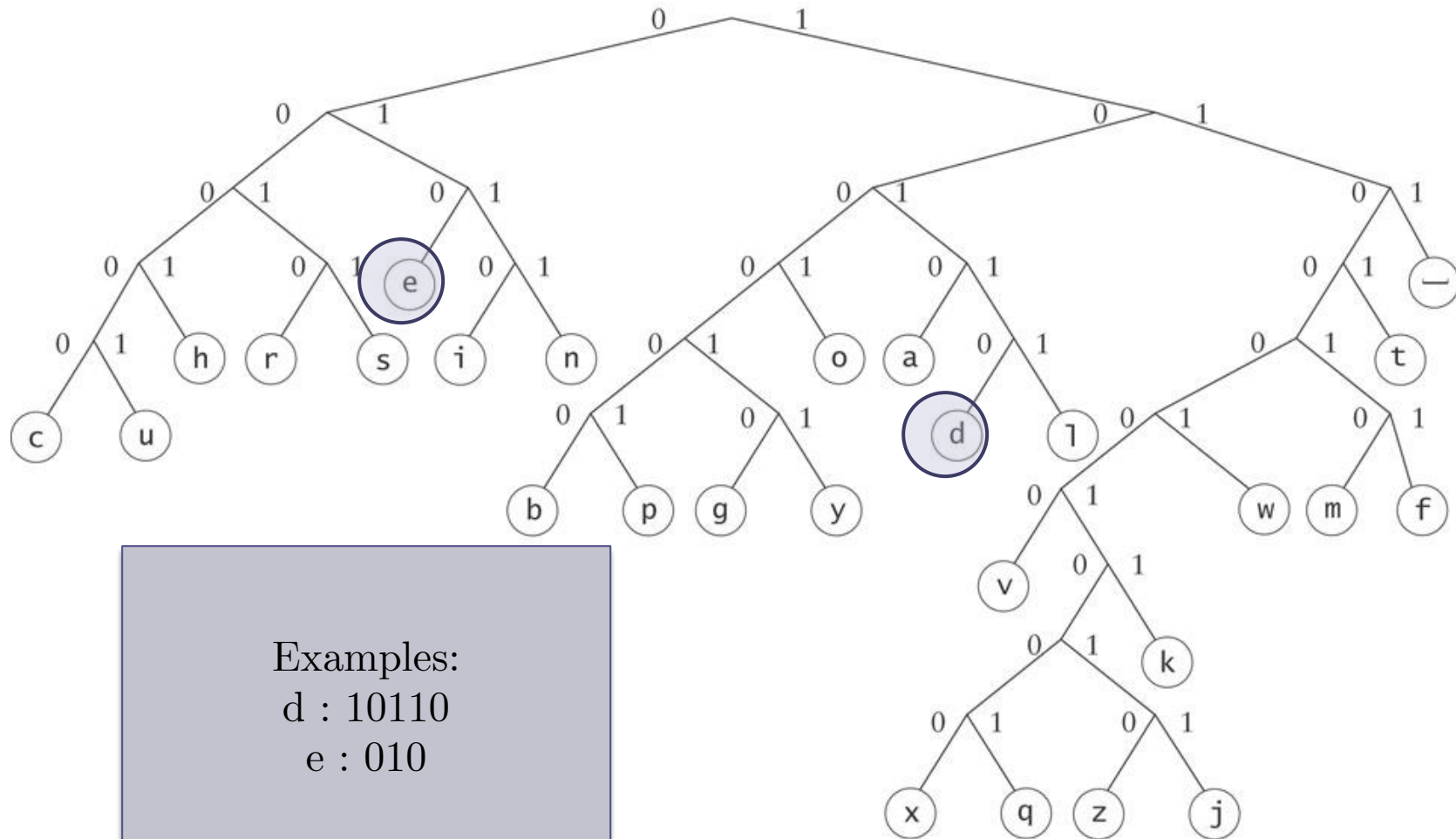
- A *Huffman tree* represents *Huffman codes* for characters that might appear in a text file
- As opposed to ASCII or Unicode, Huffman code uses different numbers of bits to encode letters; more common characters use fewer bits
- Many programs that compress files use Huffman codes

Huffman Tree (cont.)



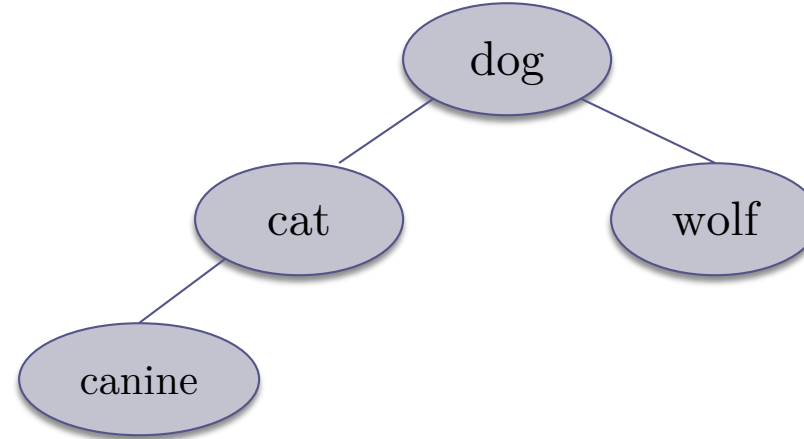
To form a code, traverse the tree from the root to the chosen character, appending 0 if you branch left, and 1 if you branch right.

Huffman Tree (cont.)



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 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 (cont.)

- A binary search tree never has to be sorted because its elements always satisfy the required order relationships
- When new elements are inserted (or removed) properly, the binary search tree 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 $O(n)$

Binary Search Tree (cont.)

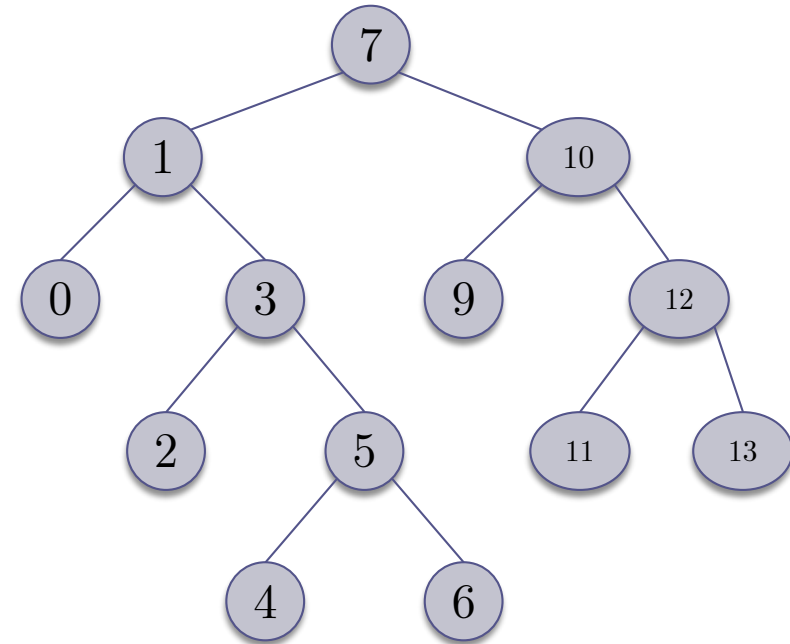
- When searching a BST, each probe has the potential to eliminate half the elements in the tree, so searching can be $O(\log n)$
- In the worst case, searching is $O(n)$

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

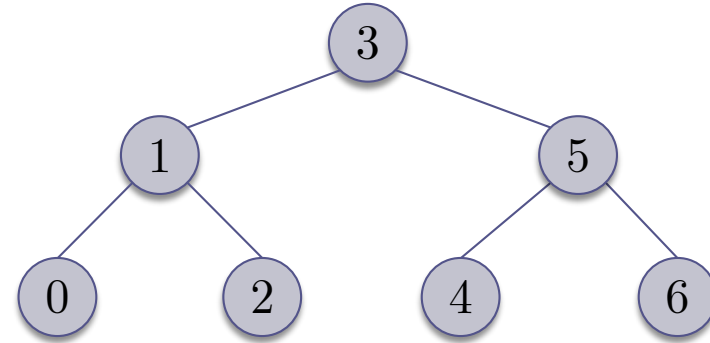
- A full binary tree is a binary tree where all nodes have either 2 children or 0 children (the leaf nodes)



Full, Perfect, and Complete Binary Trees

(cont.)

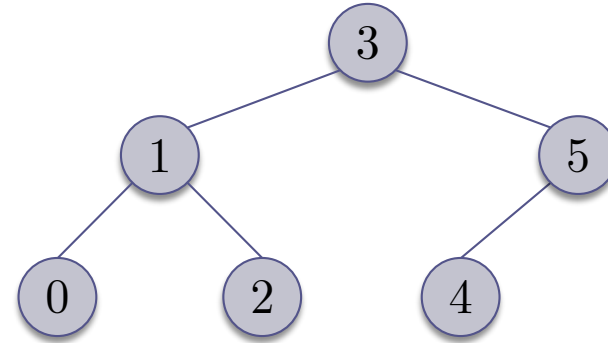
- A *perfect binary tree* is a full binary tree of height n with exactly $2^n - 1$ nodes
- In this case, $n = 3$ and $2^n - 1 = 7$



Full, Perfect, and Complete Binary Trees

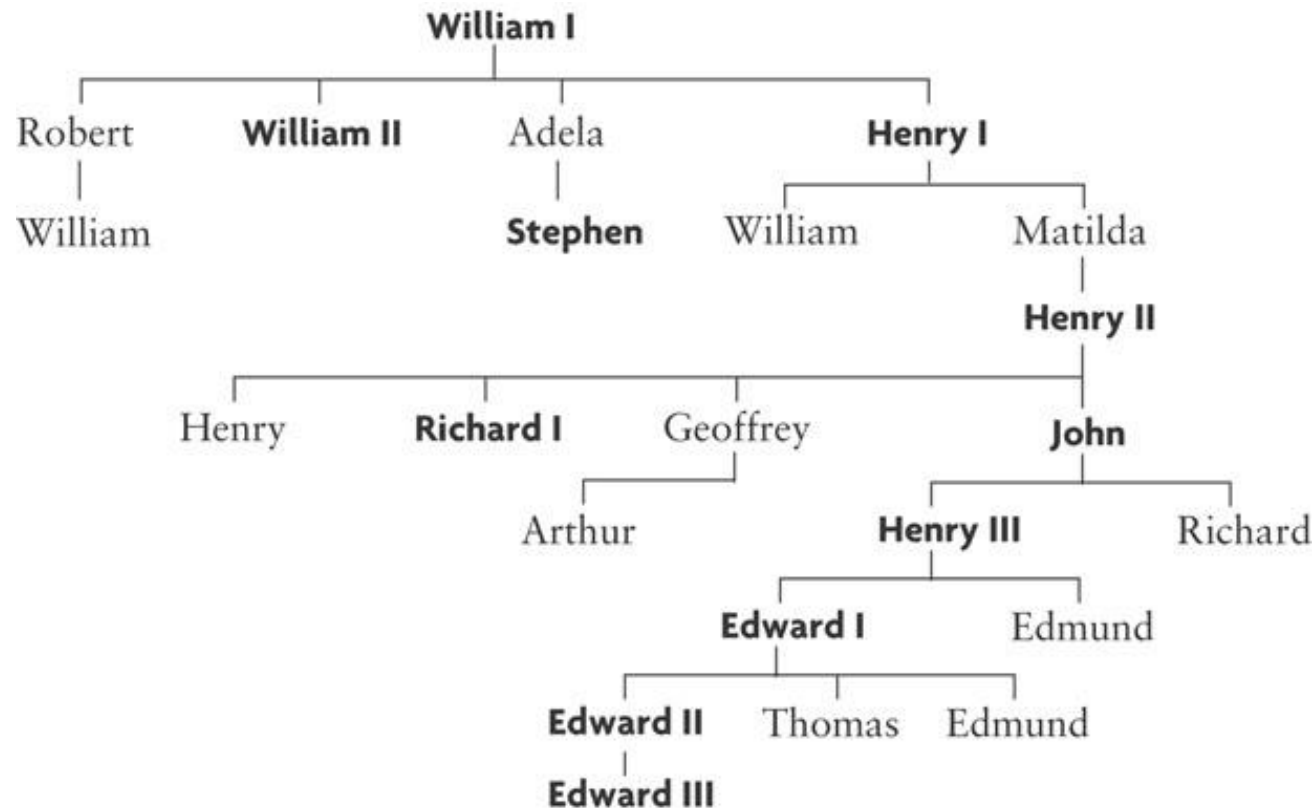
(cont.)

- 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



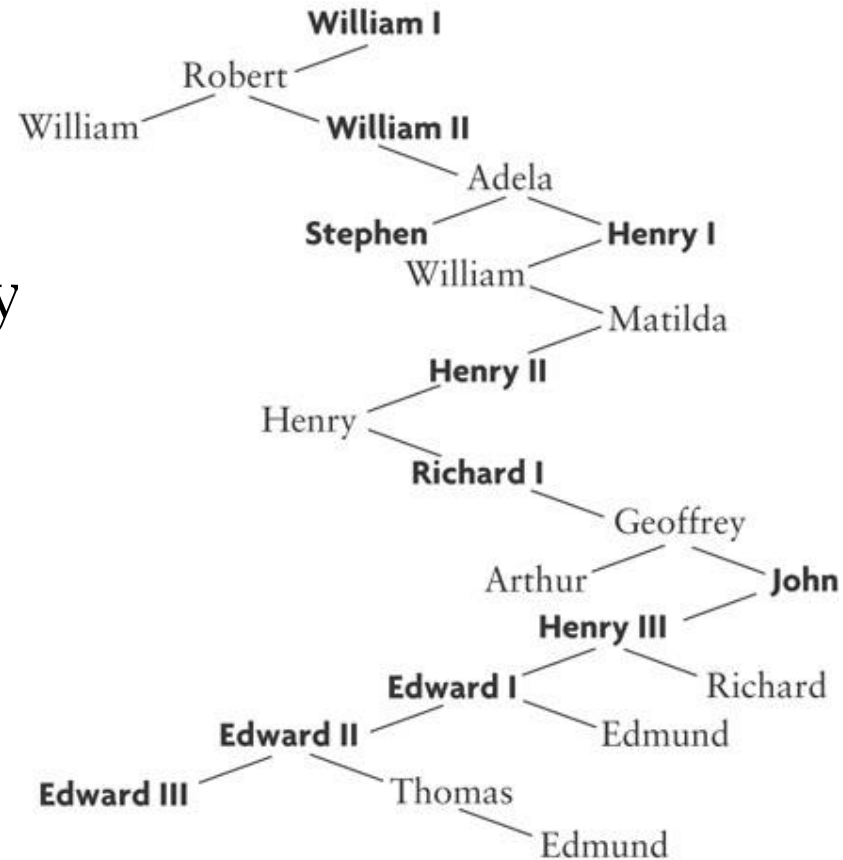
General Trees

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



General Trees (cont.)

- A general tree can be represented using 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

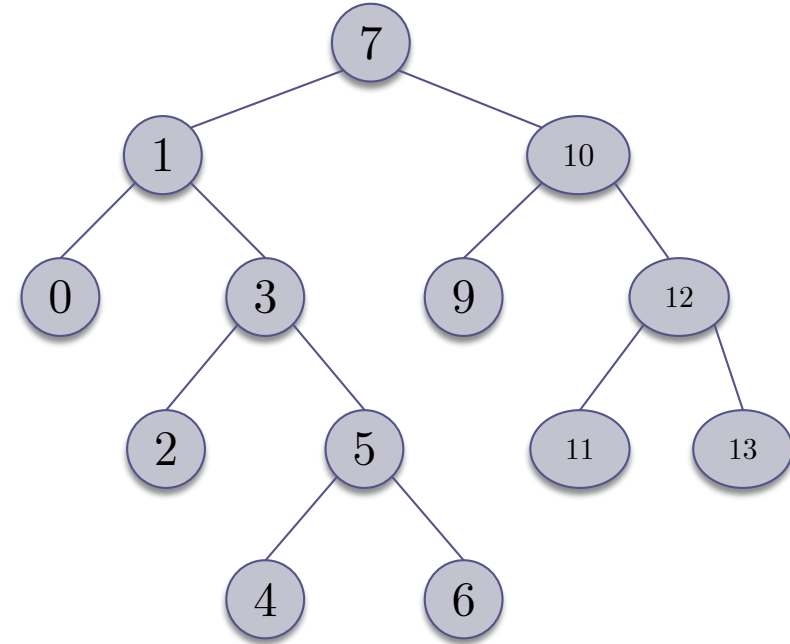
Section 6.2

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

- 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

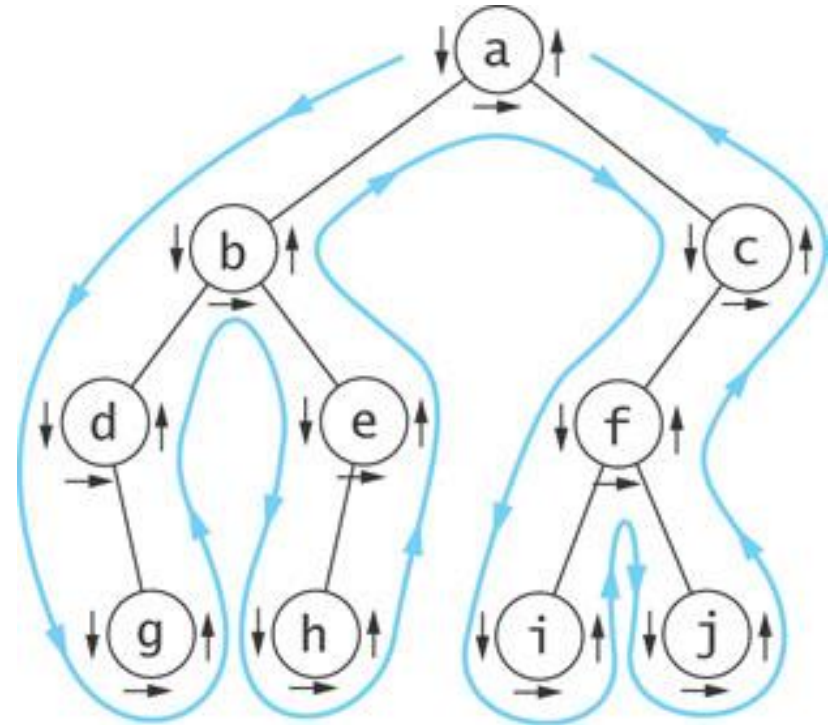
1. if the tree is empty
2. Return.
- else
3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.

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.

