**Section 21.1 Introduction**

• Dynamic data structures (p. [870](http://proquest.safaribooksonline.com/9780133813036/ch21_html#page_870)) can grow and shrink at execution time.

• Linked lists (p. [870](http://proquest.safaribooksonline.com/9780133813036/ch21_html#page_870)) are collections of data items “linked up in a chain”—insertions and deletions can be made anywhere in a linked list.

• Stacks (p. [870](http://proquest.safaribooksonline.com/9780133813036/ch21_html#page_870)) are important in compilers and operating systems—insertions and deletions are made only at the top (p. [870](http://proquest.safaribooksonline.com/9780133813036/ch21_html#page_870)) of a stack.

• In a queue, insertions are made at the tail (p. [870](http://proquest.safaribooksonline.com/9780133813036/ch21_html#page_870)) and deletions are made from the head (p. [870](http://proquest.safaribooksonline.com/9780133813036/ch21_html#page_870)).

• Binary trees (p. [870](http://proquest.safaribooksonline.com/9780133813036/ch21_html#page_870)) facilitate high-speed searching and sorting, eliminating duplicate data items efficiently, representing file-system directories and compiling expressions into machine language.

#### Section 21.2 Self-Referential Classes

• A self-referential class (p. [871](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec1_html#page_871)) contains a reference that refers to another object of the same class type. Self-referential objects can be linked together to form dynamic data structures.

#### Section 21.3 Dynamic Memory Allocation

• The limit for dynamic memory allocation (p. [871](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec1_html#page_871)) can be as large as the available physical memory in the computer or the 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 users.

• If no memory is available, an OutOfMemoryError is thrown.

#### Section 21.4 Linked Lists

• A linked list is accessed via a reference to the first node of the list. Each subsequent node is accessed via the link-reference member stored in the previous node.

• By convention, the link reference in the last node of a list is set to null to mark the end of the list.

• A node can contain data of any type, including objects of other classes.

• A linked list is appropriate when the number of data elements to be stored is unpredictable. Linked lists are dynamic, so the length of a list can increase or decrease as necessary.

• The size of a “conventional” Java array cannot be altered—it’s fixed at creation time.

• List nodes normally are not stored in contiguous memory. Rather, they’re logically contiguous.

• Packages help manage program components and facilitate software reuse.

• Packages provide a convention for unique type names that helps prevent naming conflicts (p. [884](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_884)).

• Before a type can be imported into multiple programs, it must be placed in a package. There can be only one package declaration (p. [883](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_883)) in each Java source-code file, and it must precede all other declarations and statements in the file.

• Every package name should start with your Internet domain name in reverse order. After the domain name is reversed, you can choose any other names you want for your package.

• When compiling types in a package, the javac command-line option -d (p. [884](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_884)) specifies where to store the package and causes the compiler to create the package’s directories if they do not exist.

• The package name is part of the fully qualified type name (p. [884](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_884)).

• A single-type-import declaration (p. [885](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_885)) specifies one class to import. A type-import-on-demand declaration (p. [885](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_885)) imports only the classes that the program uses from a particular package.

• The compiler uses a class loader (p. [885](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_885)) to locate the classes it needs in the classpath. The classpath consists of a list of directories or archive files, each separated by a directory separator (p. [885](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_885)).

• The classpath for the compiler and JVM can be specified by providing the -classpath option (p. [886](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_886)) to the javac or java command, or by setting the CLASSPATH environment variable. If classes must be loaded from the current directory, include a dot (.) in the classpath.

#### Section 21.5 Stacks

• A stack is a last-in, first-out (LIFO) data structure (p. [886](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec11_html#page_886)). The primary methods used to manipulate a stack are push (p. [887](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec5_html#page_887)) and pop (p. [887](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec5_html#page_887)), which add a new node to the stack’s top and remove a node from the top, respectively. Method pop returns the removed node’s data.

• When a method call is made, the called method must know how to return to its caller, so the return address is pushed onto the program-execution stack. If a series of method calls occurs, the successive return values are pushed onto the stack in last-in, first-out order.

• The program-execution stack contains the space created for local variables on each invocation of a method. When the method returns to its caller, the space for that method’s local variables is popped off the stack, and those variables are no longer available to the program.

