Chapter 21
Custom Generic Data Structures

Java How to Program, 10/e

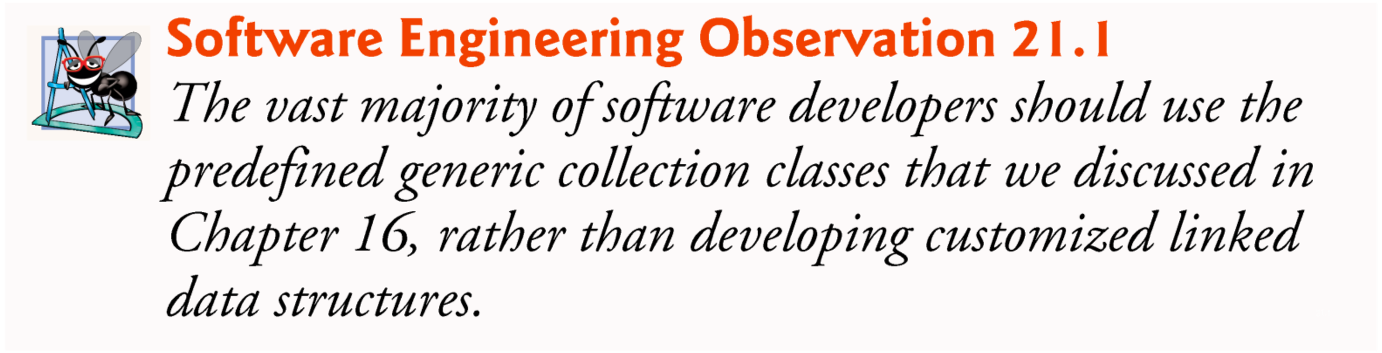
OBJECTIVES

In this chapter you'll learn:

- To form linked data structures using references, self-referential classes, recursion and generics.
- To create and manipulate dynamic data structures, such as linked lists, queues, stacks and binary trees.
- Various important applications of linked data structures.
- How to create reusable data structures with classes, inheritance and composition.
- To organize classes in packages to promote reuse.

21.1 Introduction

- Dynamic data structures grow and shrink at execution time.
- Linked lists are collections of data items "linked up in a chain"—insertions and deletions can be made anywhere in a linked list.
- Stacks are important in compilers and operating systems; insertions and deletions are made only at one end of a stack—its top.
- Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and deletions are made from the front (also referred to as the head).
- Binary trees facilitate high-speed searching and sorting of data, eliminating duplicate data items efficiently, representing file-system directories, compiling expressions into machine language and many other interesting applications.



Software Engineering Observation 21.1

The vast majority of software developers should use the predefined generic collection classes that we discussed in Chapter 16, rather than developing customized linked data structures.

21.2 Self-Referential Classes

- A self-referential class contains an instance variable that refers to another object of the same class type.
- For example, the generic class declaration

```
class Node<T>
{
   private T data;
   private Node<T> nextNode; // reference to next node

   public Node(T data) { /* constructor body */ }
   public void setData(T data) { /* method body */ }
   public T getData() { /* method body */ }
   public void setNext(Node<T> next) { /* method body */ }
   public Node<T> getNext() { /* method body */ }
} // end class Node<T>
```

declares class `Node`, which has two `private` instance variables—`data` (of the generic type `T`) and `Node<T>` variable `nextNode`.

21.2 Self-Referential Classes (cont.)

- Variable `nextNode` references a `Node<T>` object, an object of the same class being declared here—hence, the term "self-referential class."
- Field `nextNode` is a link—it "links" an object of type `Node<T>` to another object of the same type.
- Programs can link self-referential objects together to form such useful data structures as lists, queues, stacks and trees.
- Figure 21.1 illustrates two self-referential objects linked together to form a list.

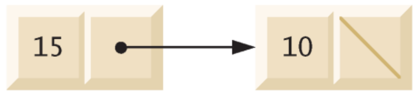


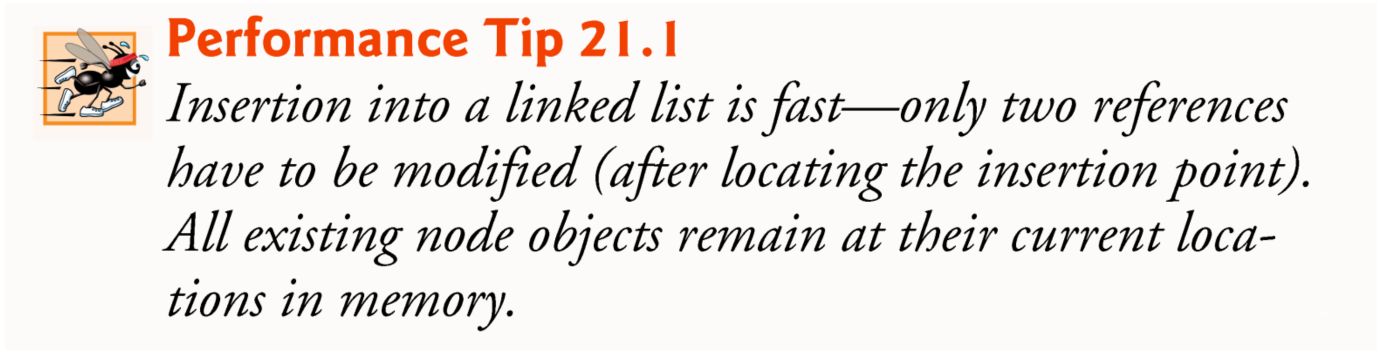
Fig. 21.1 | Self-referential-class objects linked together.

21.3 Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires dynamic memory allocation—allowing a program to obtain more memory space at execution time to hold new nodes and to release space no longer needed.
- The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available disk space in a virtual-memory system.
- Often the limits are much smaller, because the computer's available memory must be shared among many applications.

21.4 Linked Lists

- A linked list is a linear collection (i.e., a sequence) of self-referential-class objects, called nodes, connected by reference links—hence, the term “linked” list.
- Typically, a program accesses a linked list via a reference to its first node.
- The program accesses each subsequent node via the link reference stored in the previous node.
- By convention, the link reference in the last node of the list is set to `null` to indicate “end of list.”
- A linked list is appropriate when the number of data elements to be represented in the data structure is *unpredictable*.
- Linked lists become full only when the system has *insufficient memory* to satisfy dynamic storage allocation requests.



Performance Tip 21.1

Insertion into a linked list is fast—only two references have to be modified (after locating the insertion point). All existing node objects remain at their current locations in memory.