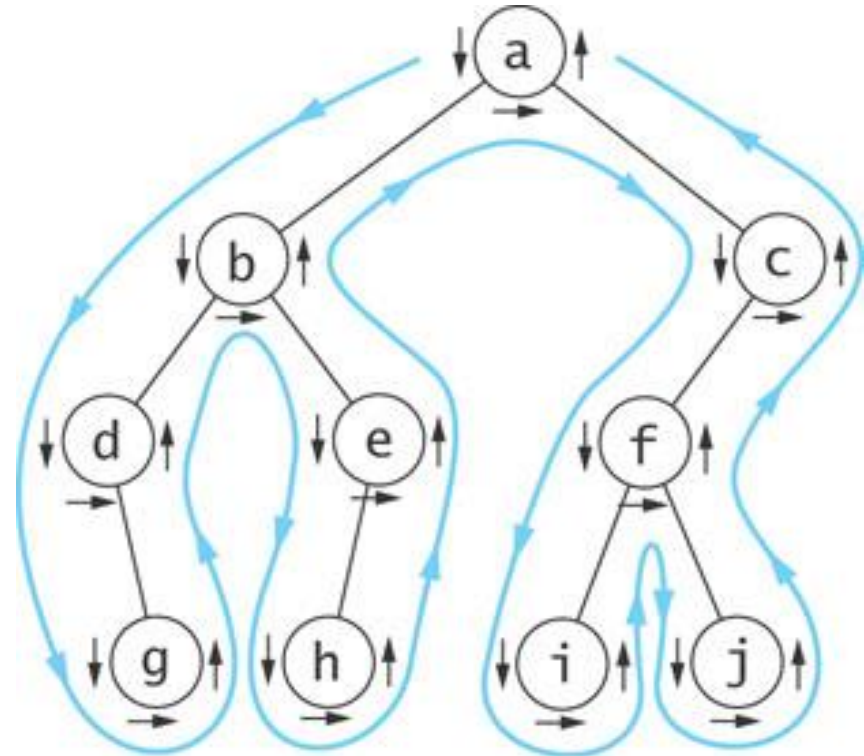
Visualizing Tree Traversals

- 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



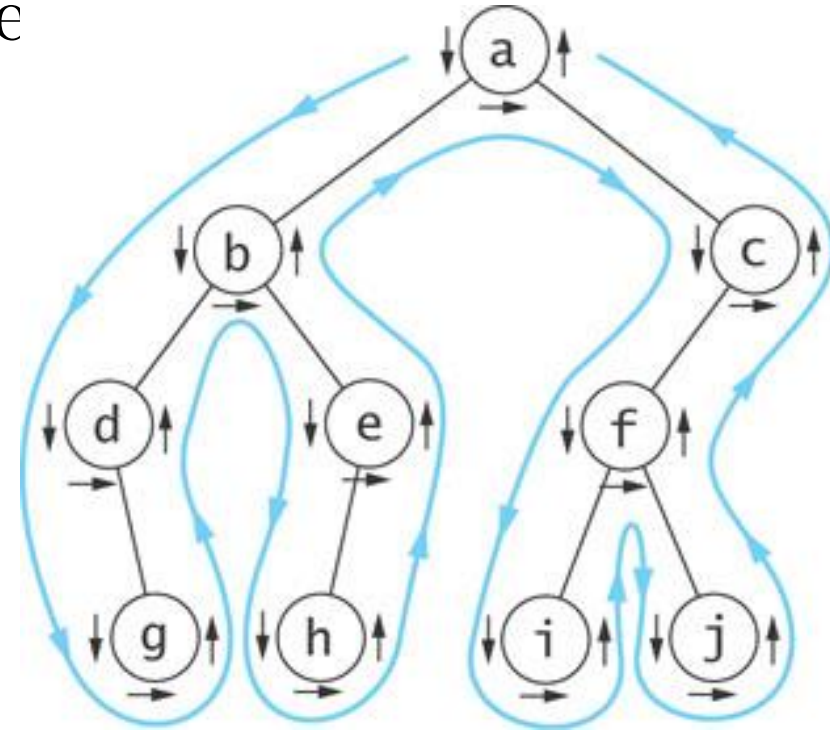
Visualizing Tree Traversals (cont.)

- A 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)



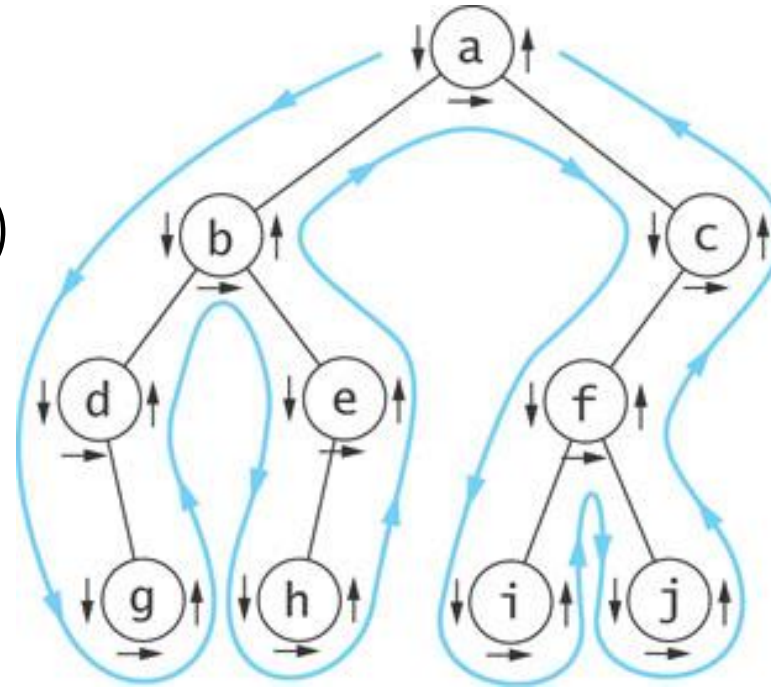
Visualizing Tree Traversals (cont.)

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



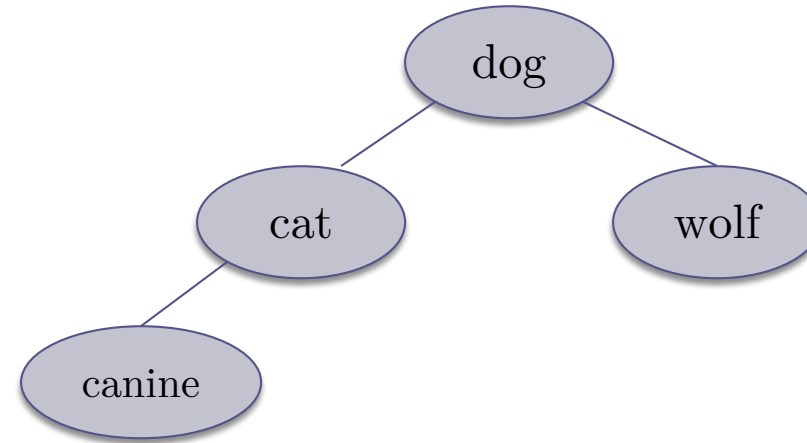
Visualizing Tree Traversals (cont.)

- 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

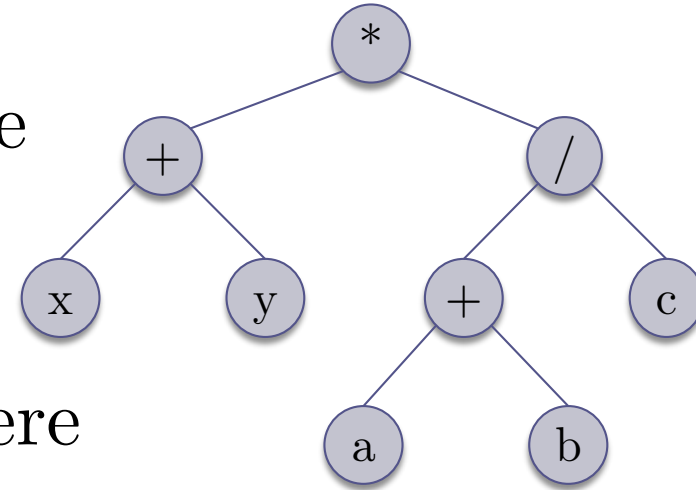


canine, cat, dog, wolf

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

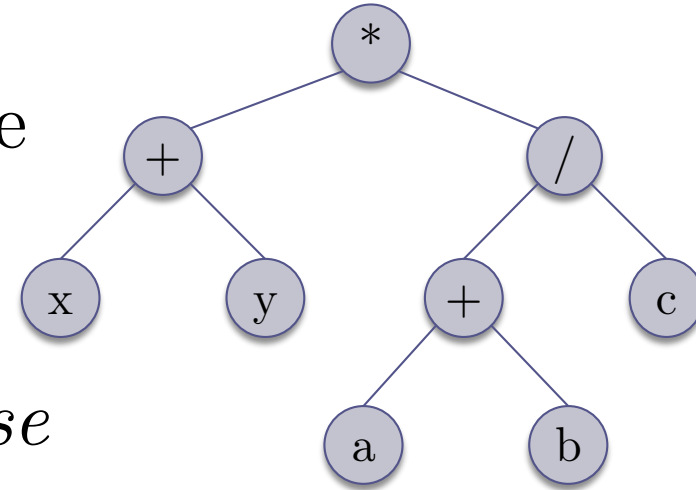
- An inorder traversal of an expression tree results in the sequence
 $x + y * a + b / c$
- If we insert parentheses where they belong, we get the infix form:

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



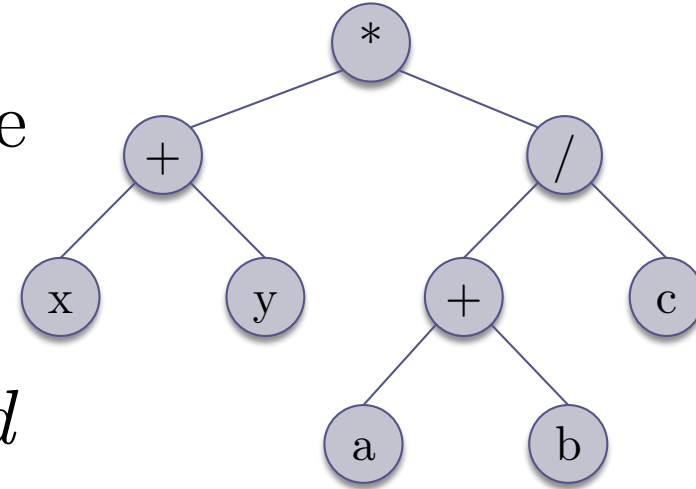
Traversals of Binary Search Trees and Expression Trees (cont.)

- 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

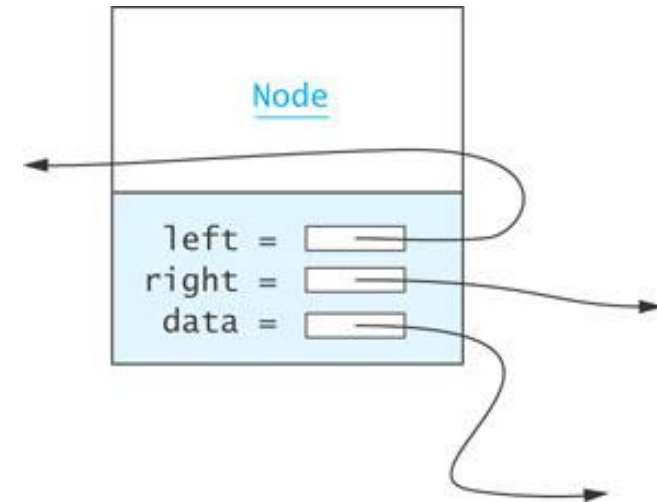


Implementing a BinaryTree Class

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

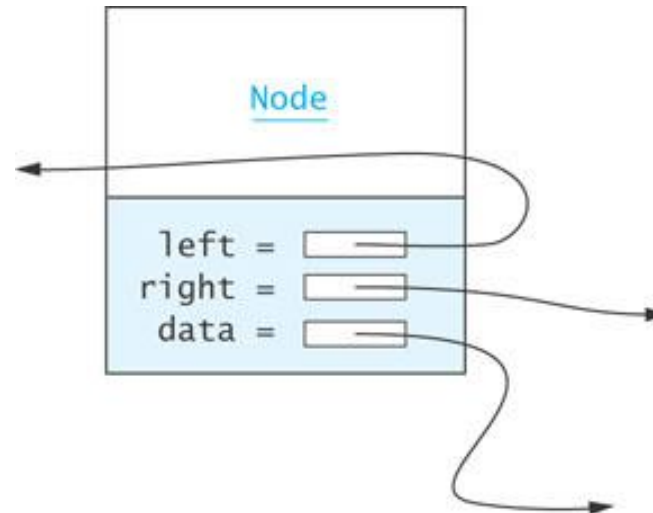


Node<E> Class (cont.)

```
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();
    }
}
```



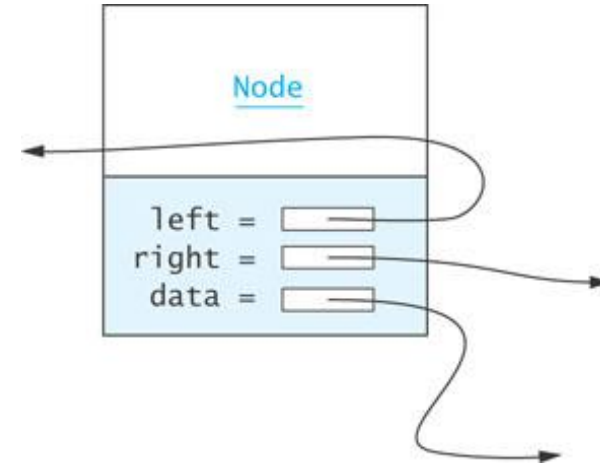
Node<E> is declared as an inner class within BinaryTree<E>

Node<E> Class (cont.)

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

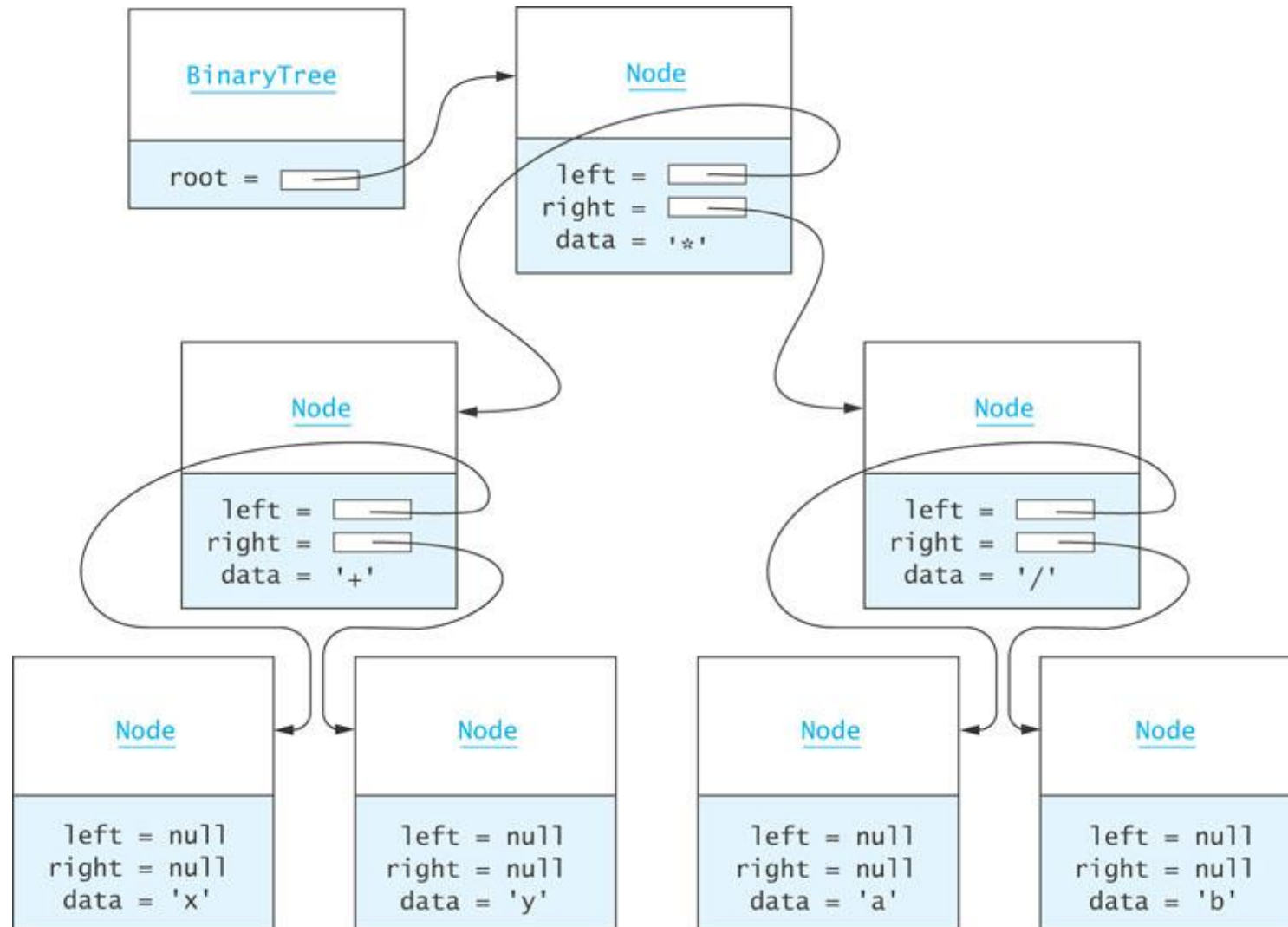
    public Node(E data) {
        this.data = data;
        left = null;
        right = null;
    }

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



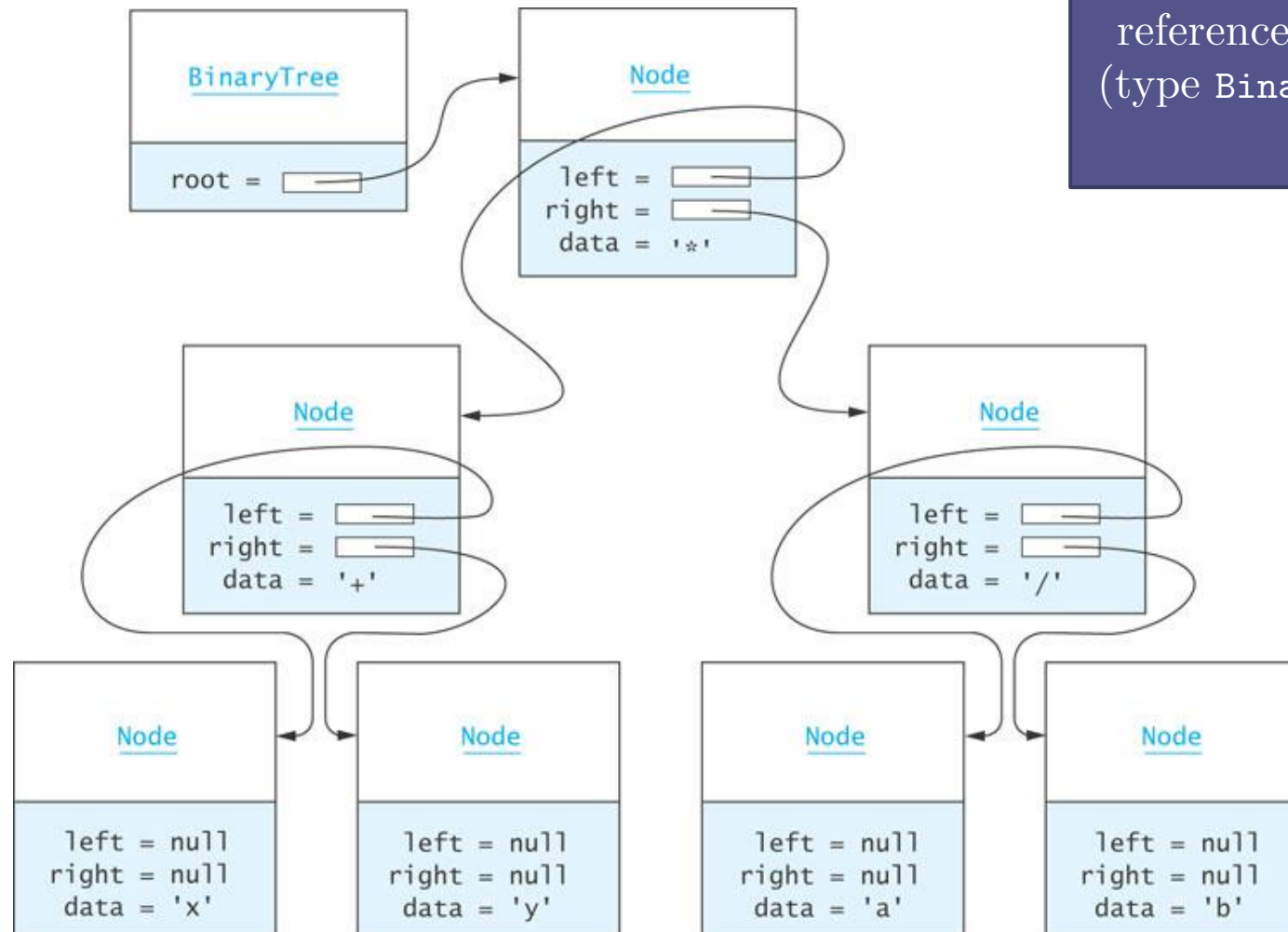
`Node<E>` is declared protected. This way we can use it as a superclass.