• Stacks are used by compilers to evaluate arithmetic expressions and generate machine-language code to process the expressions.

• The technique of implementing each stack method as a call to a List method is called delegation—the stack method invoked delegates (p. [890](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec13_html#page_890)) the call to the appropriate List method.

#### Section 21.6 Queues

• A queue (p. [890](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec13_html#page_890)) is similar to a checkout line in a supermarket—the first person in line is serviced first, and other customers enter the line only at the end and wait to be serviced.

• Queue nodes are removed only from the head (p. [890](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec13_html#page_890)) of the queue and are inserted only at the tail. For this reason, a queue is referred to as a first-in, first-out FIFOaaaa data structure.

• The insert and remove operations for a queue are known as enqueue (p. [891](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_891)) and dequeue (p. [891](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_891)).

• Queues have many uses in computer systems. Most computers have only a single processor, so only one application at a time can be serviced. Entries for the other applications are placed in a queue. The entry at the front of the queue is the next to receive service. Each entry gradually advances to the front of the queue as applications receive service.

#### Section 21.7 Trees

• A tree is a nonlinear, two-dimensional data structure. Tree nodes contain two or more links.

• A binary tree (p. [893](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_893)) is a tree whose nodes all contain two links. The root node (p. [893](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_893)) is the first node in a tree.

• Each link in the root node refers to a child (p. [893](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_893)). The left child (p. [893](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_893)) is the first node in the left subtree (p. [893](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_893)), and the right child (p. [893](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_893)) is the first node in the right subtree (p. [893](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_893)).

• The children of a node are called siblings (p. [893](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_893)). A node with no children is a leaf node (p. [893](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#page_893)).

• In a binary search tree (p. [894](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#page_894)) with no duplicate values, the values in any left subtree are less than the value in the subtree’s parent node, and the values in any right subtree are greater than the value in the subtree’s parent node. A node can be inserted only as a leaf node in a binary search tree.

• An inorder traversal (p. [894](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#page_894)) of a binary search tree processes the node values in ascending order.

• In a preorder traversal (p. [894](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#page_894)), the value in each node is processed as the node is visited. Then the values in the left subtree are processed, then the values in the right subtree.

• In a postorder traversal (p. [894](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#page_894)), the value in each node is processed after the values of its children.

• The binary search tree facilitates duplicate elimination (p. [899](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec17_html#page_899)). As the tree is created, attempts to insert a duplicate value are recognized, because a duplicate follows the same “go left” or “go right” decisions on each comparison as the original value did. Thus, the duplicate eventually is compared with a node containing the same value. The duplicate value can be discarded at this point.

• In a tightly packed tree (p. [900](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec17_html#page_900)), each level contains about twice as many elements as the previous one. So a tightly packed binary search tree with n elements has log2 n levels, and thus at most log2 ncomparisons would have to be made 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 210 > 1000. Searching a (tightly packed) 1,000,000-element binary search tree requires at most 20 comparisons, because 220 > 1,000,000.

### Self-Review Exercises

[**21.1**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec11_html#ch21ans1) Fill in the blanks in each of the following statements:

a) A self-\_\_\_\_\_\_\_\_\_\_ class is used to form dynamic data structures that can grow and shrink at execution time.

b) A(n) \_\_\_\_\_\_\_\_\_\_ is a constrained version of a linked list in which nodes can be inserted and deleted only from the start of the list.

c) A method that does not alter a linked list, but simply looks at it to determine whether it’s empty, is referred to as a(n) \_\_\_\_\_\_\_\_\_\_ method.

d) A queue is referred to as a(n) \_\_\_\_\_\_\_\_\_\_ data structure because the first nodes inserted are the first ones removed.

e) The reference to the next node in a linked list is referred to as a(n) \_\_\_\_\_\_\_\_\_\_.

f) Automatically reclaiming dynamically allocated memory in Java is called \_\_\_\_\_\_\_\_\_\_.

g) A(n) \_\_\_\_\_\_\_\_\_\_ is a constrained version of a linked list in which nodes can be inserted only at the end of the list and deleted only from the start of the list.

h) A(n) \_\_\_\_\_\_\_\_\_\_ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.

i) A stack is referred to as a(n) \_\_\_\_\_\_\_\_\_\_ data structure because the last node inserted is the first node removed.

