Trees

Based on Chapter 8,

Koffmann and Wolfgang

Chapter Outline

- How trees organize information ***hierarchically***
- Using *recursion* to process trees
- The different ways of *traversing* a tree
- *Binary* trees, *Binary Search* trees, and *Heaps*
- Implementing binary trees, binary search trees, heaps
  - Using *linked data structures* and *arrays*
- Using binary search trees for efficient data lookup
- Using *Huffman trees* for compact character storage

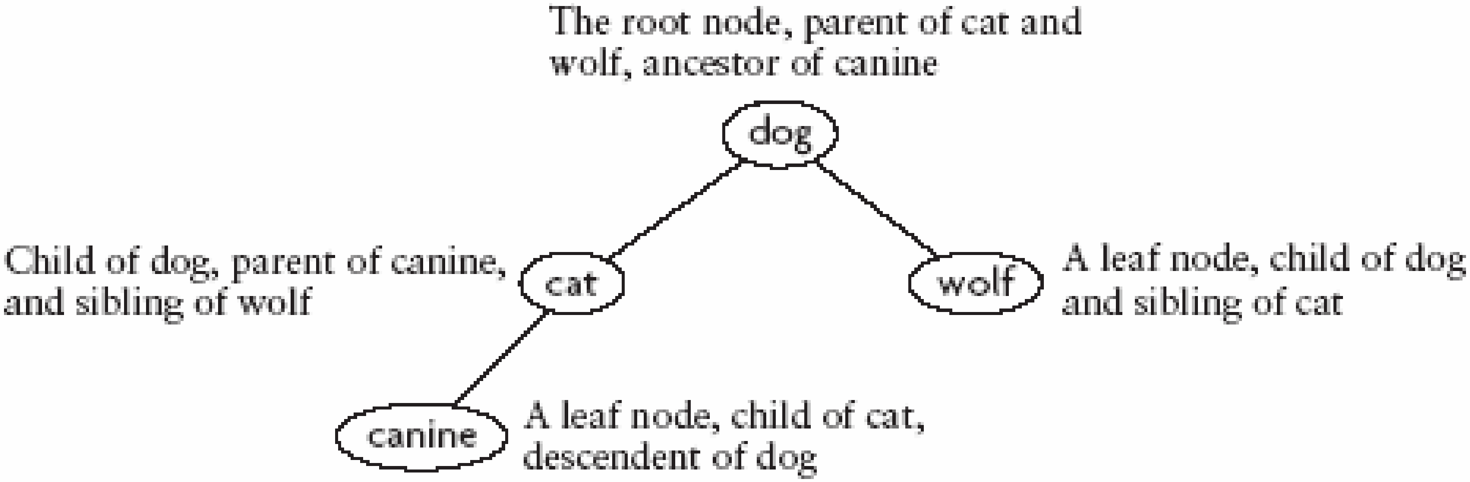
Tree Terminology

- A *tree* is a collection of elements (nodes)
- Each node may have 0 or more *successors*
  - (Unlike a list, which has 0 or 1 successor)
- Each node has *exactly one* *predecessor*
  - Except the starting / top node, called the *root*
- Links from node to its successors are called *branches*
- Successors of a node are called its *children*
- Predecessor of a node is called its *parent*
- Nodes with same parent are *siblings*
- Nodes with no children are called *leaves*

Tree Terminology (2)

- We also use words like *ancestor* and *descendent*

FIGURE 8.2
A Tree of Words



Tree Terminology (3)

- *Subtree* of a node:

  A tree whose root is a child of that node
- *Level* of a node:

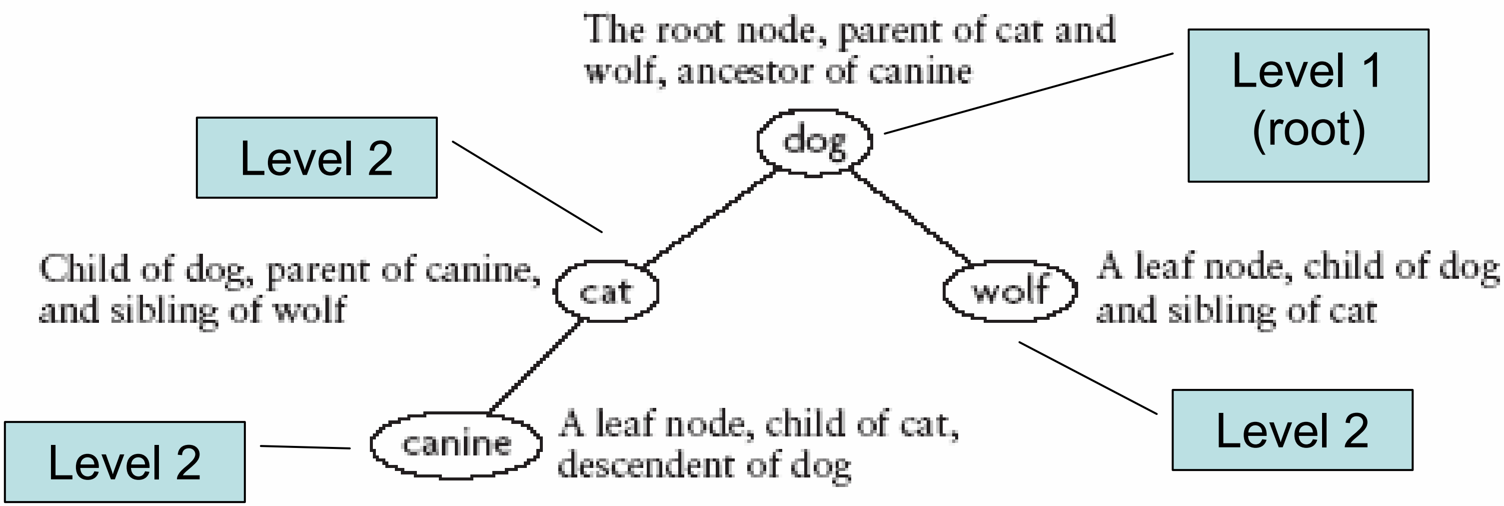
  A measure of its distance from the root:

  Level of the root = 1

  Level of other nodes = 1 + level of parent

Tree Terminology (4)

FIGURE 8.2
A Tree of Words

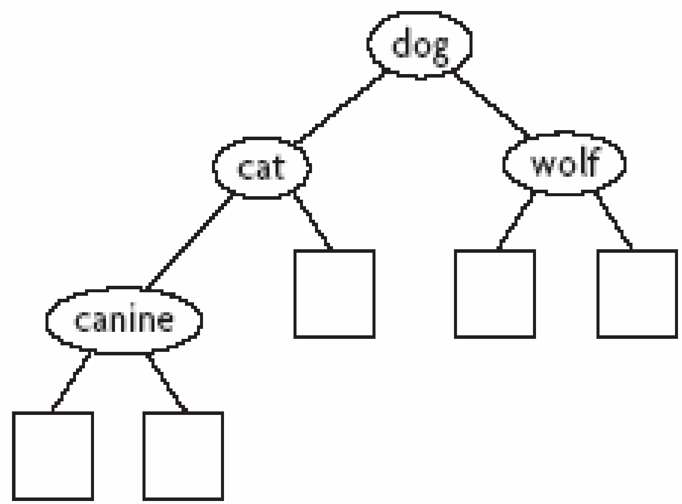


Binary Trees

- *<u>Binary</u>* tree: a node has *<u>at most 2</u>* non-empty subtrees
- Set of nodes T is a binary tree if either of these is true:
  - T is empty
  - Root of T has two subtrees, both binary trees
  - (Notice that this is a recursive definition)

FIGURE 8.3
A Tree of Words with Null Subtrees Indicated

This is a *binary* tree



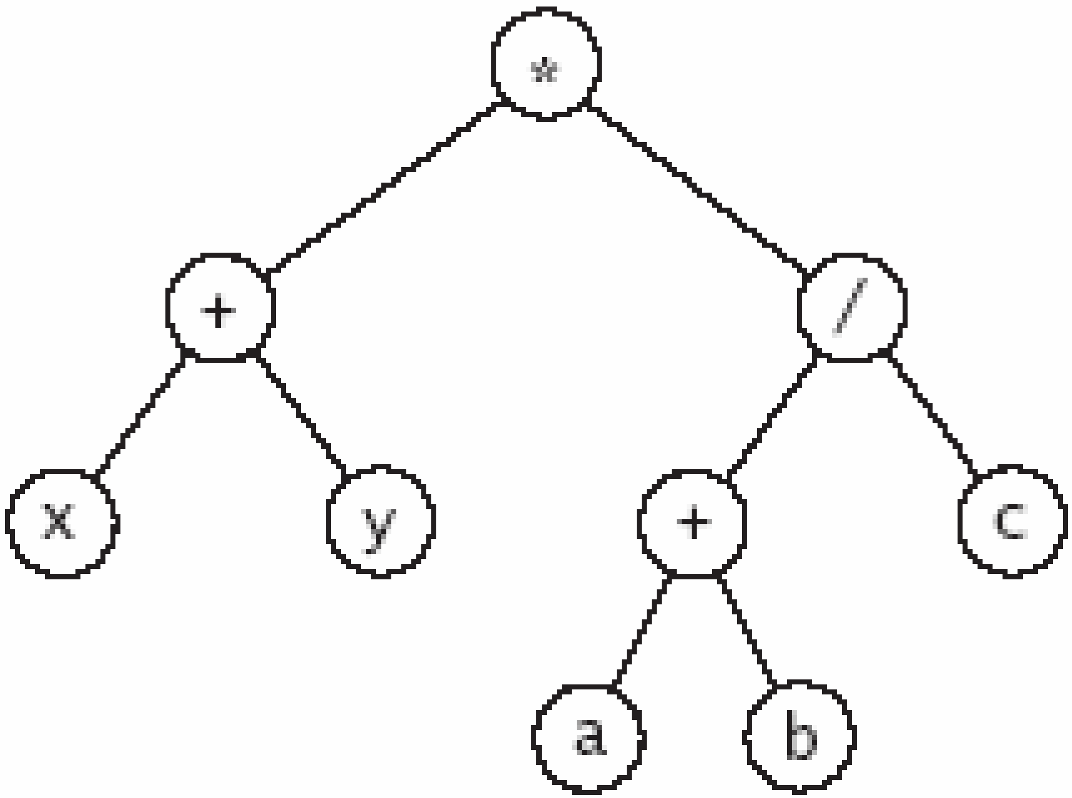
Examples of Binary Trees

- *Expression tree*
  - Non-leaf (*internal*) nodes contain operators
  - Leaf nodes contain operands
- *Huffman tree*
  - Represents *Huffman codes* for characters appearing in a file or stream
  - Huffman code may use different numbers of bits to encode different characters
  - ASCII or Unicode uses a *fixed number* of bits for each character

Examples of Binary Trees (2)