BinaryTree<E> Class (cont.)

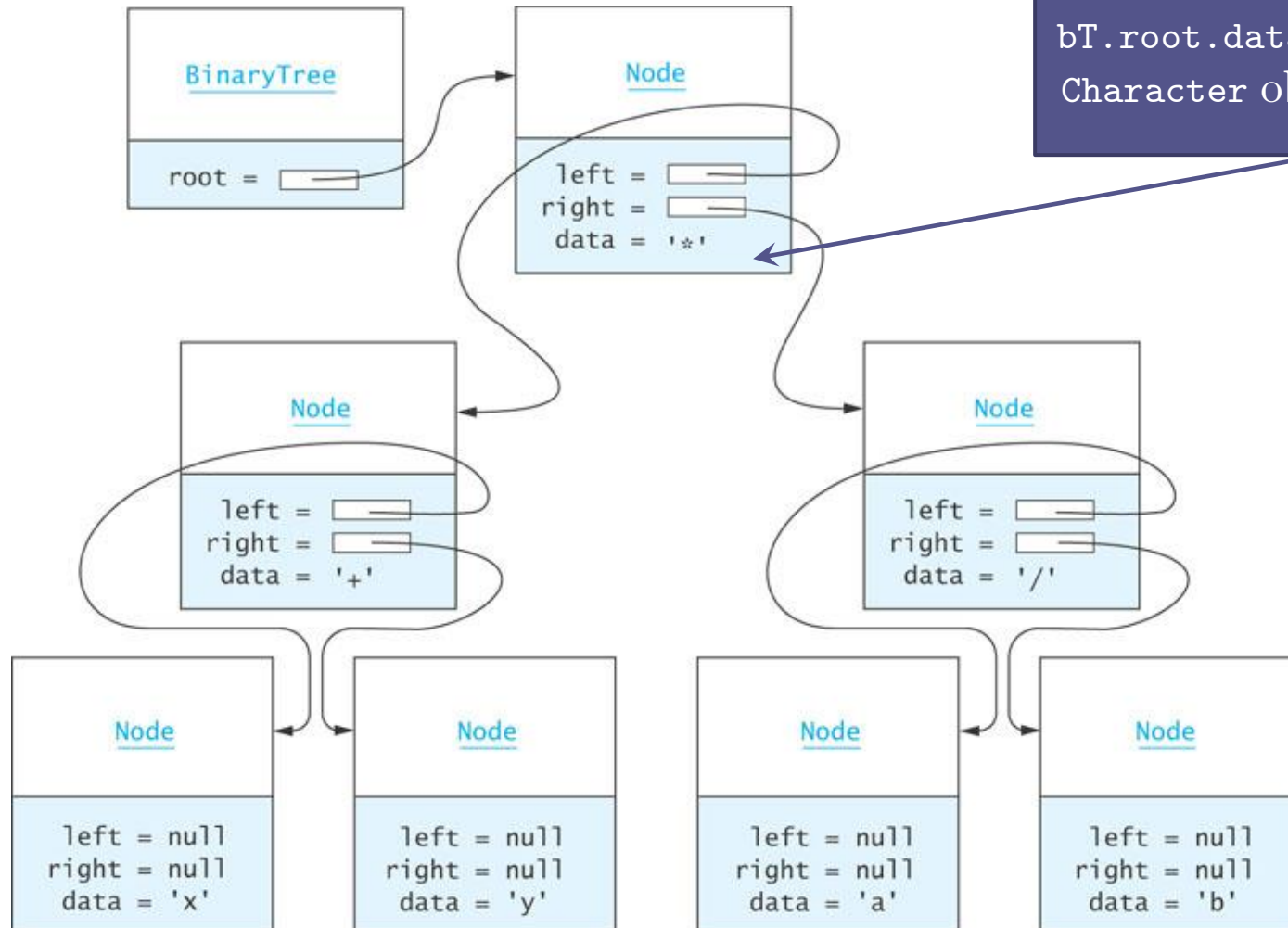


BinaryTree<E> Class (cont.)

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

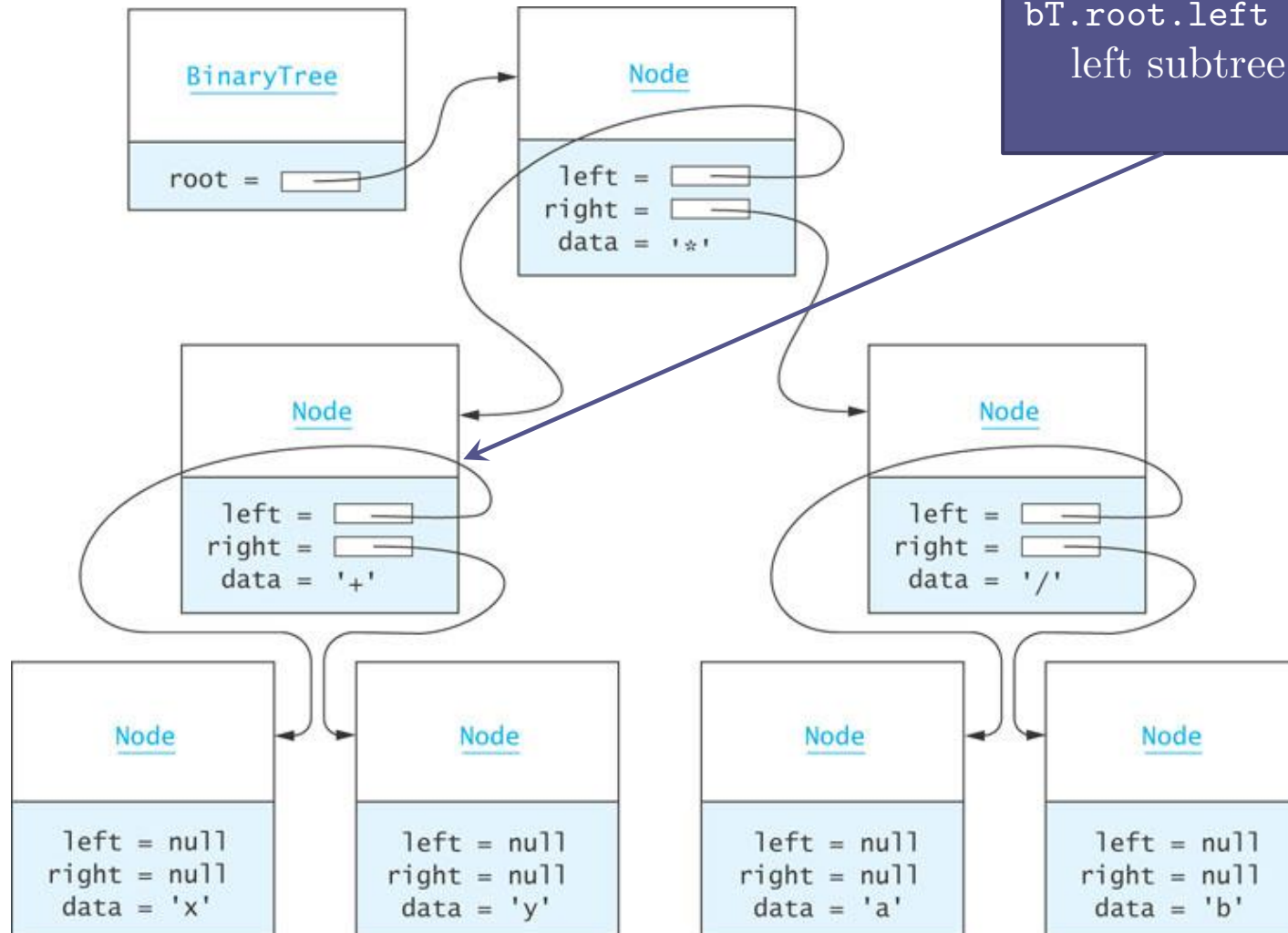


BinaryTree<E> Class (cont.)



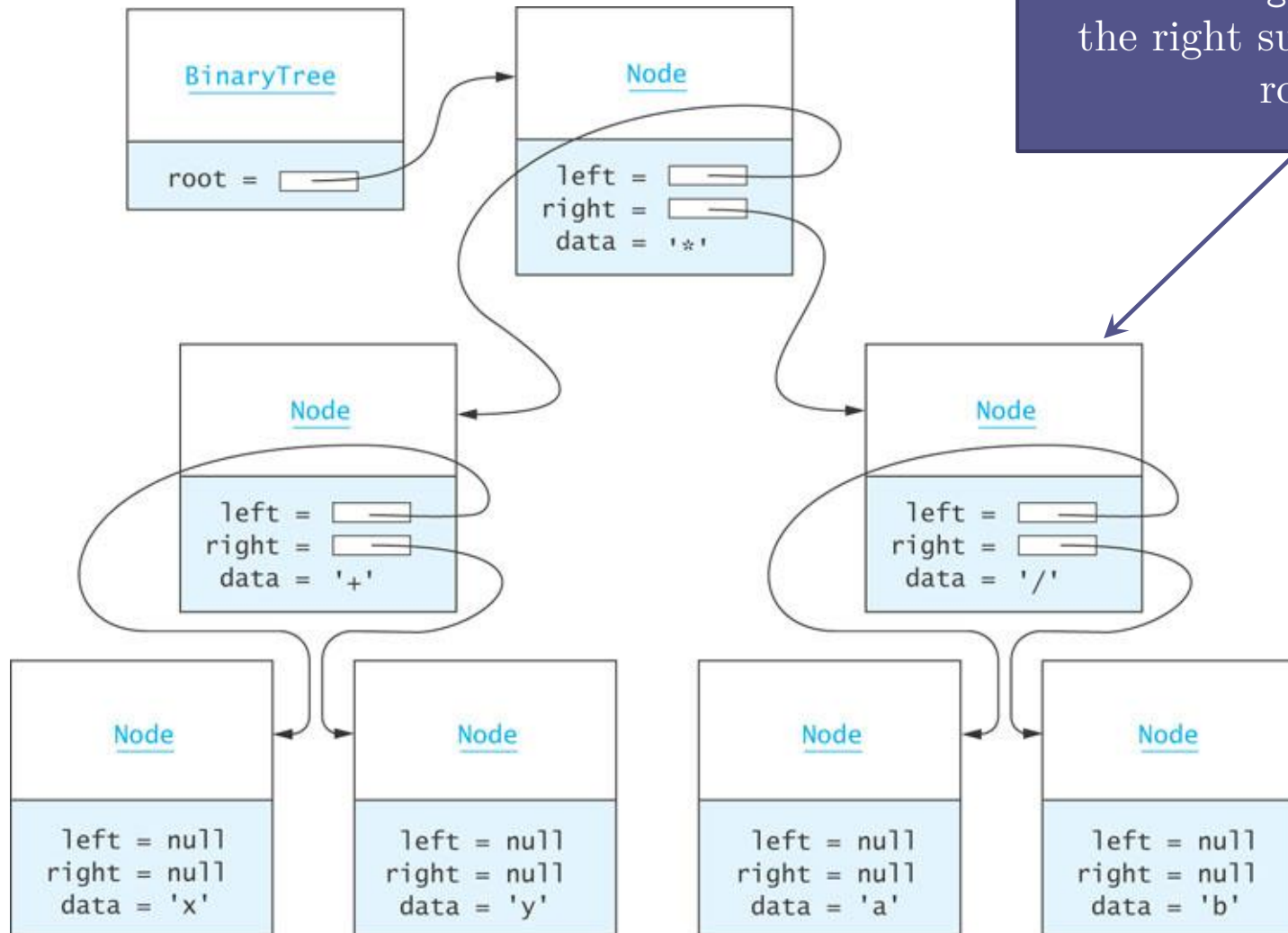
bT.root.data references the
Character object storing '*'

BinaryTree<E> Class (cont.)



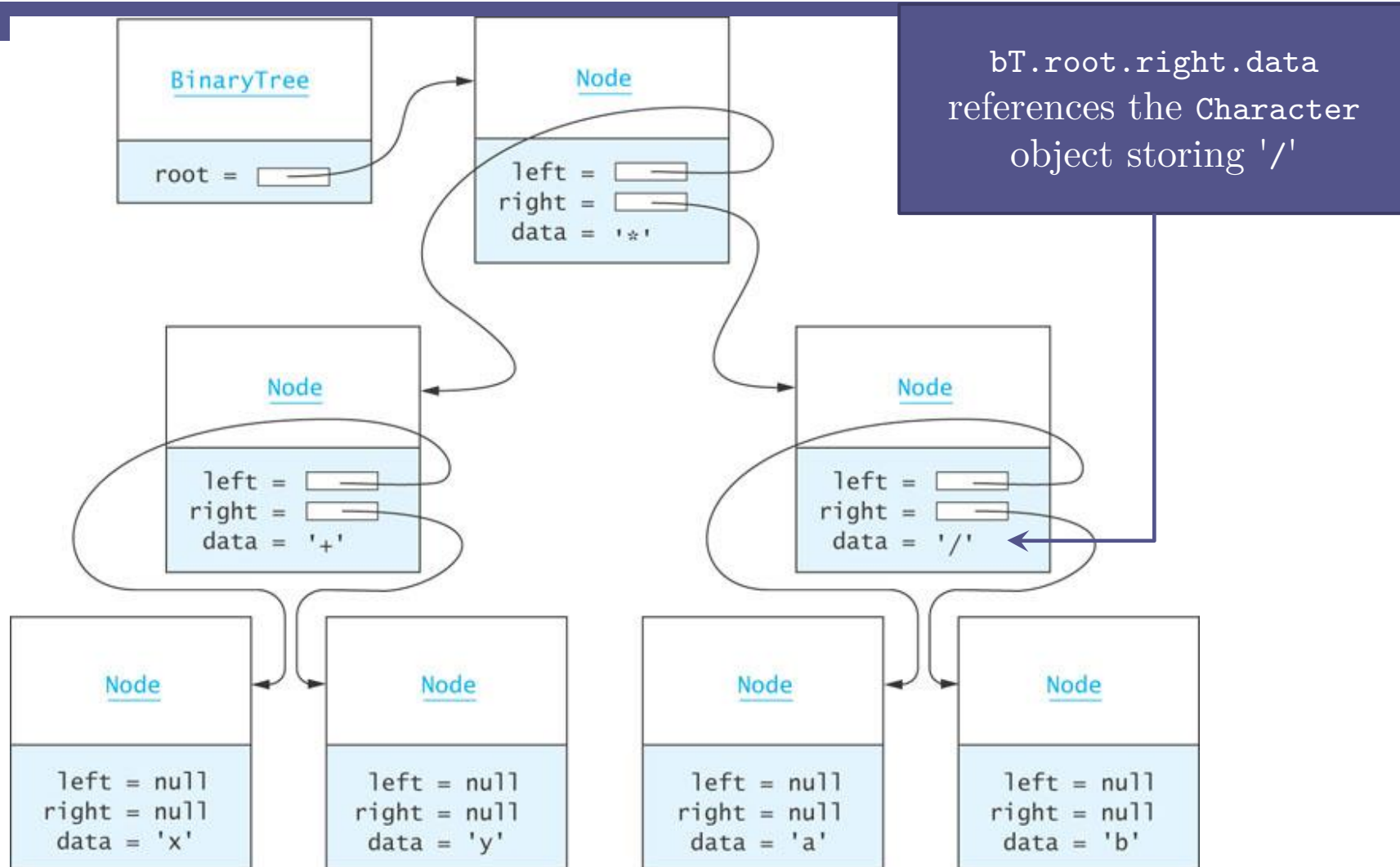
`bT.root.left` references the left subtree of the root

BinaryTree<E> Class (cont.)



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

BinaryTree<E> Class (cont.)



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`

Using ObjectOutputStream and ObjectInputStream

- ❑ The Java API includes the class `ObjectOutputStream` that will write to an external file any object that is declared to be `Serializable`
- ❑ To declare an object `Serializable`, add

`implements Serializable`

to the class declaration
- ❑ The `Serializable` interface contains no methods, but it serves to mark the class and gives you control over whether or not you want your object written to an external file

Using ObjectOutputStream and ObjectInputStream (cont.)

- To write a Serializable object to a file:

```
try {  
    ObjectOutputStream out =  
        new ObjectOutputStream(new FileOutputStream(filename));  
    out.writeObject(nameOfObject);  
} catch (Exception ex) {  
    ex.printStackTrace();  
    System.exit(1);  
}
```

- A deep copy of all the nodes of the binary tree will be written to the file

Using ObjectOutputStream and ObjectInputStream (cont.)

- To read a Serializable object from a file:

```
try {  
    ObjectInputStream in =  
        new ObjectInputStream(new FileInputStream(filename));  
  
    objectName = (objectClass)in.readObject();  
} catch (Exception ex) {  
    ex.printStackTrace();  
    System.exit(1);  
}
```


Using ObjectOutputStream and ObjectInputStream (cont.)

- ❑ Do not recompile the Java source file for a class after an object of that class has been serialized
- ❑ Even if you didn't make any changes to the class, the resulting `.class` file associated with the serialized object will have a different class signature
- ❑ When you attempt to read the object, the class signatures will not match, and you will get an exception

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() {  
    return (root.left == null && root.right == null);  
}
```

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();  
}
```

