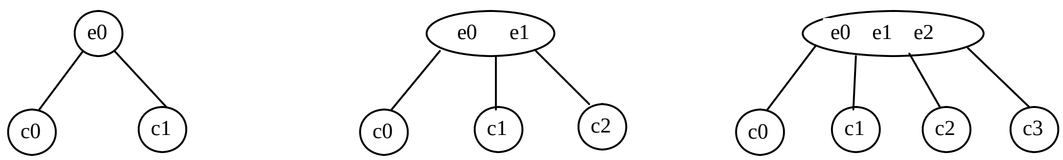
CHAPTER
9
2-4 Trees and B-Trees

Objectives

- To know what a 2-4 tree is (§9.1).
- To design the Tree24 class that implements the Tree interface (§9.2).
- To search an element in a 2-4 tree (§9.3).
- To insert an element in a 2-4 tree and know how to split a node (§9.4).
- To delete an element from a 2-4 tree and know how to perform transfer and fusion operations (§9.5).
- To traverse elements in a 2-4 tree (§9.6).
- To know how to implement the Tree24 class (§9.7).
- To use B-trees for indexing large amount of data (§9.10).

<Side Remark: completely-balanced tree>
<Side Remark: 2-node>
<Side Remark: 3-node>
<Side Remark: 4-node>

A *2-4 tree*, also known as a *2-3-4 tree*, is a *complete 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 9.1.



(a) 2-node **(b) 3-node** **(c) 4-node**

Figure 9.1
An interior node of a 2-4 tree has two, three, or four children.

<Side Remark: ordered>

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

$$E(c_0) < e_0 < E(c_1) < e_1 < E(c_2) < e_2 < E(c_3) < e_3 < E(c_4)$$

<Side Remark: E(c_k)>
<Side Remark: left subtree>
<Side Remark: right subtree>

Where E(c_k) denote the elements in c_k. Figure 9.1 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.

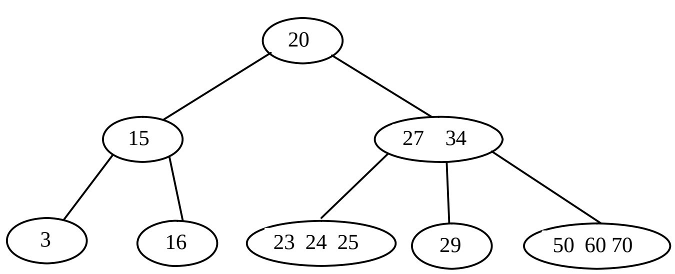


Figure 9.2
A 2-4 tree is a full complete search tree.

<Side Remark: binary vs. 2-4>

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.

9.2 Designing Classes for 2-4 Trees

The Tree24 class can be designed by implementing the Tree interface, as shown in Figure 9.3. 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 9.5.

***New Figure
<PD: UML Class Diagram>

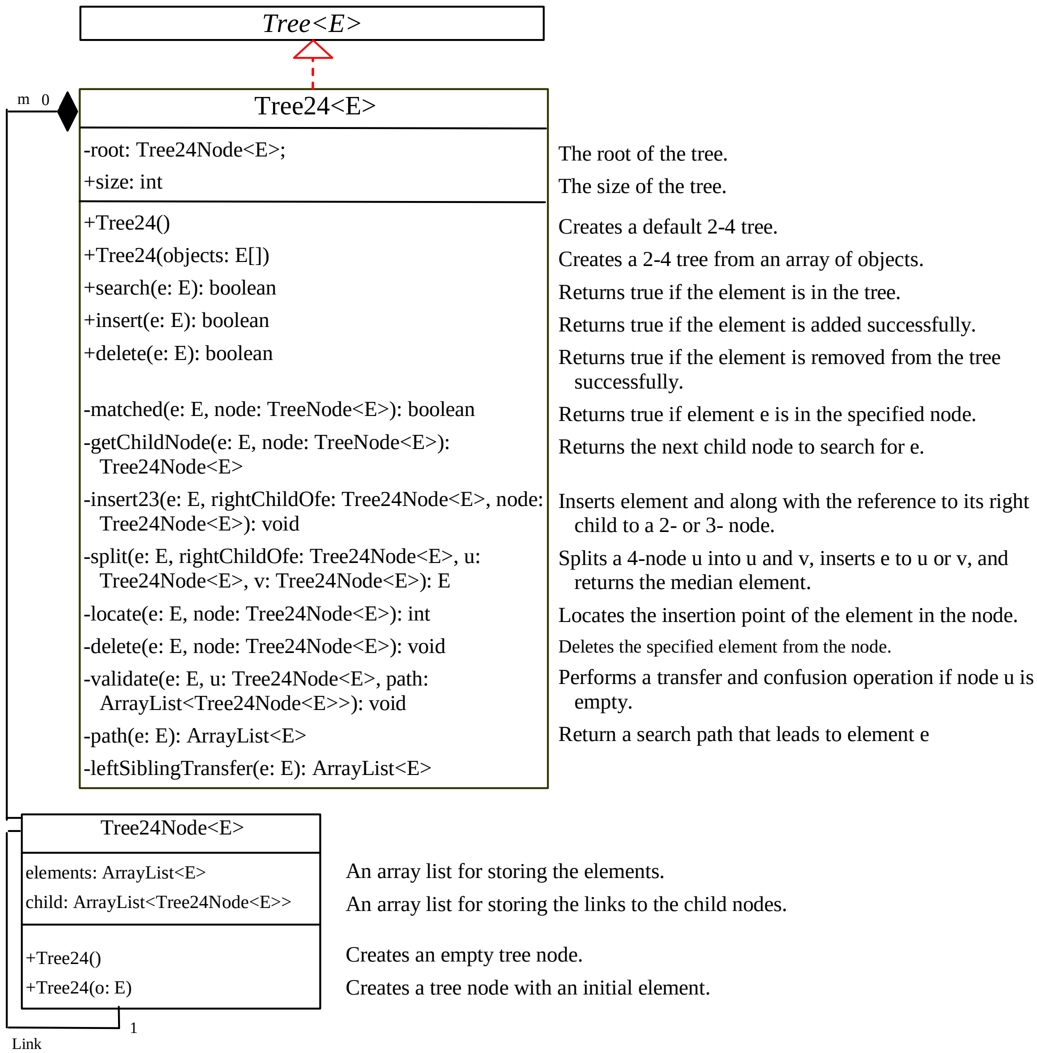


Figure 9.3
The Tree24 class implements Tree.