Performance Tip 21.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

21.4.1 Singly Linked Lists

- Linked list nodes normally are *not stored contiguously* in memory.
- Rather, they are logically contiguous.
- illustrates a linked list with several nodes.
- This diagram presents a singly linked list—each node contains one reference to the next node in the list.
- Often, linked lists are implemented as *doubly linked lists*—each node contains a reference to the next node in the list *and* a reference to the preceding one.

Performance Tip 21.3

The elements of an array are contiguous in memory. This allows immediate access to any array element, because its address can be calculated directly as its offset from the beginning of the array. Linked lists do not afford such immediate access—an element can be accessed only by traversing the list from the front (or the back in a doubly linked list).

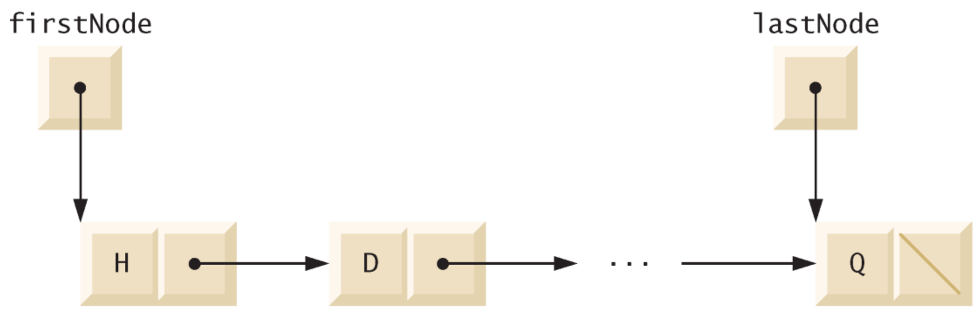


Fig. 21.2 | Linked-list graphical representation.

21.4.2 Implementing a Generic `List` Class

- The program of Figs. 21.3–21.5 uses an object of our generic `List` class to manipulate a list of miscellaneous objects.
- The program consists of four classes:
- `ListNode` (Fig. 21.3, lines 6–37),
- `List` (Fig. 21.3, lines 40–147),
- `EmptyListException` (Fig. 21.4)
- `ListTest` (Fig. 21.5).

```java
// Fig. 21.3: List.java
// ListNode and List class declarations.
package com.deitel.datastructures;

// class to represent one node in a list
class ListNode<T>
{
   // package access members; List can access these directly
   T data; // data for this node
   ListNode<T> nextNode; // reference to the next node in the list

   // constructor creates a ListNode that refers to object
   ListNode(T object)
   {
      this(object, null);
   }

   // constructor creates ListNode that refers to the specified
   // object and to the next ListNode
   ListNode(T object, ListNode<T> node)
   {
      data = object;
      nextNode = node;
   }
```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 1 of 7.)

```java

   // return reference to data in node
   T getData()
   {
      return data;
   }

   // return reference to next node in list
   ListNode<T> getNext()
   {
      return nextNode;
   }
} // end class ListNode<T>

// class List definition
public class List<T>
{
   private ListNode<T> firstNode;
   private ListNode<T> lastNode;
   private String name; // string like "list" used in printing

```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 2 of 7.)

```
   // constructor creates empty List with "list" as the name
   public List()
   {
      this("list");
   }

   // constructor creates an empty List with a name
   public List(String listName)
   {
      name = listName;
      firstNode = lastNode = null;
   }

   // insert item at front of List
   public void insertAtFront(T insertItem)
   {
      if (isEmpty()) // firstNode and lastNode refer to same object
         firstNode = lastNode = new ListNode<T>(insertItem);
      else // firstNode refers to new node
         firstNode = new ListNode<T>(insertItem, firstNode);
   }

```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 3 of 7.)

```java
   // insert item at end of List
   public void insertAtBack(T insertItem)
   {
      if (isEmpty()) // firstNode and lastNode refer to same object
         firstNode = lastNode = new ListNode<T>(insertItem);
      else // lastNode's nextNode refers to new node
         lastNode = lastNode.nextNode = new ListNode<T>(insertItem);
   }

   // remove first node from List
   public T removeFromFront() throws EmptyListException
   {
      if (isEmpty()) // throw exception if List is empty
         throw new EmptyListException(name);

      T removedItem = firstNode.data; // retrieve data being removed

      // update references firstNode and lastNode
      if (firstNode == lastNode)
         firstNode = lastNode = null;
      else
         firstNode = firstNode.nextNode;

      return removedItem; // return removed node data
   }
```

Fig. 21.3 | ListNode and List class declarations. (Part 4 of 7.)

```java

   // remove last node from List
   public T removeFromBack() throws EmptyListException
   {
      if (isEmpty()) // throw exception if List is empty
         throw new EmptyListException(name);

      T removedItem = lastNode.data; // retrieve data being removed

      // update references firstNode and lastNode
      if (firstNode == lastNode)
         firstNode = lastNode = null;
      else // locate new last node
      {
         ListNode<T> current = firstNode;

         // loop while current node does not refer to lastNode
         while (current.nextNode != lastNode)
            current = current.nextNode;

         lastNode = current; // current is new lastNode
         current.nextNode = null;
      }

```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 5 of 7.)

```
      return removedItem; // return removed node data
   }

   // determine whether list is empty
   public boolean isEmpty()
   {
      return firstNode == null; // return true if list is empty
   }

   // output list contents
   public void print()
   {
      if (isEmpty())
      {
         System.out.printf("Empty %s%n", name);
         return;
      }

      System.out.printf("The %s is: ", name);
      ListNode<T> current = firstNode;

```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 6 of 7.)

```
      // while not at end of list, output current node's data
      while (current != null)
      {
         System.out.printf("%s ", current.data);
         current = current.nextNode;
      }

      System.out.println();
   }
} // end class List<T>
```

Fig. 21.3 | `ListNode` and `List` class declarations. (Part 7 of 7.)

```java
// Fig. 21.4: EmptyListException.java
// Class EmptyListException declaration.
package com.deitel.datastructures;

public class EmptyListException extends RuntimeException
{
   // constructor
   public EmptyListException()
   {
      this("List"); // call other EmptyListException constructor
   }

   // constructor
   public EmptyListException(String name)
   {
      super(name + " is empty"); // call superclass constructor
   }
} // end class EmptyListException
```

