

2-4 TREES AND B-TREES

Objectives

- To know what a 2-4 tree is (§42.1).
- To design the **Tree24** class that implements the **Tree** interface (§42.2).
- To search an element in a 2-4 tree (§42.3).
- To insert an element in a 2-4 tree and know how to split a node (§42.4).
- To delete an element from a 2-4 tree and know how to perform transfer and fusion operations (§42.5).
- To traverse elements in a 2-4 tree (§42.6).
- To implement and test the **Tree24** class (§§42.7-42.8).
- To analyze the complexity of the 2-4 tree (§42.9).
- To use B-trees for indexing large amount of data (§42.10).

CHAPTER 42





42.1 Introduction

A 2-4 tree, also known as a 2-3-4 tree, is a completely balanced search tree with all leaf nodes appearing on the same level.

In a 2-4 tree, a node may have one, two, or three elements. An interior 2-node contains one element and two children. An interior 3-node contains two elements and three children. An interior 4-node contains three elements and four children, as shown in Figure 42.1.

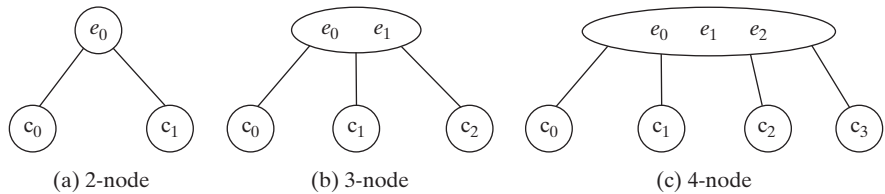


FIGURE 42.1 An interior node of a 2-4 tree has two, three, or four children.

Each child is a sub 2-4 tree, possibly empty. The root node has no parent, and leaf nodes have no children. The elements in the tree are distinct. The elements in a node are ordered such that

$$E(c_0) < e_0 < E(c_1) < e_1 < E(c_2) < e_2 < E(c_3)$$

where $E(c_k)$ denote the elements in c_k . Figure 42.2 shows an example of a 2-4 tree. c_k is called the *left subtree* of e_k and c_{k+1} is called the *right subtree* of e_k .

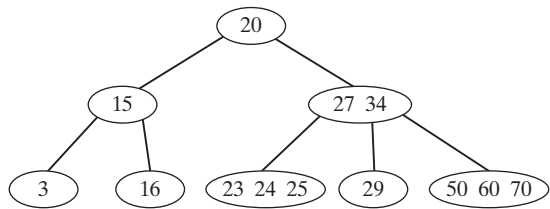


FIGURE 42.2 A 2-4 tree is a full complete search tree.

In a binary tree, each node contains one element. A 2-4 tree tends to be shorter than a corresponding binary search tree, since a 2-4 tree node may contain two or three elements.



Pedagogical Note

Run from <http://liveexample.pearsoncmg.com/dsanimation/24Tree.html> to see how a 2-4 tree works, as shown in Figure 42.3.



42.2 Designing Classes for 2-4 Trees

The **Tree24** class defines a 2-4 tree and provides methods for searching, inserting, and deleting elements.

The **Tree24** class can be designed by implementing the **Tree** interface, as shown in Figure 42.4. The **Tree** interface was defined in Listing 27.3 `Tree.java`. The **Tree24Node** class defines tree nodes. The elements in the node are stored in a list named **elements** and the links to the child nodes are stored in a list named **child**, as shown in Figure 42.5.

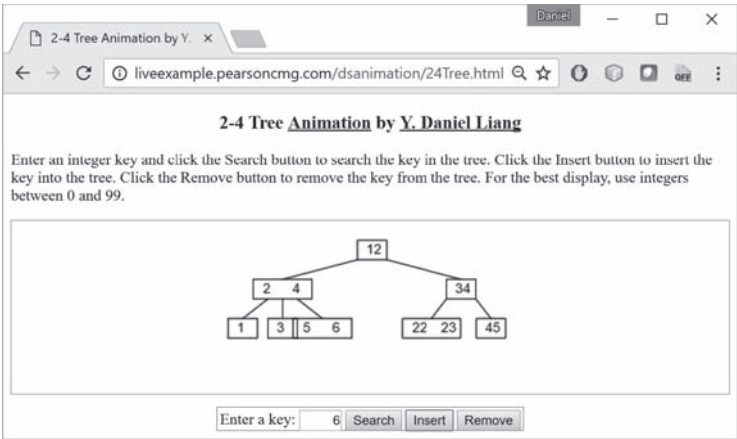


FIGURE 42.3 The animation tool enables you to insert, delete, and search elements in a 2–4 tree visually.

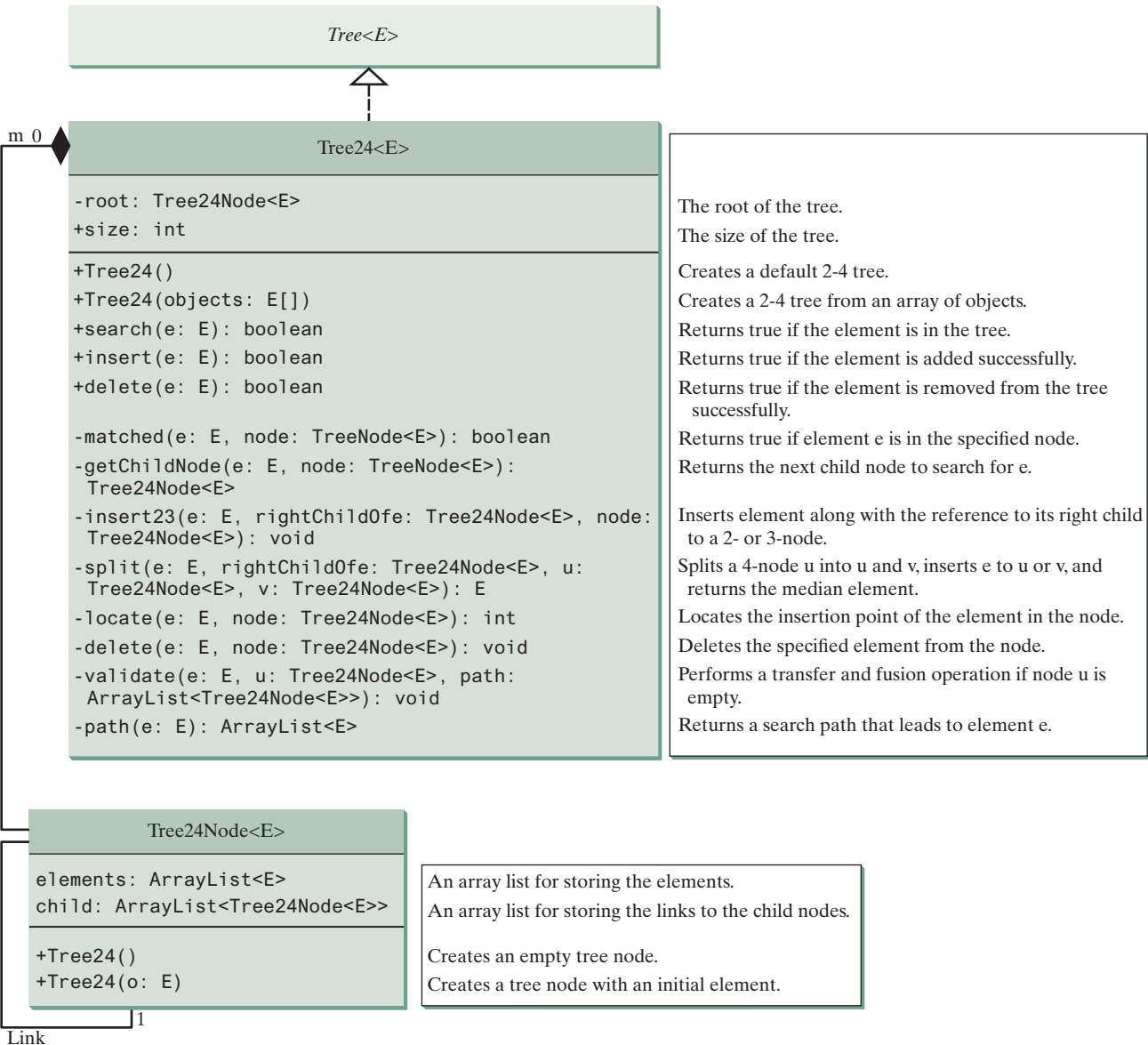


FIGURE 42.4 The **Tree24** class implements **Tree**.

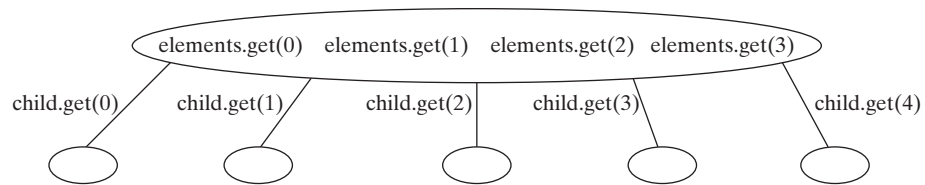


FIGURE 42.5 A 2-4 tree node stores the elements and the links to the child nodes in array lists.



- 42.2.1** What is a 2-4 tree? What are a 2-node, 3-node, and 4-node?
- 42.2.2** Describe the data fields in the `Tree24` class and those in the `Tree24Node` class.
- 42.2.3** What is the minimum number of elements in a 2-4 tree of height 5? What is the maximum number of elements in a 2-4 tree of height 5?