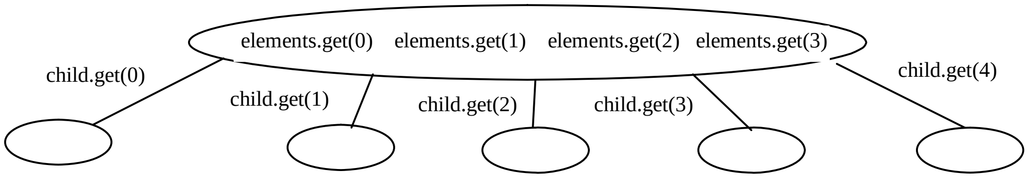


Figure 9.5
A 2-4 tree node stores the elements and the links to the child nodes in array lists.

Pedagogical NOTE
<side remark: 2-4 tree animation>
Run from www.cs.armstrong.edu/liang/jds/exercisejds/Exercise10_5.html to see how a 2-4 tree works, as shown in Figure 9.4.

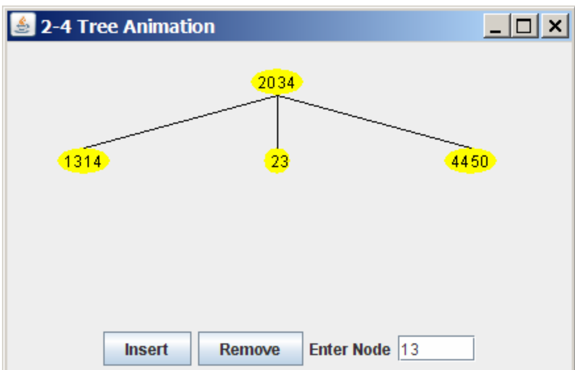


Figure 9.4
The animation tool enables you to insert, delete, and search elements in a 2-4 tree visually.
******End NOTE***

9.3 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 also 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 9.1.

Listing 9.1 Searching an Element in a 2-4 Tree
******PD: Please add line numbers in the following code******
<Side Remark line 2: start from root>
<Side Remark line 6: found>
<Side Remark line 9: search a subtree>
<Side Remark line 13: not found>

```
boolean search(E e) {
  current = root; // Start from the root

  while (current != null) {
```

```
    if (match(e, current)) { // Element is in the node
      return true; // Element is found
    }
    else {
      current = getChildNode(e, current); // Search in a subtree
    }
  }

  return false; // Element is not in the tree
}
```

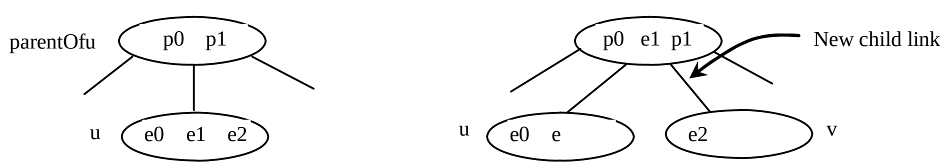
The match(e, current) method checks whether element e is in the current node. The getChildNode(e, current) method returns the root of the subtree for further search. Initially, let current point to the root (line 2). Repeat searching the element in the current node until current is null (line 4) or the element matches element (line 5) matches current.element.

9.4 Inserting an Element into a 2-4 Tree

<Side Remark: overflow>
<Side Remark: split>

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:

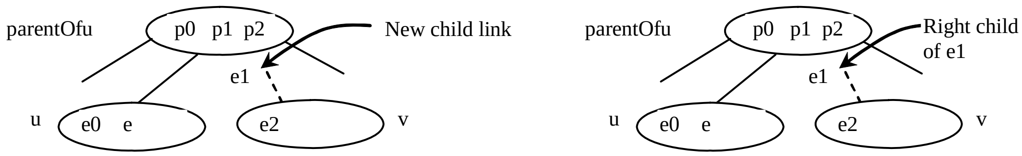
- Let u be the *leaf* 4-node in which the element will be inserted and $parentOfu$ be the parent of u, as shown in Figure 9.6(a).
- Create a new node named v, move e_2 to v.
- If $e < e_1$, insert e to u; otherwise insert e to v. Assume that $e_0 < e < e_1$, e is inserted into u, as shown in Figure 9.6(b).
- Insert e_1 along with its right child (i.e., v) to the parent node, as shown in Figure 9.6(b).



(a) Before inserting *e* (b) After inserting *e*

Figure 9.6
The splitting operation creates a new node and inserts the median element to its parent.

The parent node is a 3-node in Figure 9.6. So there is room to insert *e* to the parent node. What would happen if it is a 4-node, as shown in Figure 9.7? This would require that the parent node be split. The process is the same as splitting a leaf 4-node, except that you have to also insert the element along with its right child.