Fig. 21.4 | Class `EmptyListException` declaration.

```java
// Fig. 21.5: ListTest.java
// ListTest class to demonstrate List capabilities.
import com.deitel.datastructures.List;
import com.deitel.datastructures.EmptyListException;

public class ListTest
{
   public static void main(String[] args)
   {
      List<Integer> list = new List<>();

      // insert integers in list
      list.insertAtFront(-1);
      list.print();
      list.insertAtFront(0);
      list.print();
      list.insertAtBack(1);
      list.print();
      list.insertAtBack(5);
      list.print();

```

Fig. 21.5 | `ListTest` class to demonstrate `List` capabilities. (Part 1 of 3.)

```java
      // remove objects from list; print after each removal
      try
      {
         int removedItem = list.removeFromFront();
         System.out.printf("%n%d removed%n", removedItem);
         list.print();

         removedItem = list.removeFromFront();
         System.out.printf("%n%d removed%n", removedItem);
         list.print();

         removedItem = list.removeFromBack();
         System.out.printf("%n%d removed%n", removedItem);
         list.print();

         removedItem = list.removeFromBack();
         System.out.printf("%n%d removed%n", removedItem);
         list.print();
      }
      catch (EmptyListException emptyListException)
      {
         emptyListException.printStackTrace();
      }
   }
} // end class ListTest
```

Fig. 21.5 | `ListTest` class to demonstrate `List` capabilities. (Part 2 of 3.)

```
The list is: -1
The list is: 0 -1
The list is: 0 -1 1
The list is: 0 -1 1 5
```

```
0 removed
The list is: -1 1 5

-1 removed
The list is: 1 5

5 removed
The list is: 1

1 removed
Empty list
```

Fig. 21.5 | `ListTest` class to demonstrate `List` capabilities. (Part 3 of 3.)

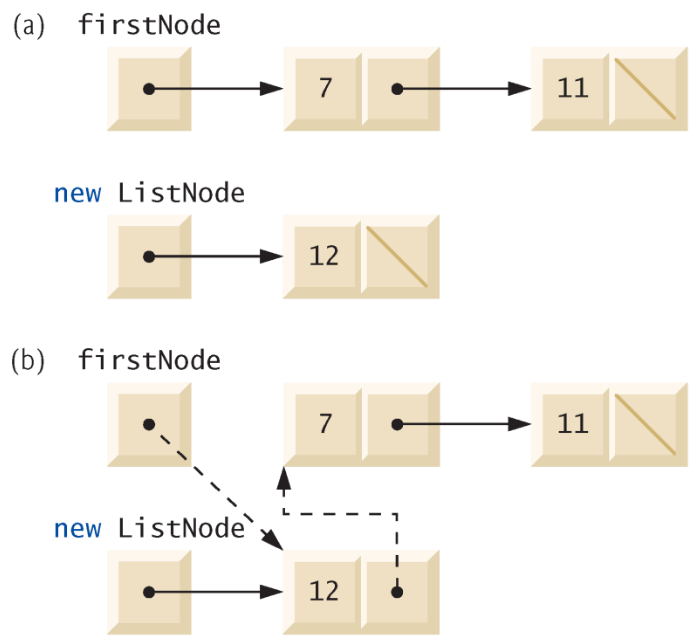


Fig. 21.6 | Graphical representation of operation `insertAtFront`.

21.4.6 List Method insertAtBack

- Method `insertAtBack` (lines 69–75 of Fig. 21.3) places a new node at the back of the list.
- The steps are:
  - Call `isEmpty` to determine whether the list is empty.
  - If the list is empty, assign to `firstNode` and `lastNode` the new `ListNode` that was initialized with `insertItem`.
  - If the list is not empty, link the new node into the list by assigning to `lastNode` and `lastNode.nextNode` the reference to the new `ListNode` that was initialized with `insertItem`.

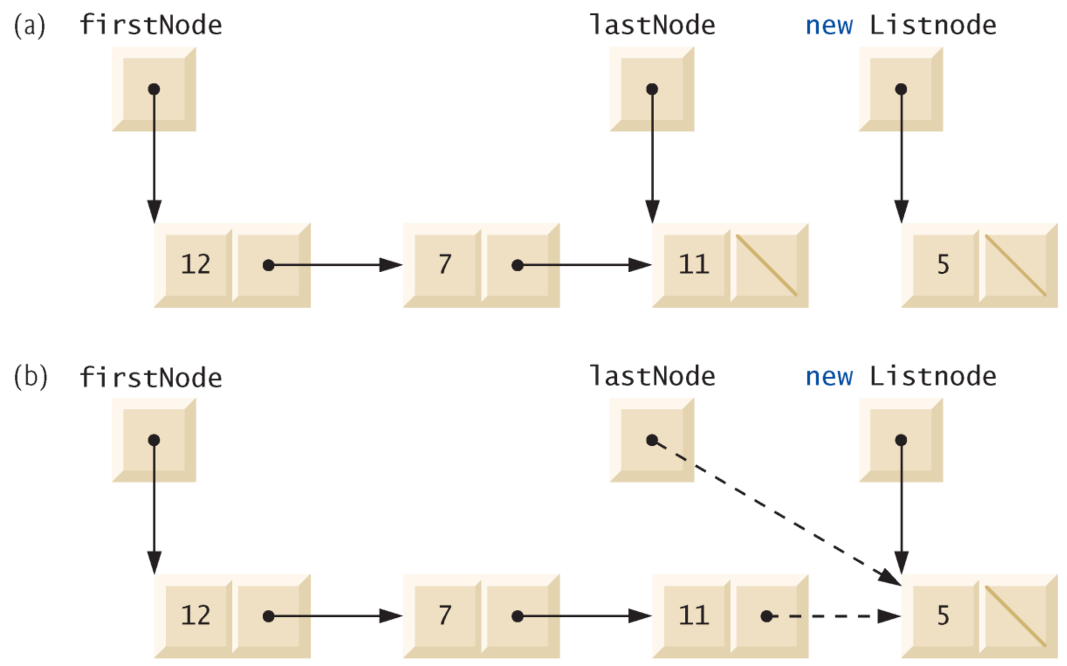


Fig. 21.7 | Graphical representation of operation `insertAtBack`.

21.4.7 `List` Method `removeFromFront`

- Method `removeFromFront` (lines 78–92 of) removes the first node of the list and returns a reference to the removed data.
- The steps are:
  - Assign `firstNode.data` to `removedItem`.
  - If `firstNode` and `lastNode` refer to the same object, the list has only one element at this time. So, the method sets `firstNode` and `lastNode` to `null` to remove the node from the list (leaving the list empty).
  - If the list has more than one node, then the method leaves reference `lastNode` as is and assigns the value of `firstNode.nextNode` to `firstNode`. Thus, `firstNode` references the node that was previously the second node in the list.
  - Return the `removedItem` reference (line 91).