j) The nodes of a(n) \_\_\_\_\_\_\_\_\_\_ tree contain two link members.

k) The first node of a tree is the \_\_\_\_\_\_\_\_\_\_ node.

l) Each link in a tree node refers to a(n) \_\_\_\_\_\_\_\_\_\_ or \_\_\_\_\_\_\_\_\_\_ of that node.

m) A tree node that has no children is called a(n) \_\_\_\_\_\_\_\_\_\_ node.

n) The three traversal algorithms we mentioned in the text for binary search trees are \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_.

o) When compiling types in a package, the javac command-line option \_\_\_\_\_\_\_\_\_\_ specifies where to store the package and causes the compiler to create the package’s directories if they do not exist.

p) The compiler uses a(n) \_\_\_\_\_\_\_\_\_\_ to locate the classes it needs in the classpath.

q) The classpath for the compiler and JVM can be specified with the \_\_\_\_\_\_\_\_\_\_ option to the javac orjava command, or by setting the \_\_\_\_\_\_\_\_\_\_ environment variable.

r) There can be only one \_\_\_\_\_\_\_\_\_\_ in a Java source-code file, and it must precede all other declarations and statements in the file.

[**21.2**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec11_html#ch21ans2) What are the differences between a linked list and a stack?

[**21.3**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec11_html#ch21ans3) What are the differences between a stack and a queue?

[**21.4**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec11_html#ch21ans4) Comment on how each of the following entities or concepts contributes to the reusability of data structures:

a) classes

b) inheritance

c) composition

[**21.5**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec11_html#ch21ans5) Provide the inorder, preorder and postorder traversals of the binary search tree of [Fig. 21.20](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec10_html#ch21fig20).

**Fig. 21.20** | Binary search tree with 15 nodes.

### Answers to Self-Review Exercises

[**21.1**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec10_html#ch21que1)

a) referential.

b) stack.

c) predicate.

d) first-in, first-out (FIFO).

e) link.

f) garbage collection.

g) queue.

h) tree

i) last-in, first-out (LIFO).

j) binary.

k) root.

l) child or subtree.

m) leaf.

n) inorder, preorder, postorder.

o) -d.

p) class loader.

q) -classpath, CLASSPATH.

r) package declaration.

[**21.2**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec10_html#ch21que2) It’s possible to insert a node anywhere in a linked list and remove a node from anywhere in a linked list. Nodes in a stack may be inserted only at the top of the stack and removed only from the top.

[**21.3**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec10_html#ch21que3) A queue is a FIFO data structure that has references to both its head and its tail, so that nodes may be inserted at the tail and deleted from the head. A stack is a LIFO data structure that has a single reference to the stack’s top, where both insertion and deletion of nodes are performed.

[**21.4**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec10_html#ch21que4)

a) Classes allow us to create as many data structure objects as we wish.

b) Inheritance enables a subclass to reuse the functionality from a superclass. Public and protected superclass methods can be accessed through a subclass to eliminate duplicate logic.

c) Composition enables a class to reuse code by storing a reference to an instance of another class in a field. Public methods of the instance can be called by methods in the class that contains the reference.

[**21.5**](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec10_html#ch21que5) The inorder traversal is

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#p904pro01a)

11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

The preorder traversal is

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#p904pro02a)

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

The postorder traversal is

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#p904pro03a)

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

### Exercises

**21.6 (Concatenating Lists)** Write a program that concatenates two linked list objects of characters. ClassListConcatenate should include a static method concatenate that takes references to both list objects as arguments and concatenates the second list to the first list.

**21.7 (Inserting into an Ordered List)** Write a program that inserts 25 random integers from 0 to 100 in order into a linked-list object. For this exercise, you’ll need to modify the List<T> class ([Fig. 21.3](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec3_html#ch21fig03)) to maintain an ordered list. Name the new version of the class SortedList.

**21.8 (Merging Ordered Lists)** Modify the SortedList class from [Exercise 21.7](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec12_html#ch21que7) to include a merge method that can merge the SortedList it receives as an argument with the SortedList that calls the method. Write an application to test method merge.