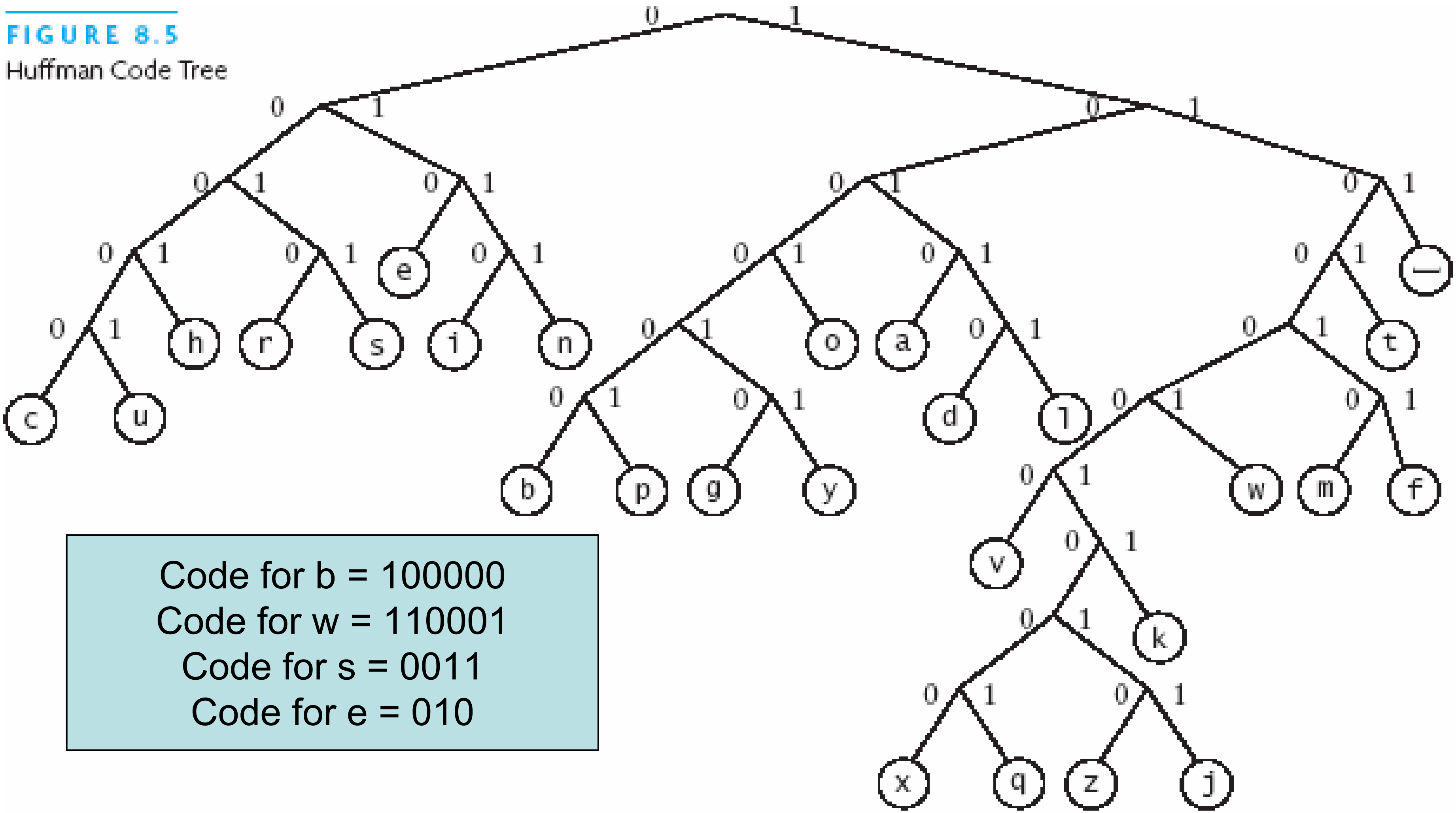
FIGURE 8.4

Expression Tree



Examples of Binary Trees (3)

FIGURE 8.5
Huffman Code Tree



Code for b = 100000
Code for w = 110001
Code for s = 0011
Code for e = 010

Examples of Binary Trees (4)

- *Binary search trees*
  - Elements in *left subtree* < element in subtree root
  - Elements in *right subtree* > element in subtree root

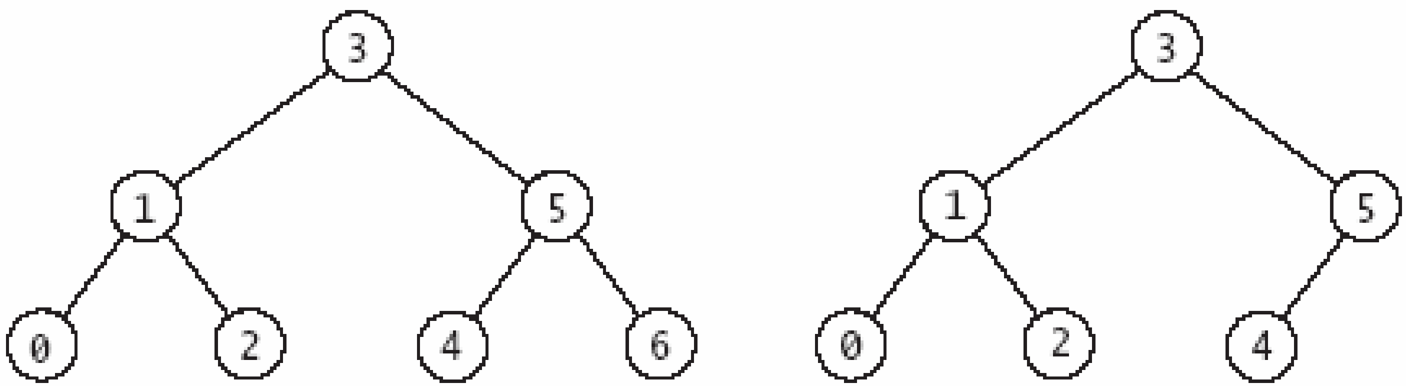
Search algorithm:

1. if the tree is empty, return **null**
2. if target equals to root node's data, return that data
3. if target < root node data, return search(left subtree)
4. otherwise return search(right subtree)

Fullness and Completeness

- (In computer science) trees grow from the *top down*
- New values inserted in new leaf nodes
- A binary tree is *full* if all leaves are at the *same level*
- In a full tree, every node has 0 or 2 non-null children

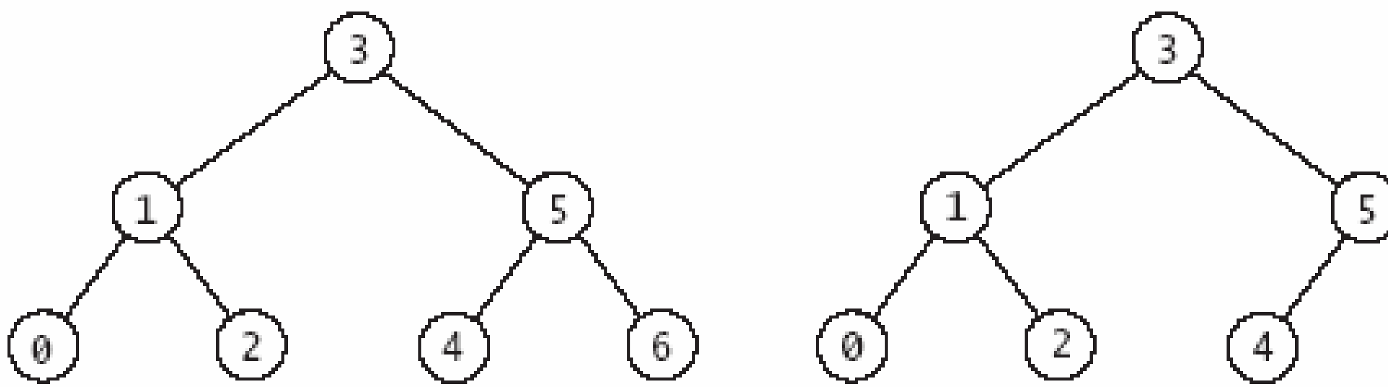
FIGURE 8.6
Full Binary Tree (Left) and Complete Binary Tree (Right) of Height 3



Fullness and Completeness (2)

- A binary tree is *complete* if:
  - All leaves are at level *h* or level *h-1* (for some *h*)
  - All level *h-1* leaves are to the right

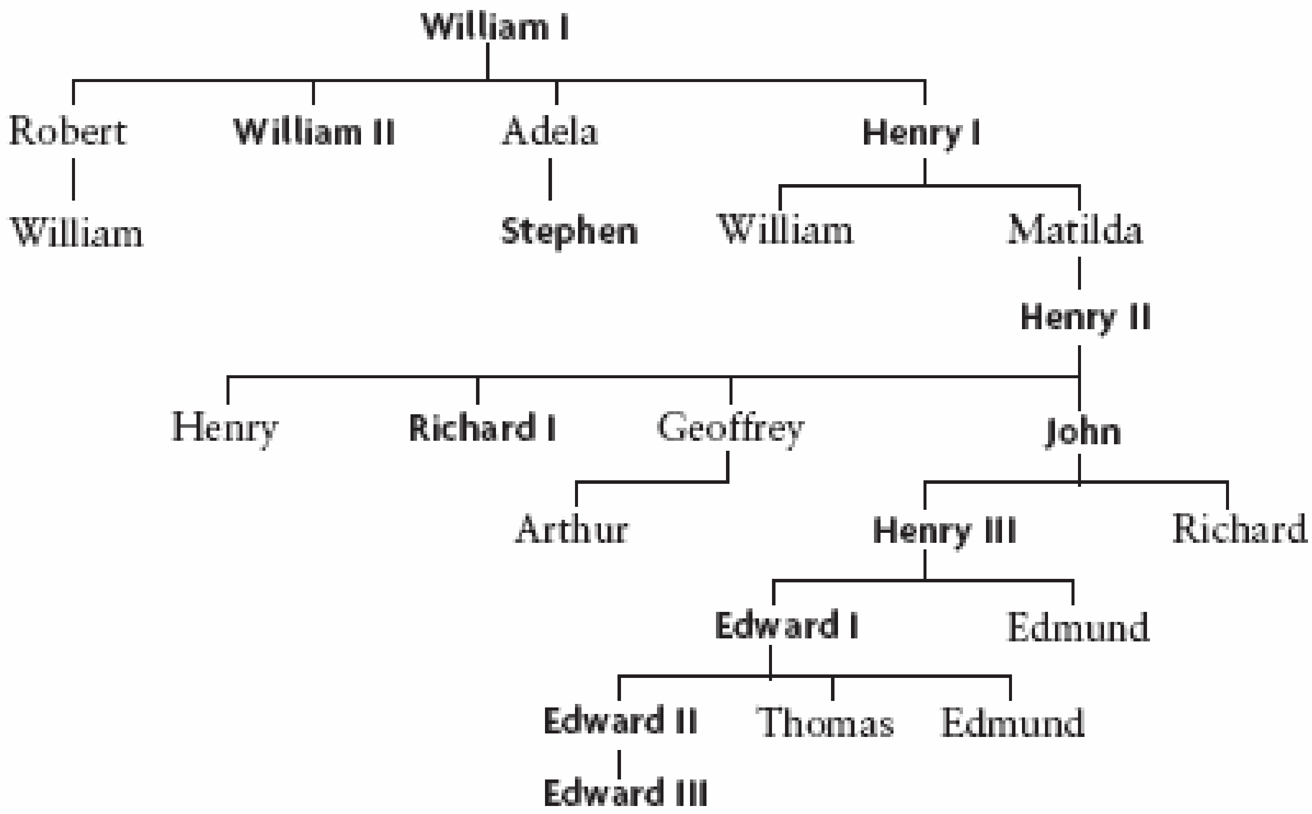
FIGURE 8.6
Full Binary Tree (Left) and Complete Binary Tree (Right) of Height 3



