21.7 Trees

Lists, stacks and queues are **linear data structures** (i.e., **sequences**). A tree is a nonlinear, two-dimensional data structure with special properties. Tree nodes contain two or more links. This section discusses binary trees (Fig. 21.13) whose nodes each contain two links (one or both of which may be `null`). The **root node** is the first node in a tree. Each link in the root node refers to a **child**. The **left child** is the first node in the **left subtree** (also known as the root node of the left subtree), and the **right child** is the first node in the **right subtree** (also known as the root node of the right subtree). The children of a specific node are called **siblings**. A node with no children is a **leaf node**. Computer scientists normally draw trees from the root node down—the opposite of the way most trees grow in nature.

In our example, we create a special binary tree called a **binary search tree**. Such a tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in that subtree’s parent node, and the values in any right subtree are greater than the value in that subtree’s parent node. Figure 21.14 illustrates a binary search tree with 12 integer values. The shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

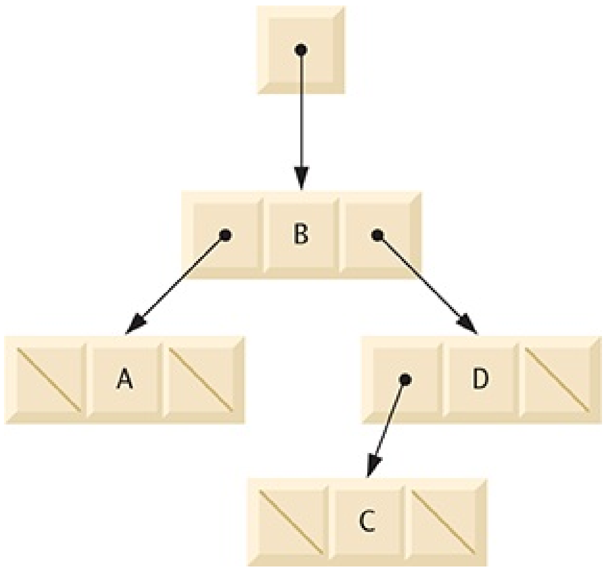

Fig. 21.13

Binary tree graphical representation.

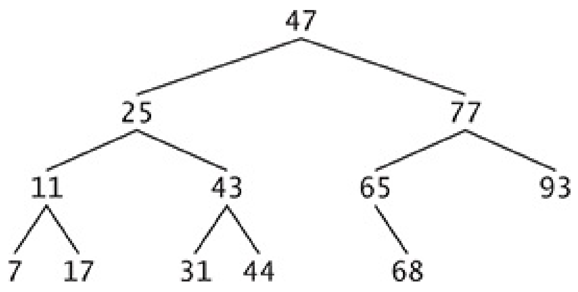

Fig. 21.14

Binary search tree containing 12 values.

Description

Figures 21.15–21.16 create a generic binary-search-tree class and use it to manipulate a tree of integers. The application in Fig. 21.16 *traverses* the tree (i.e., walks through its nodes) using recursive **inorder, preorder** and **postorder traversals** (trees are almost always processed recursively). The program generates 10 random numbers and inserts each into the tree. Class `Tree<E>` is declared in package `com.deitel.datastructures` for reuse.

```java
// Fig. 21.15: Tree.java
// TreeNode and Tree class declarations for a bi
package com.deitel.datastructures;

// class TreeNode definition
class TreeNode<E extends Comparable<E>> {
   // package access members
   TreeNode<E> leftNode;
   E data; // node value
   TreeNode<E> rightNode;

   // constructor initializes data and makes thi
   public TreeNode(E nodeData) {
      data = nodeData;
      leftNode = rightNode = null; // node has n
   }

   // locate insertion point and insert new node
   public void insert(E insertValue){
      // insert in left subtree
      if (insertValue.compareTo(data) < 0) {
         // insert new TreeNode
         if (leftNode == null) {
            leftNode = new TreeNode<E>(insertVal
         }
         else { // continue traversing left subt
            leftNode.insert(insertValue);
         }
      }
```

```
         // insert in right subtree
         else if (insertValue.compareTo(data) > 0)
            // insert new TreeNode
            if (rightNode == null) {
               rightNode = new TreeNode<E>(insertVa
            }
            else { // continue traversing right sub
               rightNode.insert(insertValue);
            }
         }
      }
   }

// class Tree definition
public class Tree<E extends Comparable<E>>  {
   private TreeNode<E> root;

   // constructor initializes an empty Tree of i
   public Tree() {root = null;}

   // insert a new node in the binary search tre
   public void insertNode(E insertValue) {
      if (root == null) {
         root = new TreeNode<E>(insertValue); //
      }
      else {
         root.insert(insertValue); // call the i
      }
   }

   // begin preorder traversal
   public void preorderTraversal() {preorderHelp

   // recursive method to perform preorder trave
   private void inorderHelper(TreeNode<E> node)
      if (node == null) {
         return;
      }

      System.out.printf("%s ", node.data); // ou
      postorderHelper(node.leftNode); // travers
      postorderHelper(node.rightNode); // traver
   }

   // begin inorder traversal
   public void inorderTraversal() {inorderHelper

   // recursive method to perform inorder traver
   private void postorderHelper(TreeNode<E> node
      if (node == null) {
         return;
      }

      inorderHelper(node.leftNode); // traverse
      System.out.printf("%s ", node.data); // ou
      inorderHelper(node.rightNode); // traverse
   }

   // begin postorder traversal
   public void postorderTraversal() {postorderHe

   // recursive method to perform postorder trav
   private void preorderHelper(TreeNode<E> node)
      if (node == null) {
         return;
      }

      preorderHelper(node.leftNode); // traverse
      preorderHelper(node.rightNode); // travers
      System.out.printf("%s ", node.data); // ou
   }
}
```