**21.9 (Copying a List Backward)** Write a static method reverseCopy that receives a List<T> as an argument and returns a copy of that List<T> with its elements reversed. Test this method in an application.

**21.10 (Printing a Sentence in Reverse Using a Stack)** Write a program that inputs a line of text and uses a stack to display the words of the line in reverse order.

**21.11 (Palindrome Tester)** Write a program that uses a stack to determine whether a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.

**21.12 (Infix-to-Postfix Converter)** Stacks are used by compilers to help in the process of evaluating expressions and generating machine-language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators and parentheses.

Humans generally write expressions like 3 + 4 and 7 / 9 in which the operator (+ or / here) is written between its operands—this is called infix notation. Computers “prefer” postfix notation, in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as 3 4 + and 7 9 /, respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation and evaluate the postfix version. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack object in support of its operation, but each uses the stack for a different purpose.

In this exercise, you’ll write a Java version of the infix-to-postfix conversion algorithm. In the next exercise, you’ll write a Java version of the postfix expression evaluation algorithm. In a later exercise, you’ll discover that code you write in this exercise can help you implement a complete working compiler.

Write class InfixToPostfixConverter to convert an ordinary infix arithmetic expression (assume a valid expression is entered) with single-digit integers such as

(6 + 2) \* 5 - 8 / 4

to a postfix expression. The postfix version (no parentheses are needed) of the this infix expression is

6 2 + 5 \* 8 4 / -

The program should read the expression into StringBuffer infix and use one of the stack classes implemented in this chapter to help create the postfix expression in StringBuffer postfix. The algorithm for creating a postfix expression is as follows:

a) Push a left parenthesis '(' onto the stack.

b) Append a right parenthesis ')' to the end of infix.

c) While the stack is not empty, read infix from left to right and do the following:

If the current character in infix is a digit, append it to postfix.

If the current character in infix is a left parenthesis, push it onto the stack.

If the current character in infix is an operator:

Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and append the popped operators to postfix.

Push the current character in infix onto the stack.

If the current character in infix is a right parenthesis:

Pop operators from the top of the stack and append them to postfix until a left parenthesis is at the top of the stack.

Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

+ addition

- subtraction

\* multiplication

/ division

^ exponentiation

% remainder

The stack should be maintained with stack nodes that each contain an instance variable and a reference to the next stack node. Some methods you may want to provide are as follows:

a) Method convertToPostfix, which converts the infix expression to postfix notation.

b) Method isOperator, which determines whether c is an operator.

c) Method precedence, which determines whether the precedence of operator1 (from the infix expression) is less than, equal to or greater than that of operator2 (from the stack). The method returns true if operator1 has lower precedence than operator2. Otherwise, false is returned.

d) Method peek (this should be added to the stack class), which returns the top value of the stack without popping the stack.

**21.13 (Postfix Evaluator)** Write class PostfixEvaluator that evaluates a postfix expression such as

6 2 + 5 \* 8 4 / -

The program should read a postfix expression consisting of digits and operators into a StringBuffer. Using modified versions of the stack methods implemented earlier in this chapter, the program should scan the expression and evaluate it (assume it’s valid). The algorithm is as follows:

a) Append a right parenthesis ')' to the end of the postfix expression. When the right parenthesis character is encountered, no further processing is necessary.

b) Until the right parenthesis is encountered, read the expression from left to right.

If the current character is a digit, do the following:

Push its integer value onto the stack (the integer value of a digit character is its

value in the Unicode character set minus the value of '0' in Unicode).

Otherwise, if the current character is an operator:

Pop the two top elements of the stack into variables x and y.

Calculate y operator x.

Push the result of the calculation onto the stack.

c) When the right parenthesis is encountered in the expression, pop the top value of the stack. This is the result of the postfix expression.

[Note: In b) above (based on the sample expression at the beginning of this exercise), if the operator is '/', the top of the stack is 4 and the next element in the stack is 40, then pop 4 into x, pop 40 into y, evaluate 40 / 4 and push the result, 10, back on the stack. This note also applies to operator '-'.] The arithmetic operations allowed in an expression are: + (addition), - (subtraction), \* (multiplication), / (division), ^(exponentiation) and % (remainder).

The stack should be maintained with one of the stack classes introduced in this chapter. You may want to provide the following methods:

a) Method evaluatePostfixExpression, which evaluates the postfix expression.

b) Method calculate, which evaluates the expression op1 operator op2.