General (Non-Binary) Trees

- Nodes can have any number of children

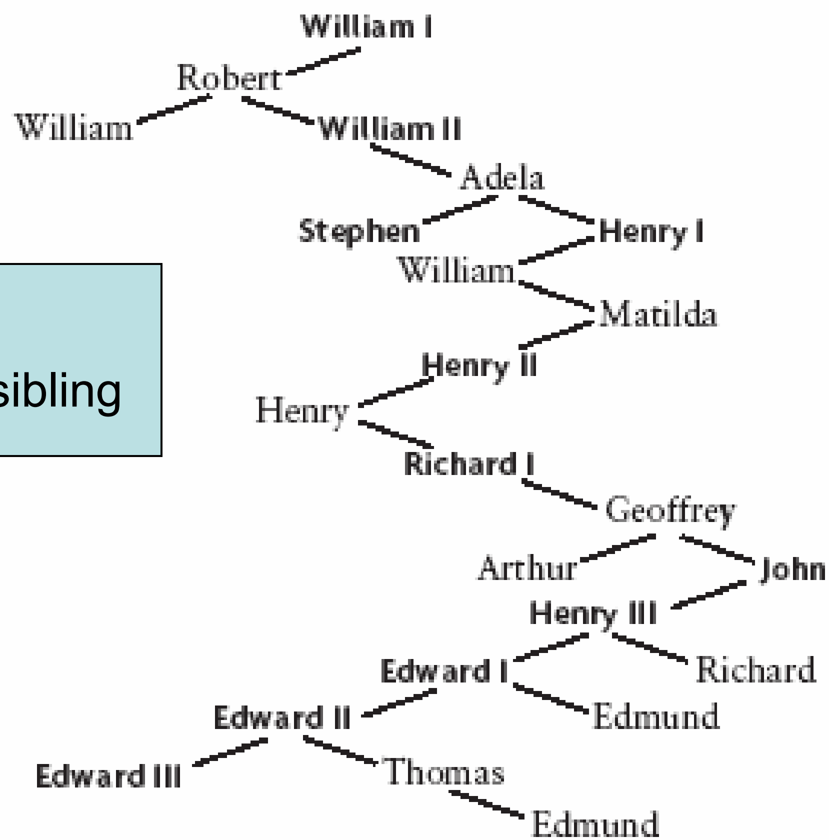
FIGURE 8.7
Family Tree for the Descendants of William I of England



General (Non-Binary) Trees (2)

- A general tree can be *__represented__* using a binary tree

FIGURE 8.8
Binary Tree Equivalent of King William's Family Tree



Left link is to first child
Right link is to next younger sibling

Traversals of Binary Trees

- Often want iterate over and process nodes of a tree
  - Can *walk* the tree and *visit* the nodes in order
  - This process is called ***tree traversal***
- Three kinds of binary tree traversal:
  - ***Pre***order
  - ***In***order
  - ***Post***order
- According to order of subtree ***root*** w.r.t. its ***children***

Binary Tree Traversals (2)

- Preorder: Visit root, traverse left, traverse right
- Inorder: Traverse left, visit root, traverse right
- Postorder: Traverse left, traverse right, visit root

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

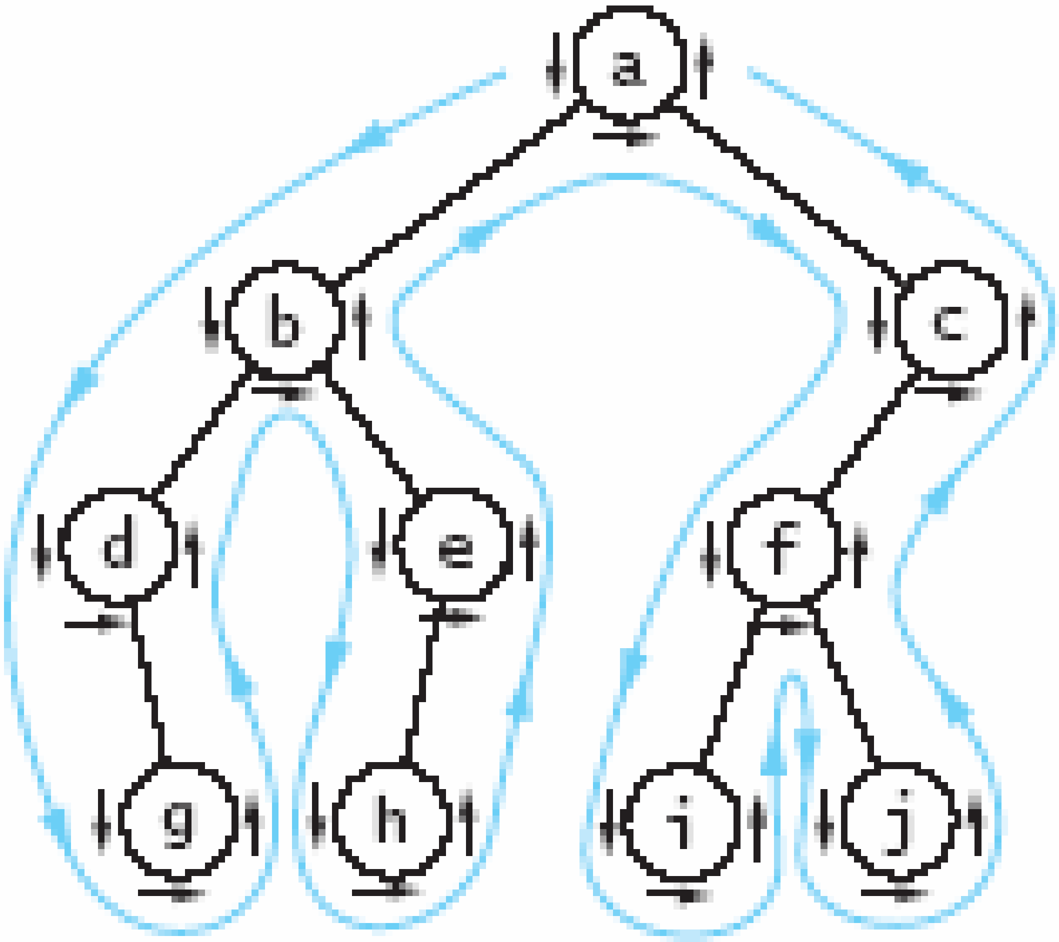
3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

Visualizing Tree Traversals

- Can visualize traversal by imagining a mouse that walks along outside the tree
- If mouse keeps the tree on its left, it traces a route called *the Euler tour:*
- *Preorder:* record node first time mouse is there
- *Inorder:* record after mouse traverses left subtree
- *Postorder:* record node last time mouse is there

Visualizing Tree Traversals (2)

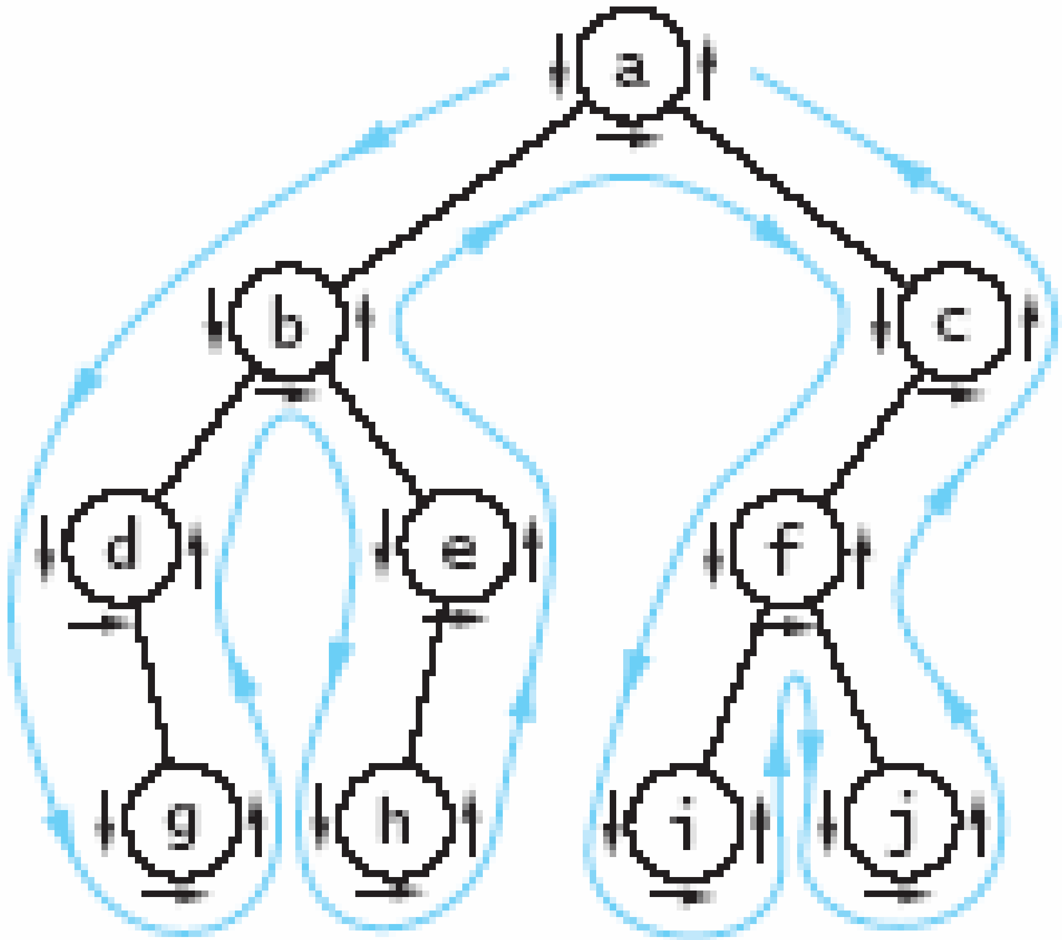
FIGURE 8.9
Traversal of a Binary Tree



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

Visualizing Tree Traversals (3)