42.3 Searching an Element

Searching an element in a 2-4 tree is similar to searching an element in a binary tree. The difference is that you have to search an element within a node in addition to searching elements along the path.

To search an element in a 2-4 tree, you start from the root and scan down. If an element is not in the node, move to an appropriate subtree. Repeat the process until a match is found or you arrive at an empty subtree. The algorithm is described in Listing 42.1.

LISTING 42.1 Searching an Element in a 2-4 tree

```

1  boolean search(E e) {
2      current = root; // Start from the root
3
4      while (current != null) {
5          if (match(e, current)) { // Element is in the node
6              return true; // Element is found
7          }
8          else {
9              current = getChildNode(e, current); // Search in a subtree
10         }
11     }
12     return false; // Element is not in the tree
13 }
  
```

The `match(e, current)` method checks whether element `e` is in the current node. The `getChildNode(e, current)` method returns the root of the subtree for further search. Initially, let `current` point to the root (line 2). Repeat searching the element in the current node until `current` is `null` (line 4) or the element matches an element in the current node.



42.4 Inserting an Element into a 2-4 tree

Inserting an element involves locating a leaf node and inserting the element into the leaf node.

To insert an element `e` to a 2-4 tree, locate a leaf node in which the element will be inserted. If the leaf node is a 2-node or 3-node, simply insert the element into the node. If the node is a 4-node, inserting a new element would cause an *overflow*. To resolve overflow, perform a *split* operation as follows:

- Let u be the leaf 4-node in which the element will be inserted and $parentOfu$ be the parent of u , as shown in Figure 42.6(a).

- Create a new node named v ; move e_2 to v .
- If $e < e_1$, insert e to u ; otherwise insert e to v . Assume that $e_0 < e < e_1$, e is inserted into u , as shown in Figure 42.6(b).
- Insert e_1 along with its right child (i.e., v) to the parent node, as shown in Figure 42.6(b).

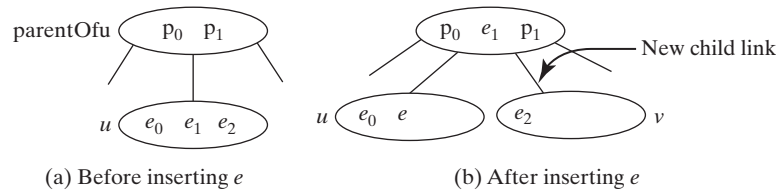


FIGURE 42.6 The splitting operation creates a new node and inserts the median element to its parent.

The parent node is a 3-node in Figure 42.6. So, there is room to insert e to the parent node. What happens if it is a 4-node, as shown in Figure 42.7? This requires that the parent node be split. The process is the same as splitting a leaf 4-node, except that you must also insert the element along with its right child.

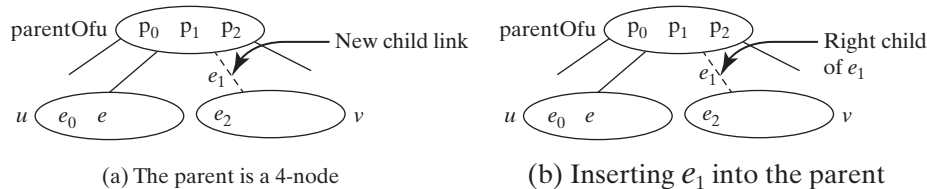


FIGURE 42.7 Insertion process continues if the parent node is a 4-node.

The algorithm can be modified as follows:

- Let u be the 4-node (*leaf or nonleaf*) in which the element will be inserted and $parentOfu$ be the parent of u , as shown in Figure 42.8(a).
- Create a new node named v , move e_2 and its children c_2 and c_3 to v .
- If $e < e_1$, insert e along with its right child link to u ; otherwise insert e along with its right child link to v , as shown in Figure 42.6(b), (c), (d) for the cases $e_0 < e < e_1$, $e_1 < e < e_2$, and $e_2 < e$, respectively.
- Insert e_1 along with its right child (i.e., v) to the parent node, recursively.

Listing 42.2 gives an algorithm for inserting an element.

LISTING 42.2 Inserting an Element to a 2–4 tree

```

1 public boolean insert(E e) {
2     if (root == null)
3         root = new Tree24Node<E>(e); // Create a new root for element
4     else {
5         Locate leafNode for inserting e
6         insert(e, null, leafNode); // The right child of e is null
7     }
8
9     size++; // Increase size
10    return true; // Element inserted
11 }
12
```

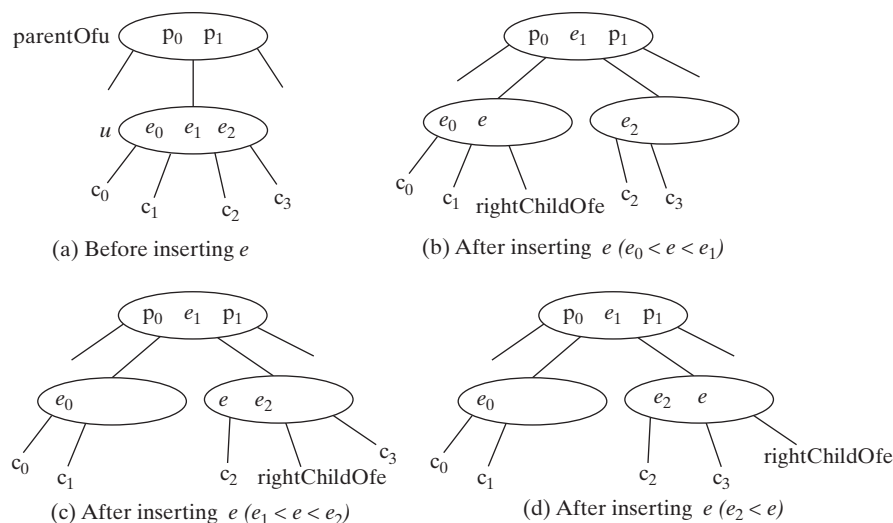


FIGURE 42.8 An interior node may be split to resolve overflow.

```

13 private void insert(E e, Tree24Node<E> rightChildOfe,
14     Tree24Node<E> u) {
15     if (u is a 2- or 3- node) { // u is a 2- or 3-node
16         insert23(e, rightChildOfe, u); // Insert e to node u
17     }
18     else { // Split a 4- node u
19         Tree24Node<E> v = new Tree24Node<E>(); // Create a new node
20         E median = split(e, rightChildOfe, u, v); // Split u
21
22         if (u == root) { // u is the root
23             root = new Tree24Node<E>(median); // New root
24             root.child.add(u); // u is the left child of median
25             root.child.add(v); // v is the right child of median
26         }
27         else {
28             Get the parent of u, parentOfu;
29             insert(median, v, parentOfu); // Inserting median to parent
30         }
31     }
32 }

```

The `insert(E e, Tree24Node<E> rightChildOfe, Tree24Node<E> u)` method inserts element e along with its right child to node u . When inserting e to a leaf node, the right child of e is `null` (line 6). If the node is a 2- or 3-node, simply insert the element to the node (lines 15–17). If the node is a 4-node, invoke the `split` method to split the node (line 20). The `split` method returns the median element. Recursively invoke the `insert` method to insert the median element to the parent node (line 29). Figure 42.9 shows the steps of inserting elements 34, 3, 50, 20, 15, 16, 25, 27, 29, and 24 into a 2-4 tree.



42.5 Deleting an Element from a 2-4 tree

Deleting an element involves locating the node that contains the element and removing the element from the node.

To delete an element from a 2-4 tree, first search the element in the tree to locate the node that contains it. If the element is not in the tree, the method returns false. Let u be the node that contains the element and $parentOfu$ be the parent of u . Consider three cases:

42.5 Deleting an Element from a 2–4 tree 42-7

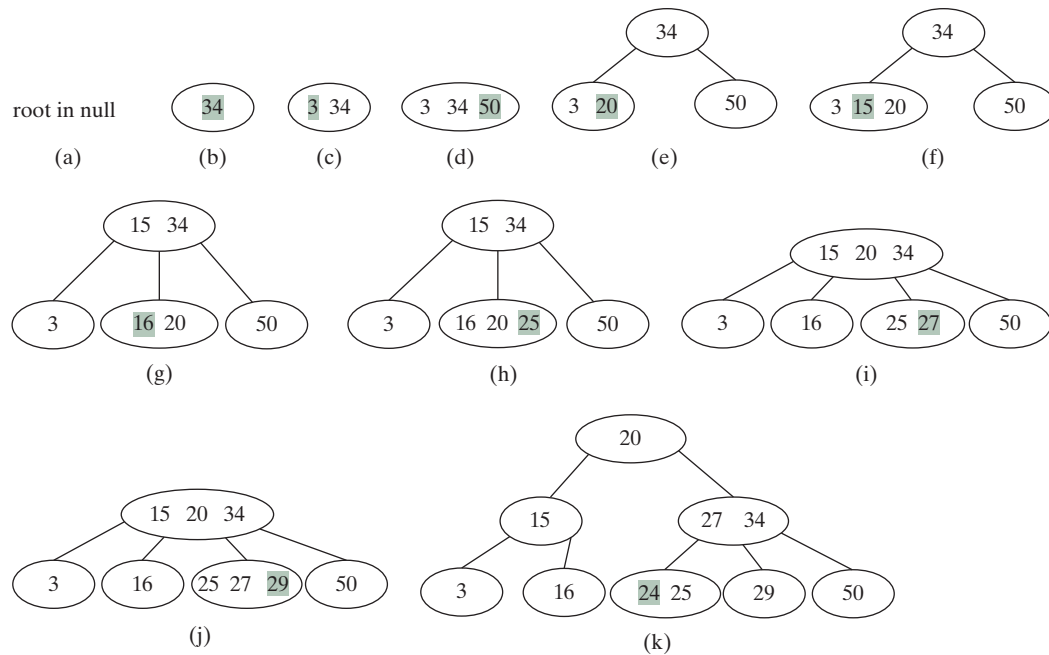


FIGURE 42.9 The tree changes after 34, 3, 50, 20, 15, 16, 25, 27, 29, and 24 are added into an empty tree.