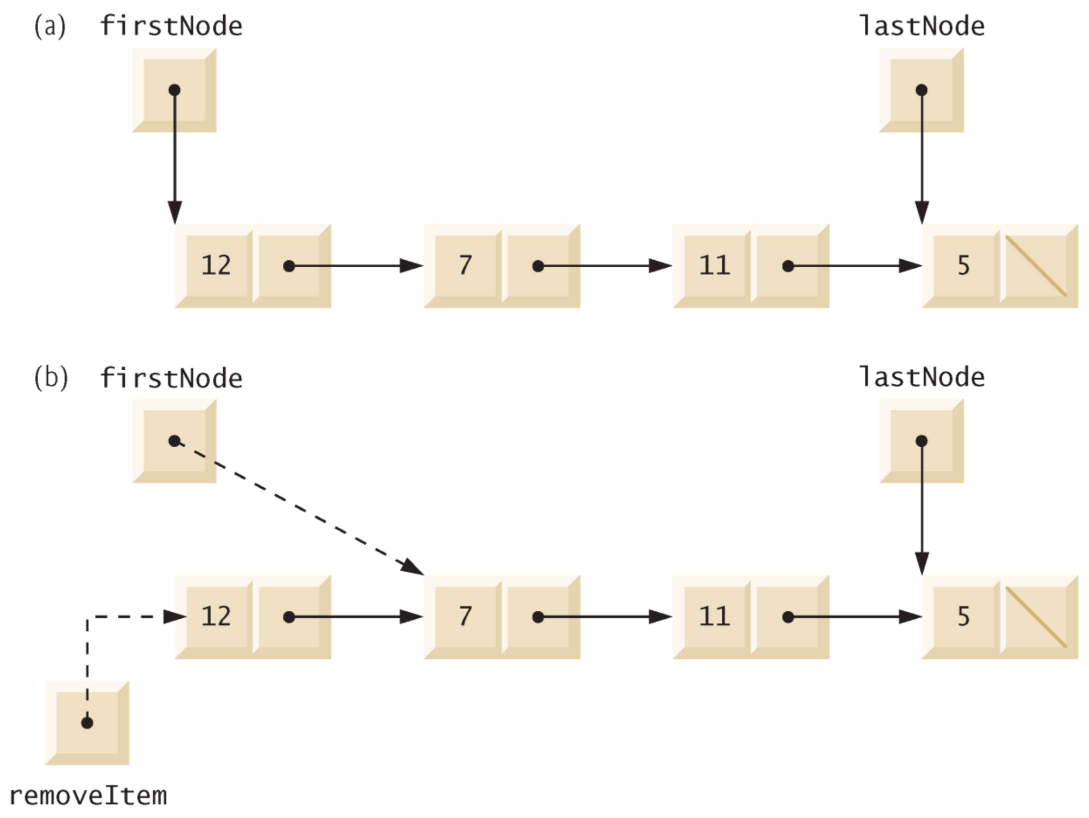


Fig. 21.8 | Graphical representation of operation `removeFromFront`.

21.4.8 List Method removeFromBack

- Method `removeFromBack` (lines 95–118 of Fig. 21.3) removes the last node of a list and returns a reference to the removed data.
- The steps are:
  - Assign `lastNode.data` to `removedItem`.
  - If the `firstNode` and `lastNode` refer to the same object, the list has only one element at this time. So, set `firstNode` and `lastNode` to `null` to remove that node from the list (leaving the list empty).
  - If the list has more than one node, create the `ListNode` reference `current` and assign it `firstNode`.
  - Now “walk the list” with `current` until it references the node before the last node. The `while` loop assigns `current.nextNode` to `current` as long as `current.nextNode` is not `lastNode`.

21.4 Linked Lists (cont.)

- After locating the second-to-last node, assign `current` to `lastNode` to update which node is last in the list.
- Set the `current.nextNode` to `null` to remove the last node from the list and terminate the list at the current node.
- Return the `removedItem` reference.

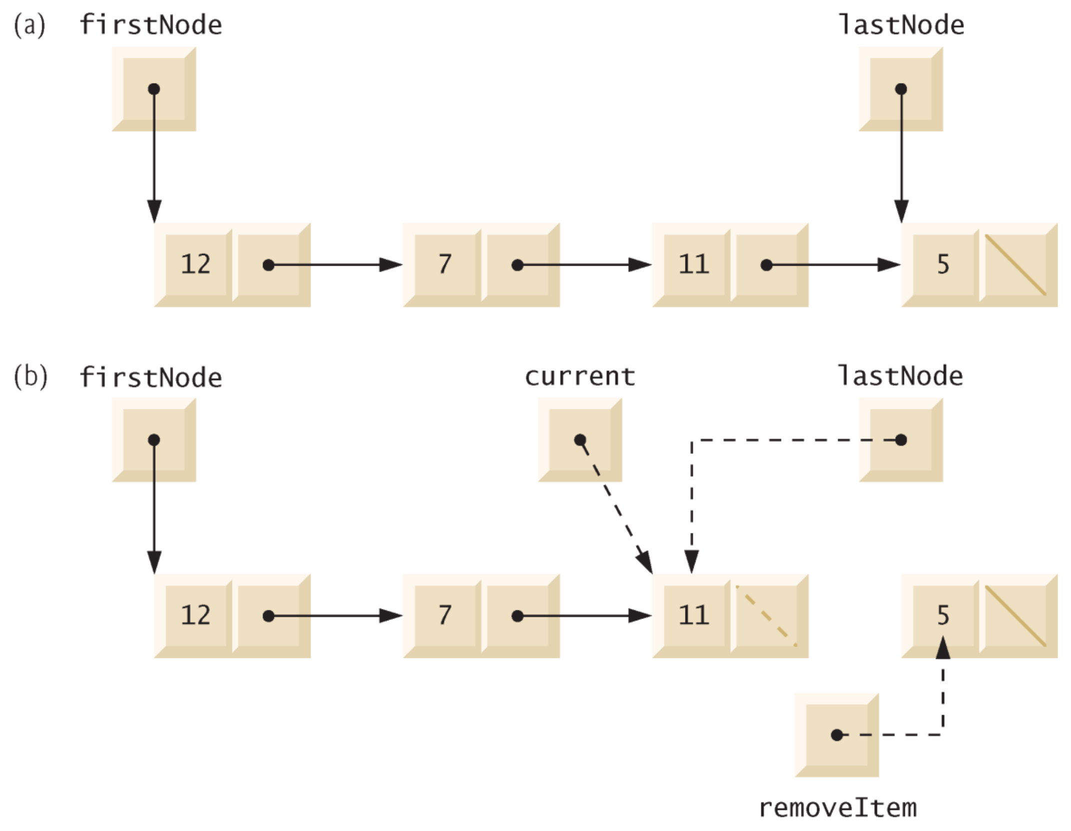


Fig. 21.9 | Graphical representation of operation `removeFromBack`.