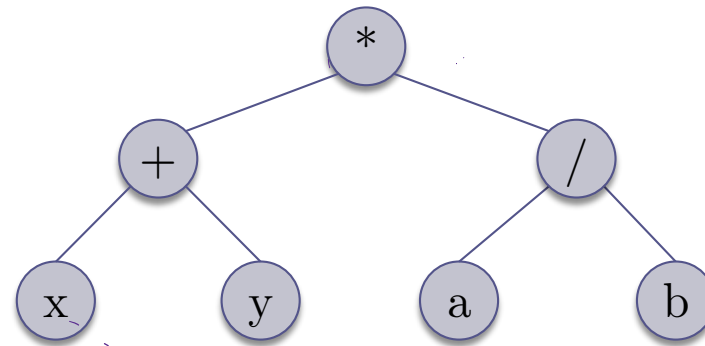
preOrderTraverse Method

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

    for (int i = 1; i < depth; i++) {
        sb.append("  "); // indentation
    }
    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
```



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

Reading a Binary Tree

- If we use a Scanner to read the individual lines created by the `toString` and `preOrderTraverse` methods, we can reconstruct the tree
- 1. Read a line that represents information at the root
- 2. Remove the leading and trailing spaces using `String.trim`
- 3. **if** it is "null"
- 4. return **null**
- else**
- 5. recursively read the left child
- 6. recursively read the right child
- 7. return a tree consisting of the root and the two children

Binary Search Trees

Section 6.4

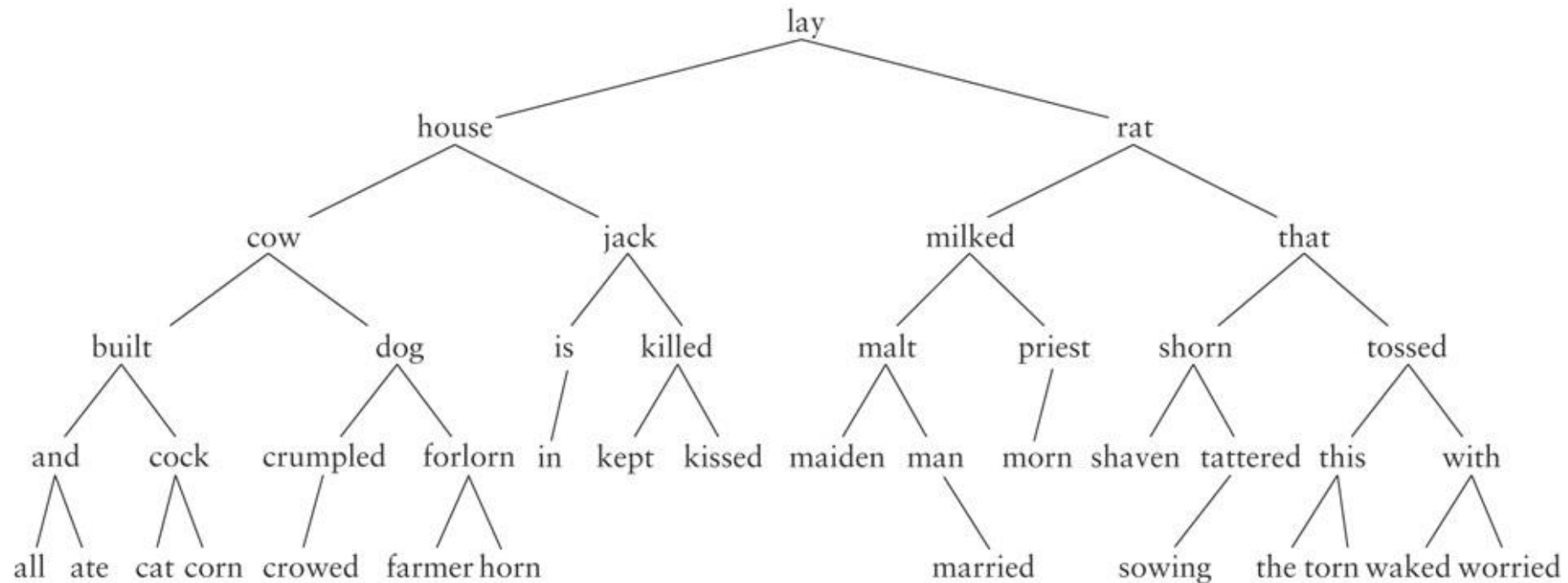
Overview of a Binary Search Tree

- 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

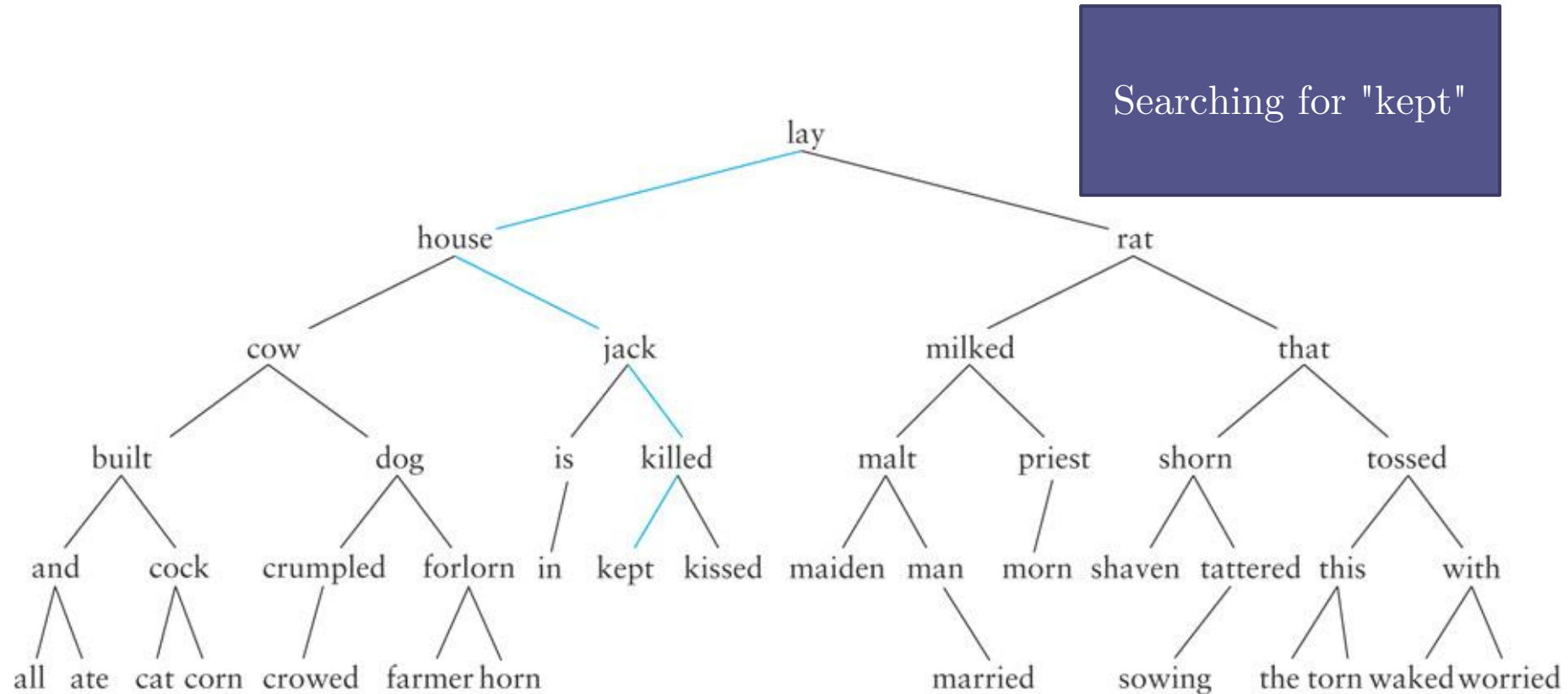
Overview of a Binary Search Tree (cont.)



Recursive Algorithm for Searching a Binary Search Tree

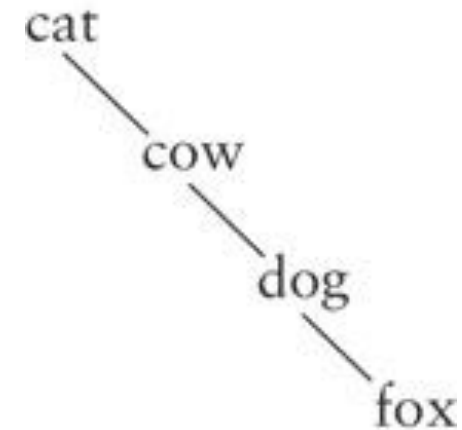
1. **if** the root is **null**
 2. the item is not in the tree; return **null**
3. Compare the value of **target** with **root.data**
4. **if** they are equal
 5. the target has been found; return the data at the root
- else if** the target is less than **root.data**
 6. return the result of searching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

Searching a Binary Tree



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 $O(n)$

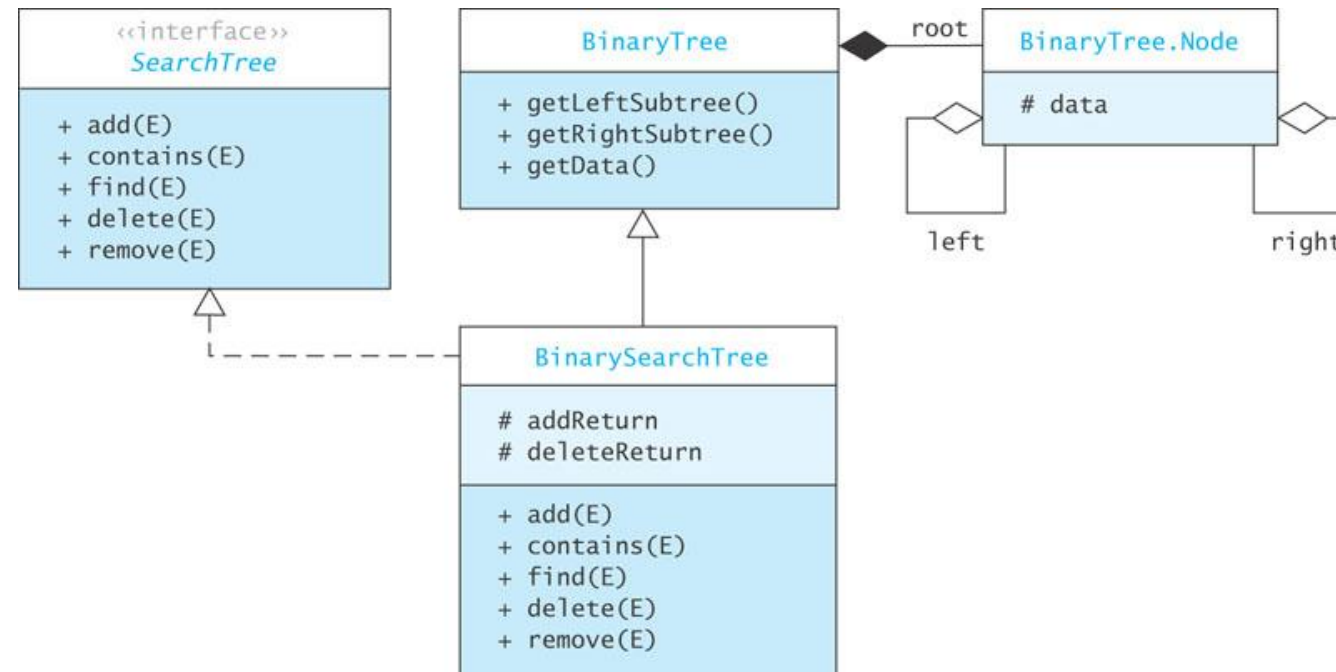


Interface SearchTree<E>

Method	Behavior
boolean add(E item)	Inserts <code>item</code> 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 <code>target</code> is found in the tree.
E find(E target)	Returns a reference to the data in the node that is equal to <code>target</code> . If no such node is found, returns null .
E delete(E target)	Removes <code>target</code> (if found) from tree and returns it; otherwise, returns null .
boolean remove(E target)	Removes <code>target</code> (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.



Implementing 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);
}

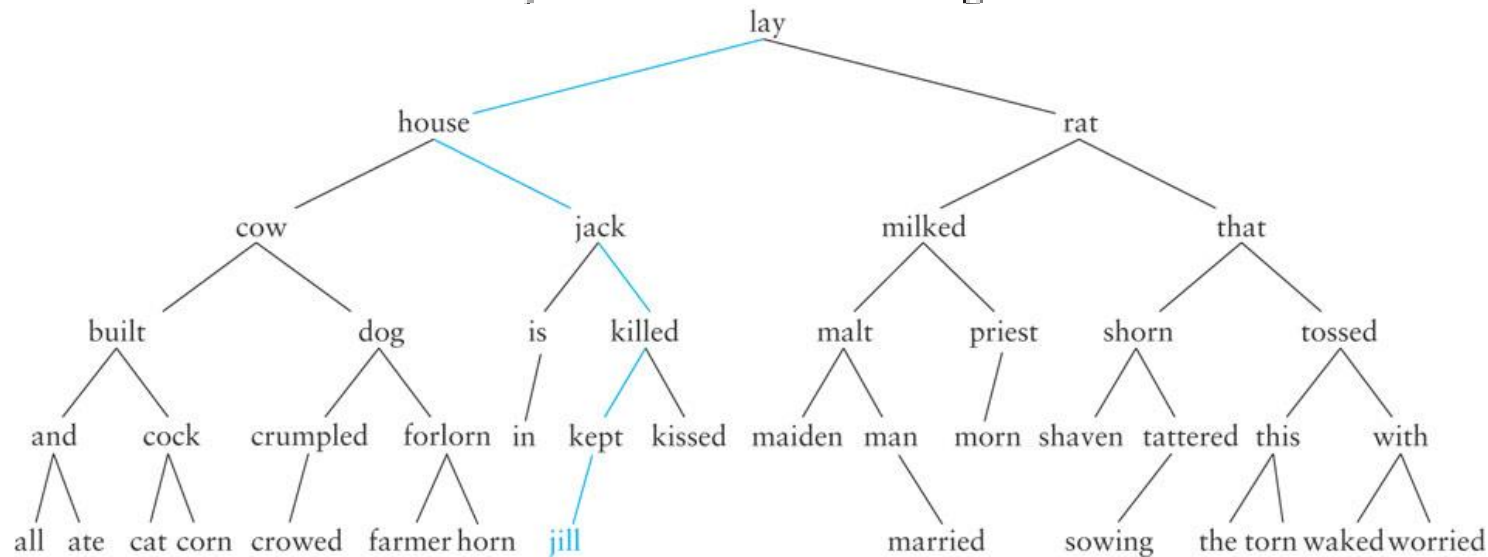
/** Recursive find method.
    @param localRoot The local subtree's root
    @param target The object being sought
    @return The object, if found, otherwise null
*/
private E find(Node<E> localRoot, E target) {
    if (localRoot == null)
        return null;

    // 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);
}
```