FIGURE 8.9
Traversal of a Binary Tree

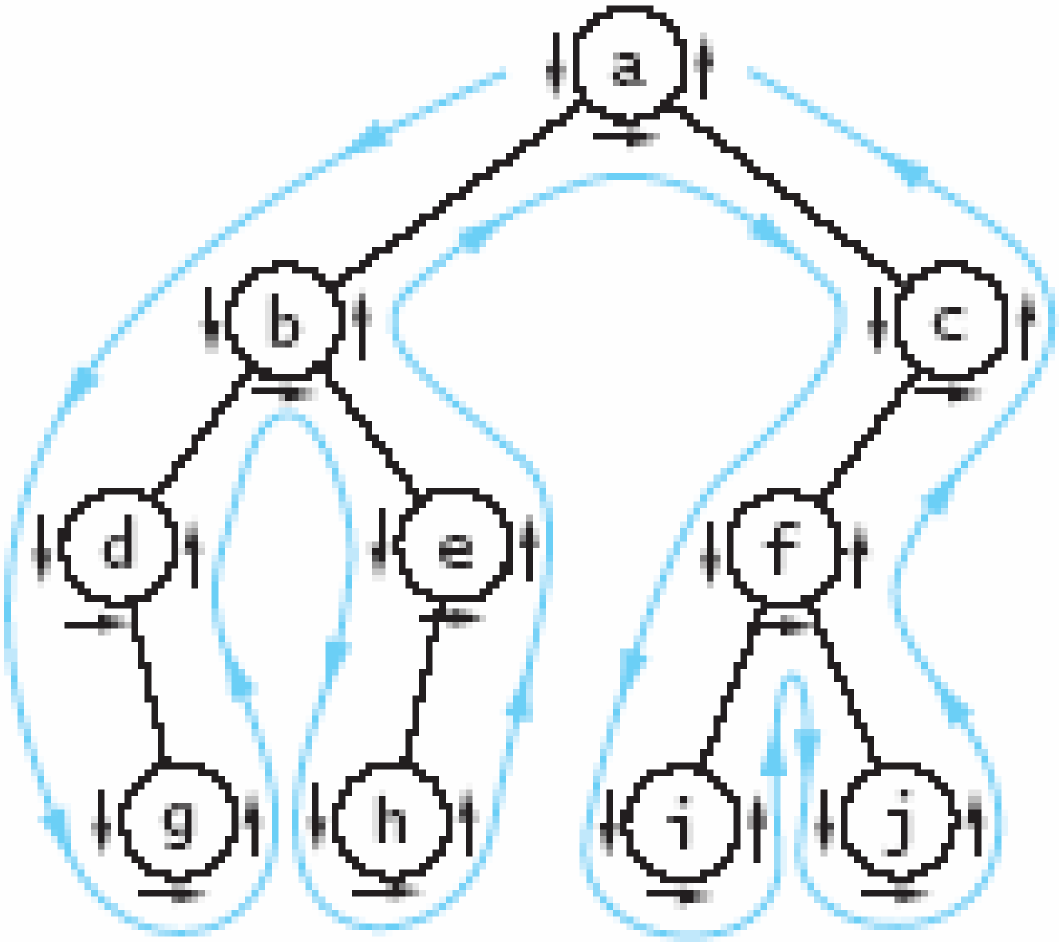


Inorder:
d, g, b, h, e,
a, i, f, j, c

Visualizing Tree Traversals (4)

FIGURE 8.9

Traversal of a Binary Tree

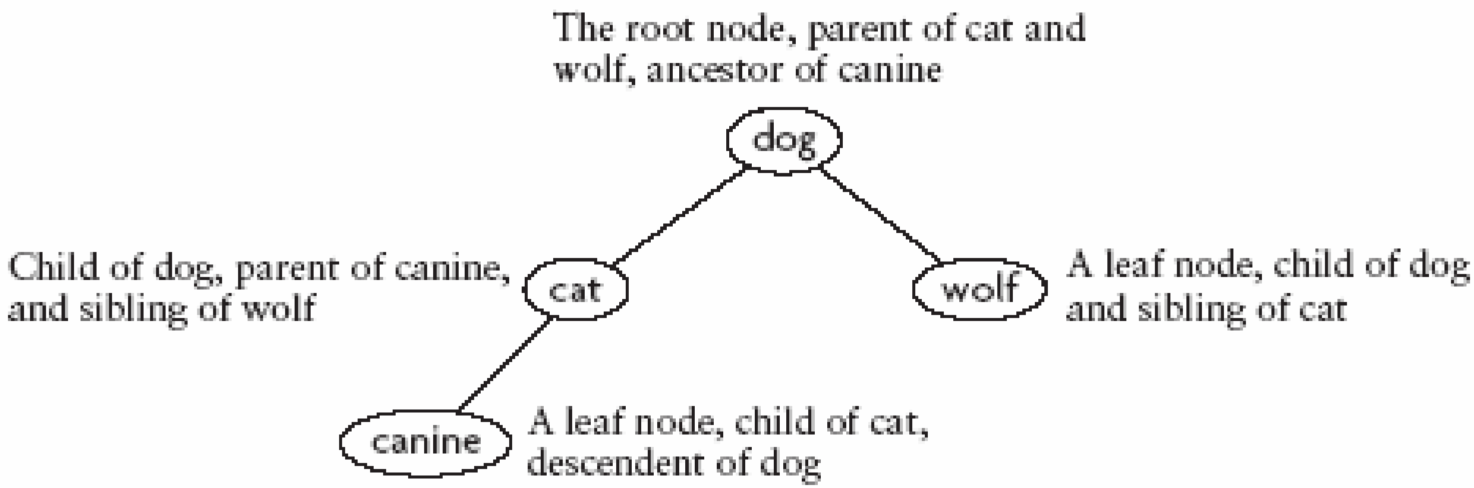


Postorder:
g, d, h, e, b,
i, j, f, c, a

Traversals of Binary Search Trees

- Inorder traversal of a binary search tree → Nodes are visited in order of increasing data value

FIGURE 8.2
A Tree of Words



Inorder traversal visits in this order: canine, cat, dog, wolf

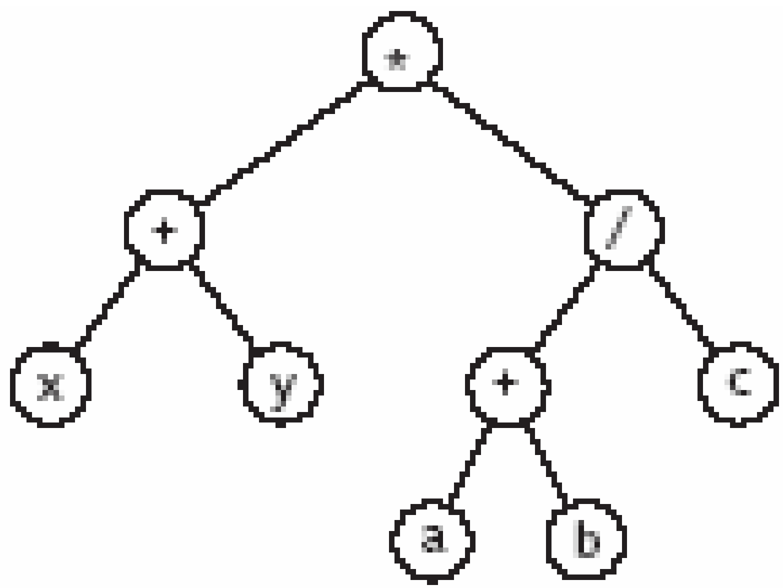
Traversals of Expression Trees

- *Inorder traversal* can insert parentheses where they belong for infix form
- *Postorder traversal* results in postfix form
- *Prefix traversal* results in prefix form

Infix form

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

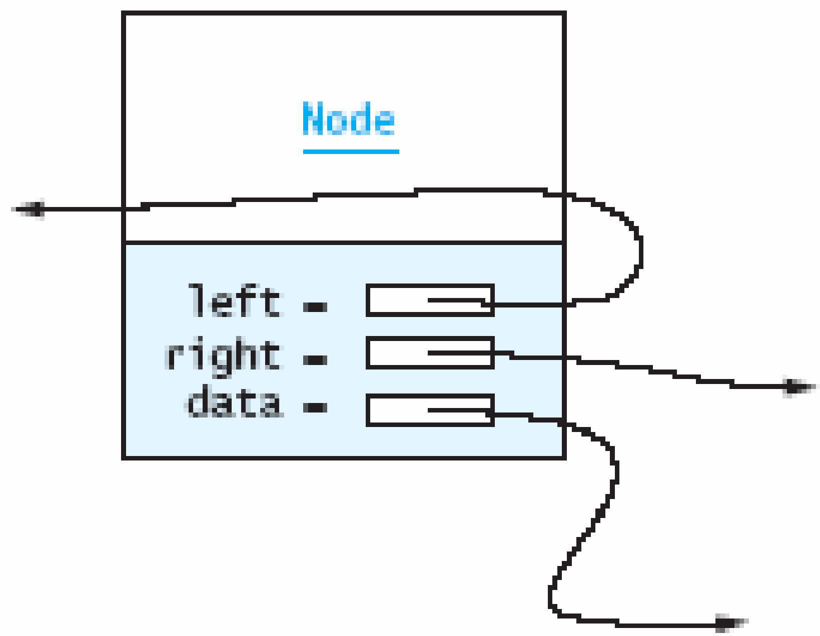
Postfix form: x y + a b + c / *
Prefix form: * + x y / + a b c



The Node<E> Class

- Like linked list, a node has *data* + *links to successors*
- *Data* is reference to object of type **E**
- Binary tree node has links for *left* and *right* subtrees