Case 1: u is a leaf 3-node or 4-node. Delete e from u .

Case 2: u is a leaf 2-node. Delete e from u . Now u is empty. This situation is known as *underflow*. To remedy an underflow, consider two subcases:

Case 2.1: u 's immediate left or right sibling is a 3- or 4-node. Let the node be w , as shown in Figure 42.10(a) (assume that w is a left sibling of u). Perform a *transfer* operation that moves an element from $\text{parentOf } u$ to u , as shown in Figure 42.10(b), and move an element from w to replace the moved element in $\text{parentOf } u$, as shown in Figure 42.10(c).

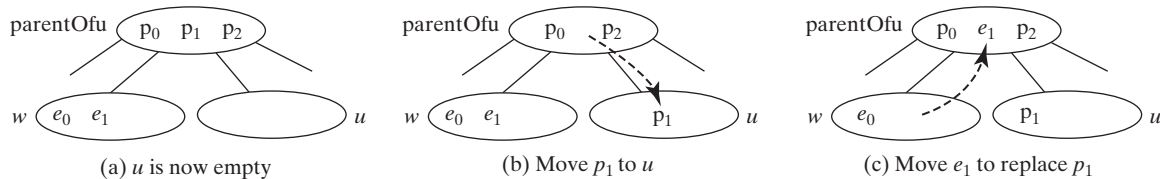


FIGURE 42.10 The transfer operation fills the empty node u .

Case 2.2: Both u 's immediate left and right siblings are 2-node if they exist (u may have only one sibling). Let the node be w , as shown in Figure 42.11(a) (assume that w is a left sibling of u). Perform a *fusion* operation that discards u and moves an element from $\text{parentOf } u$ to w , as shown in Figure 42.11(b). If $\text{parentOf } u$ becomes empty, repeat Case 2 recursively to perform a transfer or a fusion on $\text{parentOf } u$.

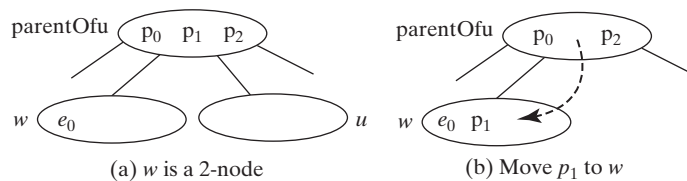


FIGURE 42.11 The fusion operation discards the empty node u .

Case 3: u is a nonleaf node. Find the rightmost leaf node in the left subtree of e . Let this node be w , as shown in Figure 42.12(a). Move the last element in w to replace e in u , as shown in Figure 42.12(b). If w becomes empty, apply a transfer or fusion operation on w .

Listing 42.3 describes the algorithm for deleting an element.

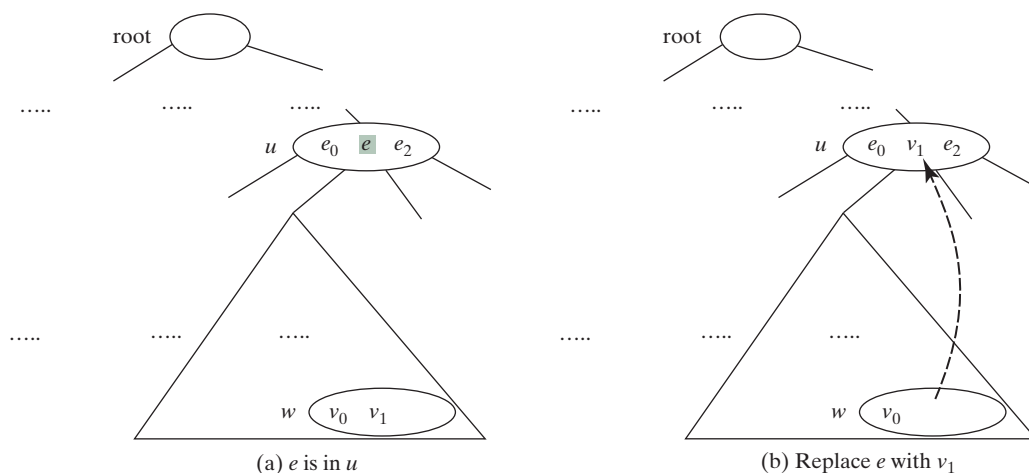


FIGURE 42.12 An element in the internal node is replaced by an element in a leaf node.

LISTING 42.3 Deleting an Element from a 2-4 tree

```

1  /** Delete the specified element from the tree */
2  public boolean delete(E e) {
3      Locate the node that contains the element e
4      if (the node is found) {
5          delete(e, node); // Delete element e from the node
6          size--; // After one element deleted
7          return true; // Element deleted successfully
8      }
9
10     return false; // Element not in the tree
11 }
12
13 /** Delete the specified element from the node */
14 private void delete(E e, Tree24Node<E> node) {
15     if (e is in a leaf node) {
16         // Get the path that leads to e from the root
17         ArrayList<Tree24Node<E>> path = path(e);
18
19         Remove e from the node;
20
21         // Check node for underflow along the path and fix it
22         validate(e, node, path); // Check underflow node
23     }
24     else { // e is in an internal node
25         Locate the rightmost node in the left subtree of node u;
26         Get the rightmost element from the rightmost node;
27
28         // Get the path that leads to e from the root
29         ArrayList<Tree24Node<E>> path = path(rightmostElement);
30
31         Replace the element in the node with the rightmost element
32     }

```



```

33     // Check node for underflow along the path and fix it
34     validate(rightmostElement, rightmostNode, path);
35 }
36 }
37
38 /** Perform a transfer or fusion operation if necessary */
39 private void validate(E e, Tree24Node<E> u,
40     ArrayList<Tree24Node<E>> path) {
41     for (int i = path.size() - 1; i >= 0; i--) {
42         if (u is not empty)
43             return; // Done, no need to perform transfer or fusion
44
45         Tree24Node<E> parentOfu = path.get(i - 1); // Get parent of u
46
47         // Check two siblings
48         if (left sibling of u has more than one element) {
49             Perform a transfer on u with its left sibling
50         }
51         else if (right sibling of u has more than one element) {
52             Perform a transfer on u with its right sibling
53         }
54         else if (u has left sibling) { // Fusion with a left sibling
55             Perform a fusion on u with its left sibling
56             u = parentOfu; // Back to the loop to check the parent node
57         }
58         else { // Fusion with right sibling (right sibling must exist)
59             Perform a fusion on u with its right sibling
60             u = parentOfu; // Back to the loop to check the parent node
61         }
62     }
63 }

```

The `delete(E e)` method locates the node that contains the element `e` and invokes the `delete(E e, Tree24Node<E> node)` method (line 5) to delete the element from the node.

If the node is a leaf node, get the path that leads to `e` from the root (line 17), delete `e` from the node (line 19), and invoke `validate` to check and fix the empty node (line 22). The `validate(E e, Tree24Node<E> u, ArrayList<Tree24Node<E>> path)` method performs a transfer or fusion operation if the node is empty. Since these operations may cause the parent of node `u` to become empty, a path is obtained in order to obtain the parents along the path from the root to node `u`, as shown in Figure 42.13.

If the node is a nonleaf node, locate the rightmost element in the left subtree of the node (lines 25–26), get the path that leads to the rightmost element from the root (line 29), replace

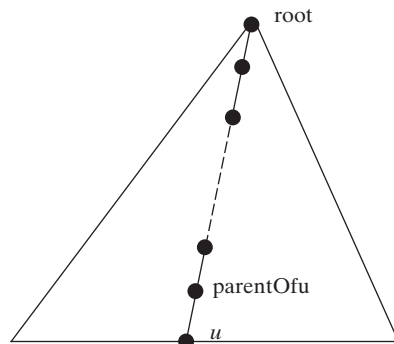


FIGURE 42.13 The nodes along the path may become empty as result of a transfer and fusion operation.

e in the node with the rightmost element (line 31), and invoke **validate** to fix the rightmost node if it is empty (line 34).

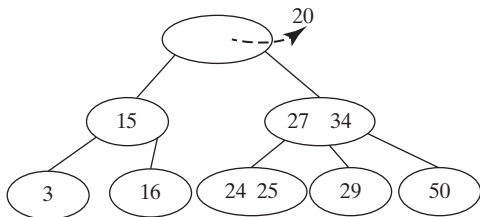
The **validate(E e, Tree24Node<E> u, ArrayList<Tree24Node<E>> path)** checks whether **u** is empty and performs a transfer or fusion operation to fix the empty node. The **validate** method exits when node is not empty (line 43). Otherwise, consider one of the following cases:

1. If **u** has a left sibling with more than one element, perform a transfer on **u** with its left sibling (line 49).
2. Otherwise, if **u** has a right sibling with more than one element, perform a transfer on **u** with its right sibling (line 52).
3. Otherwise, if **u** has a left sibling, perform a fusion on **u** with its left sibling (line 55) and reset **u** to **parentOfu** (line 56).
4. Otherwise, **u** must have a right sibling. Perform a fusion on **u** with its right sibling (line 59) and reset **u** to **parentOfu** (line 60).