21.4.9 List Method `print`

- Method `print` (lines 127–146 of Fig. 21.3) first determines whether the list is empty (lines 129–133).
- If so, `print` displays a message indicating that the list is empty and returns control to the calling method.
- Otherwise, `print` outputs the list's data. Line 136 creates `ListNode current` and initializes it with `firstNode`.

21.4.10 Creating Your Own Packages

- The Java API types (classes, interfaces and `enum`s) are organized in *packages* that group related types.
- Packages facilitate software reuse by enabling programs to `import` existing classes, rather than *copying* them into the folders of each program that uses them.
- Programmers use packages to organize program components, especially in large programs.
- Packages help you specify unique names for every type you declare, which (as we'll discuss) helps prevent class-name conflicts.

21.4.10 Creating Your Own Packages (cont.)

Steps for Declaring a Reusable Class

- Before a class can be imported into multiple programs, it must be placed in a package to make it reusable.
  - Declare one or more `public` types (classes, interfaces and `enum`s). Only public types can be reused outside the package in which they're declared.
  - Choose a unique package name and add a `package declaration` to the source-code file for each reusable type that should be part of the package.
  - Compile the types so that they're placed in the appropriate package directory.
  - Import the reusable types into a program and use them.

21.4.10 Creating Your Own Packages (cont.)

Step 1: Creating `public` Types for Reuse

- For *Step 1*, you declare the types that will be placed in the package, including both the reusable types and any supporting types.
- In Fig. 21.3, class `List` is `public`, so it's reusable outside its package. Class `ListNode`, however, is not `public`, so it can be used only by class `List` and any other types declared in the same package.
- In Fig. 21.4, class `EmptyListException` is public, so it, too, is reusable.
- If a source-code file contains more than one type, all types in the file are placed in the same package when the file is compiled.

21.4.10 Creating Your Own Packages (cont.)

Step 2: Adding the `package` Statements

- For Step 2, you provide a `package` declaration containing the package's name.
- All source-code files containing types that should be part of the same package must contain the same package declaration.
- Figures 21.3 and 21.4 each contain:
  - `package com.deitel.datastructures;`
- indicating that all the types declared in those files—`ListNode` and `List` in Fig. 21.3 and `EmptyListException` in Fig. 21.4—are part of the `com.deitel.datastructures` package.

21.4.10 Creating Your Own Packages (cont.)

Package Naming Conventions

- A package name's parts are separated by dots (.), and there typically are two or more parts.
- To ensure unique package names, you typically begin the name with your institution's or company's Internet domain name in reverse order—e.g., our domain name is `deitel.com`, so we begin our package names with `com.deitel`.
- After the reversed domain name, you can specify additional parts in a package name.
- We chose `datastructures` as the next part in our package name to indicate that classes `ListNode`, `List` and `EmptyListException` are from this data structures chapter.

21.4.10 Creating Your Own Packages (cont.)

Fully Qualified Names

- The `package` name is part of the fully qualified type name, so the name of class `List` is actually `com.deitel.datastructures.List`.
- You can use this fully qualified name in your programs, or you can import the class and use its simple name (the class name by itself—`List`) in the program.
- If another package also contains a `List` class, the fully qualified class names can be used to distinguish between the classes in the program and prevent a name conflict (also called a name collision).

21.4.10 Creating Your Own Packages (cont.)

Step 3: Compiling Packaged Types

- When a Java file containing a `package` declaration is compiled, the resulting class file is placed in a directory specified by the declaration.
- Classes in the package `com.deitel.datastructures` are placed in the directory
  - `com`
  - `    deitel`
  - `        datastructures`
- The names in the `package` declaration specify the exact location of the package's classes.

21.4.10 Creating Your Own Packages (cont.)

- The `javac` command-line option `-d` causes the compiler to create the directories based on the `package` declaration.
- The option also specifies where the top-level directory in the package name should be placed on your system—you may specify a relative or complete path to this location. For example, the command
  - `javac -d . List.java EmptyListException.java`
- specifies that the first directory in our package name (`com`) should be placed in the current directory.
  - The period (.) after -d in the preceding command represents the current directory on the Windows, UNIX, Linux and Mac OS X operating systems (and several others as well).
- Similarly, the command
  - `javac -d .. List.java EmptyListException.java`
- specifies that the first directory in our package name (`com`) should be placed in the parent directory.

21.4.10 Creating Your Own Packages (cont.)

Step 4: Importing Types from Your Package

- Once types are compiled into a package, they can be imported (*Step 4*).
- Class `ListTest` (Fig. 21.5) is in the *default package* because its `.java` file does not contain a `package` declaration.
- Because class `ListTest` is in a different package from List and `EmptyListException`, you must either import these classes so that class `ListTest` can use them (lines 3–4 of Fig. 21.5) or you must fully qualify the names `List` and `EmptyListException` everywhere they're used throughout class `ListTest`.
- For example, line 10 of Fig. 21.5 could have been written as:

```
com.deitel.datastructures.List<Integer> list =
   new com.deitel.datastructures.List<>();
```

21.4.10 Creating Your Own Packages (cont.)

Single-Type-Import vs. Type-Import-On-Demand Declarations

- Lines 3–4 of Fig. 21.5 are single-type-import declarations—they each specify one class to import. When a source-code file uses multiple classes from a package, you can import those classes with a a type-import-on-demand declaration of the form
  - import packagename.*;
- which uses an asterisk (*) at its end to inform the compiler that all public classes from the *packagename* package can be used in the file containing the `import`.
- Only those classes that are *used* are loaded at execution time.

Common Programming Error 21.1

Using the `import` *declaration* `import java.*;` *causes a compilation error. You must specify the full package name from which you want to import classes.*

Error-Prevention Tip 21.1

Using single-type-import declarations helps avoid naming conflicts by importing only the types you actually use in your code.