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;  
}
```

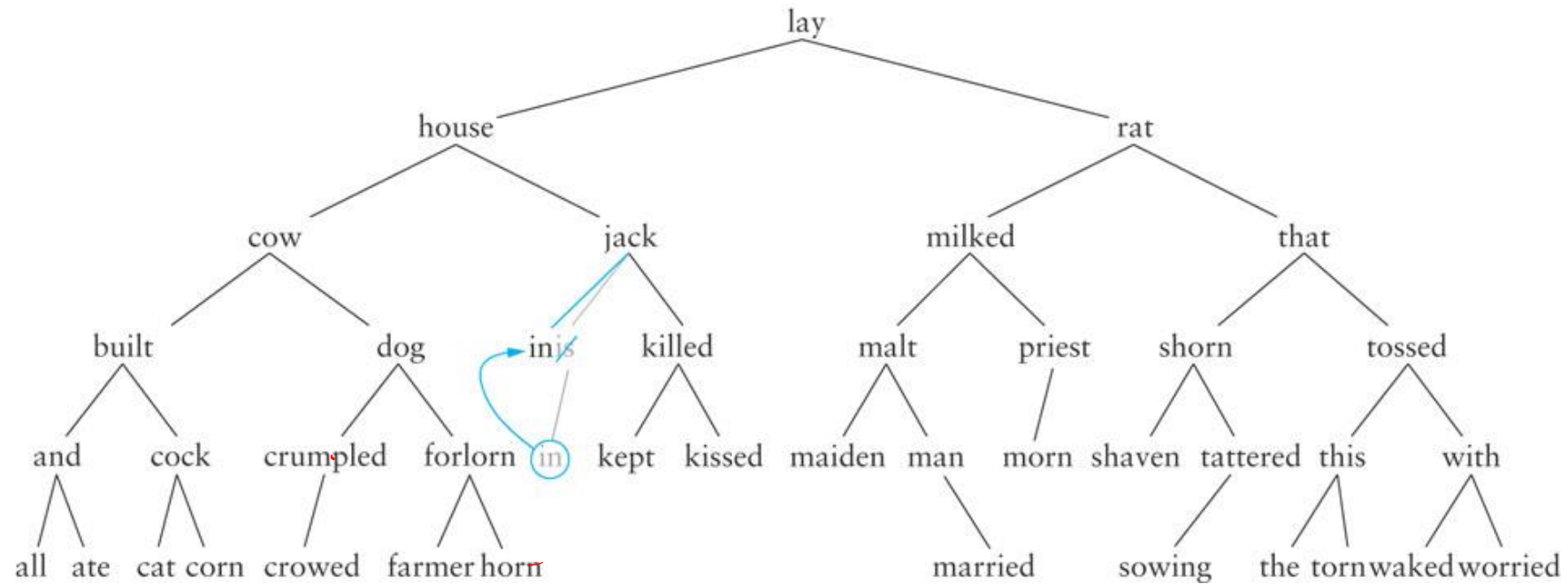
```
□ /** Recursive add method.
□ post: The data field addReturn is set true if the item is added to
□ the tree, false if the item is already in the tree.
□ @param localRoot The local root of the subtree
□ @param item The object to be inserted
□ @return The new local root that now contains the
□ inserted item
□ */
□ private Node<E> add(Node<E> localRoot, E item) {
□     if (localRoot == null) {
□         addReturn = true;
□         return new Node<E>(item);
□     } else if (item.compareTo(localRoot.data) == 0) {
□         addReturn = false;
□         return localRoot;
□     } else if (item.compareTo(localRoot.data) < 0) {
□         localRoot.left = add(localRoot.left, item);
□         return localRoot;
□     } else {
□         localRoot.right = add(localRoot.right, item);
□         return localRoot;
□     }
□ }
```

Removal from a Binary Search Tree

- 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 Binary Search Tree

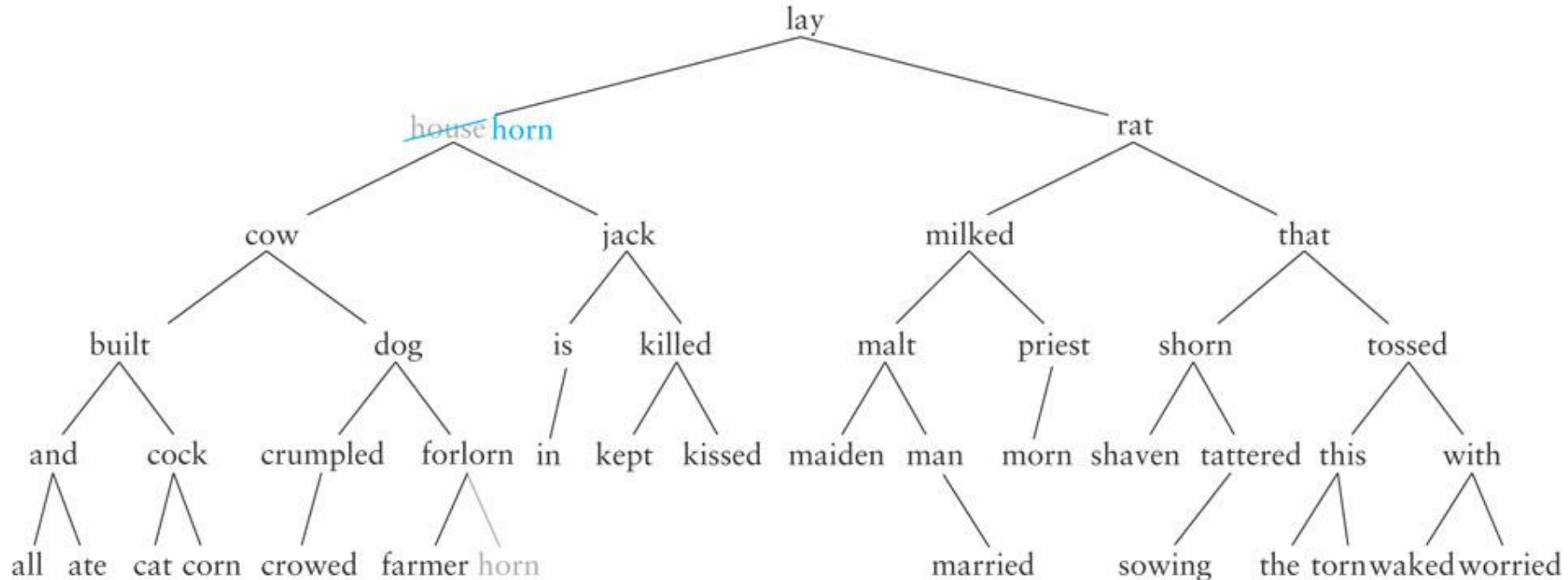
(cont.)



Removing from a Binary Search Tree

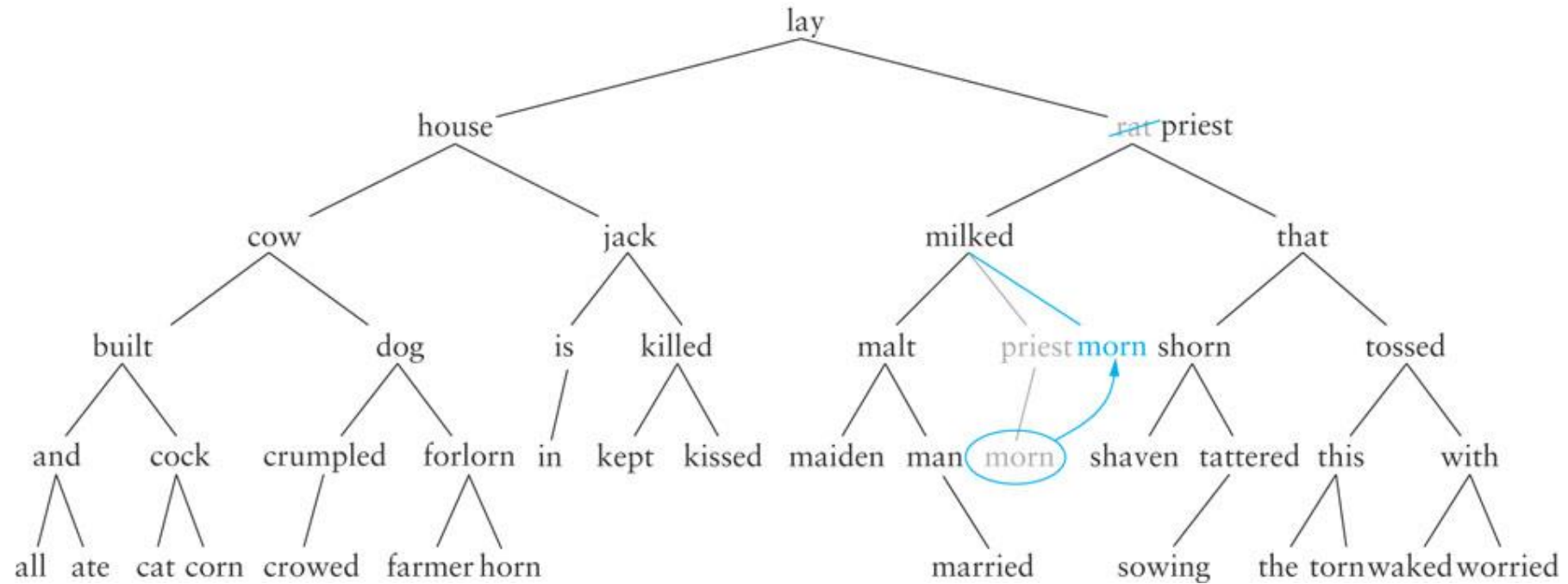
(cont.)

- 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.)



Algo: Search

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 **deleted**Return.
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.

}inary

Implementing the delete Method

- Listing 6.5 (BinarySearchTree delete Methods; pages 325-326)

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

Writing an Index for a Term Paper

- Problem: write an index for a term paper
 - ▣ The index should show each word in the paper followed by the line number on which it occurred
 - ▣ The words should be displayed in alphabetical order
 - ▣ If a word occurs on multiple lines, the line numbers should be listed in ascending order:
 - a, 003
 - a, 013
 - are, 003

Writing an Index for a Term Paper

(cont.)

□ Analysis

- ▣ Store each word and its line number as a string in a tree node
- ▣ For example, two occurrences of "java": "java, 005" and "java, 010"
- ▣ Display the words in ascending order by performing an inorder traversal

Writing an Index for a Term Paper (cont.)

□ Design

- ▣ Use `TreeSet<E>`, a class based on a binary search tree, provided in the Java API
- ▣ Write a class `IndexGenerator` with a `TreeSet<String>` data fields

Data Field	Attribute
<code>private TreeSet<String> index</code>	The search tree used to store the index.
Method	Behavior
<code>public void buildIndex(Scanner scan)</code>	Reads each word from the file scanned by <code>scan</code> and stores it in tree <code>index</code> .
<code>public void showIndex()</code>	Performs an inorder traversal of tree <code>index</code> .

Writing an Index for a Term Paper (cont.)

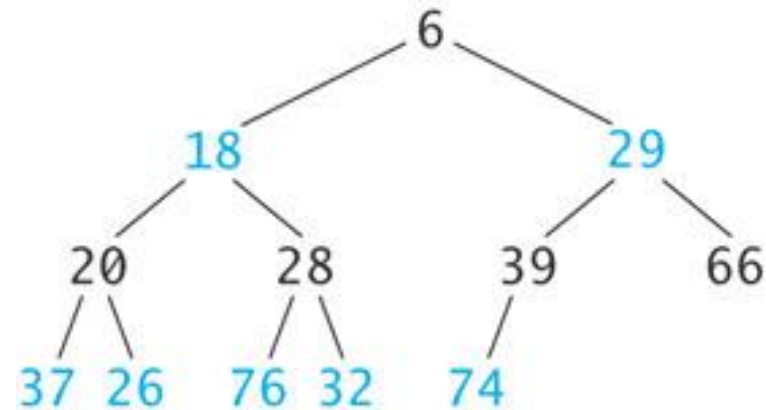
- Listing 6.7 (Class `IndexGenerator.java`; pages 330-331)

Heaps and Priority Queues

Section 6.5

Heaps and Priority Queues

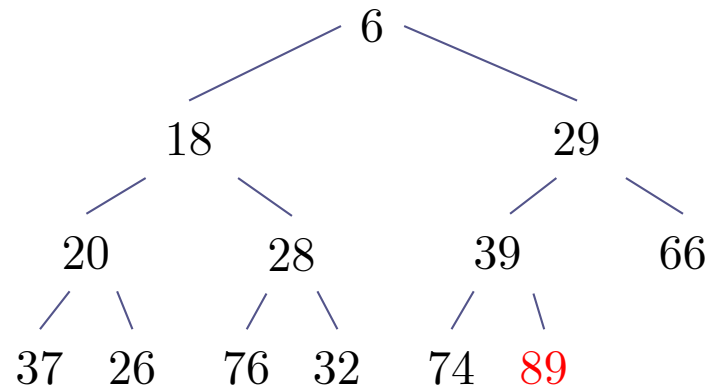
- A heap is a complete binary tree with the following properties
 - ▣ The value in the root is a smallest item in the tree
 - ▣ Every nonempty subtree is a heap



Inserting an Item into a Heap

Algorithm for Inserting in a Heap

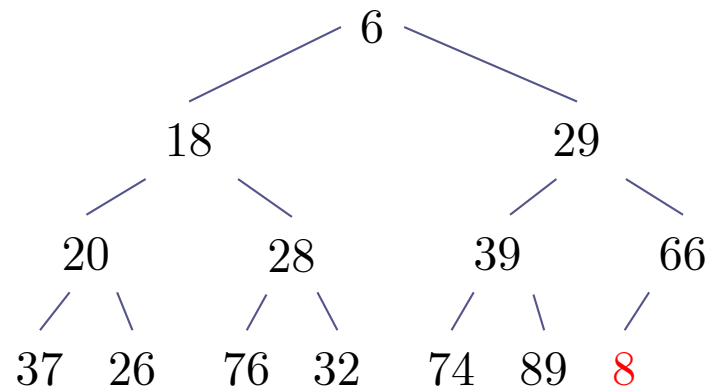
1. Insert the new item in the next position at the bottom of the heap.
2. **while** new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

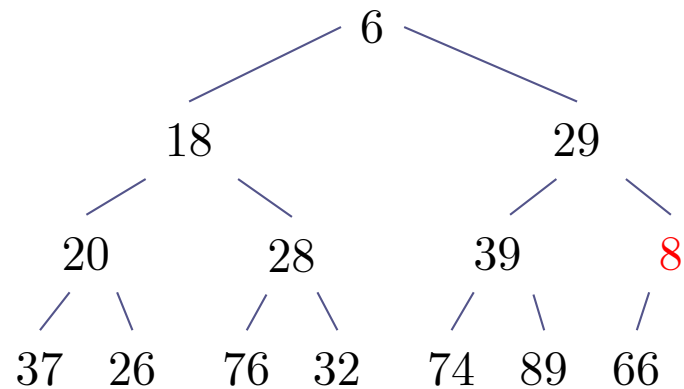
1. Insert the new item in the next position at the bottom of the heap.
2. **while** new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

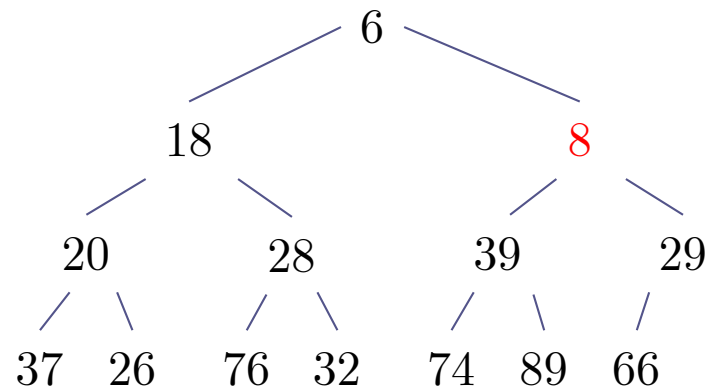
1. Insert the new item in the next position at the bottom of the heap.
2. **while** new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

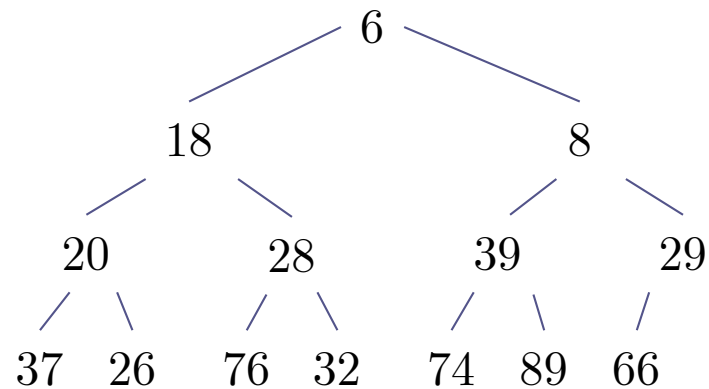
1. Insert the new item in the next position at the bottom of the heap.
2. **while** new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

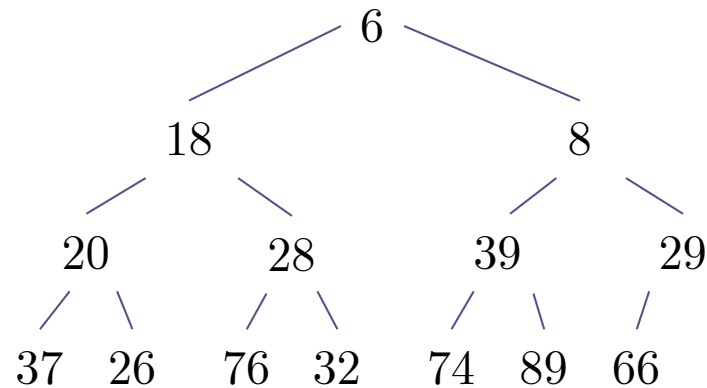
1. Insert the new item in the next position at the bottom of the heap.
2. **while** new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Removing an Item from a Heap

Algorithm for Removal from a Heap

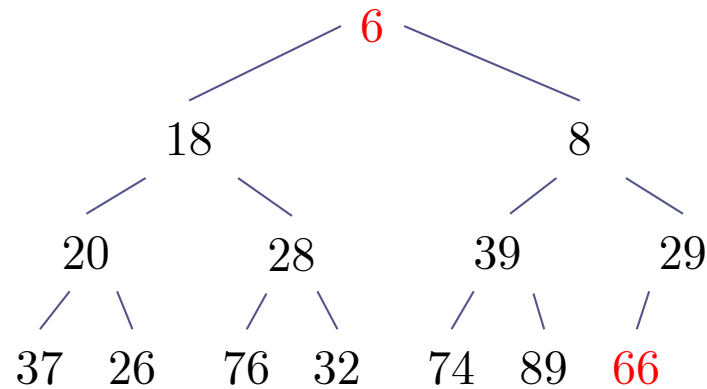
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

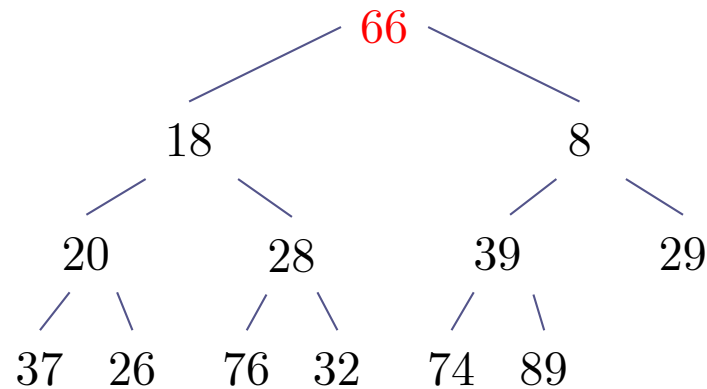
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

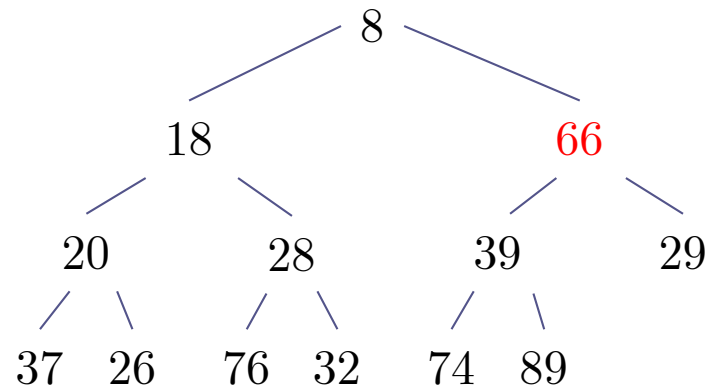
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

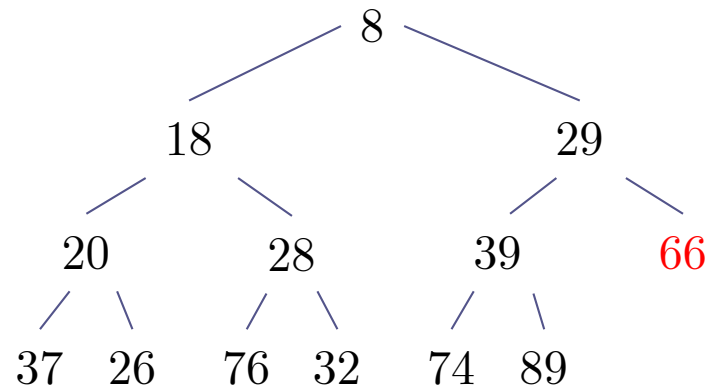
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

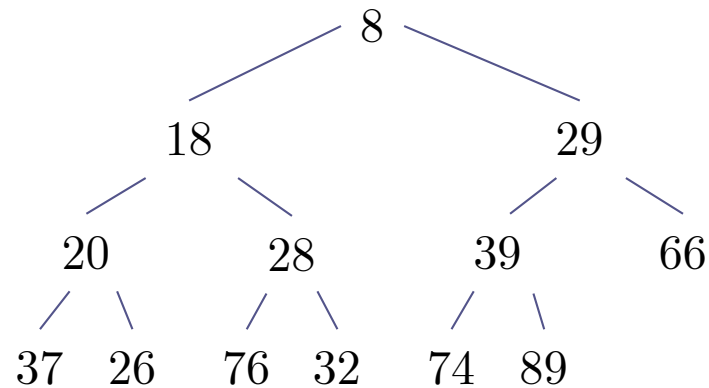