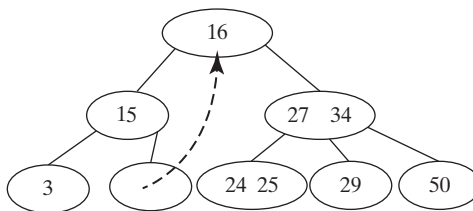
Only one of the preceding cases is executed. Afterward, a new iteration starts to perform a transfer or fusion operation on a new node **u** if needed. Figure 42.14 shows the steps of deleting elements **20**, **15**, **3**, **6**, and **34** that are deleted from a 2-4 tree in Figure 42.9(k).



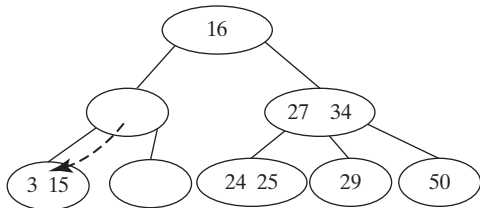
- 42.5.1** How do you search an element in a 2-4 tree?
- 42.5.2** How do you insert an element into a 2-4 tree?
- 42.5.3** How do you delete an element from a 2-4 tree?
- 42.5.4** Show the change of a 2-4 tree when inserting **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, **8**, and **6** into it, in this order.



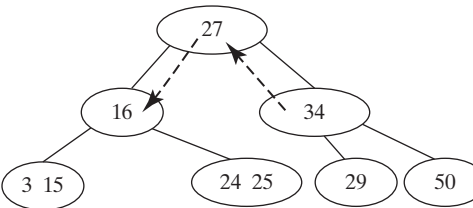
(a) Delete 20



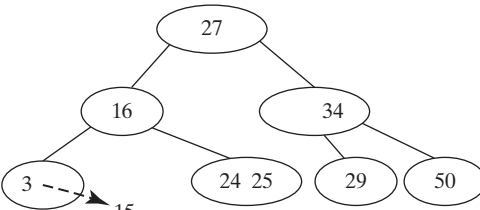
(b) Replace 20 with 16



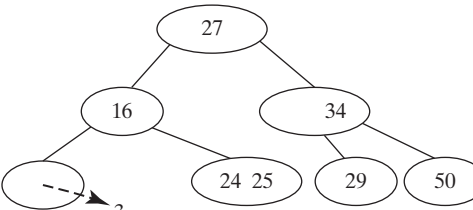
(c) Perform a fusion



(d) Perform a transfer



(e) Delete 15



(f) Delete 3

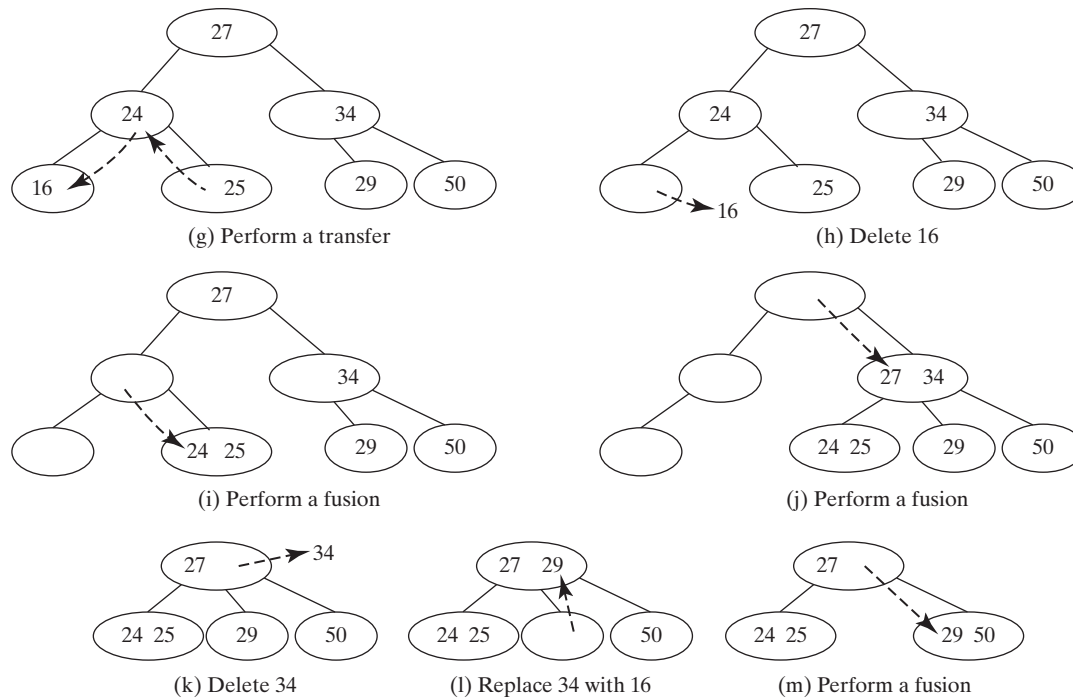


FIGURE 42.14 The tree changes after 20, 15, 3, 6, and 34 are deleted from a 2–4 tree.

- 42.5.5** For the tree built in the preceding question, show the change of the tree after deleting 1, 2, 3, 4, 10, 9, 7, 5, 8, and 6 from it, in this order.
- 42.5.6** Show the change of a B-tree of order 6 when inserting 1, 2, 3, 4, 10, 9, 7, 5, 8, 6, 17, 25, 18, 26, 14, 52, 63, 74, 80, 19, and 27 into it, in this order.
- 42.5.7** For the tree built in the preceding question, show the change of the tree after deleting 1, 2, 3, 4, 10, 9, 7, 5, and 8, and 6 from it, in this order.

42.6 Traversing Elements in a 2–4 tree

You can perform inorder, preorder, and postorder for traversing the elements in a 2–4 tree.

Inorder, preorder, and postorder traversals are useful for 2–4 trees. Inorder traversal visits the elements in increasing order. Preorder traversal visits the elements in the root, then recursively visits the subtrees from the left to right. Postorder traversal visits the subtrees from the left to right recursively, and then the elements in the root.

For example, in the 2–4 tree in Figure 42.9(k), the inorder traversal is
3 15 16 20 24 25 27 29 34 50

The preorder traversal is
20 15 3 16 27 34 24 25 29 50

The postorder traversal is
3 16 1 24 25 29 50 27 34 20



42.7 Implementing the Tree24 Class

This section gives the complete implementation for the Tree24 class.

Listing 42.4 gives the complete source code for the Tree24 class.



LISTING 42.4 Tree24.java

```

1  import java.util.ArrayList;
2
3  public class Tree24<E> extends Comparable<E>> implements Tree<E> {
4      private Tree24Node<E> root;
5      private int size;
6
7      /** Create a default 2-4 tree */
8      public Tree24() {
9      }
10
11     /** Create a 2-4 tree from an array of objects */
12     public Tree24(E[] elements) {
13         for (int i = 0; i < elements.length; i++)
14             insert(elements[i]);
15     }
16
17     @Override /** Search an element in the tree */
18     public boolean search(E e) {
19         Tree24Node<E> current = root; // Start from the root
20
21         while (current != null) {
22             if (matched(e, current)) { // Element is in the node
23                 return true; // Element found
24             }
25             else {
26                 current = getChildNode(e, current); // Search in a subtree
27             }
28         }
29
30         return false; // Element is not in the tree
31     }
32
33     /** Return true if the element is found in this node */
34     private boolean matched(E e, Tree24Node<E> node) {
35         for (int i = 0; i < node.elements.size(); i++)
36             if (node.elements.get(i).equals(e))
37                 return true; // Element found
38
39         return false; // No match in this node
40     }
41
42     /** Locate a child node to search element e */
43     private Tree24Node<E> getChildNode(E e, Tree24Node<E> node) {
44         if (node.child.size() == 0)
45             return null; // node is a leaf
46
47         int i = locate(e, node); // Locate the insertion point for e
48         return node.child.get(i); // Return the child node
49     }
50
51     @Override /** Insert element e into the tree
52     * Return true if the element is inserted successfully
53     */
54     public boolean insert(E e) {
55         if (root == null)
56             root = new Tree24Node<E>(e); // Create a new root for element
57         else {
58             // Locate the leaf node for inserting e
59             Tree24Node<E> leafNode = null;