21.4.10 Creating Your Own Packages (cont.)

Specifying the Classpath When Compiling a Program

- When compiling `ListTest`, `javac` must locate the `.class` files for classes `List` and `EmptyListException` to ensure that class `ListTest` uses them correctly.
- The compiler uses a special object called a class loader to locate the classes it needs.
  - The class loader begins by searching the standard Java classes that are bundled with the JDK.
  - Then it searches for optional packages. Java provides an extension mechanism that enables new (optional) packages to be added to Java for development and execution purposes.
  - If the class is not found in the standard Java classes or in the extension classes, the class loader searches the `classpath`—a list of directories or archive files containing reusable types.
  - Each directory or archive file is separated from the next by a directory separator—a semicolon (;) on Windows or a colon (:) on UNIX/Linux/Mac OS X.

21.4.10 Creating Your Own Packages (cont.)

- By default, the `classpath` consists only of the current directory. However, the `classpath` can be modified by
  - providing the `-classpath` listOfDirectories option to the javac compiler or
  - setting the `CLASSPATH` environment variable (a special variable that you define and the operating system maintains so that programs can search for classes in the specified locations).
- If you compile `ListTest.java` without specifying the `-classpath` option, as in
  - `javac ListTest.java`
- the class loader assumes that the additional package(s) used by the `ListTest` program are in the *current directory*.

21.4.10 Creating Your Own Packages (cont.)

- To compile `ListTest.java`, use the command
  - `javac -classpath .;.. ListTest.java`
- on Windows or the command
  - `javac -classpath .:.. ListTest.java`
- on UNIX/Linux/Mac OS X. The . in the classpath enables the class loader to locate `ListTest` in the current directory.
- The .. enables the class loader to locate the contents of package `com.deitel.datastructures` in the parent directory.

Common Programming Error 21.2

Specifying an explicit classpath eliminates the current directory from the classpath. This prevents classes in the current directory (including packages in the current directory) from loading properly. If classes must be loaded from the current directory, include a dot (`.`*) in the classpath to specify the current directory.*

Software Engineering Observation 21.2

In general, it's a better practice to use the `-classpath` *option of the compiler, rather than the* `CLASSPATH` *environment variable, to specify the classpath for a program. This enables each program to have its own classpath.*

Error-Prevention Tip 21.2

Specifying the classpath with the CLASSPATH *environment variable can cause subtle and difficult-to-locate errors in programs that use different versions of the same package.*

21.4.10 Creating Your Own Packages (cont.)

Specifying the Classpath When Executing a Program

- When you execute a program, the JVM must be able to locate the `.class` files for the program's classes.
- Like the compiler, the `java` command uses a *class loader* that searches the standard classes and extension classes first, then searches the classpath (the current directory by default).
- The classpath can be specified explicitly by using the same techniques discussed for the compiler.
- As with the compiler, it's better to specify an individual program's classpath via command-line JVM options.
- You can specify the classpath in the java command via the -`classpath` or `-cp` command-line options, followed by a list of directories or archive files.

21.4.10 Creating Your Own Packages (cont.)

- Again, if classes must be loaded from the current directory, be sure to include a dot (.) in the classpath to specify the current directory.
- To execute the `ListTest` program, use the following command:
  - `java -classpath .:.. ListTest`

21.5 Stacks

- A stack is a constrained version of a list—*new nodes can be added to and removed from a stack only at the top*.
- A stack is referred to as a last-in, first-out (LIFO) data structure.
- A stack is not required to be implemented as a linked list—it can also be implemented using an array.
- The primary methods for manipulating a stack are push and pop, which add a new node to the top of the stack and remove a node from the top of the stack, respectively.

```java
// Fig. 21.10: StackInheritance.java
// StackInheritance extends class List.
package com.deitel.datastructures;

public class StackInheritance<T> extends List<T>
{
   // constructor
   public StackInheritance()
   {
      super("stack");
   }

   // add object to stack
   public void push(T object)
   {
      insertAtFront(object);
   }

   // remove object from stack
   public T pop() throws EmptyListException
   {
      return removeFromFront();
   }
} // end class StackInheritance
```

Fig. 21.10 | StackInheritance extends class List.

```java
// Fig. 21.11: StackInheritanceTest.java
// Stack manipulation program.
import com.deitel.datastructures.StackInheritance;
import com.deitel.datastructures.EmptyListException;

public class StackInheritanceTest
{
   public static void main(String[] args)
   {
      StackInheritance<Integer> stack = new StackInheritance<>();

      // use push method
      stack.push(-1);
      stack.print();
      stack.push(0);
      stack.print();
      stack.push(1);
      stack.print();
      stack.push(5);
      stack.print();

```

Fig. 21.11 | Stack manipulation program. (Part 1 of 3.)

```
      // remove items from stack
      try
      {
         int removedItem;

         while (true)
         {
            removedItem = stack.pop(); // use pop method
            System.out.printf("%n%d popped%n", removedItem);
            stack.print();
         }
      }
      catch (EmptyListException emptyListException)
      {
         emptyListException.printStackTrace();
      }
   }
} // end class StackInheritanceTest
```

Fig. 21.11 | Stack manipulation program. (Part 2 of 3.)

```
The stack is: -1
The stack is: 0 -1

The stack is: 1  0 -1
The stack is: 5  1  0 -1

5 popped
The stack is: 1 0 -1

1 popped
The stack is: 0 -1

0 popped
The stack is: -1

-1 popped
Empty stack
com.deitel.datastructures.EmptyListException: stack is empty
        at com.deitel.datastructures.List.removeFromFront(List.java:81)
        at com.deitel.datastructures.StackInheritance.pop(
           StackInheritance.java:22)
        at StackInheritanceTest.main(StackInheritanceTest.java:29)
```

Fig. 21.11 | Stack manipulation program. (Part 3 of 3.)

21.5 Stacks (cont.)

- You can also implement a class by reusing a list class through composition.
- Next program uses a `private` `List<T>` in class `StackComposition<T>`’s declaration.
- Composition enables us to hide the `List<T>` methods that should not be in our stack’s `public` interface.
- We provide `public` interface methods that use only the required `List<T>` methods.
- Implementing each stack method as a call to a `List<T>` method is called delegation.

```java
// Fig. 21.12: StackComposition.java
// StackComposition uses a composed List object.
package com.deitel.datastructures;

public class StackComposition<T>
{
   private List<T> stackList;

   // constructor
   public StackComposition()
   {
      stackList = new List<T>("stack");
   }

   // add object to stack
   public void push(T object)
   {
      stackList.insertAtFront(object);
   }

   // remove object from stack
   public T pop() throws EmptyListException
   {
      return stackList.removeFromFront();
   }
```

Fig. 21.12 | StackComposition uses a composed List object. (Part 1 of 2.)

```java

    // determine if stack is empty
    public boolean isEmpty()
    {
       return stackList.isEmpty();
    }

    // output stack contents
    public void print()
    {
       stackList.print();
    }
 } // end class StackComposition
```

Fig. 21.12 | `StackComposition` uses a composed `List` object. (Part 2 of 2.)