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

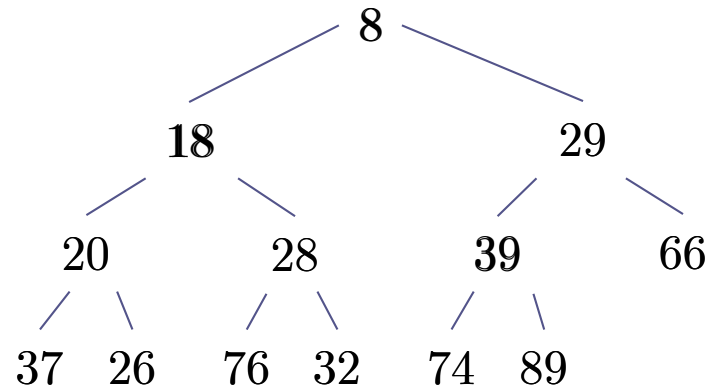
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Implementing a Heap

- Because a heap is a complete binary tree, it can be implemented efficiently using an array rather than a linked data structure

Implementing a Heap (cont.)



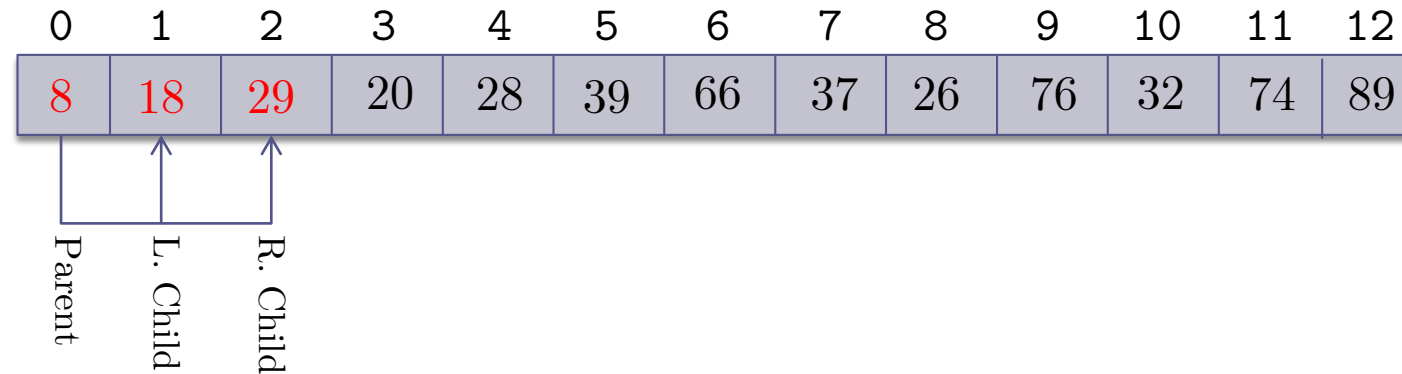
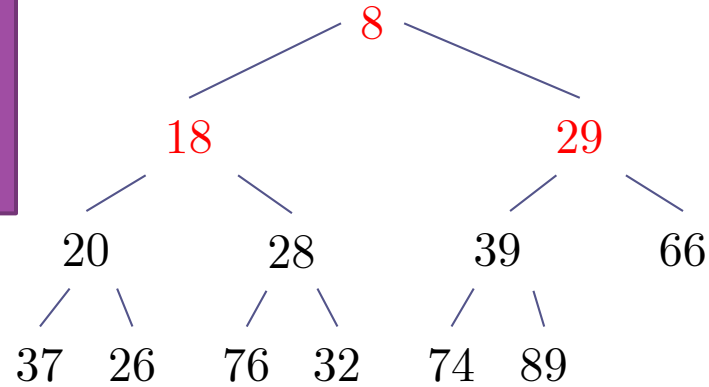
0	1	2	3	4	5	6	7	8	9	10	11	12
8	18	29	20	28	39	66	37	26	76	32	74	89

Implementing a Heap (cont.)

For a node at position p ,

L. child position: $2p + 1$

R. child position: $2p + 2$

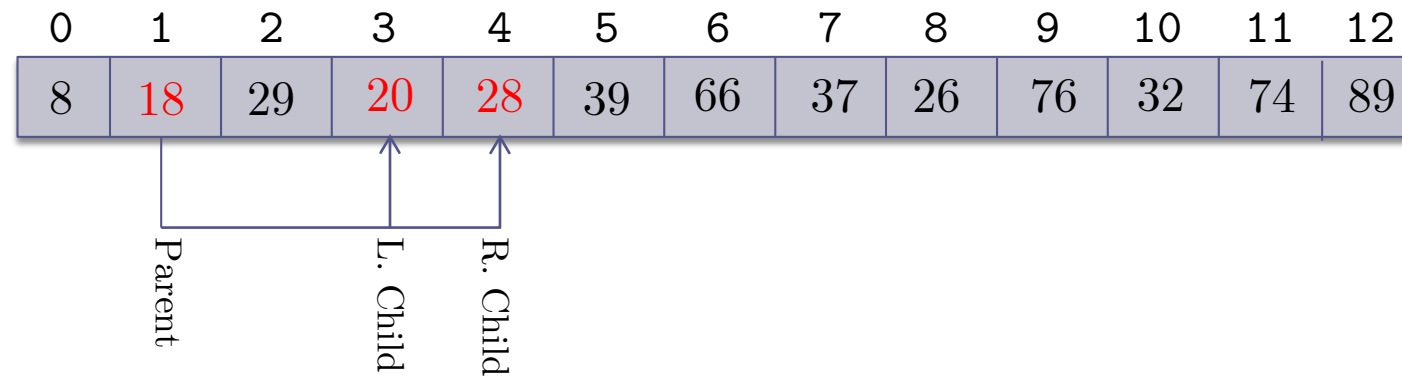
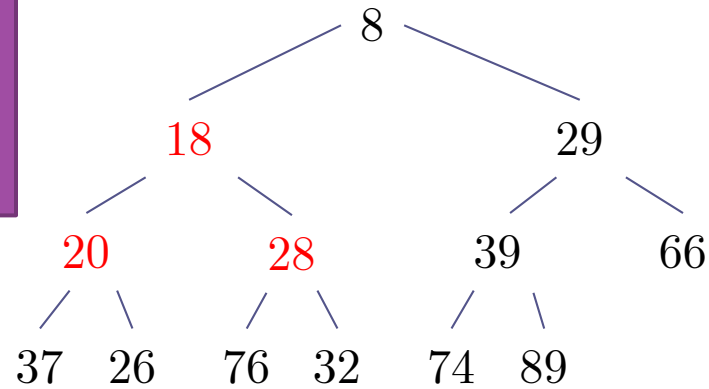


Implementing a Heap (cont.)

For a node at position p ,

L. child position: $2p + 1$

R. child position: $2p + 2$

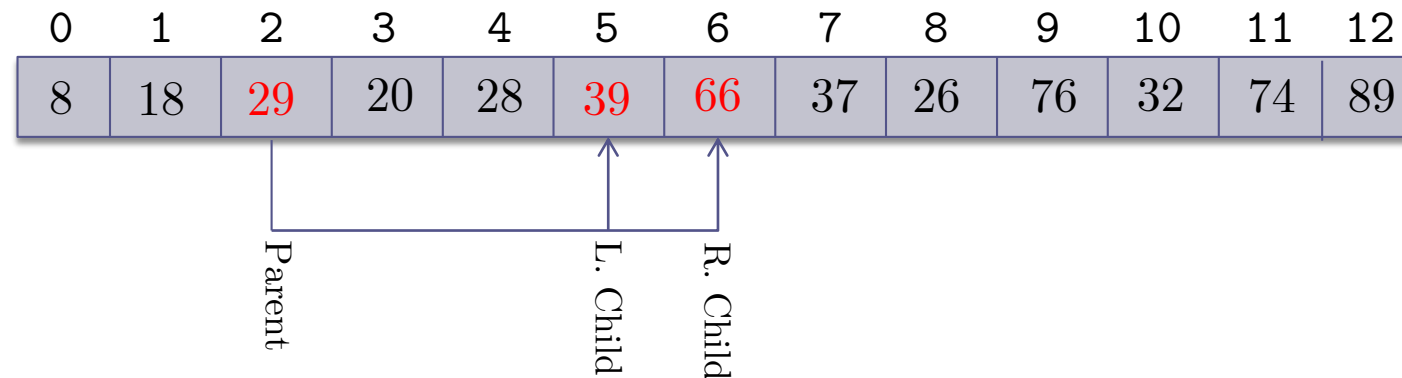
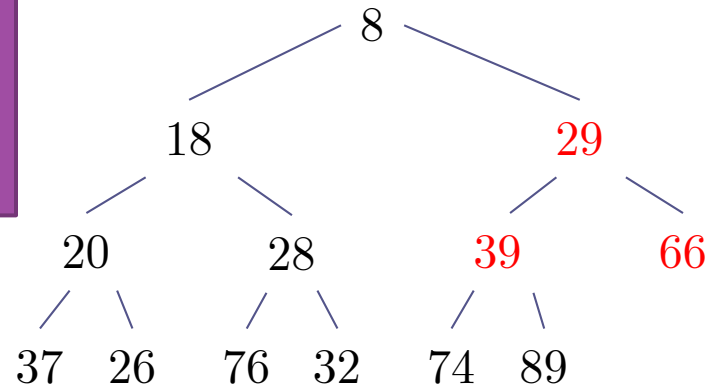


Implementing a Heap (cont.)

For a node at position
 p ,

L. child position: $2p$
 $+ 1$

R. child position: $2p$
 $+ 2$

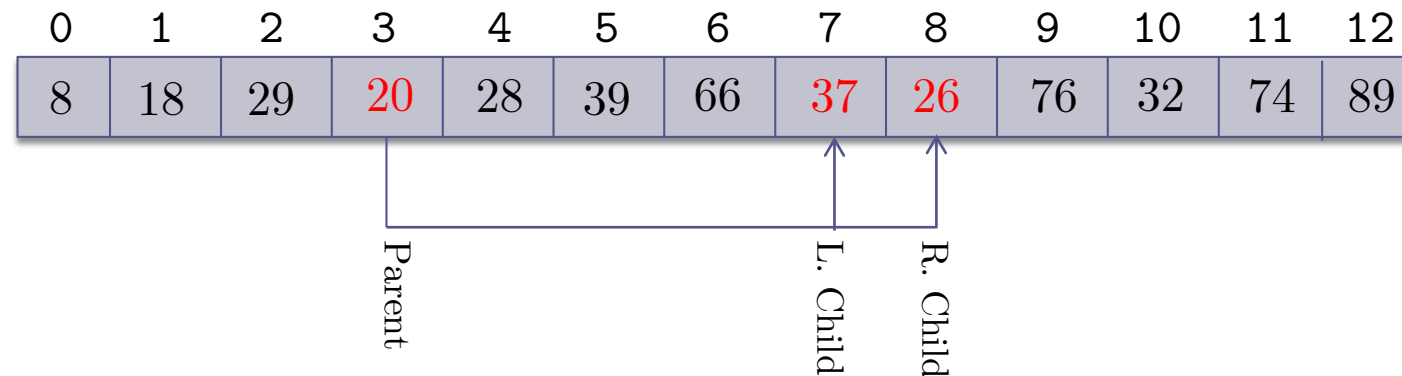
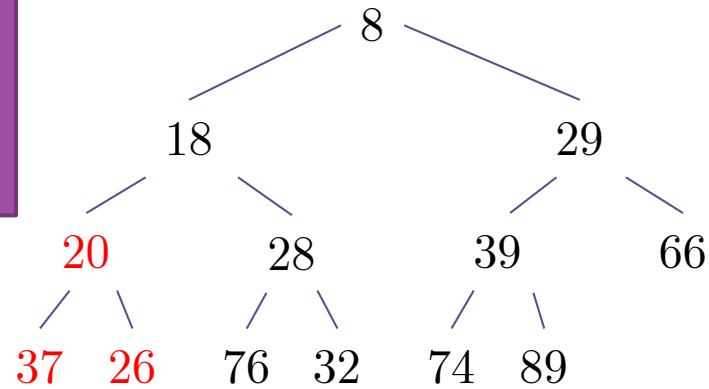


Implementing a Heap (cont.)

For a node at position p ,

L. child position: $2p + 1$

R. child position: $2p + 2$

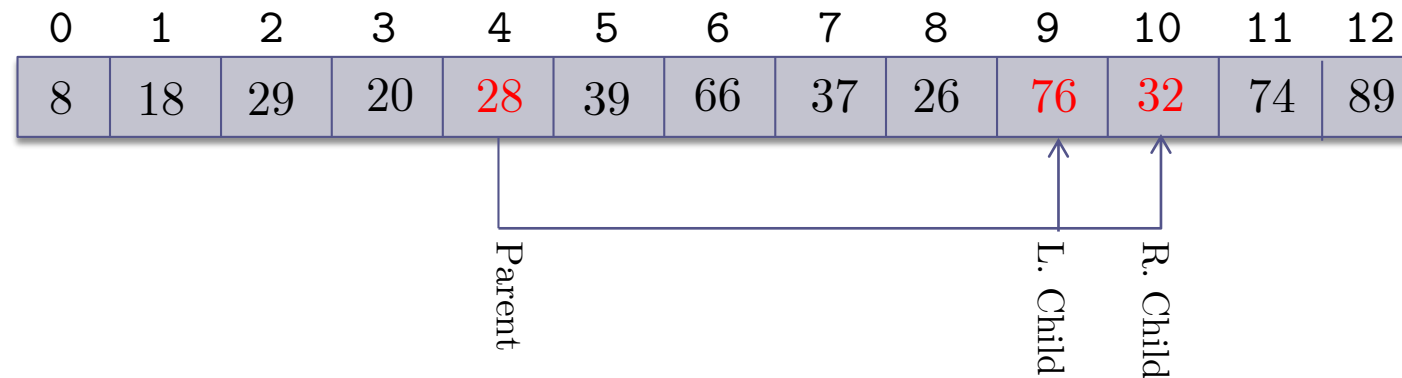
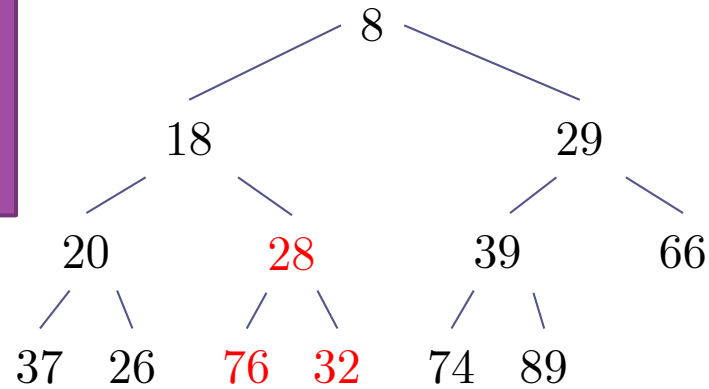


Implementing a Heap (cont.)

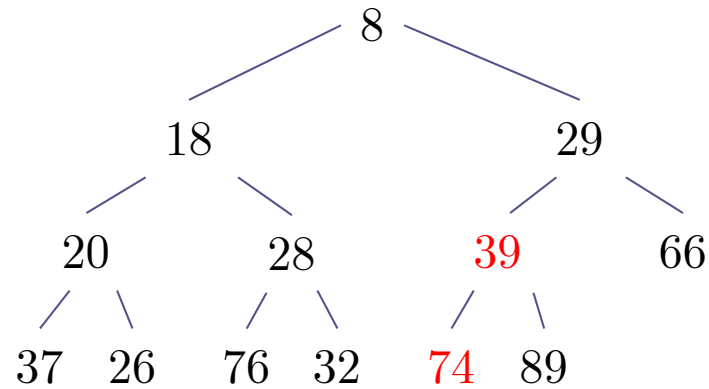
For a node at position p ,

L. child position: $2p + 1$

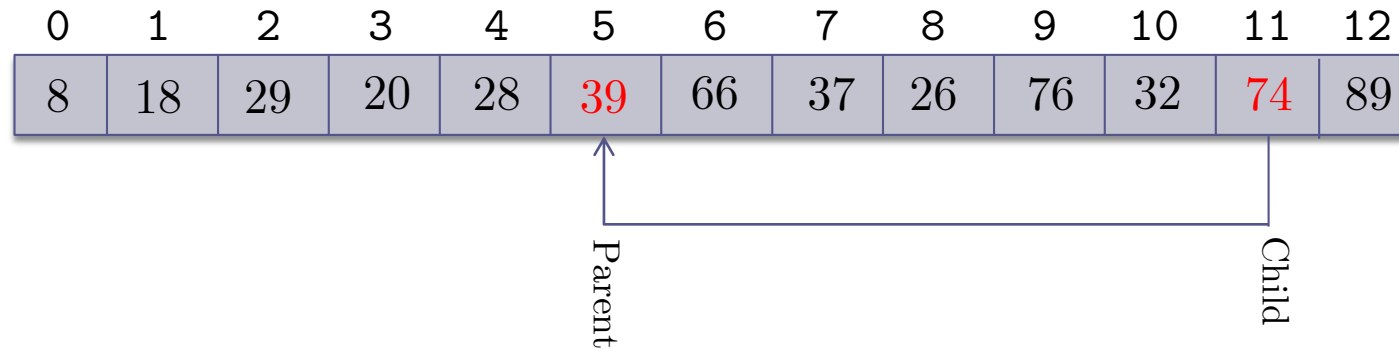
R. child position: $2p + 2$



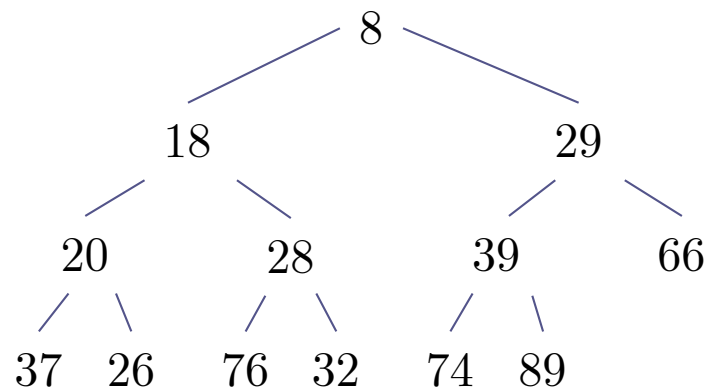
Implementing a Heap (cont.)



A node at position c
can find its parent
at
 $(c - 1)/2$



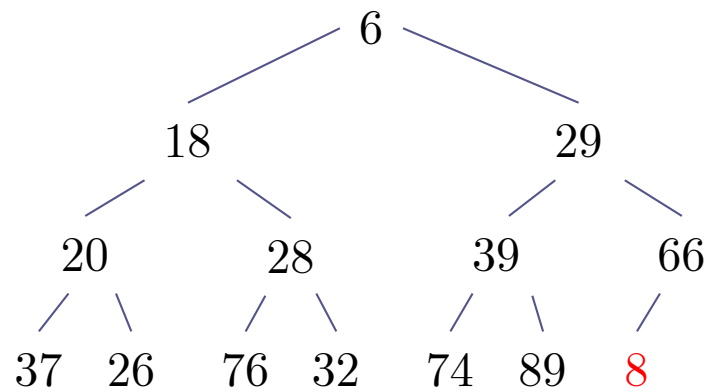
Inserting into a Heap Implemented as an ArrayList



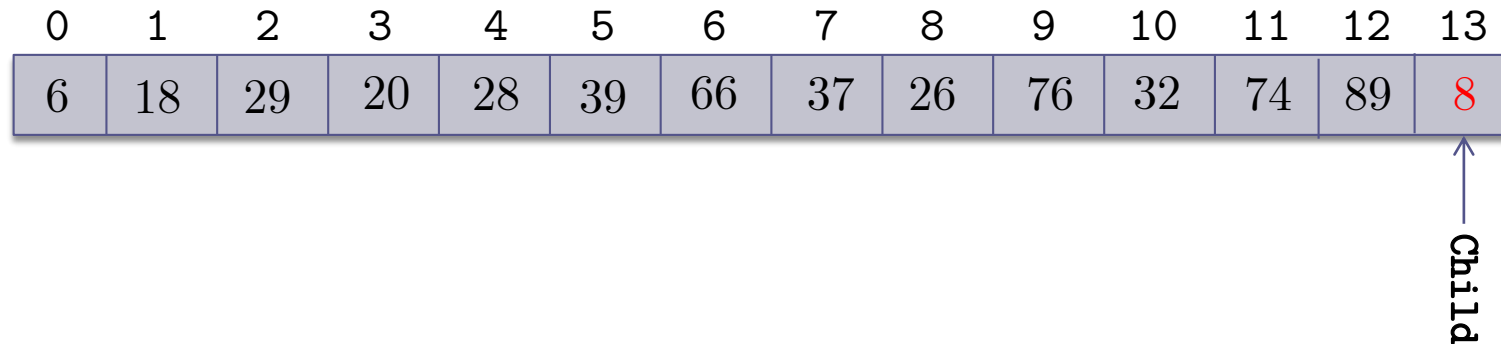
1. Insert the new element at the end of the ArrayList and set child to `table.size() - 1`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	

Inserting into a Heap Implemented as an ArrayList (cont.)

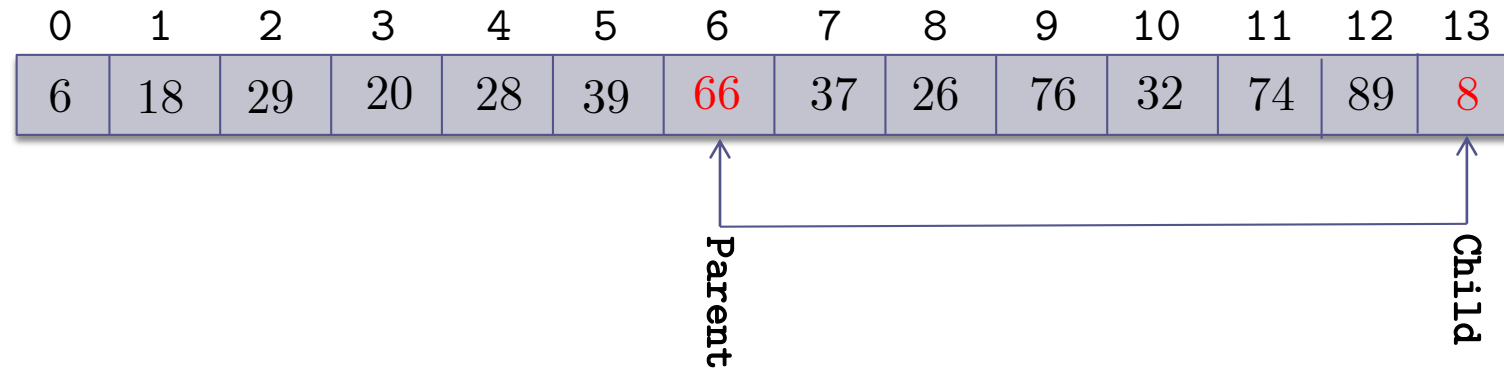
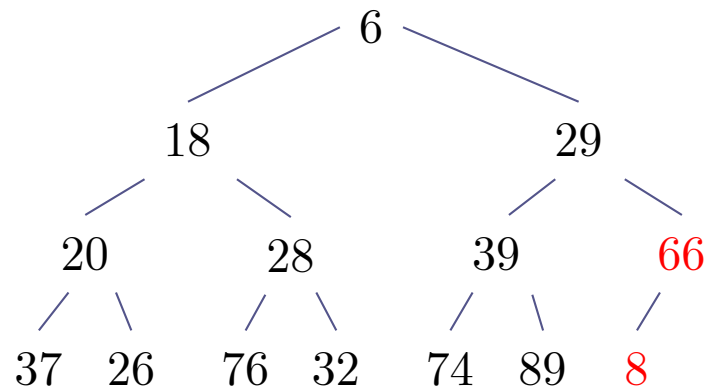


1. Insert the new element at the end of the ArrayList and set `child` to `table.size() - 1`

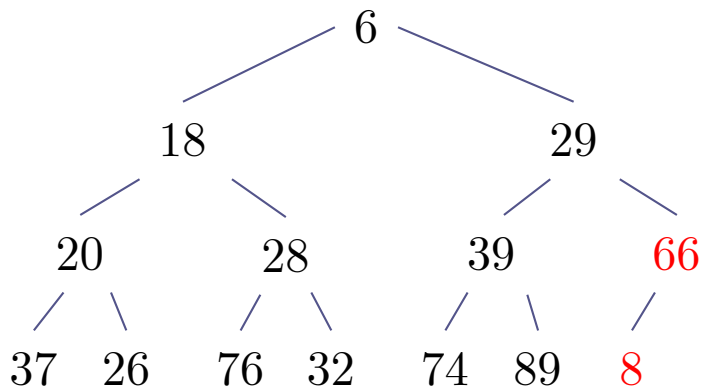


Inserting into a Heap Implemented as an ArrayList (cont.)

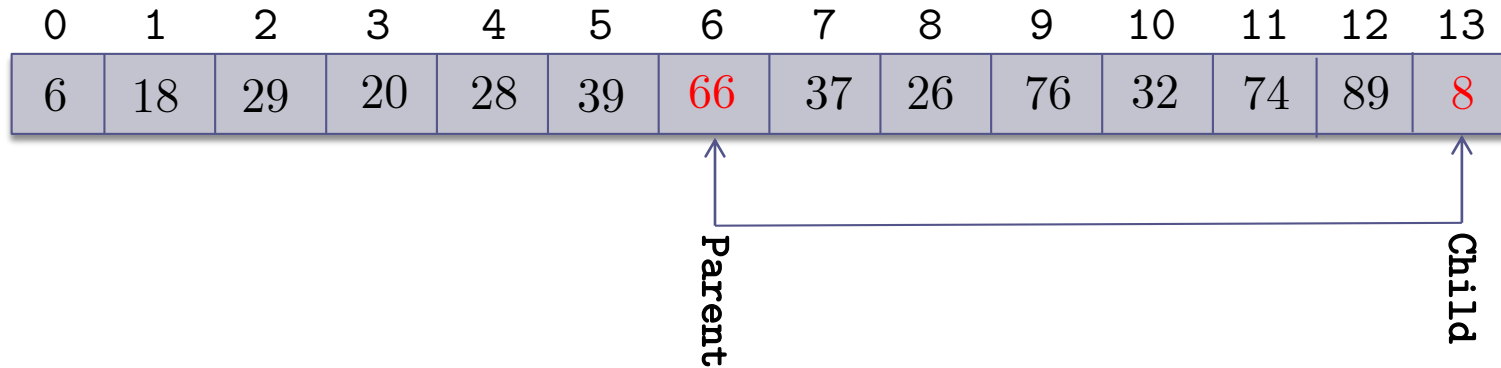
2. Set parent to $(\text{child} - 1) / 2$



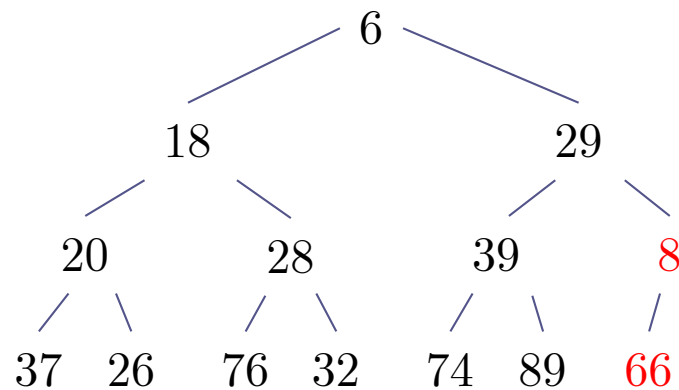
Inserting into a Heap Implemented as an ArrayList (cont.)



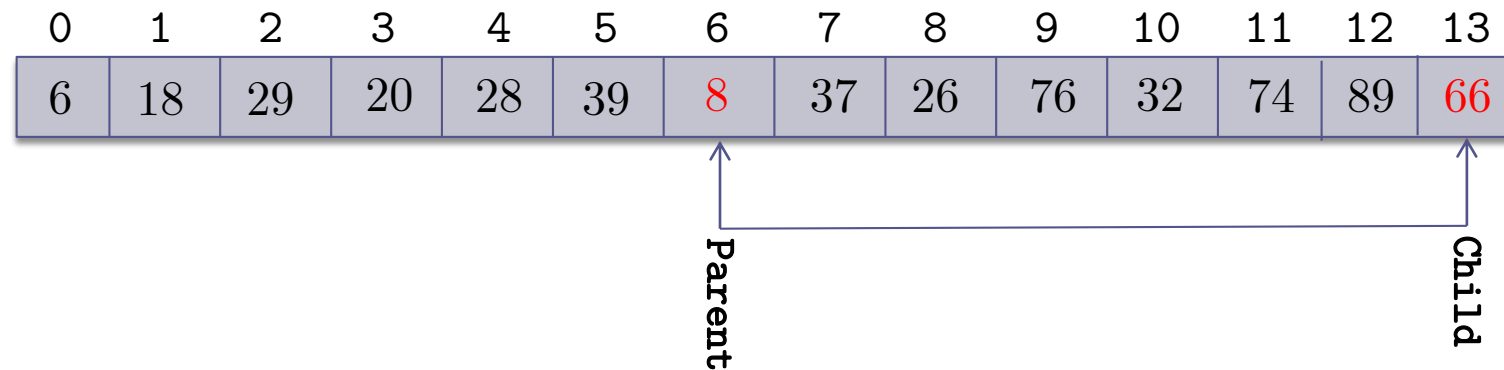
3. while (parent \geq 0
 and
 table[parent] > table[child])
4. Swap table[parent]
 and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2



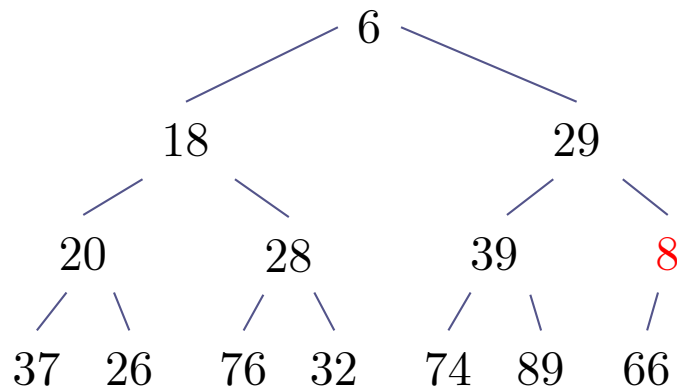
Inserting into a Heap Implemented as an ArrayList (cont.)



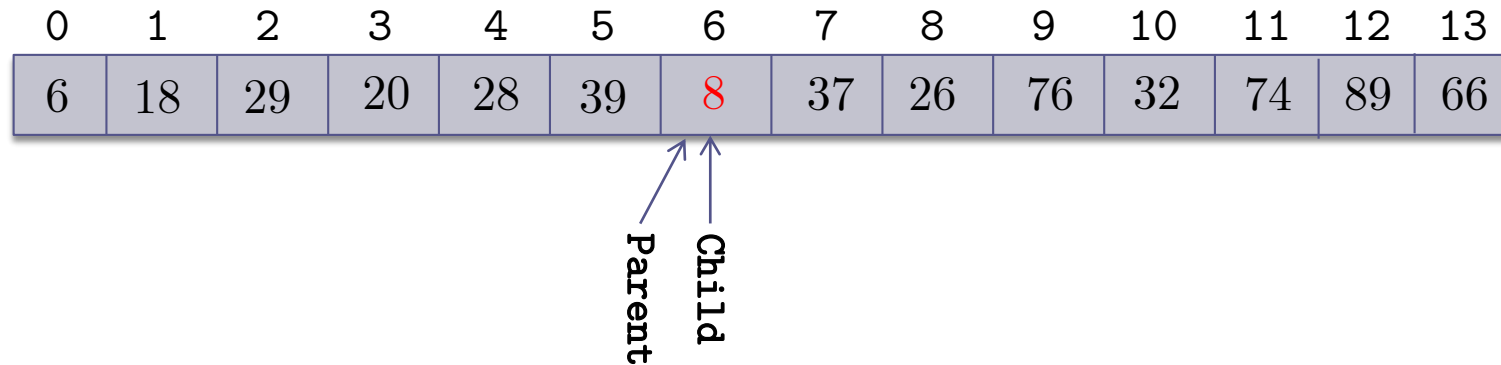
3. while (parent >= 0
and
table[parent] > table[child])
4. Swap table[parent]
and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2



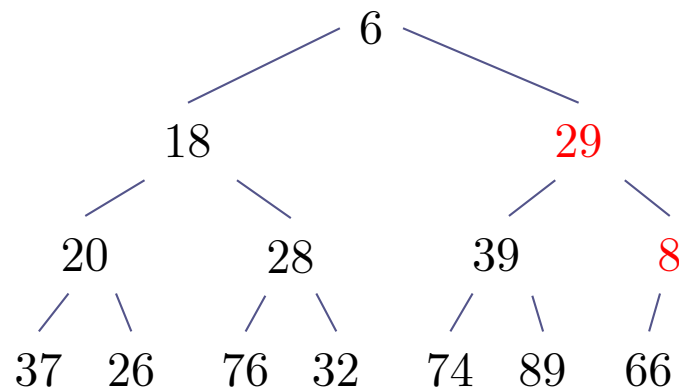
Inserting into a Heap Implemented as an ArrayList (cont.)



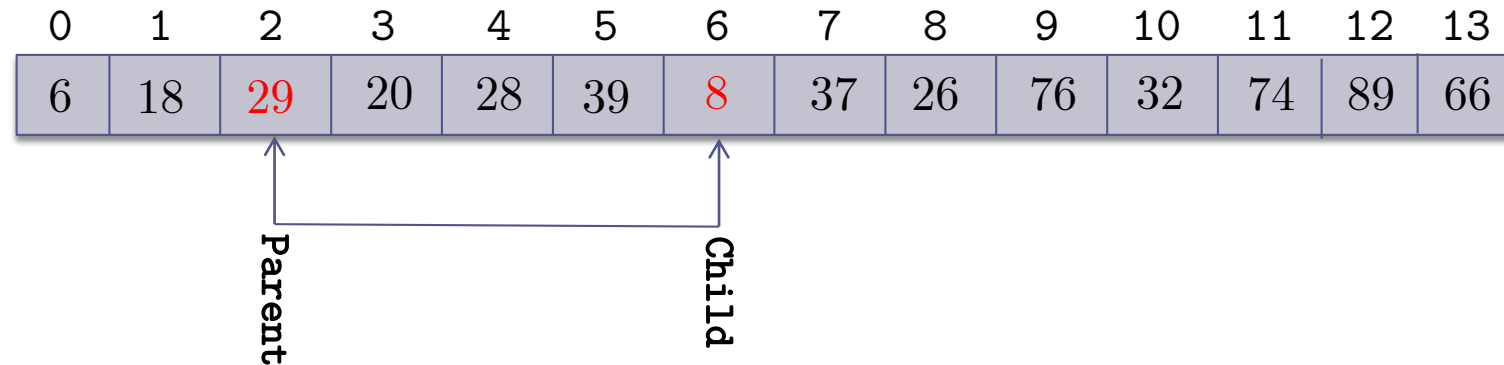
3. while (parent >= 0
 and
 table[parent] > table[child])
4. Swap table[parent]
 and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2



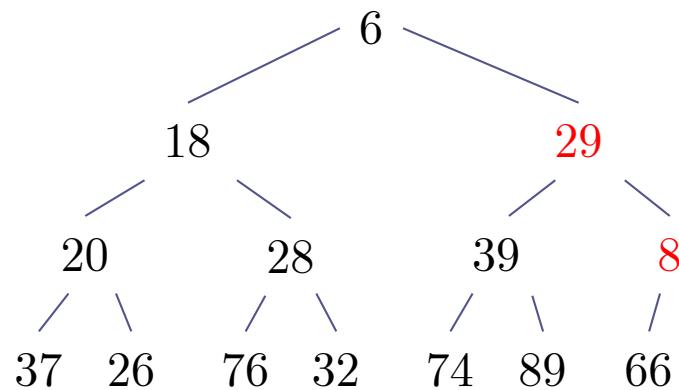
Inserting into a Heap Implemented as an ArrayList (cont.)



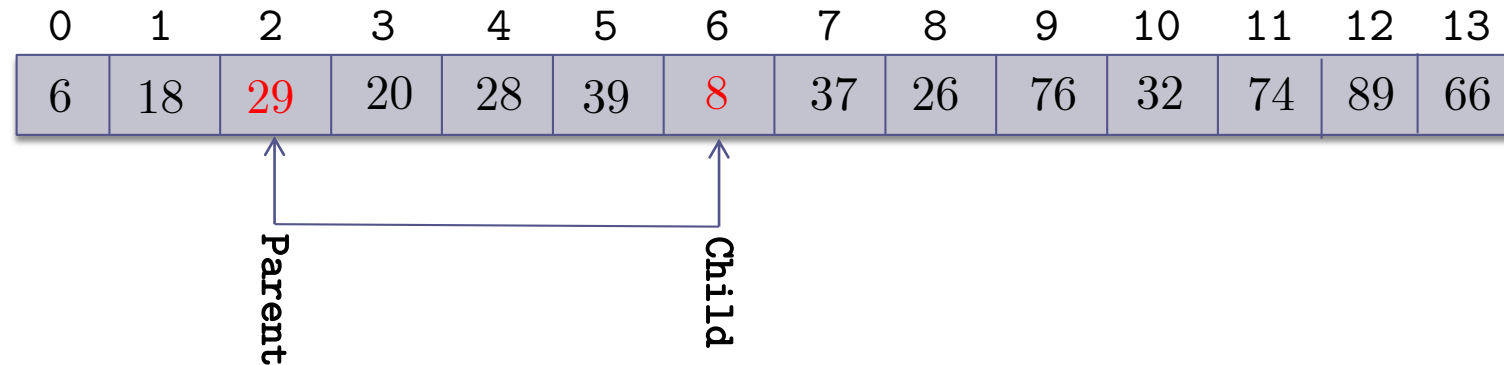
3. while (parent >= 0
and
table[parent] > table[child])
4. Swap table[parent]
and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2



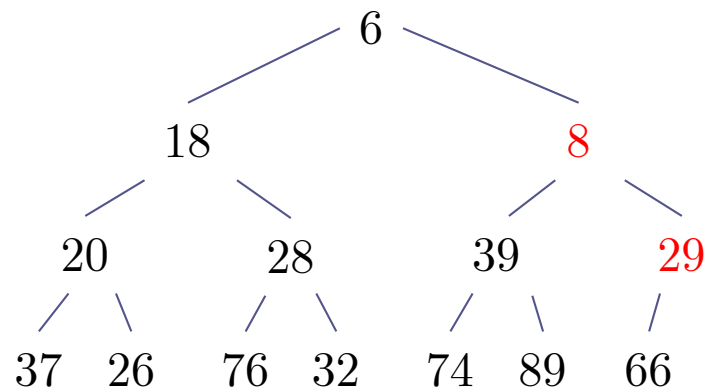
Inserting into a Heap Implemented as an ArrayList (cont.)



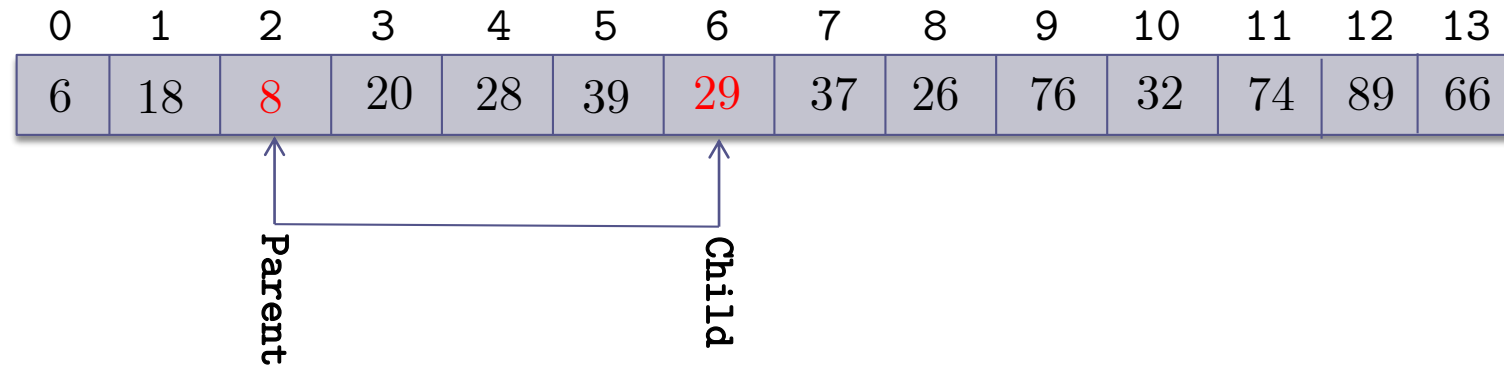
3. while (parent >= 0
and
table[parent] > table[child])
4. Swap table[parent]
and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2



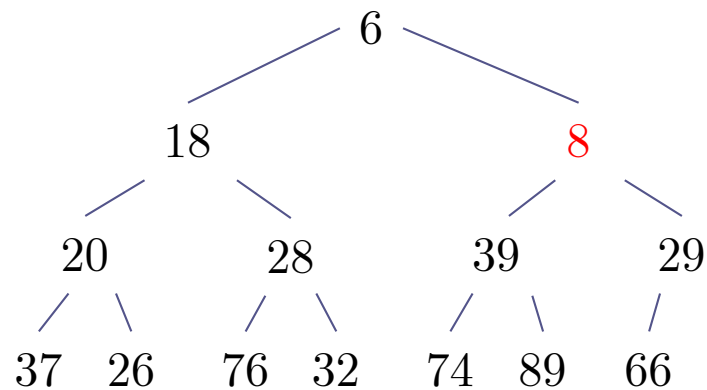
Inserting into a Heap Implemented as an ArrayList (cont.)



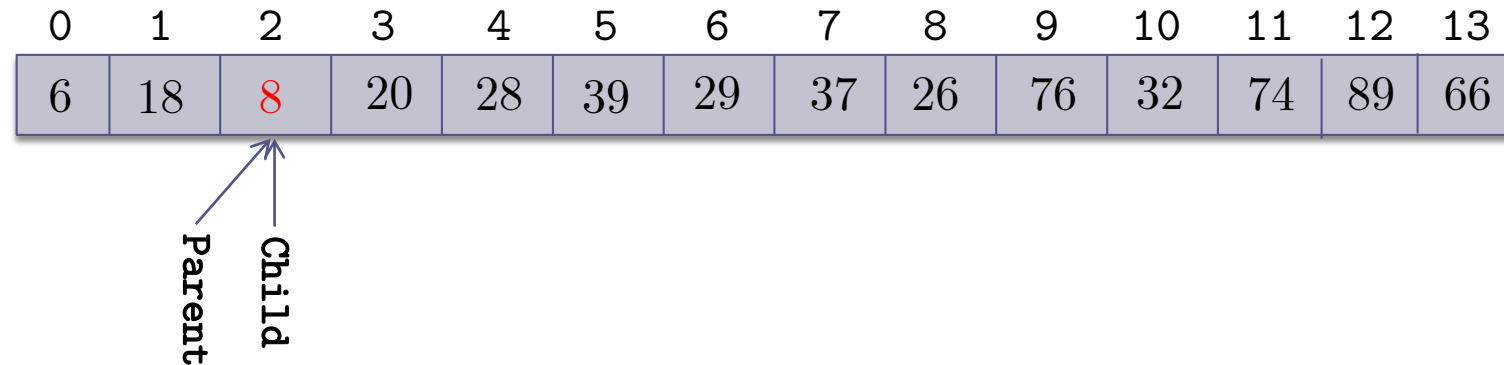
3. while (parent >= 0
and
table[parent] > table[child])
4. Swap table[parent]
and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2



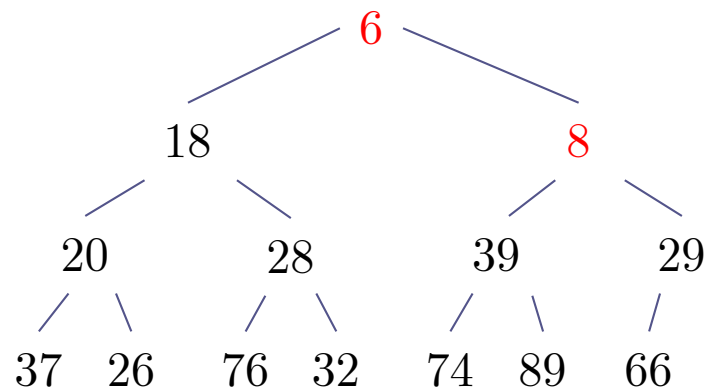
Inserting into a Heap Implemented as an ArrayList (cont.)



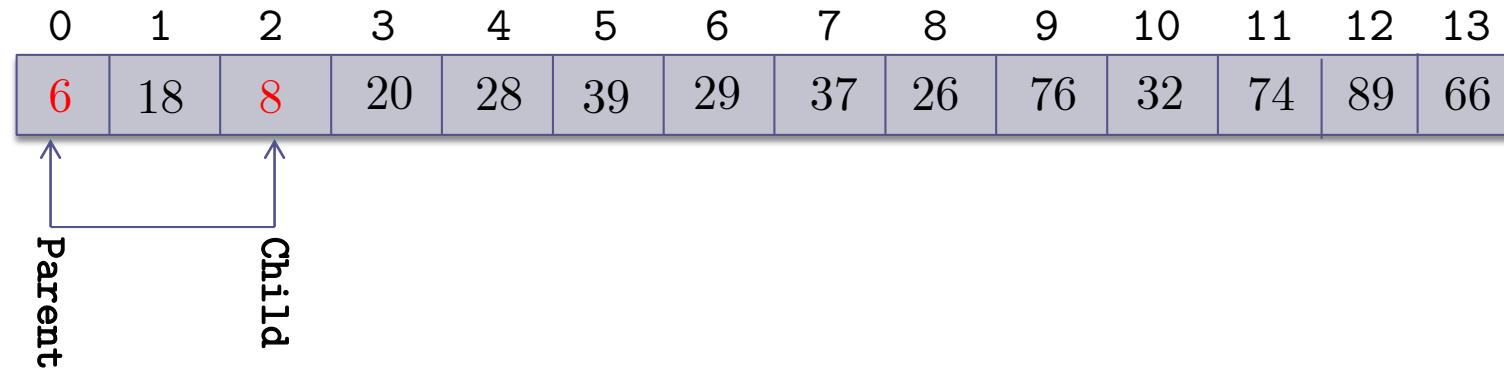
3. while (parent >= 0
and
table[parent] > table[child])
4. Swap table[parent]
and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2



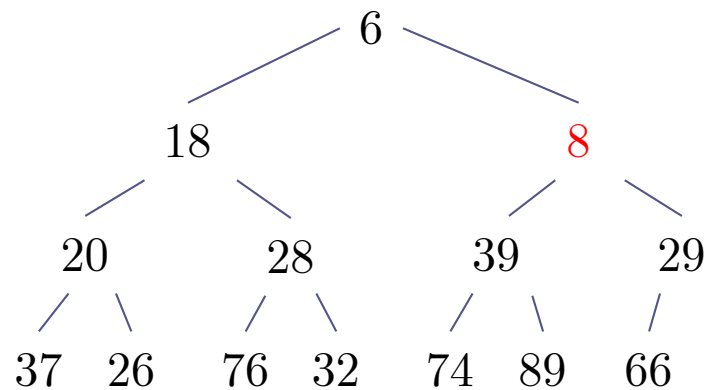
Inserting into a Heap Implemented as an ArrayList (cont.)



3. while (parent >= 0
 and
 table[parent] > table[child])
4. Swap table[parent]
 and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2



Inserting into a Heap Implemented as an ArrayList (cont.)



3. while (parent \geq 0
and
table[parent] > table[child])
4. Swap table[parent]