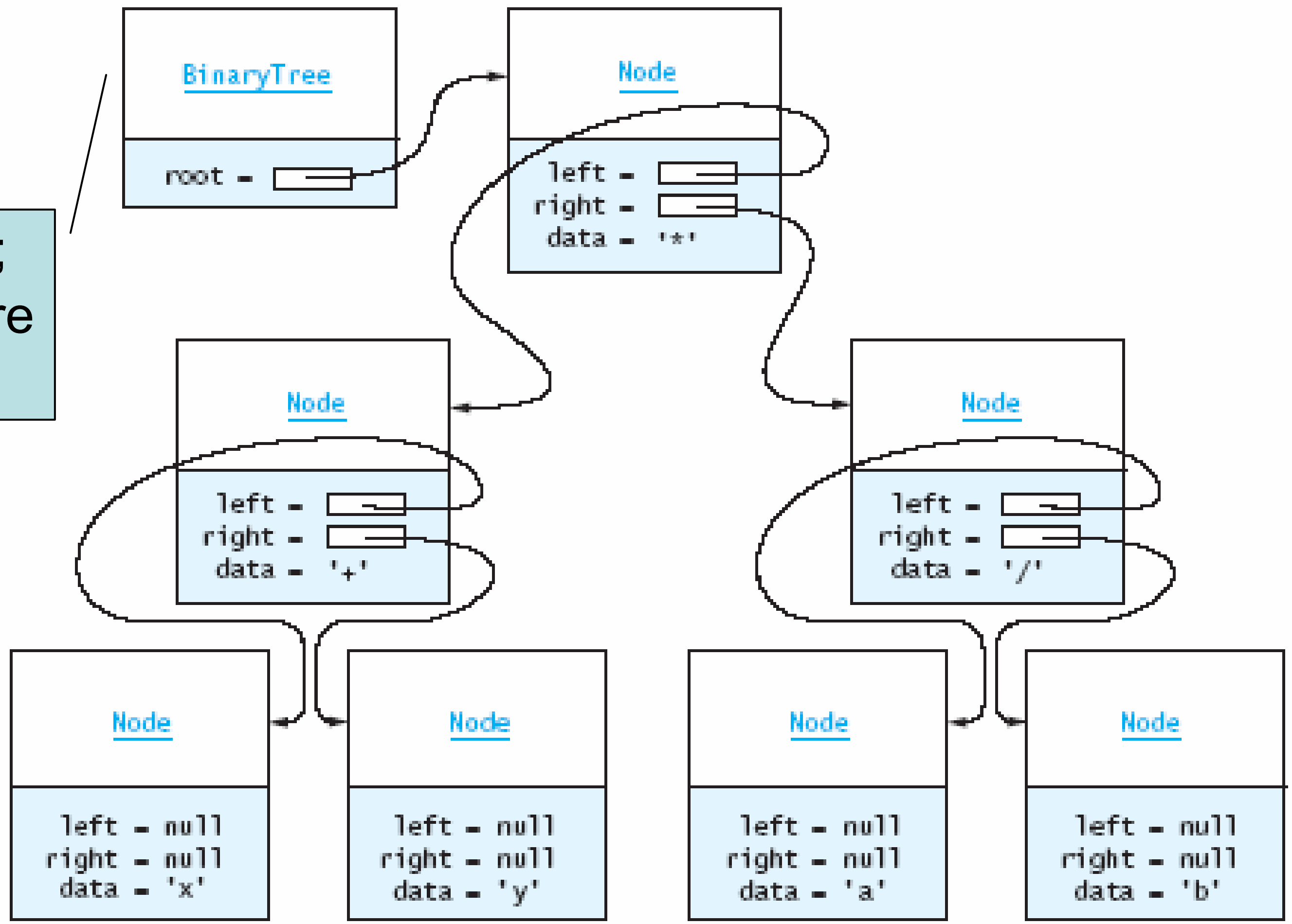
FIGURE 8.10
Linked Structure to Represent a Node



The `BinaryTree<E>` Class

FIGURE 8.11
Linked Representation of Expression Tree ((x + y) * (a / b))

BinaryTree<E>; remaining nodes are **Node<E>**



Code for Node<E>

```
protected static class Node<E>
    implements Serializable {

  protected E data;
  protected Node<E> left;
  protected Node<E> right;

  public Node (E e,
               Node<E> l, Node<E> r) {
    this.data  = e;
    this.left  = l;
    this.right = r;
  }
```

Code for Node<E> (2)

```
  public Node (E e) {
    this(e, null, null);
  }

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

}
```

Code for BinaryTree<E>

```
import java.io.*;
public class BinaryTree<E>
    implements Serializable {

  protected Node<E> root;

  protected static class Node<E> ...{...}

  public BinaryTree () { root = null; }

  protected BinaryTree (Node<E> root) {
    this.root = root;
  }
```

Code for `BinaryTree<E>` (2)

```
public BinaryTree (E data,
                   BinaryTree<E> left,
                   BinaryTree<E> right) {
  root = new Node<E>
    (e,
     (left  == null) ? null : left .root,
     (right == null) ? null : right.root);
}
```

Code for BinaryTree<E> (3)

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

public BinaryTree<E> getRightSubtree () {
  if (root == null || root.right == null)
    return null;
  return new BinaryTree<E>(root.right);
}
```

Code for BinaryTree<E> (4)

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

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

Code for BinaryTree<E> (5)

```
private void preOrderTraverse (Node<E> n,
    int d, StringBuilder sb) {
  for (int i = 1; i < d; ++i)
    sb.append(“ “);
  if (node == null)
    sb.append(“null\n”);
  else {
    sb.append(node.toString());
    sb.append(“\n”);
    preOrderTraverse(node.left , d+1, sb);
    preOrderTraverse(node.right, d+1, sb);
  }
}
```

Code for BinaryTree<E> (6)

```
public static BinaryTree<String>
    readBinaryTree (BufferedReader bR)
      throws IOException {
  String data = bR.readLine().trim();
  if (data.equals(“null”))
    return null;
  BinaryTree<String> l =
      readBinaryTree(bR);
  BinaryTree<String> r =
      readBinaryTree(bR);
  return new BinaryTree<String>(
    data, l, r);
}
```

Overview of Binary Search Tree

Binary search tree definition:

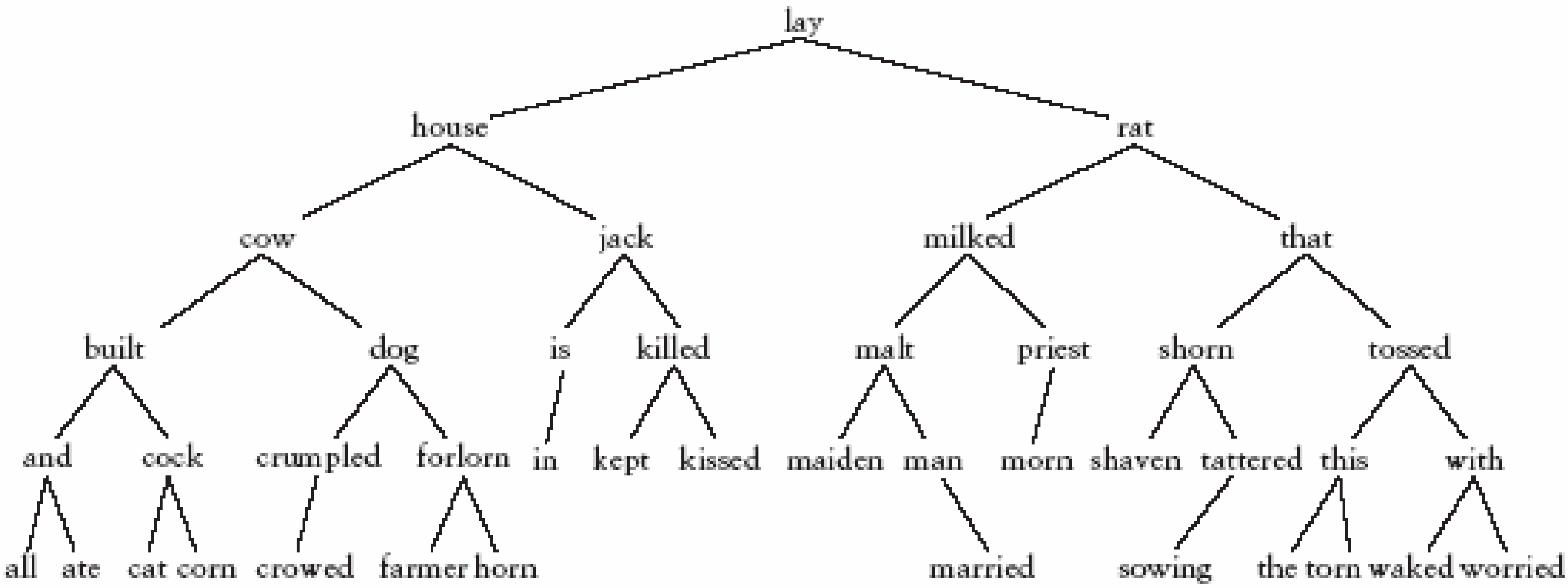
T is a binary search tree if either of these is true

- T is empty; *or*
- Root has two subtrees:
  - Each is a binary search tree
  - Value in root > all values of the left subtree
  - Value in root < all values in the right subtree

Binary Search Tree Example

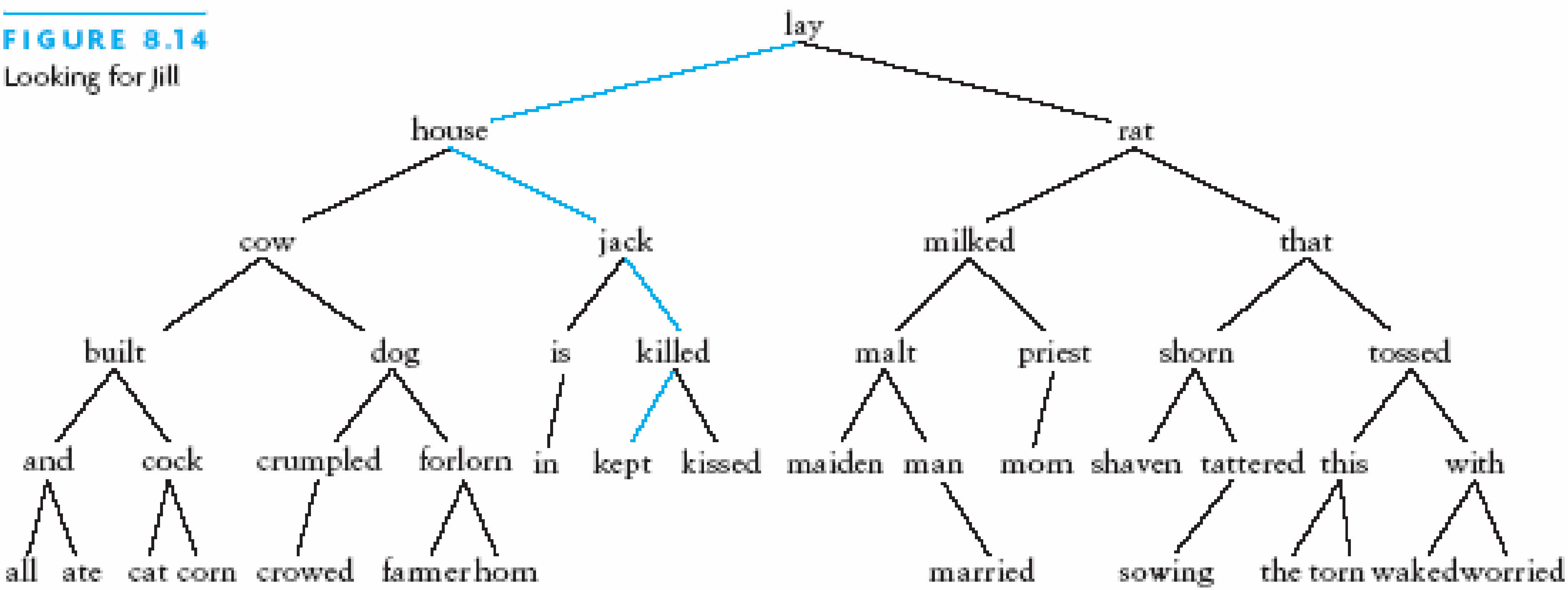
FIGURE 8.13

Binary Search Tree Containing All of the Words from "The House That Jack Built"



Searching a Binary Tree

FIGURE 8.14
Looking for Jill



Searching a Binary Tree: Algorithm

1. if root is **null**
2.   item not in tree: return **null**
3. compare **target** and **root.data**
4. if they are *equal*
5.   **target** is found, return **root.data**
6. else if **target < root.data**
7.   return search(left subtree)
8. else
9.   return search(right subtree)

Class `TreeSet<E>` and Interface `SearchTree<E>`

- Java API offers **`TreeSet<E>`**
  - Implements binary search trees
  - Has operations such as add, contains, remove

- We define **`SearchTree<E>`**, offering some of these

Code for Interface SearchTree<E>