**21.14 (Postfix Evaluator Modification)** Modify the postfix evaluator program of [Exercise 21.13](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec12_html#ch21que13) so that it can process integer operands larger than 9.

**21.15 (Supermarket Simulation)** Write a program that simulates a checkout line at a supermarket. The line is a queue object. Customers (i.e., customer objects) arrive in random integer intervals of from 1 to 4 minutes. Also, each customer is serviced in random integer intervals of from 1 to 4 minutes. Obviously, the rates need to be balanced. If the average arrival rate is larger than the average service rate, the queue will grow infinitely. Even with “balanced” rates, randomness can still cause long lines. Run the supermarket simulation for a 12-hour day (720 minutes), using the following algorithm:

a) Choose a random integer between 1 and 4 to determine the minute at which the first customer arrives.

b) At the first customer’s arrival time, do the following:

Determine customer’s service time (random integer from 1 to 4).

Begin servicing the customer.

Schedule arrival time of next customer (random integer 1 to 4 added to the current time).

c) For each simulated minute of the day, consider the following:

If the next customer arrives, proceed as follows:

Say so.

Enqueue the customer.

Schedule the arrival time of the next customer.

If service was completed for the last customer, do the following:

Say so.

Dequeue next customer to be serviced.

Determine customer’s service completion time (random integer from 1 to 4 added to the current time).

Now run your simulation for 720 minutes and answer each of the following:

a) What is the maximum number of customers in the queue at any time?

b) What is the longest wait any one customer experiences?

c) What happens if the arrival interval is changed from 1 to 4 minutes to 1 to 3 minutes?

**21.16 (Allowing Duplicates in a Binary Tree)** Modify [Figs. 21.17](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig17) and [21.18](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig18) to allow the binary tree to contain duplicates.

**21.17 (Processing a Binary Search Tree of** ***String*s)** Write a program based on the program of [Figs. 21.17](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig17) and [21.18](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig18) that inputs a line of text, tokenizes it into separate words, inserts the words in a binary search tree and prints the inorder, preorder and postorder traversals of the tree.

**21.18 (Duplicate Elimination)** In this chapter, we saw that duplicate elimination is straightforward when creating a binary search tree. Describe how you’d perform duplicate elimination when using only a one-dimensional array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.

**21.19 (Depth of a Binary Tree)** Modify [Figs. 21.17](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig17) and [21.18](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig18) so the Tree class provides a methodgetDepth that determines how many levels are in the tree. Test the method in an application that inserts 20 random integers in a Tree.

**21.20 (Recursively Print a List Backward)** Modify the List<T> class of [Fig. 21.3](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec3_html#ch21fig03) to include methodprintListBackward that recursively outputs the items in a linked-list object in reverse order. Write a test program that creates a list of integers and prints the list in reverse order.

**21.21 (Recursively Search a List)** Modify the List<T> class of [Fig. 21.3](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec3_html#ch21fig03) to include method search that recursively searches a linked-list object for a specified value. The method should return a reference to the value if it’s found; otherwise, it should return null. Use your method in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.

**21.22 (Binary Tree Delete)** In this exercise, we discuss deleting items from binary search trees. The deletion algorithm is not as straightforward as the insertion algorithm. Three cases are encountered when deleting an item—the item is contained in a leaf node (i.e., it has no children), or in a node that has one child or in a node that has two children.

If the item to be deleted is contained in a leaf node, the node is deleted and the reference in the parent node is set to null.

If the item to be deleted is contained in a node with one child, the reference in the parent node is set to reference the child node and the node containing the data item is deleted. This causes the child node to take the place of the deleted node in the tree.

The last case is the most difficult. When a node with two children is deleted, another node in the tree must take its place. However, the reference in the parent node cannot simply be assigned to reference one of the children of the node to be deleted. In most cases, the resulting binary search tree would not embody the following characteristic of binary search trees (with no duplicate values): The values in any left subtree are less than the value in the parent node, and the values in any right subtree are greater than the value in the parent node.