```

```

60     Tree24Node<E> current = root;
61     while (current != null)
62         if (matched(e, current)) {
63             return false; // Duplicate element found, nothing inserted
64         }
65         else {
66             leafNode = current;
67             current = getChildNode(e, current);
68         }
69
70     // Insert the element e into the leaf node
71     insert(e, null, leafNode); // The right child of e is null
72 }
73
74 size++; // Increase size
75 return true; // Element inserted
76 }
77
78 /** Insert element e into node u */
79 private void insert(E e, Tree24Node<E> rightChildOfe,
80     Tree24Node<E> u) {
81     // Get the search path that leads to element e
82     ArrayList<Tree24Node<E>> path = path(e);
83
84     for (int i = path.size() - 1; i >= 0; i--) {
85         if (u.elements.size() < 3) { // u is a 2-node or 3-node
86             insert23(e, rightChildOfe, u); // Insert e to node u
87             break; // No further insertion to u's parent needed
88         }
89         else {
90             Tree24Node<E> v = new Tree24Node<E>(); // Create a new node
91             E median = split(e, rightChildOfe, u, v); // Split u
92
93             if (u == root) {
94                 root = new Tree24Node<E>(median); // New root
95                 root.child.add(u); // u is the left child of median
96                 root.child.add(v); // v is the right child of median
97                 break; // No further insertion to u's parent needed
98             }
99             else {
100                 // Use new values for the next iteration in the for loop
101                 e = median; // Element to be inserted to parent
102                 rightChildOfe = v; // Right child of the element
103                 u = path.get(i - 1); // New node to insert element
104             }
105         }
106     }
107 }
108
109 /** Insert element to a 2- or 3- and return the insertion point */
110 private void insert23(E e, Tree24Node<E> rightChildOfe,
111     Tree24Node<E> node) {
112     int i = this.locate(e, node); // Locate where to insert
113     node.elements.add(i, e); // Insert the element into the node
114     if (rightChildOfe != null)
115         node.child.add(i + 1, rightChildOfe); // Insert the child link
116 }
117
118 /** Split a 4-node u into u and v and insert e to u or v */
119 private E split(E e, Tree24Node<E> rightChildOfe,

```

```

120     Tree24Node<E> u, Tree24Node<E> v) {
121     // Move the last element in node u to node v
122     v.elements.add(u.elements.remove(2));
123     E median = u.elements.remove(1);
124
125     // Split children for a nonleaf node
126     // Move the last two children in node u to node v
127     if (u.child.size() == 0) {
128         v.child.add(u.child.remove(2));
129         v.child.add(u.child.remove(2));
130     }
131
132     // Insert e into a 2- or 3- node u or v.
133     if (e.compareTo(median) < 0)
134         insert23(e, rightChildOfe, u);
135     else
136         insert23(e, rightChildOfe, v);
137
138     return median; // Return the median element
139 }
140
141 /** Return a search path that leads to element e */
142 private ArrayList<Tree24Node<E>=> path(E e) {
143     ArrayList<Tree24Node<E>=> list = new ArrayList<Tree24Node<E>=>();
144     Tree24Node<E> current = root; // Start from the root
145
146     while (current != null) {
147         list.add(current); // Add the node to the list
148         if (matched(e, current)) {
149             break; // Element found
150         }
151         else {
152             current = getChildNode(e, current);
153         }
154     }
155
156     return list; // Return an array of nodes
157 }
158
159 @Override /** Delete the specified element from the tree */
160 public boolean delete(E e) {
161     // Locate the node that contains the element e
162     Tree24Node<E> node = root;
163     while (node != null)
164         if (matched(e, node)) {
165             delete(e, node); // Delete element e from node
166             size--; // After one element deleted
167             return true; // Element deleted successfully
168         }
169         else {
170             node = getChildNode(e, node);
171         }
172
173     return false; // Element not in the tree
174 }
175
176 /** Delete the specified element from the node */
177 private void delete(E e, Tree24Node<E> node) {
178     if (node.child.size() == 0) { // e is in a leaf node
179         // Get the path that leads to e from the root
180         ArrayList<Tree24Node<E>=> path = path(e);

```

```

181
182     node.elements.remove(e); // Remove element e
183
184     if (node == root) { // Special case
185         if (node.elements.size() == 0)
186             root = null; // Empty tree
187         return; // Done
188     }
189
190     validate(e, node, path); // Check underflow node
191 }
192 else { // e is in an internal node
193     // Locate the rightmost node in the left subtree of the node
194     int index = locate(e, node); // Index of e in node
195     Tree24Node<E> current = node.child.get(index);
196     while (current.child.size() > 0) {
197         current = current.child.get(current.child.size() - 1);
198     }
199     E rightmostElement =
200         current.elements.get(current.elements.size() - 1);
201
202     // Get the path that leads to e from the root
203     ArrayList<Tree24Node<E>> path = path(rightmostElement);
204
205     // Replace the deleted element with the rightmost element
206     node.elements.set(index, current.elements.remove(
207         current.elements.size() - 1));
208
209     validate(rightmostElement, current, path); // Check underflow
210 }
211 }
212
213 /** Perform transfer and confusion operations if necessary */
214 private void validate(E e, Tree24Node<E> u,
215     ArrayList<Tree24Node<E>> path) {
216     for (int i = path.size() - 1; u.elements.size() == 0; i--) {
217         Tree24Node<E> parentOfu = path.get(i - 1); // Get parent of u
218         int k = locate(e, parentOfu); // Index of e in the parent node
219
220         // Check two siblings
221         if (k > 0 && parentOfu.child.get(k - 1).elements.size() > 1) {
222             leftSiblingTransfer(k, u, parentOfu);
223         }
224         else if (k + 1 < parentOfu.child.size() &&
225             parentOfu.child.get(k + 1).elements.size() > 1) {
226             rightSiblingTransfer(k, u, parentOfu);
227         }
228         else if (k - 1 == 0) { // Fusion with a left sibling
229             // Get left sibling of node u
230             Tree24Node<E> leftNode = parentOfu.child.get(k - 1);
231
232             // Perform a fusion with left sibling on node u
233             leftSiblingFusion(k, leftNode, u, parentOfu);
234
235             // Done when root becomes empty
236             if (parentOfu == root && parentOfu.elements.size() == 0) {
237                 root = leftNode;
238                 break;
239             }
240
241             u = parentOfu; // Back to the loop to check the parent node

```

```

242     }
243     else { // Fusion with right sibling (right sibling must exist)
244         // Get left sibling of node u
245         Tree24Node<E> rightNode = parentOfu.child.get(k + 1);
246
247         // Perform a fusion with right sibling on node u
248         rightSiblingFusion(k, rightNode, u, parentOfu);
249
250         // Done when root becomes empty
251         if (parentOfu == root && parentOfu.elements.size() == 0) {
252             root = rightNode;
253             break;
254         }
255
256         u = parentOfu; // Back to the loop to check the parent node
257     }
258 }
259 }
260
261 /** Locate the insertion point of the element in the node */
262 private int locate(E o, Tree24Node<E> node) {
263     for (int i = 0; i < node.elements.size(); i++) {
264         if (o.compareTo(node.elements.get(i)) <= 0) {
265             return i;
266         }
267     }
268
269     return node.elements.size();
270 }
271
272 /** Perform a transfer with a left sibling */
273 private void leftSiblingTransfer(int k,
274     Tree24Node<E> u, Tree24Node<E> parentOfu) {
275     // Move an element from the parent to u
276     u.elements.add(0, parentOfu.elements.get(k - 1));
277
278     // Move an element from the left node to the parent
279     Tree24Node<E> leftNode = parentOfu.child.get(k - 1);
280     parentOfu.elements.set(k - 1,
281         leftNode.elements.remove(leftNode.elements.size() - 1));
282
283     // Move the child link from left sibling to the node
284     if (leftNode.child.size() > 0)
285         u.child.add(0, leftNode.child.remove(
286             leftNode.child.size() - 1));
287 }
288
289 /** Perform a transfer with a right sibling */
290 private void rightSiblingTransfer(int k,
291     Tree24Node<E> u, Tree24Node<E> parentOfu) {
292     // Transfer an element from the parent to u
293     u.elements.add(parentOfu.elements.get(k));
294
295     // Transfer an element from the right node to the parent
296     Tree24Node<E> rightNode = parentOfu.child.get(k + 1);
297     parentOfu.elements.set(k, rightNode.elements.remove(0));
298
299     // Move the child link from right sibling to the node
300     if (rightNode.child.size() > 0)
301         u.child.add(rightNode.child.remove(0));
302 }

```



```

303
304  /** Perform a fusion with a left sibling */
305  private void leftSiblingFusion(int k, Tree24Node<E> leftNode,
306      Tree24Node<E> u, Tree24Node<E> parentOfu) {
307      // Transfer an element from the parent to the left sibling
308      leftNode.elements.add(parentOfu.elements.remove(k - 1));
309
310      // Remove the link to the empty node
311      parentOfu.child.remove(k);
312
313      // Adjust child links for nonleaf node
314      if (u.child.size() > 0)
315          leftNode.child.add(u.child.remove(0));
316  }
317
318  /** Perform a fusion with a right sibling */
319  private void rightSiblingFusion(int k, Tree24Node<E> rightNode,
320      Tree24Node<E> u, Tree24Node<E> parentOfu) {
321      // Transfer an element from the parent to the right sibling
322      rightNode.elements.add(0, parentOfu.elements.remove(k));
323
324      // Remove the link to the empty node
325      parentOfu.child.remove(k);
326
327      // Adjust child links for nonleaf node
328      if (u.child.size() > 0)
329          rightNode.child.add(0, u.child.remove(0));
330  }
331
332  /** Get the number of nodes in the tree */
333  public int getSize() {
334      return size;
335  }
336
337  /** Preorder traversal from the root */
338  public void preorder() {
339      preorder(root);
340  }
341
342  /** Preorder traversal from a subtree */
343  private void preorder(Tree24Node<E> root) {
344      if (root == null) return;
345      for (int i = 0; i < root.elements.size(); i++)
346          System.out.print(root.elements.get(i) + " ");
347
348      for (int i = 0; i < root.child.size(); i++)
349          preorder(root.child.get(i));
350  }
351
352  /** Inorder traversal from the root */
353  public void inorder() {
354      // Left as exercise
355  }
356
357  /** Postorder traversal from the root */
358  public void postorder() {
359      // Left as exercise
360  }
361
362  /** Return true if the tree is empty */
363  public boolean isEmpty() {