```
public interface SearchTree<E> {
  // add/remove say whether tree changed
  public boolean add (E e);
  boolean remove (E e);
  // contains tests whether e is in tree
  public boolean contains (E e);
  // find and delete return e if present,
  //   or null if it is not present
  public E find (E e);
  public E delete (E e);
}
```

Code for BinarySearchTree<E>

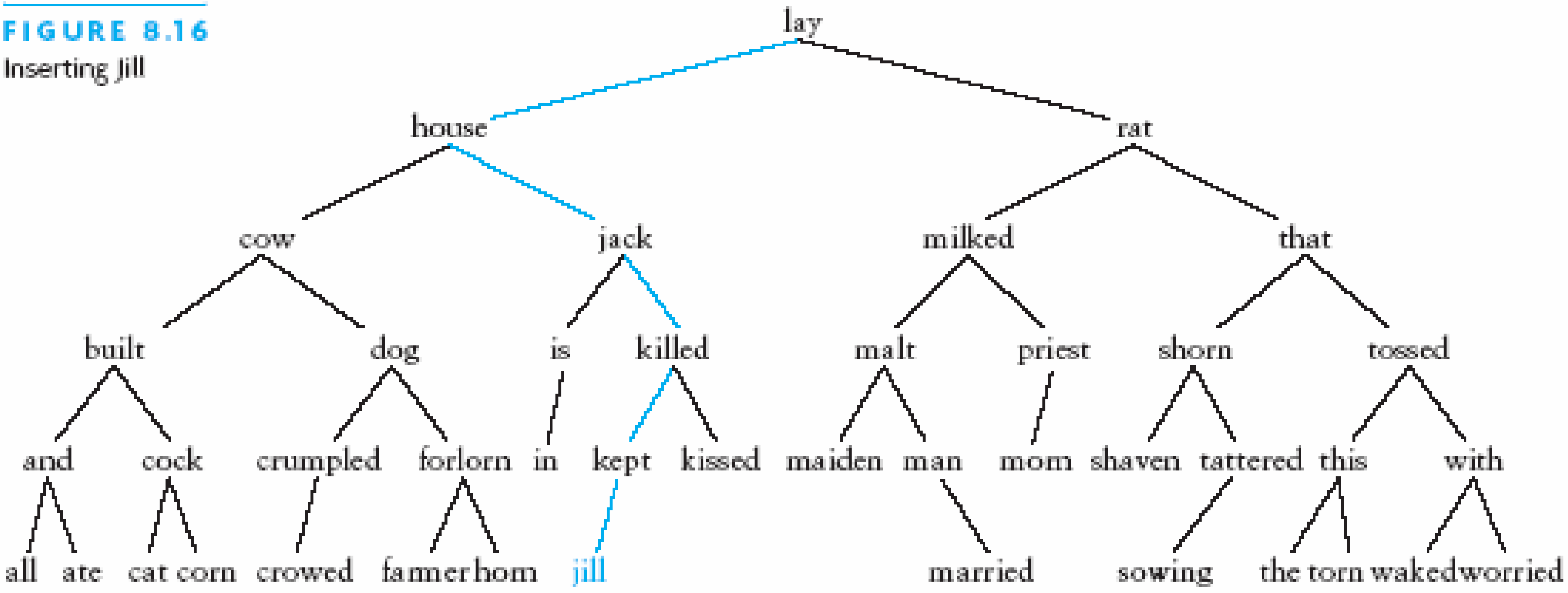
```
public class BinarySearchTree<
      E extends Comparable<E>>
    extends BinaryTree<E>
    implements SearchTree<E> {
  protected boolean addReturn;
  protected E       deleteReturn;
  // these two hold the “extra” return
  //  values from the SearchTree adding
  //  and deleting methods

  ...
}
```

BinarySearchTree.find(E e)

```
public E find (E e) {
  return find(root, e);
}
private E find (Node<E> n, E e) {
  if (n == null) return null;
  int comp = e.compareTo(n.data);
  if (comp == 0) return n.data;
  if (comp < 0)
    return find(n.left , e);
  else
    return find(n.right, e);
}
```

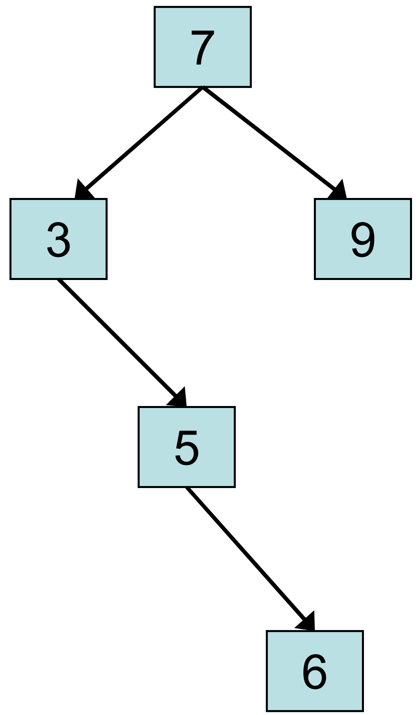
Insertion into a Binary Search Tree

FIGURE 8.16
Inserting Jill



Example Growing a Binary Search Tree

1. insert 7
2. insert 3
3. insert 9
4. insert 5
5. insert 6



Binary Search Tree Add Algorithm

1. if root is **null**
2. replace empty tree with new data leaf; return **true**
3. if **item equals root.data**
4. return **false**
5. if **item < root.data**
6. return insert(left subtree, item)
7. else
8. return insert(right subtree, item)

BinarySearchTree.add(E e)

```
public boolean add (E e) {
  root = add(root, item);
  return addReturn;
}
```

BinarySearchTree.add(E e) (2)

```
private Node<E> add (Node<E> n, E e) {
  if (n == null) { addReturn = true;
    return new Node<E>(e);
  }
  int comp = e.compareTo(n.data);
  if (comp == 0)
    addReturn = false;
  else if (comp < 0)
    n.left  = add(n.left , e);
  else
    n.right = add(n.right, e);
  return n;
}
```

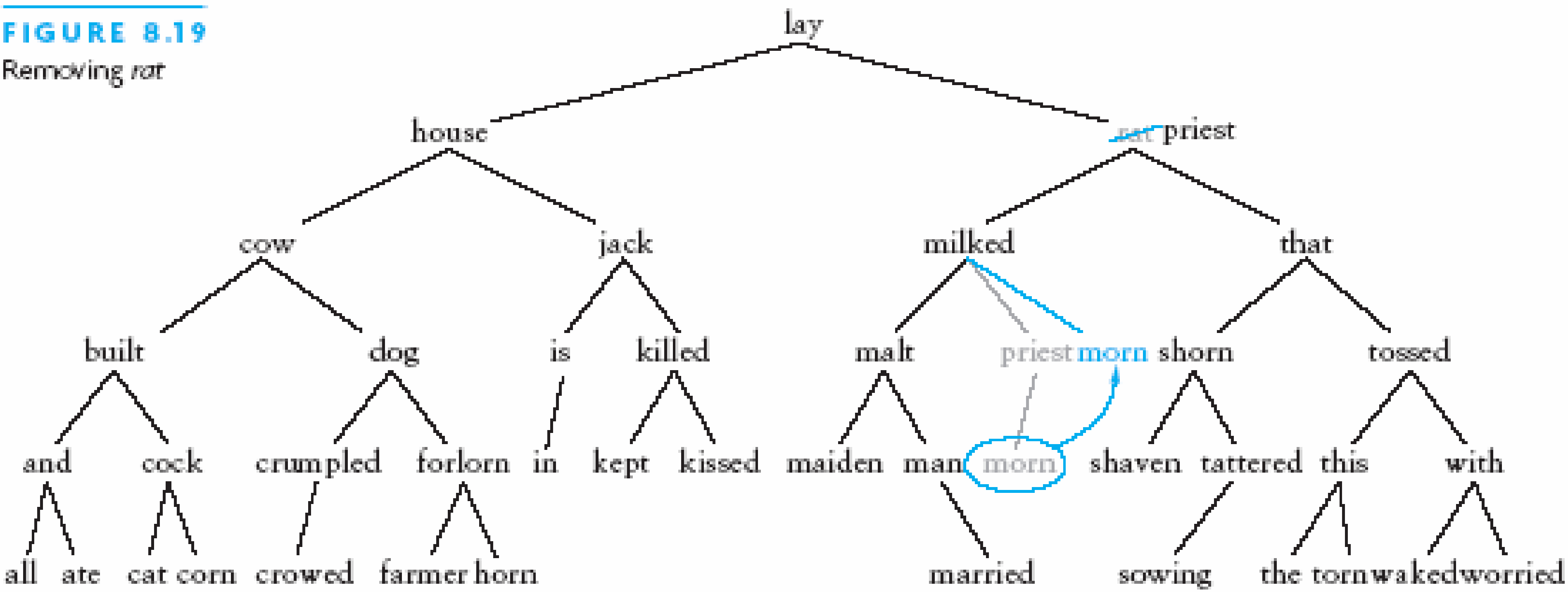
Removing from a Binary Search Tree

- Item not present: do nothing
- Item present in leaf: remove leaf (change to null)
- Item in non-leaf with one child:

  Replace current node with that child
- Item in non-leaf with *two children*?
  - Find *largest item* in the *left subtree*
  - Recursively remove it
  - Use it as the parent of the two subtrees
  - (Could use smallest item in right subtree)

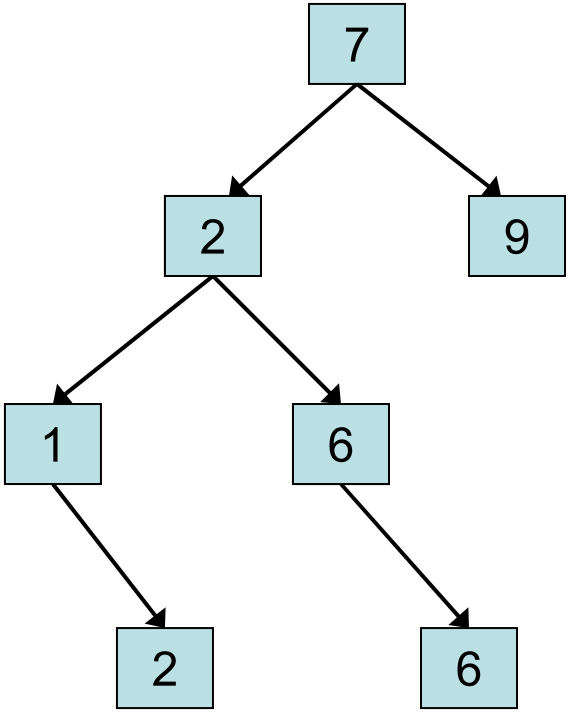
Removing from a Binary Search Tree (2)

FIGURE 8.19
Removing *rat*



Example Shrinking a Binary Search Tree

1. delete 9
2. delete 5
3. delete 3



BinarySearchTree.delete(E e)