Which node is used as a replacement node to maintain this characteristic? It’s either the node containing the largest value in the tree less than the value in the node being deleted, or the node containing the smallest value in the tree greater than the value in the node being deleted. Let’s consider the node with the smaller value. In a binary search tree, the largest value less than a parent’s value is located in the left subtree of the parent node and is guaranteed to be contained in the rightmost node of the subtree. This node is located by walking down the left subtree to the right until the reference to the right child of the current node is null. We’re now referencing the replacement node, which is either a leaf node or a node with one child to its left. If the replacement node is a leaf node, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.

b) Set the reference in the parent of the node being deleted to reference the replacement node.

c) Set the reference in the parent of the replacement node to null.

d) Set the reference to the right subtree in the replacement node to reference the right subtree of the node to be deleted.

e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

The deletion steps for a replacement node with a left child are similar to those for a replacement node with no children, but the algorithm also must move the child into the replacement node’s position in the tree. If the replacement node is a node with a left child, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.

b) Set the reference in the parent of the node being deleted to refer to the replacement node.

c) Set the reference in the parent of the replacement node to reference the left child of the replacement node.

d) Set the reference to the right subtree in the replacement node to reference the right subtree of the node to be deleted.

e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

Write method deleteNode, which takes as its argument the value to delete. Method deleteNode should locate in the tree the node containing the value to delete and use the algorithms discussed here to delete the node. If the value is not found in the tree, the method should display a message saying so. Modify the program of [Figs. 21.17](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig17) and [21.18](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig18) to use this method. After deleting an item, call the methodsinorderTraversal, preorderTraversal and postorderTraversal to confirm that the delete operation was performed correctly.

**21.23 (Binary Tree Search)** Modify class Tree of [Fig. 21.17](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig17) to include method contains, which attempts to locate a specified value in a binary-search-tree object. The method should take as an argument a search key to locate. If the node containing the search key is found, the method should return a reference to that node’s data; otherwise, it should return null.

**21.24****(Level-Order Binary Tree Traversal)** The program of [Figs. 21.17](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig17) and [21.18](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig18) illustrated three recursive methods of traversing a binary tree—inorder, preorder and postorder traversals. This exercise presents the level-order traversal of a binary tree, in which the node values are printed level by level, starting at the root node level. The nodes on each level are printed from left to right. The level-order traversal is not a recursive algorithm. It uses a queue object to control the output of the nodes. The algorithm is as follows:

a) Insert the root node in the queue.

b) While there are nodes left in the queue, do the following:

Get the next node in the queue.

Print the node’s value.

If the reference to the left child of the node is not null:

Insert the left child node in the queue.

If the reference to the right child of the node is not null:

Insert the right child node in the queue.

Write method levelOrder to perform a level-order traversal of a binary tree object. Modify the program of[Figs. 21.17](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig17) and [21.18](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig18) to use this method. [Note: You’ll also need to use the queue-processing methods of[Fig. 21.13](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#ch21fig13) in this program.]

**21.25 (Printing Trees)** Modify class Tree of [Fig. 21.17](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec7_html#ch21fig17) to include a recursive outputTree method to display a binary tree object. The method should output the tree row by row, with the top of the tree at the left of the screen and the bottom of the tree toward the right. Each row is output vertically. For example, the binary tree illustrated in [Fig. 21.20](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec10_html#ch21fig20) is output as shown in [Fig. 21.21](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec12_html#ch21fig21).

               99  
          97  
               92  
     83  
               72  
          71  
               69  
49  
               44  
          40  
               32  
     28  
               19  
          18  
               11

**Fig. 21.21** | Sample output of recursive method outputTree.

The rightmost leaf node appears at the top of the output in the rightmost column and the root node appears at the left of the output. Each column starts five spaces to the right of the preceding column. Method outputTree should receive an argument totalSpaces representing the number of spaces preceding the value to be output. (This variable should start at zero so that the root node is output at the left of the screen.) The method uses a modified inorder traversal to output the tree—it starts at the rightmost node in the tree and works back to the left. The algorithm is as follows:

While the reference to the current node is not null, perform the following:

Recursively call outputTree with the right subtree of the current node and totalSpaces + 5.

Use a for statement to count from 1 to totalSpaces and output spaces.

Output the value in the current node.

Set the reference to the current node to refer to the left subtree of the current node.