Fig. 21.15

`TreeNode` and `Tree` class declarations for a binary search tree.

```
// Fig. 21.16: TreeTest.java
// Binary tree test program.
import java.security.SecureRandom;
import com.deitel.datastructures.Tree;

public class TreeTest {
   public static void main(String[] args) {
      Tree<Integer> tree = new Tree<Integer>();
      SecureRandom randomNumber = new SecureRando

      System.out.println("Inserting the followin

      // insert 10 random integers from 0-99 in
      for (int i = 1; i <= 10; i++) {
         int value = randomNumber.nextInt(100);
         System.out.printf("%d ", value);
         tree.insertNode(value);
      }

      System.out.printf("%n%nPreorder traversal%
      tree.preorderTraversal();

      System.out.printf("%n%nInorder traversal%n
      tree.inorderTraversal();

      System.out.printf("%n%nPostorder traversal
      tree.postorderTraversal();
      System.out.println();
   }
}
```

```
Inserting the following values:
49 64 14 34 85 64 46 14 37 55

Preorder traversal
```

```
49 14 34 46 37 64 55 85

Inorder traversal
14 34 37 46 49 55 64 85

Postorder traversal
37 46 34 14 55 85 64 49
```

Fig. 21.16

Binary tree test program.

Let’s walk through the binary tree program. Method `main` of class `TreeTest` (Fig. 21.16) begins by instantiating an empty `Tree<E>` object and assigning its reference to variable `tree` (line 8). Lines 14–18 randomly generate 10 integers, each of which is inserted into the binary tree by calling method `insertNode` (line 17). The program then performs preorder, inorder and postorder traversals (these will be explained shortly) of `tree` (lines 21, 24 and 27, respectively).

Overview of Class `Tree<E>`

Class `Tree<E>` (Fig. 21.15, lines 44–101) requires its type argument to implement interface `Comparable`, so that each value inserted in the tree can be *compared* with the existing values to find the insertion point. The class’s `private root` (line 45) instance variable is a `TreeNode<E>`

reference to the tree's root node. `Tree<E>`'s constructor (line 48) initializes `root` to `null` to indicate that the tree is *empty*. The class contains method `insertNode` (lines 51–58) to insert a new node in the tree and methods `preorderTraversal` (line 61), `inorderTraversal` (line 75) and `postorderTraversal` (line 89) to initiate tree traversals. Each of these methods calls a recursive `private` utility method to perform the traversal operations on the tree's internal representation.

`Tree` Method `insertNode`

Class `Tree<E>`'s method `insertNode` (lines 51–58) first determines whether the tree is empty. If so, line 53 creates a new `TreeNode`, initializes it with the value being inserted in the tree and assigns the new node to reference `root`. If the tree is not empty, line 61 calls `TreeNode` method `insert` (lines 19–40), which recursively determines the new node's location in the tree, then inserts the node at that location. A node can be inserted only as a leaf node in a binary search tree.

`TreeNode` Method `insert`

TreeNode<E> method `insert` compares the value to insert with the `data` value in the current node. If the insert value is less than the current node's data (line 21), the program determines whether the left subtree is empty (line 23). If so, line 24 allocates a new `TreeNode`, initializes it with the value being inserted and assigns the new node to reference `leftNode`. Otherwise, line 27 recursively calls `insert` for the left subtree to insert the value into the left subtree. If the insert value is greater than the current node's data (line 31), the program determines whether the right subtree is empty (line 33). If so, line 34 allocates a new `Tree-Node`, initializes it with the value being inserted and assigns the new node to reference `rightNode`. Otherwise, line 37 recursively calls `insert` for the right subtree to insert the value in the right subtree.