```
public E delete (E e) {
  root = delete(root, item);
  return deleteReturn;
}
private Node<E> delete (Node<E> n E e) {
  if (n == null) {
    deleteReturn = null;
    return null;
  }
  int comp = e.compareTo(n.data);
  ...
```

BinarySearchTree.delete(E e) (2)

```
  if (comp < 0) {
    n.left  = delete(n.left , e);
    return n;
  } else if (comp > 0) {
    n.right = delete(n.right, e);
    return n;
  } else {
    // item is in n.data
    deleteReturn = n.data;
    ...
  }
}
```

BinarySearchTree.delete(E e) (3)

```
// deleting value in n: adjust tree
if (n.left == null) {
  return n.right;  // ok if also null
} else if (n.right == null) {
  return n.left;
} else {
  // case where node has two children
  ...
}
```

BinarySearchTree.delete(E e) (4)

```
// case where node has two children
if (n.left.right == null) {
  // largest to left is in left child
  n.data = n.left.data;
  n.left = n.left.left;
  return n;
} else {
  n.data = findLargestChild(n.left);
  return n;
}
```

findLargestChild

```
// finds and removes largest value under n
private E findLargestChild (Node<E> n) {
  // Note: called only for n.right != null
  if (n.right.right == null) {
    // no right child: this value largest
    E e = n.right.data;
    n.right = n.right.left;  // ok if null
    return e;
  }
  return findLargestChild(n.right);
}
```

Using Binary Search Trees

- Want an index of words in a paper, by line #
- Put word/line # strings into a binary search tree
  - "java, 0005", "a, 0013", and so on
- Performance?
  - Average BST performance is O(log n)
    - Its rare worst case is O(n)
    - Happens (for example) with sorted input
  - Ordered list performance is O(n)
- Later consider *guaranteeing* O(log n) trees

Using Binary Search Trees: Code

```
public class IndexGen {
  private TreeSet<String> index;
  public IndexGen () {
    index = new TreeSet<String>();
  }

  public void showIndex () {
    for (String next : index)
      System.out.println(next);
  }
  ...
}
```

Using Binary Search Trees: Code (2)

```
public void build (BufferedReader bR)
    throws IOException {
  int lineNum = 0;
  String line;
  while ((line = bR.readLine()) != null) {
    // process line
  }
}
```

Using Binary Search Trees: Code (3)

```
// processing a line:
StringTokenizer tokens =
  new StringTokenizer(line, ...);
while (tokens.hasNextToken()) {
  String tok =
    tokens.nextToken().toLowerCase();
  index.add(
    String.format("%s, %04d",
                  tok, lineNum));
}
```

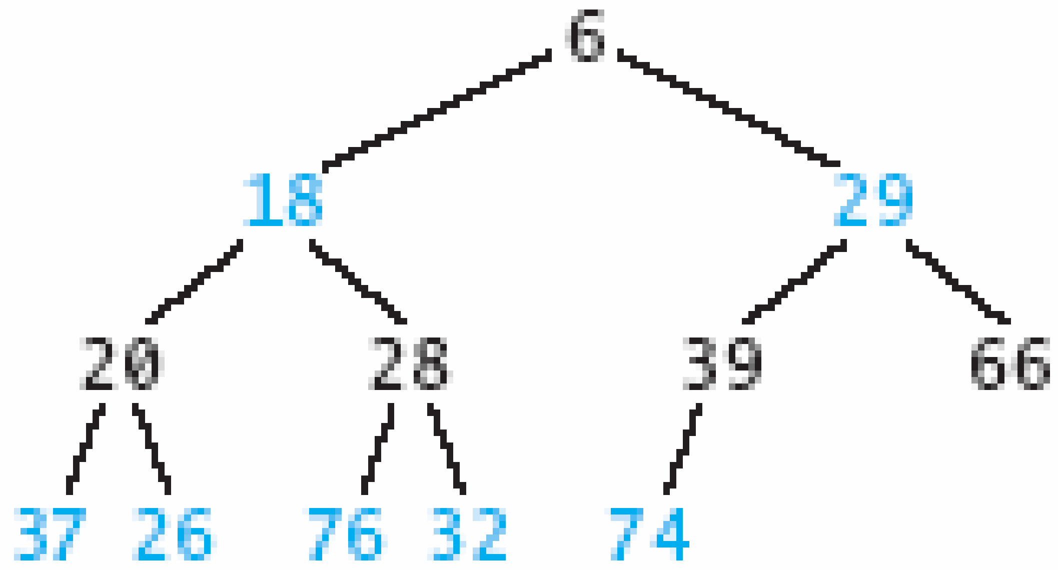
Heaps

- A ***heap*** orders its node, but in a way different from a binary search tree
- A complete tree is a *heap* if
  - The value in the root is the smallest of the tree
  - Every subtree is also a heap
- Equivalently, a complete tree is a heap if
  - Node value < child value, for each child of the node
- **Note:** This use of the word “heap” is entirely different from the heap that is the allocation area in Java

Example of a Heap

FIGURE 8.20

Example of a Heap



Inserting an Item into a Heap

1. Insert the item in the next position across the bottom of the complete tree: *preserve completeness*
2. *Restore "heap-ness":*
    1. **while** new item not root and < parent
    2. swap new item with parent

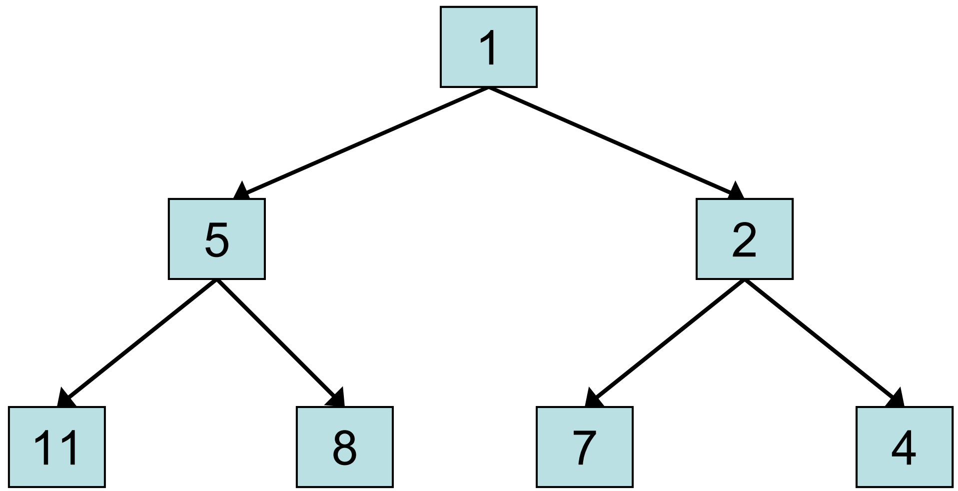
Example Inserting into a Heap

Insert 1

Add as leaf

Swap up

Swap up



Removing an Item from a Heap

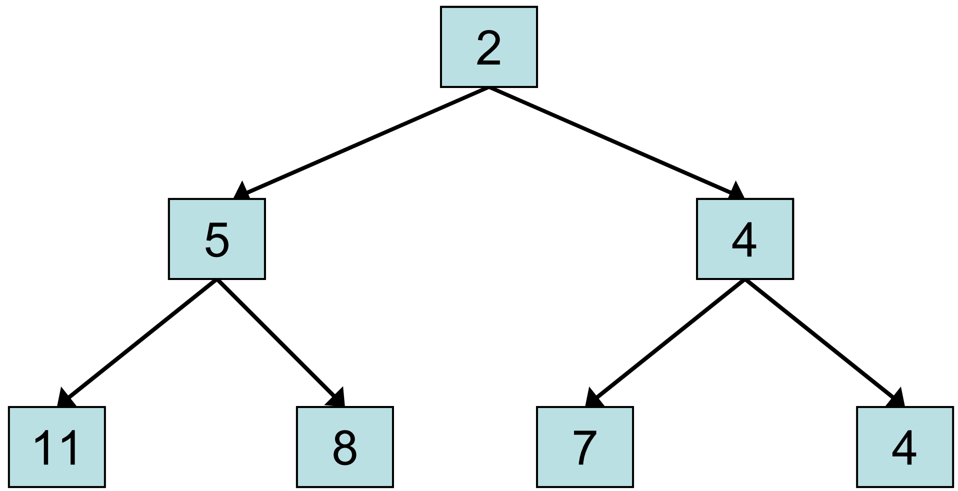
- Removing an item is always from the *top:*
  - Remove the *root* (minimum element):
    - Leaves a “hole”:
  - Fill the “hole” with the last item (lower right-hand) L
    - *Preserve completeness*
  - Swap L with smallest child, as necessary
    - *Restore “heap-ness”*

Example Removing From a Heap

Remove: returns 1

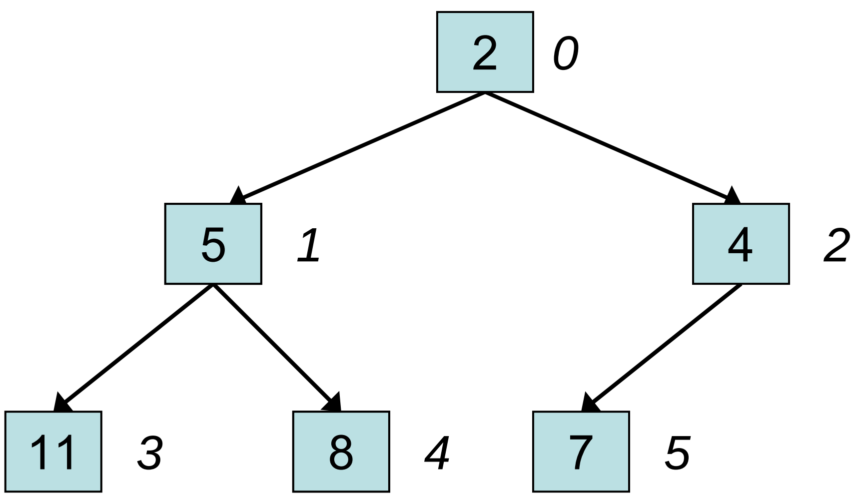
Move 4 to root

Swap down



Implementing a Heap

- Recall: a heap is a *complete binary tree*
  - (plus the heap ordering property)
- A complete binary tree fits nicely in an *array:*
  - The root is at index 0
  - Children of node at index i are at indices 2i+1, 2i+2



0	1	2	3	4	5
2	5	4	11	8	7

Inserting into an Array(List) Heap

1. Insert new item at end; set **child** to size-1
2. Set **parent** to (**child** – 1)/2
3. **while** (**parent** ≥ 0 and a[**parent**] > a[**child**])
4. Swap a[**parent**] and a[**child**]
5. Set **child** equal to **parent**
6. Set **parent** to (**child** – 1) / 2

Deleting from an Array(List) Heap

1. Set a[0] to a[size-1], and shrink size by 1
2. Set **parent** to 0
3. **while** (**true**)
4. Set **lc** to 2 * **parent** + 1, and **rc** to **lc** + 1
5. If **lc** ≥ size, break out of **while** (we're done)
6. Set **minc** to **lc**
7. If **rc** < size and a[**rc**] < a[**lc**], set **minc** to **rc**
8. If a[**parent**] ≤ a[**minc**], break (we're done)
9. Swap a[**parent**] and a[**minc**]; set **parent** to **minc**

Performance of Array(List) Heap

- A *complete* tree of height h has:
  - Less than 2^h nodes
  - At least 2^{h-1} nodes
- Thus complete tree of n nodes has height O(log n)
- Insertion and deletion *at most* do constant amount of work at each level
- Thus these operations are O(log n), *always*
- Heap forms the basis of *heapsort* algorithm (Ch. 10)
- Heap also useful for *priority queues*

Priority Queues

- A *priority queue* de-queues items in *priority order*
  - Not in order of entry into the queue (not FIFO)
- Heap is an efficient implementation of priority queue
  - Operations cost at most O(log n)