Increment totalSpaces by 5.

**21.26 (Insert/Delete Anywhere in a Linked List)** Our linked-list class allowed insertions and deletions at only the front and the back of the linked list. These capabilities were convenient for us when we used inheritance or composition to produce a stack class and a queue class with minimal code simply by reusing the list class. Linked lists are normally more general than those we provided. Modify the linked-list class we developed in this chapter to handle insertions and deletions anywhere in the list. Create diagrams comparable to [Figs. 21.6](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec6_html#ch21fig06) (insertAtFront), [21.7](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec7_html#ch21fig07) (insertAtBack), [21.8](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec8_html#ch21fig08) (removeFromFront) and [21.9](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec9_html#ch21fig09)(removeFromBack) that show how to insert a new node in the middle of a linked list and how to remove an existing node from the middle of a linked list.

**21.27****(Lists and Queues without Tail References)** Our linked-list implementation ([Fig. 21.3](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec3_html#ch21fig03)) used both afirstNode and a lastNode. The lastNode was useful for the insertAtBack and removeFromBack methods of the List class. The insertAtBack method corresponds to the enqueue method of the Queue class. Rewrite theList class so that it does not use a lastNode. Thus, any operations on the tail of a list must begin searching the list from the front. Does this affect our implementation of the Queue class ([Fig. 21.13](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#ch21fig13))?

**21.28****(Performance of Binary Tree Sorting and Searching)** One problem with the binary tree sort is that the order in which the data is inserted affects the shape of the tree—for the same collection of data, different orderings can yield binary trees of dramatically different shapes. The performance of the binary tree sorting and searching algorithms is sensitive to the shape of the binary tree. What shape would a binary tree have if its data were inserted in increasing order? in decreasing order? What shape should the tree have to achieve maximal searching performance?

**21.29 (Indexed Lists)** As presented in the text, linked lists must be searched sequentially. For large lists, this can result in poor performance. A common technique for improving list-searching performance is to create and maintain an index to the list. An index is a set of references to key places in the list. For example, an application that searches a large list of names could improve performance by creating an index with 26 entries—one for each letter of the alphabet. A search operation for a last name beginning with ‘Y’ would then first search the index to determine where the ‘Y’ entries began, then “jump into” the list at that point and search linearly until the desired name was found. This would be much faster than searching the linked list from the beginning. Use the List class of [Fig. 21.3](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec3_html#ch21fig03) as the basis of an IndexedList class. Write a program that demonstrates the operation of indexed lists. Be sure to include methods insertInIndexedList,searchIndexedList and deleteFromIndexedList.

**21.30****(Queue Class that Inherits from a List Class)** In [Section 21.5](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec5_html#ch21lev1sec5), we created a stack class from classList with inheritance ([Fig. 21.10](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec12_html#ch21fig10)) and with composition ([Fig. 21.12](http://proquest.safaribooksonline.com/9780133813036/ch21lev2sec13_html#ch21fig12)). In [Section 21.6](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#ch21lev1sec6) we created a queue class from class List with composition ([Fig. 21.13](http://proquest.safaribooksonline.com/9780133813036/ch21lev1sec6_html#ch21fig13)). Create a queue class by inheriting from class List. What are the differences between this class and the one we created with composition?

### Special Section: Building Your Own Compiler

In [Exercises 7.36](http://proquest.safaribooksonline.com/9780133813036/ch07lev2sec54_html#ch07que36)–[7.38](http://proquest.safaribooksonline.com/9780133813036/ch07lev2sec54_html#ch07que38), we introduced Simpletron Machine Language (SML), and you implemented a Simpletron computer simulator to execute SML programs. In Exercises 21.31–21.35, we build a compiler that converts programs written in a high-level programming language to SML. This section “ties” together the entire programming process. You’ll write programs in this new high-level language, compile them on the compiler you build and run them on the simulator you built in [Exercise 7.37](http://proquest.safaribooksonline.com/9780133813036/ch07lev2sec54_html#ch07que37). You should make every effort to implement your compiler in an object-oriented manner. [Note: Due to the size of the descriptions for Exercises 21.31–21.35, we’ve posted them in a PDF document located at [www.deitel.com/books/jhtp10/](http://www.deitel.com/books/jhtp10/).]