```

```

364     return root == null;
365 }
366
367 @Override /** Remove all elements from the tree */
368 public void clear() {
369     root = null;
370     size = 0;
371 }
372
373 /** Return an iterator to traverse elements in the tree */
374 public java.util.Iterator iterator() {
375     // Left as exercise
376     return null;
377 }
378
379 /** Define a 2-4 tree node */
380 protected static class Tree24Node<E extends Comparable<E>> {
381     // elements has maximum three values
382     ArrayList<E> elements = new ArrayList<E>(3);
383     // Each has maximum four children
384     ArrayList<Tree24Node<E>> child
385         = new ArrayList<Tree24Node<E>>(4);
386
387     /** Create an empty Tree24 node */
388     Tree24Node() {
389     }
390
391     /** Create a Tree24 node with an initial element */
392     Tree24Node(E o) {
393         elements.add(o);
394     }
395 }
396 }

```

The **Tree24** class contains the data fields **root** and **size** (lines 4–5). **root** references the root node and **size** stores the number of elements in the tree.

The **Tree24** class has two constructors: a no-arg constructor (lines 8–9) that constructs an empty tree and a constructor that creates an initial **Tree24** from an array of elements (lines 12–15).

The **search** method (lines 18–31) searches an element in the tree. It returns **true** (line 23) if the element is in the tree and returns **false** if the search arrives at an empty subtree (line 30).

The **matched(e, node)** method (lines 34–40) checks where the element **e** is in the node.

The **getChildNode(e, node)** method (lines 43–49) returns the root of a subtree where **e** should be searched.

The **insert(E e)** method inserts an element in a tree (lines 54–76). If the tree is empty, a new root is created (line 56). The method locates a leaf node in which the element will be inserted and invokes **insert(e, null, leafNode)** to insert the element (line 71).

The **insert(e, rightChildOfe, u)** method inserts an element into node **u** (lines 79–107). The method first invokes **path(e)** (line 82) to obtain a search path from the root to node **u**. Each iteration of the **for** loop considers **u** and its parent **parentOfu** (lines 84–106). If **u** is a 2-node or 3-node, invoke **insert23(e, rightChildOfe, u)** to insert **e** and its child link **rightChildOfe** into **u** (line 86). No split is needed (line 87). Otherwise, create a new node **v** (line 90) and invoke **split(e, rightChildOfe, u, v)** (line 91) to split **u** into **u** and **v**. The **split** method inserts **e** into either **u** and **v** and returns the median in the original **u**. If **u** is the root, create a new root to hold median, and set **u** and **v** as the left and right children for median (lines 95–96). If **u** is not the root, insert median to **parentOfu** in the next iteration (lines 101–103).

The `insert23(e, rightChildOfe, node)` method inserts `e` along with the reference to its right child into the node (lines 110–116). The method first invokes `locate(e, node)` (line 112) to locate an insertion point, then insert `e` into the node (line 113). If `rightChildOfe` is not `null`, it is inserted into the child list of the node (line 115).

The `split(e, rightChildOfe, u, v)` method splits a 4-node `u` (lines 119–139). This is accomplished as follows: (1) move the last element from `u` to `v` and remove the median element from `u` (lines 122–123); (2) move the last two child links from `u` to `v` (lines 127–130) if `u` is a nonleaf node; (3) if `e < median`, insert `e` into `u`; otherwise, insert `e` into `v` (lines 133–136); and (4) return median (line 138).

The `path(e)` method returns an `ArrayList` of nodes searched from the root in order to locate `e` (lines 142–157). If `e` is in the tree, the last node in the path contains `e`. Otherwise, the last node is where `e` should be inserted.

The `delete(E e)` method deletes an element from the tree (lines 160–174). The method first locates the node that contains `e` and invokes `delete(e, node)` to delete `e` from the node (line 165). If the element is not in the tree, return `false` (line 173).

The `delete(e, node)` method deletes an element from node `u` (lines 177–211). If the node is a leaf node, obtain the path that leads to `e` (line 180), delete `e` (line 182), set root to `null` if the tree becomes empty (lines 184–188), and invoke `validate` to apply transfer and fusion operation on empty nodes (line 190). If the node is a nonleaf node, locate the rightmost element (lines 194–200), obtain the path that leads to `e` (line 203), replace `e` with the rightmost element (lines 206–207), and invoke `validate` to apply transfer and fusion operations on empty nodes (line 209).

The `validate(e, u, path)` method ensures that the tree is a valid 2–4 tree (lines 214–259). The `for` loop terminates when `u` is not empty (line 216). The loop body is executed to fix the empty node `u` by performing a transfer or fusion operation. If a left sibling with more than one element exists, perform a transfer on `u` with the left sibling (line 222). Otherwise, if a right sibling with more than one element exists, perform a transfer on `u` with the left sibling (line 226). Otherwise, if a left sibling exists, perform a fusion on `u` with the left sibling (lines 230–239), and validate `parentOfu` in the next loop iteration (line 241). Otherwise, perform a fusion on `u` with the right sibling.

The `locate(e, node)` method locates the index of `e` in the node (lines 262–270).

The `leftSiblingTransfer(k, u, parentOfu)` method performs a transfer on `u` with its left sibling (lines 273–287). The `rightSiblingTransfer(k, u, parentOfu)` method performs a transfer on `u` with its right sibling (lines 290–302). The `leftSiblingFusion(k, leftNode, u, parentOfu)` method performs a fusion on `u` with its left sibling `leftNode` (lines 305–316). The `rightSiblingFusion(k, rightNode, u, parentOfu)` method performs a fusion on `u` with its right sibling `rightNode` (lines 319–330).

The `preorder()` method displays all the elements in the tree in preorder (lines 338–350).

The inner class `Tree24Node` defines a class for a node in the tree (lines 374–389).

42.8 Testing the **Tree24** Class

*This section writes a test program for using the **Tree24** class.*

Listing 42.5 gives a test program. The program creates a 2–4 tree and inserts elements in lines 6–20, and deletes elements in lines 22–56.



LISTING 42.5 TestTree24.java

```
1 public class TestTree24 {
2     public static void main(String[] args) {
3         // Create a 2-4 tree
4         Tree24<Integer> tree = new Tree24<Integer>();
5     }
```

```

6      tree.insert(34);
7      tree.insert(3);
8      tree.insert(50);
9      tree.insert(20);
10     tree.insert(15);
11     tree.insert(16);
12     tree.insert(25);
13     tree.insert(27);
14     tree.insert(29);
15     tree.insert(24);
16     System.out.print("\nAfter inserting 24:");
17     printTree(tree);
18     tree.insert(23);
19     tree.insert(22);
20     tree.insert(60);
21     tree.insert(70);
22     System.out.print("\nAfter inserting 70:");
23     printTree(tree);
24
25     tree.delete(34);
26     System.out.print("\nAfter deleting 34:");
27     printTree(tree);
28
29     tree.delete(25);
30     System.out.print("\nAfter deleting 25:");
31     printTree(tree);
32
33     tree.delete(50);
34     System.out.print("\nAfter deleting 50:");
35     printTree(tree);
36
37     tree.delete(16);
38     System.out.print("\nAfter deleting 16:");
39     printTree(tree);
40
41     tree.delete(3);
42     System.out.print("\nAfter deleting 3:");
43     printTree(tree);
44
45     tree.delete(15);
46     System.out.print("\nAfter deleting 15:");
47     printTree(tree);
48 }
49
50 public static <E extends Comparable<E>>
51     void printTree(Tree<E> tree) {
52         // Traverse tree
53         System.out.print("\nPreorder: ");
54         tree.preorder();
55         System.out.print("\nThe number of nodes is " + tree.getSize());
56         System.out.println();
57     }
58 }

```



After inserting 24:
 Preorder: 20 15 3 16 27 34 24 25 29 50
 The number of nodes is 10

```

After inserting 70:
Preorder: 20 15 3 16 24 27 34 22 23 25 29 50 60 70
The number of nodes is 14

After deleting 34:
Preorder: 20 15 3 16 24 27 50 22 23 25 29 60 70
The number of nodes is 13

After deleting 25:
Preorder: 20 15 3 16 23 27 50 22 24 29 60 70
The number of nodes is 12

After deleting 50:
Preorder: 20 15 3 16 23 27 60 22 24 29 70
The number of nodes is 11

After deleting 16:
Preorder: 23 20 3 15 22 27 60 24 29 70
The number of nodes is 10

After deleting 3:
Preorder: 23 20 15 22 27 60 24 29 70
The number of nodes is 9

After deleting 15:
Preorder: 27 23 20 22 24 60 29 70
The number of nodes is 8

```

Figure 42.15 shows how the tree evolves as elements are added. After **34**, **3**, **50**, **20**, **15**, **16**, **25**, **27**, **29**, and **24** are added to the tree, it is as shown in Figure 42.15(a). After inserting **23**, **22**, **60**, and **70**, the tree is as shown in Figure 42.15(b). After deleting **34**, the tree is as shown in Figure 42.15(c). After deleting **25**, the tree is as shown in Figure 42.15(d). After deleting **50**, the tree is as shown in Figure 42.15(e). After deleting **16**, the tree is as shown in Figure 42.15(f). After deleting **3**, the tree is as shown in Figure 42.15(g). After deleting **15**, the tree is as shown in Figure 42.15(h).