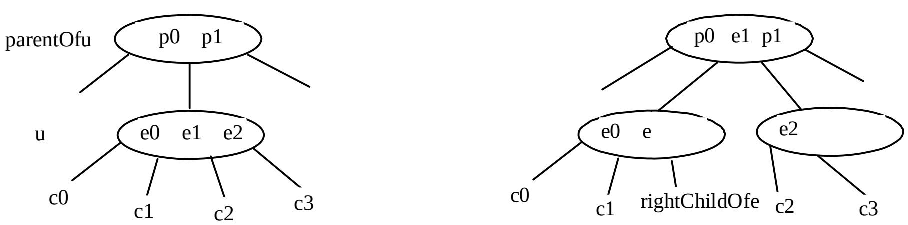


(a) The parent is a 4-node (b) Inserting e_1 into the parent

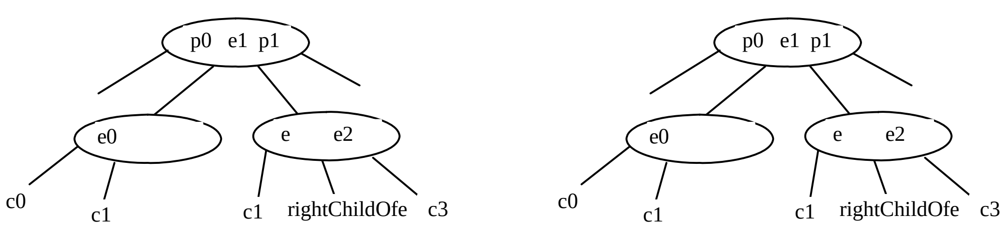
Figure 9.7
Insertion process continues if the parent node is a 4-node.

The algorithm can be modified as follows:

- Let u be the 4-node (*leaf or non-leaf*) in which the element will be inserted and $parentOfu$ be the parent of u, as shown in Figure 9.8(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 9.6(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.



(a) Before inserting e (b) After inserting e ($e_0 < e < e_1$)



(c) After inserting e ($e_1 < e < e_2$) (d) After inserting e ($e_2 < e$)

Figure 9.8
An interior node may be split to resolve overflow.

Listing 9.2 gives an algorithm for inserting an element.

Listing 9.2 Inserting an Element to a 2-4 Tree

PD: Please add line numbers in the following code

<Side Remark line 3: create a new node>
<Side Remark line 5: search e>
<Side Remark line 6: insert e>
<Side Remark line 9: one element added>
<Side Remark line 10: element added>
<Side Remark line 13: insert to a node>
<Side Remark line 15: a 2- or 3- node>
<Side Remark line 20: split 4-node>
<Side Remark line 23: new root>
<Side Remark line 29: insert median to parent>

```
public boolean insert(E e) {
  if (root == null)
    root = new Tree24Node<E>(e); // Create a new root for element
  else {
    Locate leafNode for inserting e
    insert(e, null, leafNode); // The right child of e is null
  }

  size++; // Increase size
  return true; // Element inserted
}

private void insert(E e, Tree24Node<E> rightChildOfe,
    Tree24Node<E> u) {
  if (u is a 2- or 3- node) { // u is a 2- or 3- node
    insert23(e, rightChildOfe, u); // Insert e to node u
  }
  else { // Split a 4-node u
    Tree24Node<E> v = new Tree24Node<E>(); // Create a new node
    E median = split(e, rightChildOfe, u, v); // Split u

    if (u == root) { // u is the root
      root = new Tree24Node<E>(median); // New root
      root.child.add(u); // u is the left child of median
      root.child.add(v); // v is the right child of median
    }
    else {
      Get the parent of u, parentOfu;
      insert(median, v, parentOfu); // Inserting median to parent
    }
  }
}
```

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 9.9 shows the steps of inserting elements.

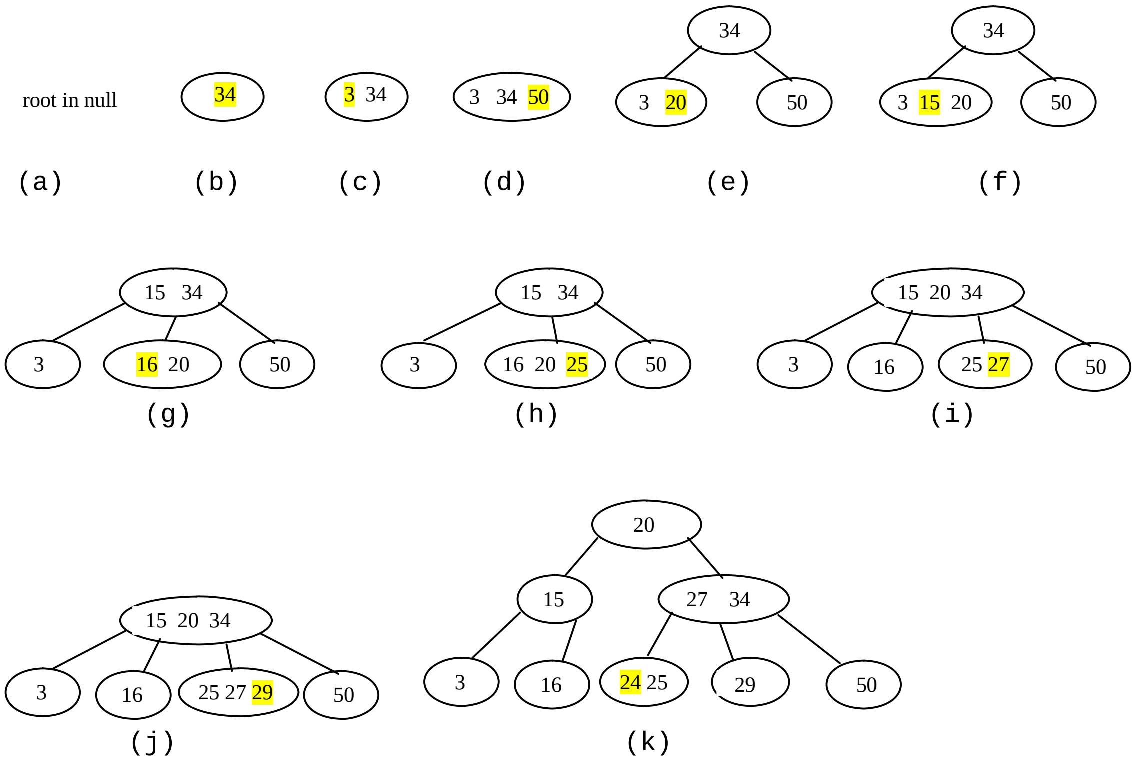


Figure 9.9
The tree changes after adding 34, 3, 50, 20, 15, 16, 25, 27, 29, and 24 into an empty tree.