21.6 Queues

- A queue is similar to a checkout line in a supermarket—the cashier services the person at the beginning of the line first.
- Other customers enter the line only at the end and wait for service.
- *Queue nodes are removed only from the head (or front) of the queue and are inserted only at the tail (or end).*
- For this reason, a queue is a first-in, first-out (FIFO) data structure.
- The insert and remove operations are known as enqueue and dequeue.
- Next program creates a `Queue<T>` class that contains a `List<T>` () object and provides methods `enqueue`, `dequeue`, `isEmpty` and `print`.

```java
// Fig. 21.13: Queue.java
// Queue uses class List.
package com.deitel.datastructures;

public class Queue
{
   private List<T> queueList;

   // constructor
   public Queue()
   {
      queueList = new List<T>("queue");
   }

   // add object to queue
   public void enqueue(T object)
   {
      queueList.insertAtBack(object);
   }

```

Fig. 21.13 | Queue uses class List. (Part 1 of 2.)

```
    // remove object from queue
    public T dequeue() throws EmptyListException
    {
        return queueList.removeFromFront();
    }

    // determine if queue is empty
    public boolean isEmpty()
    {
        return queueList.isEmpty();
    }

    // output queue contents
    public void print()
    {
        queueList.print();
    }
} // end class Queue
```

Fig. 21.13 | `Queue` uses class `List`. (Part 2 of 2.)

```java
// Fig. 21.14: QueueTest.java
// Class QueueTest.
import com.deitel.datastructures.Queue;
import com.deitel.datastructures.EmptyListException;

public class QueueTest
{
   public static void main(String[] args)
   {
      Queue<Integer> queue = new Queue<>();

      // use enqueue method
      queue.enqueue(-1);
      queue.print();
      queue.enqueue(0);
      queue.print();
      queue.enqueue(1);
      queue.print();
      queue.enqueue(5);
      queue.print();

```

Fig. 21.14 | Queue processing program. (Part 1 of 3.)

```
         // remove objects from queue
         try
         {
            int removedItem;

            while (true)
            {
               removedItem = queue.dequeue(); // use dequeue method
               System.out.printf("%n%d dequeued%n", removedItem);
               queue.print();
            }
         }
         catch (EmptyListException emptyListException)
         {
            emptyListException.printStackTrace();
         }
      }
   } // end class QueueTest
```

Fig. 21.14 | Queue processing program. (Part 2 of 3.)

```
The queue is: -1
The queue is: -1 0
The queue is: -1 0 1
The queue is: -1 0 1 5

-1 dequeued
The queue is: 0 1 5

0 dequeued
The queue is: 1 5

1 dequeued
The queue is: 5

5 dequeued
Empty queue
com.deitel.datastructures.EmptyListException: queue is empty
        at com.deitel.datastructures.List.removeFromFront(List.java:81)
        at com.deitel.datastructures.Queue.dequeue(Queue.java:24)
        at QueueTest.main(QueueTest.java:29)
```

Fig. 21.14 | Queue processing program. (Part 3 of 3.)

21.7 Trees

- A tree is a nonlinear, two-dimensional data structure with special properties.
- Tree nodes contain two or more links.
- Binary tree 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 called a leaf node.
- Computer scientists normally draw trees from the root node down—the opposite of the way most trees grow in nature.

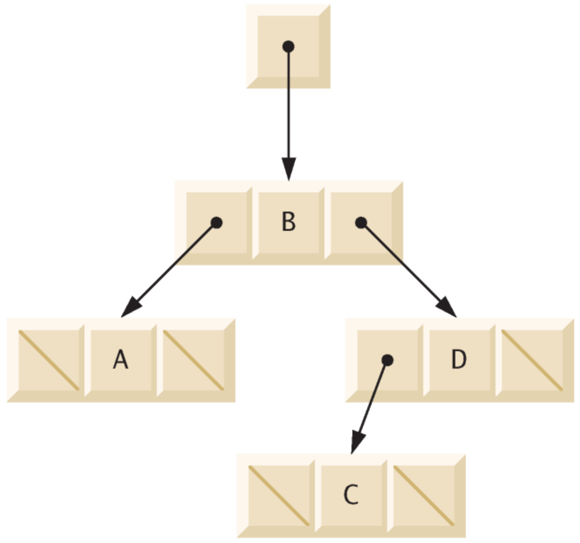


Fig. 21.15 | Binary tree graphical representation.

21.7 Trees (Cont.)

- A binary search 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.
- Fig. 21.16 illustrates a binary search tree with 12 integer values.
- Figs. 21.17 and 21.18 create a generic binary search tree class and use it to manipulate a tree of integers.
- The application in traverses the tree (i.e., walks through all its nodes) three ways—using recursive inorder, preorder and postorder traversals (trees are almost always processed recursively).

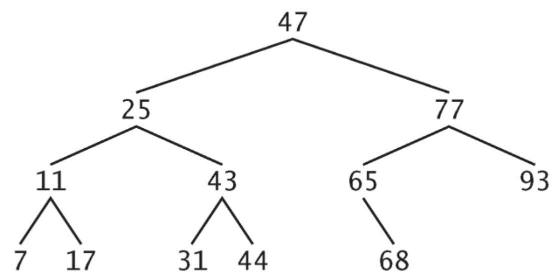


Fig. 21.16 | Binary search tree containing 12 values.

```java
// Fig. 21.17: Tree.java
// TreeNode and Tree class declarations for a binary search tree.
package com.deitel.datastructures;

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

   // constructor initializes data and makes this a leaf node
   public TreeNode(T nodeData)
   {
      data = nodeData;
      leftNode = rightNode = null; // node has no children
   }

```

Fig. 21.17 | TreeNode and Tree class declarations for a binary search tree. (Part 1 of 6.)

```java
   // locate insertion point and insert new node; ignore duplicate values
   public void insert(T insertValue)
   {
      // insert in left subtree
      if (insertValue.compareTo(data) < 0)
      {
         // insert new TreeNode
         if (leftNode == null)
            leftNode = new TreeNode<T>(insertValue);
         else // continue traversing left subtree recursively
            leftNode.insert(insertValue);
      }
      // insert in right subtree
      else if (insertValue.compareTo(data) > 0)
      {
         // insert new TreeNode
         if (rightNode == null)
            rightNode = new TreeNode<T>(insertValue);
         else // continue traversing right subtree recursively
            rightNode.insert(insertValue);
      }
   }
} // end class TreeNode
```