42.9 Time-Complexity Analysis

Search, insertion, and deletion operations take $O(\log n)$ time in a 2–4 tree.

Since a 2–4 tree is a completely balanced binary tree, its height is at most $O(\log n)$. The **search**, **insert**, and **delete** methods operate on the nodes along a path in the tree. It takes a constant time to search an element within a node. So, the **search** method takes $O(\log n)$ time. For the **insert** method, the time for splitting a node takes a constant time. So, the **insert** method takes $O(\log n)$ time. For the **delete** method, it takes a constant time to perform a transfer and fusion operation. So, the **delete** method takes $O(\log n)$ time.



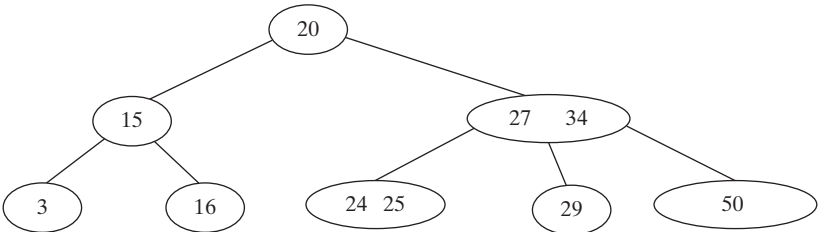
42.10 B-Tree

A B-tree is a generalization of a 2–4 tree.

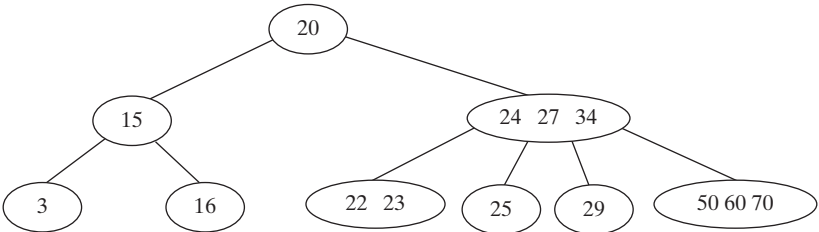
So far we assume that the entire data set is stored in main memory. What if the data set is too large and cannot fit in the main memory, as in the case with most databases, where data is stored on disks? Suppose you use an AVL tree to organize a million records in a database table. To find a record, the average number of nodes traversed is $\log_2 1,000,000 \approx 20$.



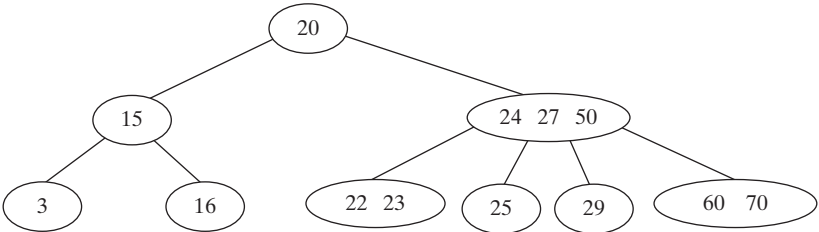
This is fine if all nodes are stored in main memory. However, for nodes stored on a disk, this means 20 disk reads. Disk I/O is expensive, and it is thousands of times slower than memory access. To improve performance, we need to reduce the number of disk I/Os. An efficient data structure for performing search, insertion, and deletion for data stored on secondary storage such as hard disks is the B-tree, which is a generalization of the 2-4 tree.



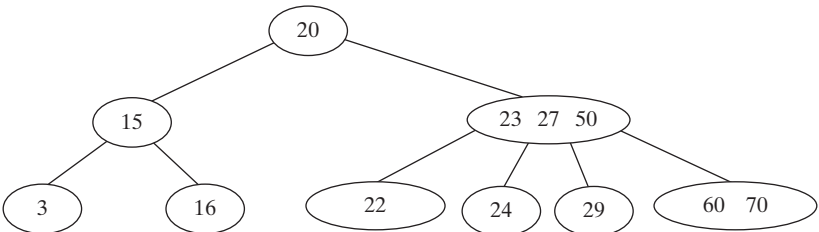
(a) After inserting 34, 3, 50, 20, 15, 16, 25, 27, 29, and 24, in this order



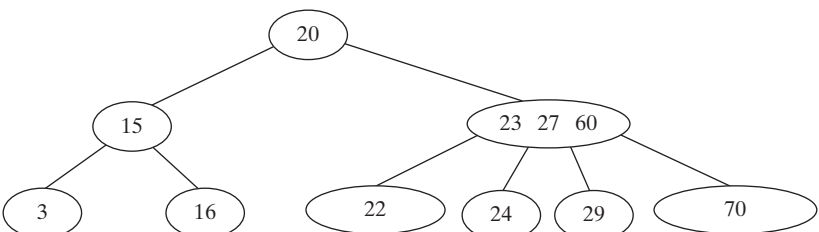
(b) After inserting 23, 22, 60, and 70



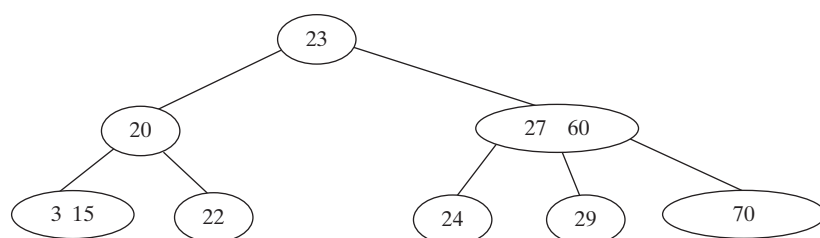
(c) After deleting 34



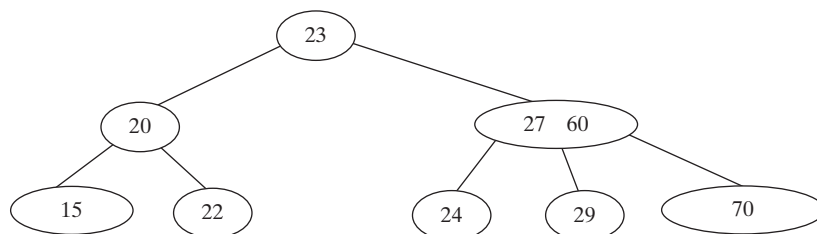
(d) After deleting 25



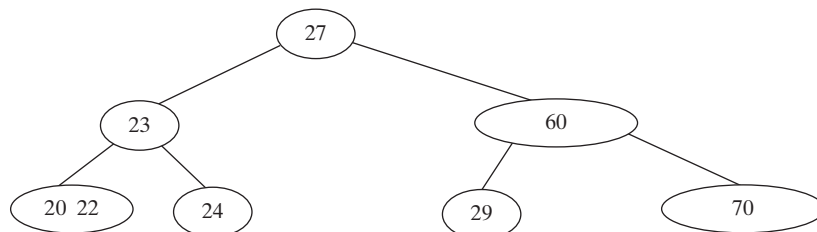
(e) After deleting 50



(f) After deleting 16



(g) After deleting 3



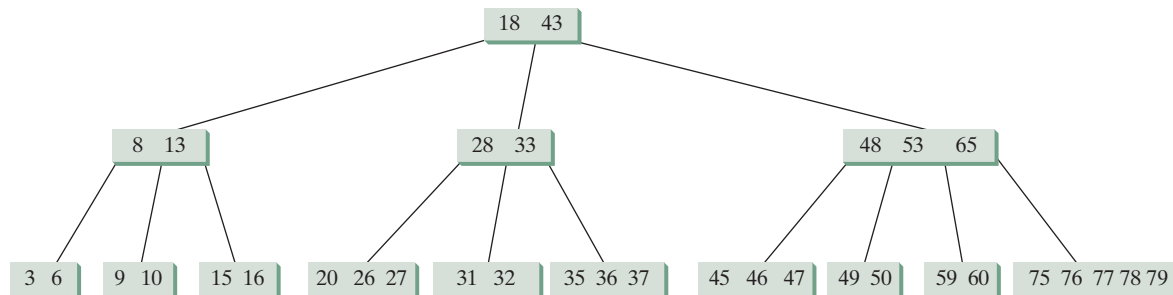
(h) After deleting 15

FIGURE 42.15 The tree evolves as elements are inserted and deleted.

A B-tree of order d is defined as follows:

1. Each node except the root contains between $\lceil d/2 \rceil - 1$ and $d - 1$ keys.
2. The root may contain up to $d - 1$ keys.
3. A nonleaf node with k keys has $k + 1$ children.
4. All leaf nodes have the same depth.

Figure 42.16 shows a B-tree of order 6. For simplicity, we use integers to represent keys. Each key is associated with a pointer that points to the actual record in the database. For simplicity, the pointers to the records in the database are omitted in the figure.

**FIGURE 42.16** In a B-tree of order 6, each node except the root may contain between 2 and 5 keys.

Note that a B-tree is a search tree. The keys in each node are placed in increasing order. Each key in an interior node has a left subtree and a right subtree, as shown in Figure 42.17. All keys in the left subtree are less than the key in the parent node, and all keys in the right subtree are greater than the key in the parent node.