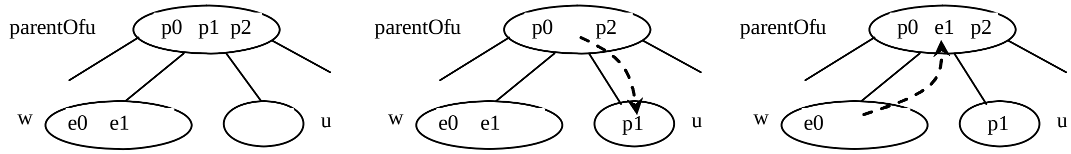
9.5 Deleting an Element from a 2-4 Tree

To delete an element from a 2-4 tree, first search the element in the tree to locate the node that contains the element. 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:

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

<Side Remark: underflow>
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:

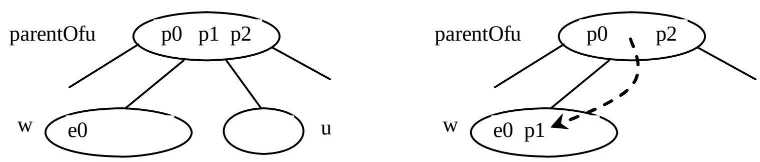
<Side Remark: transfer>
Case 2.1: u's immediate left or right sibling is a 3- or 4- node. Let the node be w, as shown in Figure 9.10(a) (assume that w is a left sibling of u). Perform a *transfer* operation that moves an element from $parentOfu$ to u, as shown in Figure 9.10(b), and move an element from w to replace the moved element in $parentOfu$, as shown in Figure 9.10(c).



(a) u is now empty (b) Move p_1 to u (c) Move e_1 to replace p_1

Figure 9.10
The transfer operation fills the empty node u.

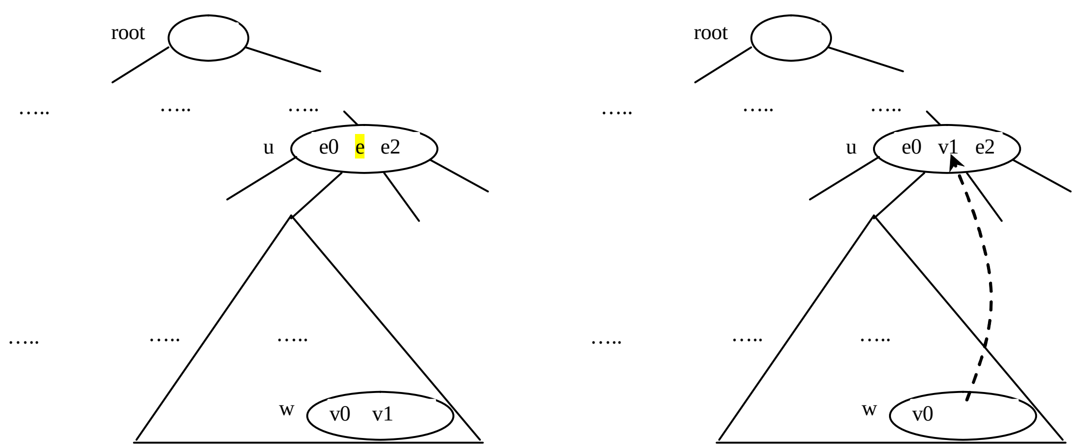
<Side Remark: fusion>
Case 2.2: Both u's immediate left and right sibling are 2-node if exist (u may have only one sibling). Let the node be w, as shown in Figure 9.11(a) (assume that w is a left sibling of u). Perform a *fusion* operation that discards u and moves an element from *parentOfu* to w, as shown in Figure 9.11(b). If *parentOfu* becomes empty, repeat Case 2 recursively to perform a transfer or a fusion on *parentOfu*.



(a) w is a 2-node (b) Move p_1 to w

Figure 9.11
The fusion operation fills the empty node u.

<Side Remark: internal node>
Case 3: u is a non-leaf node. Find the rightmost leaf node in the left subtree of e. Let this node be w, as shown in Figure 9.12(a). Move the last element in w to replace e in u, as shown in Figure 9.12(b). If w becomes empty, apply a transfer or fusion operation on w.



(a) e is in u (b) Replace e with v1

Figure 9.12

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

Listing 9.3 describes the algorithm for deleting an element.

Listing 9.3 Deleting an Element from a 2-4 Tree

******PD: Please add line numbers in the following code******

<Side Remark line 3: create a new node>
<Side Remark line 5: search e>
<Side Remark line 6: insert e>
<Side Remark line 9: one element added>
<Side Remark line 10: element added>
<Side Remark line 13: insert to a node>
<Side Remark line 15: a 2- or 3- node>
<Side Remark line 20: split 4-node>
<Side Remark line 23: new root>
<Side Remark line 29: insert median to parent>

```
/** Delete the specified element from the tree */
public boolean delete(E e) {
  Locate the node that contains the element e
  if (the node is found) {
    delete(e, node); // Delete element e from the node
    size--; // After one element deleted
    return true; // Element deleted successfully
  }
  else {
    node = getChildNode(e, node);
  }

  return false; // Element not in the tree
}

/** Delete the specified element from the node */
private void delete(E e, Tree24Node<E> node) {
  if (e is in a leaf node) {
    // Get the path that leads to e from the root
    ArrayList<Tree24Node<E>> path = path(e);

    Remove e from the node;

    // Check node for underflow along the path and fix it
    validate(e, node, path); // Check underflow node
  }
  else { // e is in an internal node
    Locate the rightmost node in the left subtree of node u;
    Get the rightmost element from the rightmost node;

    // Get the path that leads to e from the root
    ArrayList<Tree24Node<E>> path = path(rightmostElement);

    Replace the element in the node with the rightmost element

    // Check node for underflow along the path and fix it
```

```
    validate(rightmostElement, rightmostNode, path);
  }
}

/** Perform transfer and confusion operations if necessary */
private void validate(E e, Tree24Node<E> u,
    ArrayList<Tree24Node<E>> path) {
  for (int i = path.size() - 1; i >= 0; i--) {
    if (u is not empty)
      return; // Done, no need to perform transfer or fusion

    Tree24Node<E> parentOfu = path.get(i - 1); // Get parent of u

    // Check two siblings
    if (left sibling of u has more than one element) {
      Perform a transfer on u with its left sibling
    }
    else if (right sibling of u has more than one element) {
      Perform a transfer on u with its right sibling
    }
    else if (u has left sibling) { // Fusion with a left sibling
      Perform a fusion on u with its left sibling
      u = parentOfu; // Back to the loop to check the parent node
    }
    else { // Fusion with right sibling (right sibling must exist)
      Perform a fusion on u with its right sibling
      u = parentOfu; // Back to the loop to check the parent node
    }
  }
}
```