and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	8	20	28	39	29	37	26	76	32	74	89	66

Removal from a Heap Implemented as an ArrayList

Removing an Element from a Heap Implemented as an ArrayList

1. Remove the last element (i.e., the one at $\text{size}() - 1$) and set the item at 0 to this value.
2. Set `parent` to 0.
3. **while (true)**
4. Set `leftChild` to $(2 * \text{parent}) + 1$ and `rightChild` to $\text{leftChild} + 1$.
5. **if** `leftChild >= table.size()`
6. Break out of loop.
7. Assume `minChild` (the smaller child) is `leftChild`.
8. **if** `rightChild < table.size()` and
 `table[rightChild] < table[leftChild]`
9. Set `minChild` to `rightChild`.
10. **if** `table[parent] > table[minChild]`
11. Swap `table[parent]` and `table[minChild]`.
12. Set `parent` to `minChild`.
- else**
13. Break out of loop.

Performance of the Heap

- `remove` traces a path from the root to a leaf
- `insert` traces a path from a leaf to the root
- This requires at most h steps where h is the height of the tree
- The largest *full* tree of height h has $2^h - 1$ nodes
- The smallest *complete* tree of height h has $2^{(h-1)}$ nodes
- Both `insert` and `remove` are $O(\log n)$

Priority Queues

- The heap is used to implement a special kind of queue called a priority queue
- The heap is not very useful as an ADT on its own
 - ▣ We will not create a `Heap` interface or code a class that implements it
 - ▣ Instead, we will incorporate its algorithms when we implement a priority queue class and heapsort
- Sometimes a FIFO queue may not be the best way to implement a waiting line
- A priority queue is a data structure in which only the highest-priority item is accessible

Priority Queues (cont.)

- In a print queue, sometimes it is more appropriate to print a short document that arrived after a very long document
- A *priority queue* is a data structure in which only the highest-priority item is accessible (as opposed to the first item entered)

Insertion into a Priority Queue

pages = 1
title = "web page 1"

pages = 4
title = "history paper"

After inserting document with 3 pages

pages = 1
title = "web page 1"

pages = 3
title = "Lab1"

pages = 4
title = "history paper"

After inserting document with 1 page

pages = 1
title = "web page 1"

pages = 1
title = "receipt"

pages = 3
title = "Lab1"

pages = 4
title = "history paper"

PriorityQueue Class

- Java provides a `PriorityQueue<E>` class that implements the `Queue<E>` interface given in Chapter 4.

Method	Behavior
<code>boolean offer(E item)</code>	Inserts an item into the queue. Returns true if successful; returns false if the item could not be inserted.
<code>E remove()</code>	Removes the smallest entry and returns it if the queue is not empty. If the queue is empty, throws a <code>NoSuchElementException</code> .
<code>E poll()</code>	Removes the smallest entry and returns it. If the queue is empty, returns null .
<code>E peek()</code>	Returns the smallest entry without removing it. If the queue is empty, returns null .
<code>E element()</code>	Returns the smallest entry without removing it. If the queue is empty, throws a <code>NoSuchElementException</code> .

Using a Heap as the Basis of a Priority Queue

- In a priority queue, just like a heap, the smallest item always is removed first
- Because heap insertion and removal is $O(\log n)$, a heap can be the basis of a very efficient implementation of a priority queue
- While the `java.util.PriorityQueue` uses an `Object[]` array, we will use an `ArrayList` for our custom priority queue, `KWPriorityQueue`

Design of a KWPriorityQueue Class

Data Field	Attribute
<code>ArrayList<E> theData</code>	An <code>ArrayList</code> to hold the data.
<code>Comparator<E> comparator</code>	An optional object that implements the <code>Comparator<E></code> interface by providing a <code>compare</code> method.
Method	Behavior
<code>KWPriorityQueue()</code>	Constructs a heap-based priority queue that uses the elements' natural ordering.
<code>KWPriorityQueue (Comparator<E> comp)</code>	Constructs a heap-based priority queue that uses the <code>compare</code> method of <code>Comparator comp</code> to determine the ordering of the elements.
<code>private int compare(E left, E right)</code>	Compares two objects and returns a negative number if object <code>left</code> is less than object <code>right</code> , zero if they are equal, and a positive number if object <code>left</code> is greater than object <code>right</code> .
<code>private void swap(int i, int j)</code>	Exchanges the object references in <code>theData</code> at indexes <code>i</code> and <code>j</code> .

Design of a KWPriorityQueue Class (cont.)