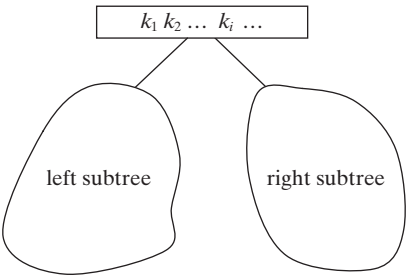


FIGURE 42.17 The keys in the left (right) subtree of key k_i are less than (greater than) k_i .

The basic unit of the IO operations on a disk is a block. When you read data from a disk, the whole block that contains the data is read. You should choose an appropriate order d so that a node can fit in a single disk block. This will minimize the number of disk IOs.

A 2-4 tree is actually a B-tree of order 4. The techniques for insertion and deletion in a 2-4 tree can be easily generalized for a B-tree.

Inserting a key to a B-tree is similar to what was done for a 2-4 tree. First locate the leaf node in which the key will be inserted. Insert the key to the node. After the insertion, if the leaf node has d keys, an overflow occurs. To resolve overflow, perform a *split* operation similar to the one used in a 2-4 tree, as follows:

Let u denote the node needed to be split and let m denote the median key in the node. Create a new node and move all keys greater than m to this new node. Insert m to the parent node of u . Now u becomes the left child of m and v becomes the right child of m , as shown in Figure 42.18. If inserting m into the parent node of u causes an overflow, repeat the same split process on the parent node.

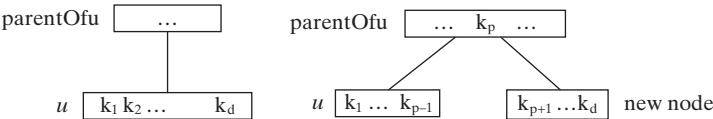


FIGURE 42.18 (a) After inserting a new key to node u . (b) The median key k_p is inserted to **parentOfu**.

A key k can be deleted from a B-tree in the same way as in a 2-4 tree. First locate the node u that contains the key. Consider two cases:

Case 1: If u is a leaf node, remove the key from u . After the removal, if u has less than $\lceil d/2 \rceil - 1$ keys, an underflow occurs. To remedy an underflow, perform a transfer with a sibling

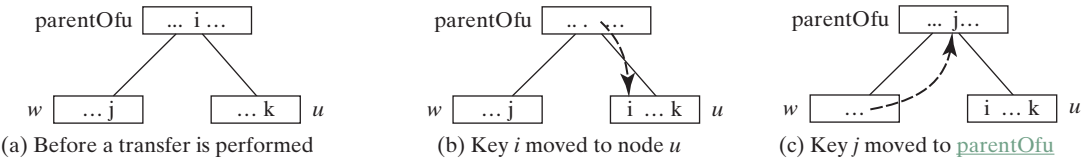


FIGURE 42.19 The transfer operation transfers a key from the **parentOfu** to u and transfers a key from u 's sibling **parentOfu**.

w of u that has more than $\lceil d/2 \rceil - 1$ keys if such sibling exists, as shown in Figure 42.19. Otherwise, perform a fusion with a sibling w of u , as shown in Figure 42.20.

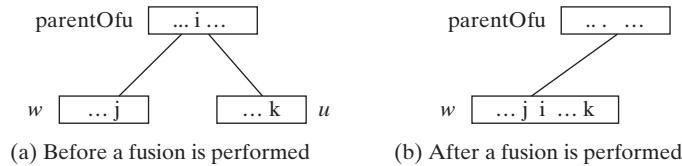


FIGURE 42.20 The fusion operation moves key i from the **parentOfu** u to w and moves all keys in u to w .

Case 2: u is a nonleaf node. Find the rightmost leaf node in the left subtree of k . Let this node be w , as shown in Figure 42.21(a). Move the last key in w to replace k in u , as shown in Figure 42.21(b). If w becomes underflow, apply a transfer or fusion operation on w .

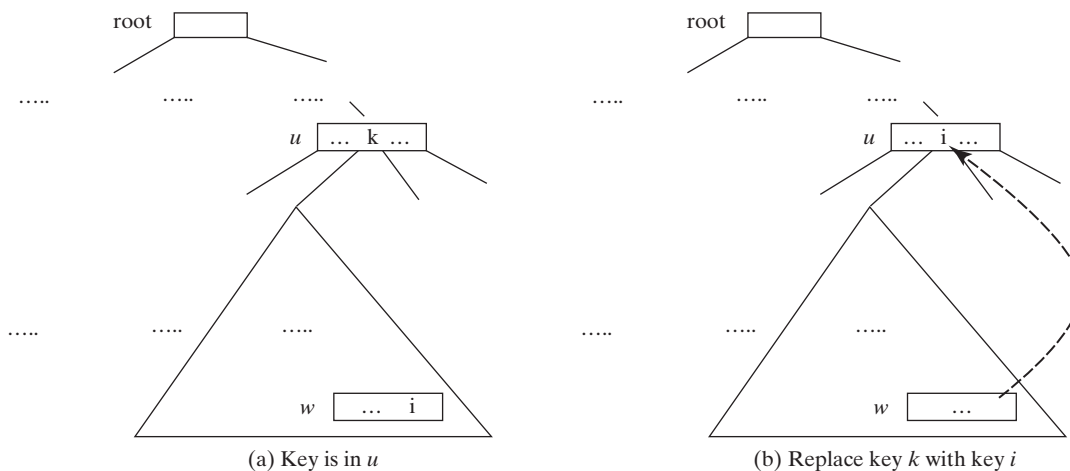


FIGURE 42.21 A key in the internal node is replaced by an element in a leaf node.

The performance of a B-tree depends on the number of disk IOs (i.e., the number of nodes accessed). The number of nodes accessed for search, insertion, and deletion operations depends on the height of the tree. In the worst case, each node contains $\lceil d/2 \rceil - 1$ keys. So, the height of the tree is $\log_{\lceil d/2 \rceil} n$, where n is the number of keys. In the best case, each node contains $d - 1$ keys. So, the height of the tree is $\log_d n$. Consider a B-tree of order 12 for 10 million keys. The height of the tree is between $\log_6 10,000,000 \approx 7$ and $\log_{12} 10,000,000 \approx 9$. So, for search, insertion, and deletion operations, the maximum number of nodes visited is 42. If you use an AVL tree, the maximum number of nodes visited is $\log_2 10,000,000 \approx 24$.

KEY TERMS

2–3–4 tree 42-2

2–4 tree 42-2

2-node 42-2

3-node 42-2

4-node 42-2

B-tree 42-11

fusion operation 42-7

split operation 42-4

transfer operation 42-7

CHAPTER SUMMARY

- 1. A 2–4 tree is a completely balanced search tree. In a 2–4 tree, a node may have one, two, or three elements.
- 2. Searching an element in a 2–4 tree is similar to searching an element in a binary tree. The difference is that you have searched an element within a node.
- 3. To insert an element to a 2–4 tree, locate a leaf node in which the element will be inserted. If the leaf node is a 2- or 3-node, simply insert the element into the node. If the node is a 4-node, split the node.
- 4. The process of deleting an element from a 2–4 tree is similar to that of deleting an element from a binary tree. The difference is that you have to perform transfer or fusion operations for empty nodes.
- 5. The height of a 2–4 tree is $O(\log n)$. So, the time complexities for the search, insert, and delete methods are $O(\log n)$.
- 6. A B-tree is a generalization of the 2–4 tree. Each node in a B-tree of order d can have between $\lceil d/2 \rceil - 1$ and $d - 1$ keys except the root. 2–4 trees are flatter than AVL trees and B-trees are flatter than 2–4 trees. B-trees are efficient for creating indexes for data in database systems where large amounts of data are stored on disks.



QUIZ

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

- *42.1 (Implement *inorder*) The *inorder* method in *Tree24* is left as an exercise. Implement it.
- 42.2 (Implement *postorder*) The *postorder* method in *Tree24* is left as an exercise. Implement it.
- 42.3 (Implement *iterator*) The *iterator* method in *Tree24* is left as an exercise. Implement it to iterate the elements using *inorder*.
- *42.4 (Display a 2–4 tree graphically) Write a GUI program that displays a 2–4 tree.
- ***42.5 (2–4 tree animation) Write a GUI program that animates the 2–4 tree *insert*, *delete*, and *search* methods, as shown in Figure 42.4.
- **42.6 (Parent reference for *Tree24*) Redefine *Tree24Node* to add a reference to a node’s parent, as shown below:

| Tree24Node<E> | |
|---------------------------------|---------------------------------------------------------|
| elements: ArrayList<E> | An array list for storing the elements. |
| child: ArrayList<Tree24Node<E>> | An array list for storing the links to the child nodes. |
| parent: Tree24Node<E> | Refers to the parent of this node. |
| +Tree24() | Creates an empty tree node. |
| +Tree24(o: E) | Creates a tree node with an initial element. |

Add the following two new methods in **Tree24**:

```
public Tree24Node<E> getParent(Tree24Node<E> node)
    Returns the parent for the specified node.
public ArrayList<Tree24Node<E>> getPath(Tree24Node<E> node)
    Returns the path from the specified node to the root in an
    array list.
```

Write a test program that adds numbers **1, 2, ..., 100** to the tree and displays the paths for all leaf nodes.

*****42.7** (The *BTree* class) Design and implement a class for B-trees.