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 20), delete the e from the node (line 22), and invoke validate to check and fix empty node (line 25). 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 9.13.

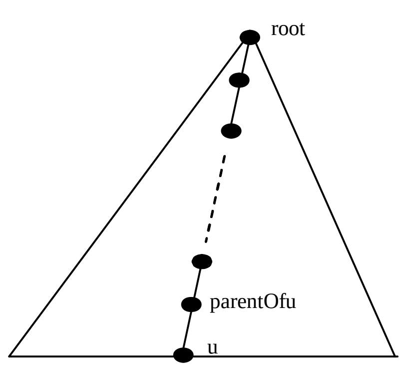


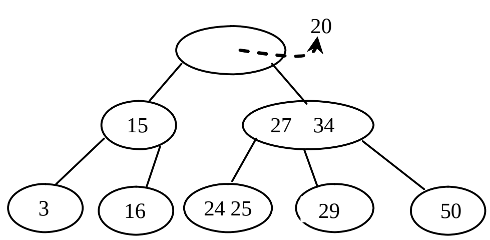
Figure 9.13
The nodes along the path may become empty as result of transfer and fusion operations.

If the node is a non-leaf node, locate the rightmost element in the left subtree of the node (lines 28-29), get the path that leads to the rightmost element from the root (line 32), replace e in the node with the rightmost element (line 34), and invoke validate to fix the rightmost node if it is empty (line 37).

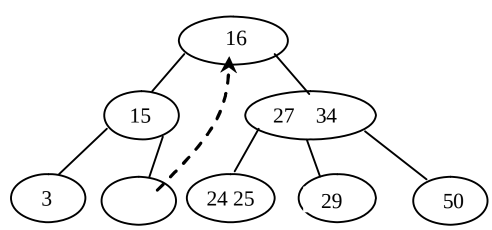
The validate(E e, Tree24Node<E> u, ArrayList<Tree24Node<E>> path) checks whether u is empty and performs a transfer or fusion operation to fix the empty node. The validate method exits when node is not empty (line 46). Otherwise, consider one of the following cases:

1. If u has a left sibling with more than one element, perform a transfer on u with its left sibling (line 52).
2. Otherwise, if u has a right sibling with more than one element, perform a transfer on u with its right sibling (line 55).
3. Otherwise, if u has a left sibling, perform a fusion on u with its left sibling (line 58), and reset u to parentOfu (line 59).
4. Otherwise, u must have a right sibling. Perform a fusion on u with its right sibling (line 62), and reset u to parentOfu (line 63).

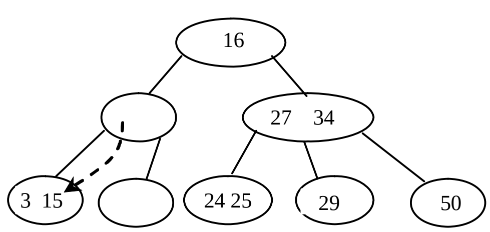
Only one of the preceding cases is executed. Afterwards, a new iteration starts to perform a transfer or fusion operation on a new node u if needed. Figure 9.14 shows the steps of deleting elements.