```
public boolean offer (E e);  // insert
public E remove ();  // return smallest
public E poll ();    // smallest or null
public E peek ();    // smallest or null
public E element (); // smallest
```

Design of Class PriQueue<E>

```
ArrayList<E> data;  // holds the data

PriQueue ()  // uses natural order of E
PriQueue (int n, Comparator<E> comp)
    // uses size n, uses comp to order

private int compare (E left, E right)
    // compares two E objects: -1, 0, 1
private void swap (int i, int j)
    // swaps elements at positions i and j
```

Implementing `PriQueue<E>`

```
public class PriQueue<E>
    extends AbstractQueue<E>
    implements Queue<E> {
  private ArrayList<E> data;
  Comparator<E> comparator = null;

  public PriQueue () {
    data = new ArrayList<E>();
  }
  ...
```

Implementing `PriQueue<E>` (2)

```
public PriQueue (int n,
                 Comparator<E> comp) {
  if (n < 1)
    throw new IllegalArgumentException();
  data = new ArrayList<E>(n);
  this.comp = comp;
}
```

Implementing `PriQueue<E>` (3)

```
private int compare (E lft, E rt) {
  if (comp != null)
    return comp.compare(lft, rt);
  else
    return
      ((Comparable<E>)lft).compareTo(rt);
}
```

Implementing `PriQueue<E>` (4)

```
public boolean offer (E e) {
  data.add(e);
  int child = data.size() – 1;
  int parent = (child – 1) / 2;
  while (parent >= 0 &&
         compare(data.get(parent),
                 data.get(child)) > 0) {
    swap(parent, child);
    child = parent;
    parent = (child – 1) / 2;
  }
  return true;
}
```

Implementing `PriQueue<E>` (5)

```
public E poll () {
  if (isEmpty()) return null;
  E result = data.get(0);
  if (data.size() == 1) {
    data.remove(0);
    return result;
  }
  // remove last item and store in posn 0
  data.set(0, data.remove(data.size()-1));
  int parent = 0;
  while (true)
    ... swap down as necessary ...
  return result;
```

Implementing PriQueue<E> (6)

```
// swapping down
int lc = 2 * parent + 1;
if (lc >= data.size()) break;
int rc = lc + 1;
int minc = lc;
if (rc < data.size() &&
    compare(data.get(lc),
            data.get(rc)) > 0)
  minc = rc;
if (compare(data.get(parent),
            data.get(minc)) <= 0) break;
swap(parent, minc);
parent = minc;
```

Huffman Trees

Problem Input: A set of symbols, each with a frequency of occurrence.

Desired output: A Huffman tree giving a code that minimizes the bit length of strings consisting of those symbols with that frequency of occurrence.

Strategy: Starting with single-symbol trees, repeatedly combine the two lowest-frequency trees, giving one new tree of frequency = sum of the two frequencies. Stop when we have a single tree.

Huffman Trees (2)

Implementation approach:

- Use a *priority queue* to find lowest frequency trees
- Use binary trees to represent the Huffman (de)coding trees

Example: b=13, c=22, d=32 a=64 e=103

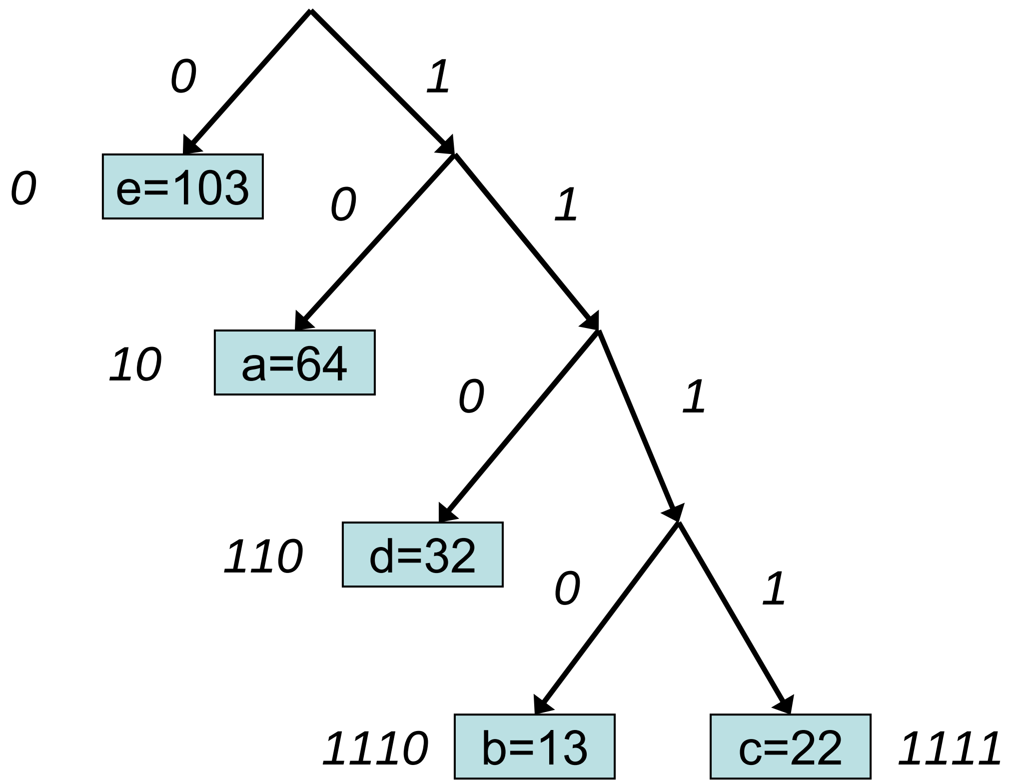
Combine b and c: bc=35

Combine d and bc: d(bc)=67

Combine a and d(bc): a(d(bc))=131

Combine e and a(d(bc)): e(a(d(bc)))=234 ... done

Huffman Tree Example



Design of Class HuffTree

```
private BinaryTree<HuffData> huffTree;
  // HuffData are pairs: weight, symbol

buildTree (HuffData[] input)
String decode (String message)
printcode (Printstream out)
```

Implementing `HuffTree`

```
public class HuffTree
    implements Serializable {
  public static class HuffData
      implements Serializable {
    private double weight;
    private Character symbol;
    public HuffData (double weight,
                     Character symbol) {
      this.weight = weight;
      this.symbol = symbol;
    }
  }
```

Implementing HuffTree (2)

```
private BinaryTree<HuffData> tree;

private static class CompHuff
    implements
      Comparator<BinaryTree<HuffData>> {
  public int compare (
      BinaryTree<HuffData> lft,
      BinaryTree<HuffData> rt) {
    double wL = lft.getData().weight;
    double wR = rt .getdata().weight;
    return Double.compare(wL, wR);
  }
}
```

Implementing `HuffTree` (3)

```
public void buildTree (HuffData[] syms) {
  Queue<BinaryTree<HuffData>> q =
    new PriQueue<BinaryTree<HuffData>>(
      syms.length, new CompHuffTree());
  for (HuffData sym : syms) {
    BinaryTree<HuffData> tree =
      new BinaryTree<HuffData>(sym);
    q.offer(tree);
  }
  ... on to second half ...
```

Implementing HuffTree (4)

```
while (q.size() > 1) {
  BinaryTree<HuffData> lft = q.poll();
  BinaryTree<HuffData> rt  = q.poll();
  double wl = lft.getData().weight;
  double wr = rt .getData().weight;
  HuffData sum =
    new HuffData(wl+wr, null);
  BinaryTree<HuffData> nTree =
    new BinaryTree<HuffData>
      (sum, lft, rt);
  q.offer(nTree);
}
this.tree = q.poll();
```

Implementing HuffTree (5)

```
private void printCode
    (PrintStream out, String code,
     BinaryTree<HuffData> tree) {
  HuffData data = tree.getData;
  if (data.symbol != null) {
    if (data.symbols.equals(“ “))
      out.println(“space: “ + code);
    else
      out.println(data.symbol+”: “+code);
  } else {
    printCode(out,code+”0”, tree.left ());
    printCode(out,code+”1”, tree.right());
  } }
```

Implementing HuffTree (6)

```
public string decode (String msg) {
  StringBuilder res = new StringBuilder();
  BinaryTree<HuffData> curr = tree;
  for (int i = 0; i < msg.length(); ++i) {
    if (msg.charAt(i) == ‘1’)
      curr = curr.getRightSubtree();
    else
      curr = curr.getLeftSubtree();
    if (curr.isLeaf()) {
      HuffData data = curr.getData();
      res.append(data.symbol);
      curr = tree;
    } } return res.toString();
```