Fig. 21.17 | TreeNode and Tree class declarations for a binary search tree. (Part 2 of 6.)

```

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

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

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

```

Fig. 21.17 | `TreeNode` and `Tree` class declarations for a binary search tree. (Part 3 of 6.)

```java
   // begin preorder traversal
   public void preorderTraversal()
   {
      preorderHelper(root);
   }

   // recursive method to perform preorder traversal
   private void preorderHelper(TreeNode<T> node)
   {
      if (node == null)
         return;

      System.out.printf("%s ", node.data); // output node data
      preorderHelper(node.leftNode); // traverse left subtree
      preorderHelper(node.rightNode); // traverse right subtree
   }

   // begin inorder traversal
   public void inorderTraversal()
   {
      inorderHelper(root);
   }

```

Fig. 21.17 | `TreeNode` and `Tree` class declarations for a binary search tree. (Part 4 of 6.)

```java
   // recursive method to perform inorder traversal
   private void inorderHelper(TreeNode<T> node)
   {
      if (node == null)
         return;

      inorderHelper(node.leftNode); // traverse left subtree
      System.out.printf("%s ", node.data); // output node data
      inorderHelper(node.rightNode); // traverse right subtree
   }

   // begin postorder traversal
   public void postorderTraversal()
   {
      postorderHelper(root);
   }

```

Fig. 21.17 | `TreeNode` and `Tree` class declarations for a binary search tree. (Part 5 of 6.)

```java
    // recursive method to perform postorder traversal
    private void postorderHelper(TreeNode<T> node)
    {
       if (node == null)
          return;

       postorderHelper(node.leftNode); // traverse left subtree
       postorderHelper(node.rightNode); // traverse right subtree
       System.out.printf("%s ", node.data); // output node data
    }
 } // end class Tree
```

Fig. 21.17 | TreeNode and Tree class declarations for a binary search tree. (Part 6 of 6.)

```java
// Fig. 21.18: 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 SecureRandom();

      System.out.println("Inserting the following values: ");

      // insert 10 random integers from 0-99 in tree
      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%n");
      tree.preorderTraversal();
```

Fig. 21.18 | Binary tree test program. (Part 1 of 2.)

```java

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

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

```
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. 21.18 | Binary tree test program. (Part 2 of 2.)

21.7 Trees (Cont.)

- Class `Tree` requires `Comparable` objects, so that each value inserted in the tree can be compared with the existing values to determine the insertion point.
- Class `Tree`'s method `insertNode` (lines 56–62) first determines whether the tree is empty.
- If so, line 59 allocates a new `TreeNode`, initializes the node 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 21–41).
- This method uses recursion to determine the location for the new node in the tree and inserts the node at that location.
- A node can be inserted only as a leaf node in a binary search tree.

21.7 Trees (Cont.)

- `TreeNode` method `insert` compares the value to insert with the `data` value in the root node.
  - If the insert value is less than the root node data, the program determines if the left subtree is empty.
    - If so, allocates a new `TreeNode`, initializes it with the value being inserted and assigns the new node to reference `leftNode`.
    - Otherwise, recursively calls `insert` for the left subtree to insert the value into the left subtree.
  - If the insert value is greater than the root node data, the program determines if the right subtree is empty.
    - If so, allocates a new `TreeNode`, initializes it with the value being inserted and assigns the new node to reference `rightNode`.
    - Otherwise, recursively calls `insert` for the right subtree to insert the value in the right subtree.
  - If the `insertValue` is already in the tree, it's simply ignored.

21.7 Trees (Cont.)

- Methods `inorderTraversal`, `preorderTraversal` and `postorderTraversal` call `Tree` helper methods `inorderHelper`, `preorderHelper` and `postorderHelper`, respectively, to traverse the tree and print the node values.
  - Helper methods in class `Tree` enable you to start a traversal without having to pass the `root` node to the method.
  - Reference `root` is an implementation detail that a programmer should not be able to access.
- Methods `inorderTraversal`, `preorderTraversal` and `postorderTraversal` simply take the private `root` reference and pass it to the appropriate helper method to initiate a traversal of the tree.

21.7 Trees (Cont.)

- The base case for each helper method determines whether the reference it receives is `null` and, if so, returns immediately.
- Method `inorderHelper` defines the steps for an inorder traversal:
  - Traverse the left subtree with a call to `inorderHelper`
  - Process the value in the node
  - Traverse the right subtree with a call to `inorderHelper`
- 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. 21.19 is
  - `6 13 17 27 33 42 48`
- Note that the inorder traversal of a binary search tree 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.

21.7 Trees (Cont.)

- Method `preorderHelper` defines the steps for a preorder traversal:
    - Process the value in the node
    - Traverse the left subtree with a call to `preorderHelper`
    - Traverse the right subtree with a call to `preorderHelper`
- The preorder traversal processes the value in each node as the node is visited.
- After processing the value in a particular node, the preorder traversal processes the values in the left subtree, then processes the values in the right subtree.
- The preorder traversal of the tree in Fig. 21.19 is
    - `27 13 6 17 42 33 48`

21.7 Trees (Cont.)

- Method `postorderHelper` defines the steps for a postorder traversal:
  - Traverse the left subtree with a call to `postorderHelper`
  - Traverse the right subtree with a call to `postorderHelper`
  - Process the value in the node
- The postorder traversal processes the value in each node after the values of all that node's children are processed.
- The `post-orderTraversal` of the tree in Fig. 21.19 is
  - `6 17 13 33 48 42 27`

21.7 Trees (Cont.)

- A binary search tree facilitates duplicate elimination.
- Insert operation recognizes a duplicate, because a duplicate follows the same "go left" or "go right" decisions on each comparison as the original value did.
- 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.
  - A tightly packed binary search tree (e.g.,) 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 $2^{20} > 1{,}000{,}000$.

21.7 Trees (Cont.)

- 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.

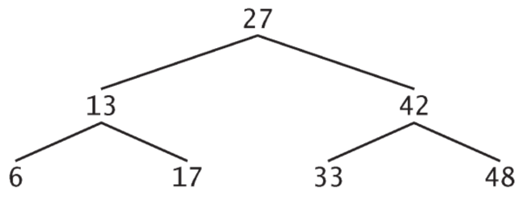


Fig. 21.19 | Binary search tree with seven values.