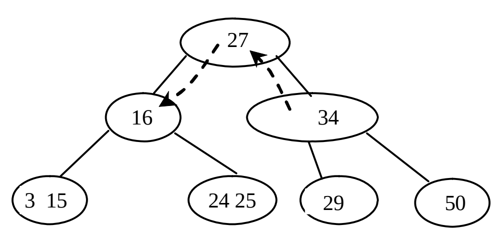
(a) Delete 20



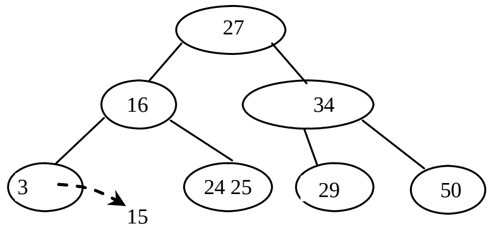
(b) Replace 20 with 16



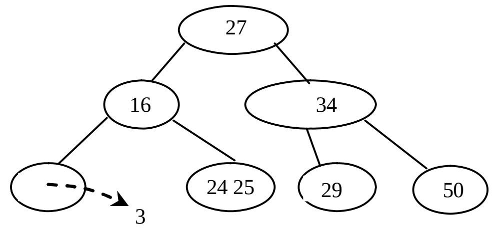
(c) Perform a fusion



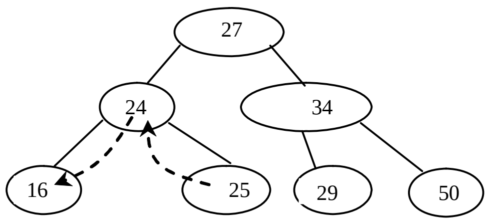
(d) Perform a transfer



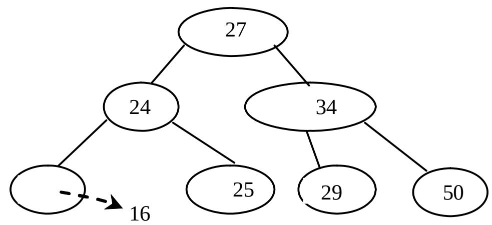
(e) Delete 15



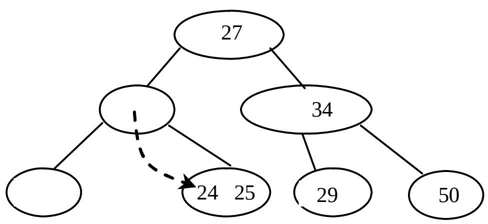
(f) Delete 3



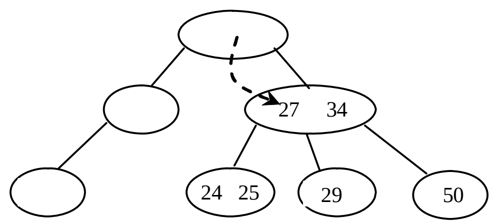
(g) Perform a transfer



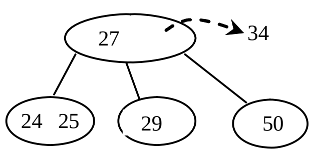
(h) Delete 16



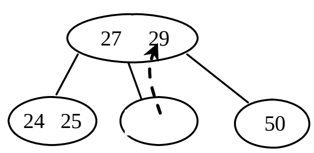
(i) Perform a fusion



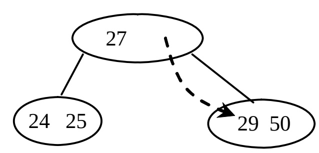
(j) Perform a fusion



(k) Delete 34



(l) Replace 34 with 16



(m) Perform a fusion

Figure 9.14
The tree changes after deleting 20, 15, 3, 6, and 34 from a 2-4 tree.

9.6 Traversing Elements in a 2-4 Tree

Inorder and preorder 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 visit the subtrees from the left to right.

For example, in the 2-4 tree in Figure 9.9(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

9.7 Implementing the Tree24 Class

Listing 9.4 gives the complete source code for the Tree24 class.

Listing 9.4 Tree24.java

******PD: Please add line numbers in the following code******

******Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.***

<Side Remark line 4: root>
<Side Remark line 5: size>
<Side Remark line 8: no-arg constructor>
<Side Remark line 12: constructor>
<Side Remark line 18: search>
<Side Remark line 22: found?>
<Side Remark line 26: next subtree>
<Side Remark line 34: find a match>
<Side Remark line 36: matched?>
<Side Remark line 43: next subtree>
<Side Remark line 44: leaf node?>
<Side Remark line 47: insertion point>
<Side Remark line 54: insert to tree>
<Side Remark line 55: empty tree?>
<Side Remark line 59: find leaf node>
<Side Remark line 71: insert to node>
<Side Remark line 79: insert to node>
<Side Remark line 85: no overflow>
<Side Remark line 90: overflow>
<Side Remark line 91: split>
<Side Remark line 93: u is root?>
<Side Remark line 101: insert to parentOfu>
<Side Remark line 110: insert to node>
<Side Remark line 112: insertion point>
<Side Remark line 119: split>
<Side Remark line 123: get median>
<Side Remark line 127: insert e>
<Side Remark line 133: insert rightChildOfe>
<Side Remark line 138: return median>
<Side Remark line 142: get path>
<Side Remark line 147: add node searched>
<Side Remark line 156: return path>
<Side Remark line 160: delete from tree>
<Side Remark line 162: locate the node>
<Side Remark line 164: found?>
<Side Remark line 165: delete from node>
<Side Remark line 177: delete from node>
<Side Remark line 178: leaf node?>
<Side Remark line 182: delete e>
<Side Remark line 184: node is root?>
<Side Remark line 190: validate tree>