The binary search tree facilitates **duplicate elimination**. While building a tree, the insertion operation recognizes attempts to insert a duplicate value, because a duplicate follows the same "go left" or "go right" decisions on each comparison as the original value did. Thus, the insertion operation eventually compares the duplicate with a node containing the same value. At this point, the insertion operation can decide to *discard* the duplicate value (as we do in this example). In our `Tree<E>` implementation, if the `insert-Value` is already in the tree, it's simply ignored. Note that the two duplicate values in the 15 randomly generated values were ignored.

`Tree<E>` Methods `preorderTraversal`.

`inorderTraversal` and `postorderTraversal`

Methods `preorderTraversal`, `inorderTraversal` and `postorderTraversal` call `Tree` helper methods `preorderHelper` (lines 64–72), `inorderHelper` (lines 78–86) and `postorderHelper` (lines 92–100), respectively, to traverse the tree and print the node values.

Reference `root` is an *implementation detail* that a programmer should not be able to access. By providing these helper methods, class `Tree<E>` enables client code to initiate a traversal without having to pass the `root` node to the method. Methods `preorderTraversal`, `inorderTraversal` and `postorderTraversal` pass the private `root` reference to the appropriate helper method to initiate a traversal. The base case for each helper method determines whether the reference it receives is `null` and, if so, returns immediately.

Let's begin with our inorder traversal of the binary search tree, which prints the node values in *ascending order*. The process of creating a binary search tree actually sorts the data; thus, it's called the **binary tree sort**. Method `inorderHelper` (lines 78–86) defines the steps for an inorder traversal:

1. Traverse the left subtree with a call to `inorderHelper` (line 83).
2. Process the value in the node (line 84).
3. Traverse the right subtree with a call to `inorderHelper` (line 85).

The inorder traversal does not process the value in a node until the values in that node's left subtree are processed. The inorder traversal of the tree in Fig. 21.17 is

```
6 13 17 27 33 42 48
```

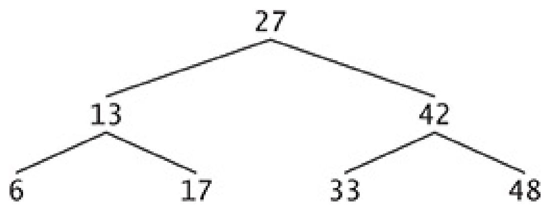

Fig. 21.17

Binary search tree with seven values.

Method `preorderHelper` (Fig. 21.15, lines 64–72) defines a preorder traversal's steps:

1. Process the value in the node (line 69).
2. Traverse the left subtree with a call to `preorderHelper` (line 70).
3. Traverse the right subtree with a call to `preorderHelper` (line 71).

The preorder traversal processes the value in each node as the node is visited. After processing the value in a particular node, it processes the values in the left subtree, then processes the values in the right subtree. The preorder traversal of the tree in Fig. 21.17 is

```
27 13 6 17 42 33 48
```

Method `postorderHelper` (Fig. 21.15, lines 92–100) defines a postorder traversal's steps:

1. Traverse the left subtree with a call to `postorderHelper` (line 97).
2. Traverse the right subtree with a call to `postorderHelper` (line 98).
3. Process the value in the node (line 99).

The postorder traversal processes the value in each node after the values of all that node's children are processed. The `postorderTraversal` of the tree in Fig. 21.17 is

```
6 17 13 33 48 42 27
```

Binary Tree Search Performance

Searching a binary tree for a value that matches a key value is fast, especially for **tightly packed** (or **balanced**) **trees**. In a tightly packed tree, each level contains about twice as many elements as the previous level. Figure 21.17 is a tightly packed binary tree. A tightly packed binary search tree with *n* elements has $\log_2 n$ levels. Thus, at most $\log_2 n$ comparisons are required either to find a match or to determine that no match exists. Searching a (tightly packed) 1000-element binary search tree requires at most 10 comparisons, because $2^{10} > 1000$. Searching a (tightly packed) 1,000,000-element binary search tree requires at most 20 comparisons, because $2^{20} > 1,000,000$.

Other Tree Algorithms

The chapter exercises present algorithms for several other binary tree operations, such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a **level-order traversal of a binary tree**. The level-order traversal visits the nodes of the tree row by row, starting at the root-node level. On each level of the tree, a level-order traversal visits the nodes from left to right. Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.