```
import java.util.*;

/** The KWPriorityQueue implements the Queue interface
    by building a heap in an ArrayList. The heap is structured
    so that the "smallest" item is at the top.
 */
public class KWPriorityQueue<E> extends AbstractQueue<E>
    implements Queue<E> {

    // Data Fields
    /** The ArrayList to hold the data. */
    private ArrayList<E> theData;
    /** An optional reference to a Comparator object. */
    Comparator<E> comparator = null;

    // Methods
    // Constructor
    public KWPriorityQueue() {
        theData = new ArrayList<E>();
    }
    . . .
```

offer Method

```
/** Insert an item into the priority queue.  
pre: The ArrayList theData is in heap order.  
post: The item is in the priority queue and  
     theData is in heap order.  
@param item The item to be inserted  
@throws NullPointerException if the item to be inserted is null.  
*/  
@Override  
public boolean offer(E item) {  
    // Add the item to the heap.  
    theData.add(item);  
    // child is newly inserted item.  
    int child = theData.size() - 1;  
    int parent = (child - 1) / 2; // Find child's parent.  
    // Reheap  
    while (parent >= 0 && compare(theData.get(parent),  
                                  theData.get(child)) > 0) {  
        swap(parent, child);  
        child = parent;  
        parent = (child - 1) / 2;  
    }  
    return true;  
}
```

poll Method

```
/** Remove an item from the priority queue  
pre: The ArrayList theData is in heap order.  
post: Removed smallest item, theData is in heap order.  
@return The item with the smallest priority value or null if empty.  
*/  
@Override  
public E poll() {  
    if (isEmpty()) {  
        return null;  
    }  
    // Save the top of the heap.  
    E result = theData.get(0);  
    // If only one item then remove it.  
    if (theData.size() == 1) {  
        theData.remove(0);  
        return result;  
    }  
    // Continues on the next slide
```

```

    /* Remove the last item from the ArrayList and place it into
       the first position. */
    theData.set(0, theData.remove(theData.size() - 1));
    // The parent starts at the top.
    int parent = 0;
    while (true) {
        int leftChild = 2 * parent + 1;
        if (leftChild >= theData.size()) {
            break; // Out of heap.
        }
        int rightChild = leftChild + 1;
        int minChild = leftChild; // Assume leftChild is smaller.
        // See whether rightChild is smaller.
        if (rightChild < theData.size()
            && compare(theData.get(leftChild),
                       theData.get(rightChild)) > 0) {
            minChild = rightChild;
        }
        // assert: minChild is the index of the smaller child.
        // Move smaller child up heap if necessary.
        if (compare(theData.get(parent),
                    theData.get(minChild)) > 0) {
            swap(parent, minChild);
            parent = minChild;
        } else { // Heap property is restored.
            break;
        }
    }
    return result;
}

```

Other Methods

- The `iterator` and `size` methods are implemented via delegation to the corresponding `ArrayList` methods
- Method `isEmpty` tests whether the result of calling method `size` is 0 and is inherited from class `AbstractCollection`
- The implementations of methods `peek` and `remove` are left as exercises

Using a Comparator

- To use an ordering that is different from the natural ordering, provide a constructor that has a `Comparator<E>` parameter

```
/** Creates a heap-based priority queue with the specified initial  
    capacity that orders its elements according to the specified  
    comparator.  
    @param cap The initial capacity for this priority queue  
    @param comp The comparator used to order this priority queue  
    @throws IllegalArgumentException if cap is less than 1  
    */  
public KWPriorityQueue(int cap, Comparator<E> comp) {  
    if (cap < 1)  
        throw new IllegalArgumentException();  
    theData = new ArrayList<E>();  
    comparator = comp;  
}
```

compare Method

- ❑ If data field `comparator` references a `Comparator<E>` object, method `compare` delegates the task to the object's `compare` method
- ❑ If `comparator` is `null`, it will delegate to method `compareTo`

compare Method (cont.)

```
/** Compare two items using either a Comparator object's compare method
    or their natural ordering using method compareTo.
    pre: If comparator is null, left and right implement Comparable<E>.
    @param left One item
    @param right The other item
    @return Negative int if left less than right,
            0 if left equals right,
            positive int if left > right
    @throws ClassCastException if items are not Comparable
 */
private int compare(E left, E right) {
    if (comparator != null) { // A Comparator is defined.
        return comparator.compare(left, right);
    } else { // Use left's compareTo method.
        return ((Comparable<E>) left).compareTo(right);
    }
}
```

PrintDocuments Example

- The class `PrintDocument` is used to define documents to be printed on a printer
- We want to order documents by a value that is a function of both size and time submitted
- In the client program, use

```
Queue printQueue =  
    new PriorityQueue(new ComparePrintDocuments());
```

PrintDocuments Example (cont.)

ComparePrintDocuments.java

```
import java.util.Comparator;

/** Class to compare PrintDocuments based on both
    their size and time stamp.
 */
public class ComparePrintDocuments implements Comparator<PrintDocument> {
    /** Weight factor for size. */
    private static final double P1 = 0.8;
    /** Weight factor for time. */
    private static final double P2 = 0.2;

    /** Compare two PrintDocuments.
        @param left The left-hand side of the comparison
        @param right The right-hand side of the comparison
        @return -1 if left < right; 0 if left == right;
                and +1 if left > right
    */
    public int compare(PrintDocument left, PrintDocument right) {
        return Double.compare(orderValue(left), orderValue(right));
    }

    /** Compute the order value for a print document.
        @param pd The PrintDocument
        @return The order value based on the size and time stamp
    */
    private double orderValue(PrintDocument pd) {
        return P1 * pd.getSize() + P2 * pd.getTimeStamp();
    }
}
```



Huffman Trees

Section 6.6

Huffman Trees

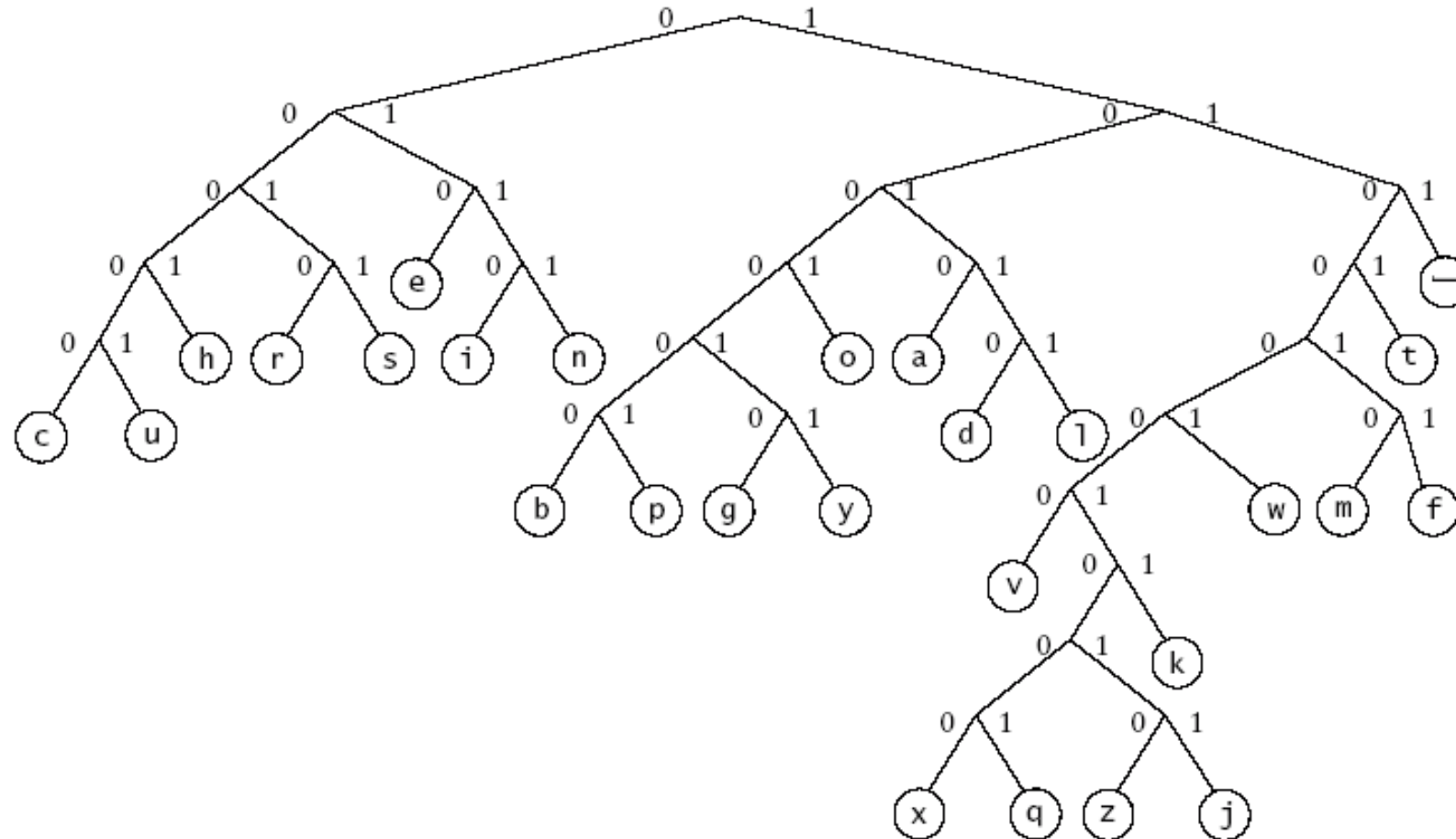
- A Huffman tree can be implemented using a binary tree and a PriorityQueue
- A straight binary encoding of an alphabet assigns a unique binary number to each symbol in the alphabet
 - ▣ Unicode is an example of such a coding
- The message “go eagles” requires 144 bits in Unicode but only 38 bits using Huffman coding

Huffman Trees (cont.)

Symbol	Frequency	Symbol	Frequency	Symbol	Frequency
—	186	h	47	g	15
e	103	d	32	p	15
t	80	l	32	b	13
a	64	u	23	v	8
o	63	c	22	k	5
i	57	f	21	j	1
n	57	m	20	q	1
s	51	w	18	x	1
r	48	y	16	z	1

Huffman Trees (cont.)

Huffman Tree Based on Frequency of Letters in English Text



Building a Custom Huffman Tree

- Suppose we want to build a custom Huffman tree for a file
- **Input:** an array of objects such that each object contains a reference to a symbol occurring in that file and the frequency of occurrence (weight) for the symbol in that file

Building a Custom Huffman Tree (cont.)

□ Analysis:

- ▣ Each node will have storage for two data items:
 - the weight of the node and
 - the symbol associated with the node
- ▣ All symbols will be stored in leaf nodes
- ▣ For nodes that are not leaf nodes, the symbol part has no meaning
- ▣ The weight of a leaf node will be the frequency of the symbol stored at that node
- ▣ The weight of an interior node will be the sum of frequencies of all leaf nodes in the subtree rooted at the interior node

Building a Custom Huffman Tree (cont.)

□ Analysis:

- ▣ A priority queue will be the key data structure in our Huffman tree
- ▣ We will store individual symbols and subtrees of multiple symbols in order by their priority (frequency of occurrence)

Building a Custom Huffman Tree (cont.)

13	22	32	64	103
b	c	d	a	e

32	35	64	103
d		a	e

13	22
b	c

64	67	103
a		e

32	35
d	

13	22
b	c

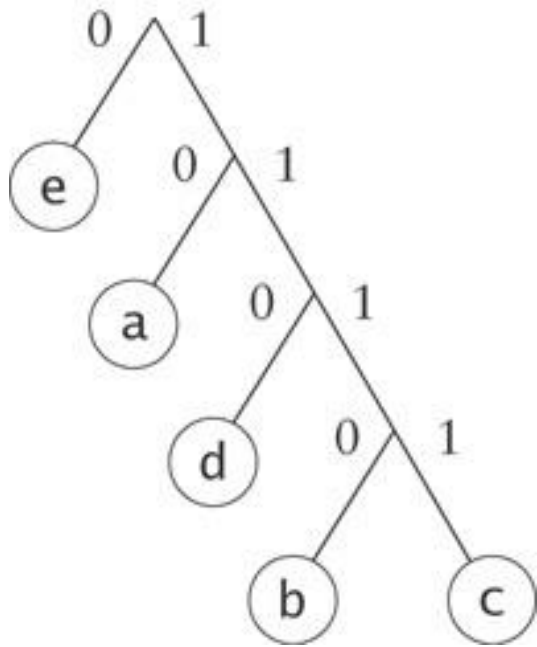
103	131
e	

64	67
a	

32	35
d	

13	22
b	c

Building a Custom Huffman Tree (cont.)



Symbol	Code
a	10
b	1110
c	1111
d	110
e	0

Design

Algorithm for Building a Huffman Tree

1. Construct a set of trees with root nodes that contain each of the individual symbols and their weights.
2. Place the set of trees into a priority queue.
3. **while** the priority queue has more than one item
 4. Remove the two trees with the smallest weights.
 5. Combine them into a new binary tree in which the weight of the tree root is the sum of the weights of its children.
 6. Insert the newly created tree back into the priority queue.

Design (cont.)

Data Field	Attribute
BinaryTree<HuffData> huffTree	A reference to the Huffman tree.
Method	Behavior
buildTree(HuffData[] input)	Builds the Huffman tree using the given alphabet and weights.
String decode(String message)	Decodes a message using the generated Huffman tree.
printCode(PrintStream out)	Outputs the resulting code.

Implementation

- Listing 6.9 (Class HuffmanTree; page 349)
- Listing 6.10 (The buildTree Method (HuffmanTree.java); pages 350-351)
- Listing 6.11 (The decode Method (HuffmanTree.java); page 352)