<Side Remark line 192: non-leaf node>
<Side Remark line 194: rightmost element>
<Side Remark line 206: replace element>
<Side Remark line 209: validate tree>
<Side Remark line 214: validate tree>
<Side Remark line 222: transfer with left sibling>
<Side Remark line 226: transfer with right sibling>
<Side Remark line 233: fusion with left sibling>
<Side Remark line 248: fusion with right sibling>
<Side Remark line 262: locate insertion point>
<Side Remark line 273: transfer with left sibling>
<Side Remark line 290: transfer with right sibling>
<Side Remark line 305: fusion with left sibling>
<Side Remark line 319: fusion with right sibling>
<Side Remark line 338: preorder >
<Side Remark line 343: recursive preorder >
<Side Remark line 343: recursive preorder >
<Side Remark line 363: inner Tree24Node class>
<Side Remark line 365: element list>
<Side Remark line 367: child list>

```
import java.util.ArrayList;

public class Tree24<E extends Comparable<E>> implements Tree<E> {
  private Tree24Node<E> root;
  private int size;

  /** Create a default 2-4 tree */
  public Tree24() {
  }

  /** Create a 2-4 tree from an array of objects */
  public Tree24(E[] elements) {
    for (int i = 0; i < elements.length; i++)
      insert(elements[i]);
  }

  /** Search an element in the tree */
  public boolean search(E e) {
    Tree24Node<E> current = root; // Start from the root

    while (current != null) {
      if (matched(e, current)) { // Element is in the node
        return true; // Element found
      }
      else {
        current = getChildNode(e, current); // Search in a subtree
      }
    }

    return false; // Element is not in the tree
  }
```

```
  /** Return true if the element is found in this node */
  private boolean matched(E e, Tree24Node<E> node) {
    for (int i = 0; i < node.elements.size(); i++)
      if (node.elements.get(i).equals(e))
        return true; // Element found

    return false; // No match in this node
  }

  /** Locate a child node to search element e */
  private Tree24Node<E> getChildNode(E e, Tree24Node<E> node) {
    if (node.child.size() == 0)
      return null; // node is a leaf

    int i = locate(e, node); // Locate the insertion point for e
    return node.child.get(i); // Return the child node
  }

  /** Insert element e into the tree
   *  Return true if the element is inserted successfully
   */
  public boolean insert(E e) {
    if (root == null)
      root = new Tree24Node<E>(e); // Create a new root for element
    else {
      // Locate the leaf node for inserting e
      Tree24Node<E> leafNode = null;
      Tree24Node<E> current = root;
      while (current != null)
        if (matched(e, current)) {
          return false; // Duplicate element found, nothing inserted
        }
        else {
          leafNode = current;
          current = getChildNode(e, current);
        }

      // Insert the element e into the leaf node
      insert(e, null, leafNode); // The right child of e is null
    }

    size++; // Increase size
    return true; // Element inserted
  }

  /** Insert element e into node u */
  private void insert(E e, Tree24Node<E> rightChildOfe,
      Tree24Node<E> u) {
    // Get the search path that leads to element e
    ArrayList<Tree24Node<E>> path = path(e);

    for (int i = path.size() - 1; i >= 0; i--) {
      if (u.elements.size() < 3) { // u is a 2-node or 3-node
        insert23(e, rightChildOfe, u); // Insert e to node u
        break; // No further insertion to u's parent needed
      }
```

```
      else {
        Tree24Node<E> v = new Tree24Node<E>(); // Create a new node
        E median = split(e, rightChildOfe, u, v); // Split u

        if (u == root) {
          root = new Tree24Node<E>(median); // New root
          root.child.add(u); // u is the left child of median
          root.child.add(v); // v is the right child of median
          break; // No further insertion to u's parent needed
        }
        else {
          // Use new values for the next iteration in the for loop
          e = median; // Element to be inserted to parent
          rightChildOfe = v; // Right child of the element
          u = path.get(i - 1); // New node to insert element
        }
      }
    }
  }

  /** Insert element to a 2- or 3- and return the insertion point */
  private void insert23(E e, Tree24Node<E> rightChildOfe,
      Tree24Node<E> node) {
    int i = this.locate(e, node); // Locate where to insert
    node.elements.add(i, e); // Insert the element into the node
    if (rightChildOfe != null)
      node.child.add(i + 1, rightChildOfe); // Insert the child link
  }

  /** Split a 4-node u into u and v and insert e to u or v */
  private E split(E e, Tree24Node<E> rightChildOfe,
      Tree24Node<E> u, Tree24Node<E> v) {
    // Move the last element in node u to node v
    v.elements.add(u.elements.remove(2));
    E median = u.elements.remove(1);

    // Split children for a non-leaf node
    // Move the last two children in node u to node v
    if (u.child.size() > 0) {
      v.child.add(u.child.remove(2));
      v.child.add(u.child.remove(2));
    }

    // Insert e into a 2- or 3- node u or v.
    if (e.compareTo(median) < 0)
      insert23(e, rightChildOfe, u);
    else
      insert23(e, rightChildOfe, v);

    return median; // Return the median element
  }

  /** Return a search path that leads to element e */
  private ArrayList<Tree24Node<E>> path(E e) {
    ArrayList<Tree24Node<E>> list = new ArrayList<Tree24Node<E>>();
    Tree24Node<E> current = root; // Start from the root
```

```
    while (current != null) {
      list.add(current); // Add the node to the list
      if (matched(e, current)) {
        break; // Element found
      }
      else {
        current = getChildNode(e, current);
      }
    }

    return list; // Return an array of nodes
  }

  /** Delete the specified element from the tree */
  public boolean delete(E e) {
    // Locate the node that contains the element e
    Tree24Node<E> node = root;
    while (node != null)
      if (matched(e, node)) {
        delete(e, node); // Delete element e from node
        size--; // After one element deleted
        return true; // Element deleted successfully
      }
      else {
        node = getChildNode(e, node);
      }

    return false; // Element not in the tree
  }

  /** Delete the specified element from the node */
  private void delete(E e, Tree24Node<E> node) {
    if (node.child.size() == 0) { // e is in a leaf node
      // Get the path that leads to e from the root
      ArrayList<Tree24Node<E>> path = path(e);

      node.elements.remove(e); // Remove element e

      if (node == root) { // Special case
        if (node.elements.size() == 0)
          root = null; // Empty tree
        return; // Done
      }

      validate(e, node, path); // Check underflow node
    }
    else { // e is in an internal node
      // Locate the rightmost node in the left subtree of the node
      int index = locate(e, node); // Index of e in node
      Tree24Node<E> current = node.child.get(index);
      while (current.child.size() > 0) {
        current = current.child.get(current.child.size() - 1);
      }
      E rightmostElement =
        current.elements.get(current.elements.size() - 1);
```

```
      // Get the path that leads to e from the root
      ArrayList<Tree24Node<E>> path = path(rightmostElement);

      // Replace the deleted element with the rightmost element
      node.elements.set(index, current.elements.remove(
        current.elements.size() - 1));

      validate(rightmostElement, current, path); // Check underflow
    }
  }

  /** Perform transfer and confusion operations if necessary */
  private void validate(E e, Tree24Node<E> u,
      ArrayList<Tree24Node<E>> path) {
    for (int i = path.size() - 1; u.elements.size() == 0; i--) {
      Tree24Node<E> parentOfu = path.get(i - 1); // Get parent of u
      int k = locate(e, parentOfu); // Index of e in the parent node

      // Check two siblings
      if (k > 0 && parentOfu.child.get(k - 1).elements.size() > 1) {
        leftSiblingTransfer(k, u, parentOfu);
      }
      else if (k + 1 < parentOfu.child.size() &&
          parentOfu.child.get(k + 1).elements.size() > 1) {
        rightSiblingTransfer(k, u, parentOfu);
      }
      else if (k - 1 >= 0) { // Fusion with a left sibling
        // Get left sibling of node u
        Tree24Node<E> leftNode = parentOfu.child.get(k - 1);

        // Perform a fusion with left sibling on node u
        leftSiblingFusion(k, leftNode, u, parentOfu);

        // Done when root becomes empty
        if (parentOfu == root && parentOfu.elements.size() == 0) {
          root = leftNode;
          break;
        }

        u = parentOfu; // Back to the loop to check the parent node
      }
      else { // Fusion with right sibling (right sibling must exist)
        // Get left sibling of node u
        Tree24Node<E> rightNode = parentOfu.child.get(k + 1);

        // Perform a fusion with right sibling on node u
        rightSiblingFusion(k, rightNode, u, parentOfu);

        // Done when root becomes empty
        if (parentOfu == root && parentOfu.elements.size() == 0) {
          root = rightNode;
          break;
        }

        u = parentOfu; // Back to the loop to check the parent node
```

```
      }
    }
  }

  /** Locate the insertion point of the element in the node */
  private int locate(E o, Tree24Node<E> node) {
    for (int i = 0; i < node.elements.size(); i++) {
      if (o.compareTo(node.elements.get(i)) <= 0) {
        return i;
      }
    }

    return node.elements.size();
  }

  /** Perform a transfer with a left sibling */
  private void leftSiblingTransfer(int k,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Move an element from the parent to u
    u.elements.add(0, parentOfu.elements.get(k - 1));

    // Move an element from the left node to the parent
    Tree24Node<E> leftNode = parentOfu.child.get(k - 1);
    parentOfu.elements.set(k - 1,
      leftNode.elements.remove(leftNode.elements.size() - 1));

    // Move the child link from left sibling to the node
    if (leftNode.child.size() > 0)
      u.child.add(0, leftNode.child.remove(
        leftNode.child.size() - 1));
  }

  /** Perform a transfer with a right sibling */
  private void rightSiblingTransfer(int k,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Transfer an element from the parent to u
    u.elements.add(parentOfu.elements.get(k));

    // Transfer an element from the right node to the parent
    Tree24Node<E> rightNode = parentOfu.child.get(k + 1);
    parentOfu.elements.set(k, rightNode.elements.remove(0));

    // Move the child link from right sibling to the node
    if (rightNode.child.size() > 0)
      u.child.add(rightNode.child.remove(0));
  }

  /** Perform a fusion with a left sibling */
  private void leftSiblingFusion(int k, Tree24Node<E> leftNode,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Transfer an element from the parent to the left sibling
    leftNode.elements.add(parentOfu.elements.remove(k - 1));

    // Remove the link to the empty node
    parentOfu.child.remove(k);
```

```
    // Adjust child links for non-leaf node
    if (u.child.size() > 0)
      leftNode.child.add(u.child.remove(0));
  }

  /** Perform a fusion with a right sibling */
  private void rightSiblingFusion(int k, Tree24Node<E> rightNode,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Transfer an element from the parent to the right sibling
    rightNode.elements.add(0, parentOfu.elements.remove(k));

    // Remove the link to the empty node
    parentOfu.child.remove(k);

    // Adjust child links for non-leaf node
    if (u.child.size() > 0)
      rightNode.child.add(0, u.child.remove(0));
  }

  /** Get the number of nodes in the tree */
  public int getSize() {
    return size;
  }

  /** Preorder traversal from the root */
  public void preorder() {
    preorder(root);
  }

  /** Preorder traversal from a subtree */
  private void preorder(Tree24Node<E> root) {
    if (root == null)return;
    for (int i = 0; i < root.elements.size(); i++)
      System.out.print(root.elements.get(i) + " ");

    for (int i = 0; i < root.child.size(); i++)
      preorder(root.child.get(i));
  }

  /** Inorder traversal from the root*/
  public void inorder() {
    // Left as exercise
  }

  /** Postorder traversal from the root */
  public void postorder() {
    // Left as exercise
  }

  /** Return true if the tree is empty */
  public boolean isEmpty() {
    return root == null;
  }

  /** Return an iterator to traverse elements in the tree */
  public java.util.Iterator iterator() {
```

```
    // Left as exercise
    return null;
  }

  /** Define a 2-4 tree node */
  protected static class Tree24Node<E extends Comparable<E>> {
    // elements has maximum three values
    ArrayList<E> elements = new ArrayList<E>(3);
    // Each has maximum four childres
    ArrayList<Tree24Node<E>> child
      = new ArrayList<Tree24Node<E>>(4);

    /** Create an empty Tree24 node */
    Tree24Node() {
    }

    /** Create a Tree24 node with an initial element */
    Tree24Node(E o) {
      elements.add(o);
    }
  }
}
```

<Side Remark: root>
<Side Remark: size>
The Tree24 class contains the data fields root and size (lines 6-7). root references the root node and size stores the number of elements in the tree.

<Side Remark: constructors>
The Tree24 class has two constructors: a no-arg constructor (lines 10-11) that constructs an empty tree and a constructor that creates an initial Tree24 from an array of elements (lines 12-17).

<Side Remark: search>
The search method (lines 20-33) searches an element in the tree. It returns true (line 25) if the element is in the tree and returns false if the search arrives at an empty subtree (line 32).

<Side Remark: matched>
The matched(e, node) method (lines 36-42) checks where the element e is in the node.

<Side Remark: getChildNode>
The getChildNode(e, node) method (lines 45-51) returns the root of a subtree where e should be searched.

<Side Remark: insert(e)>
The insert(E e) method inserts an element in a tree (lines 56-78). If the tree is empty, a new root is created (line 58). The method locates a leaf node in which the element will be inserted and invokes insert(e, null, leafNode) to insert the element (line 73).

<Side Remark: insert(e, rightChildOfe, u)>

The insert(e, rightChildOfe, u) method inserts an element into node u (lines 81-111). The method first invokes path(e) (line 84) to obtain a search path from the root to node u. Each iteration of the for loop considers u and its parent parentOfu (lines 108-110). 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 88). No split is needed (line 89). Otherwise, create a new node v (line 92) and invoke split(e, rightChildOfe, u, v) (line 93) 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 96-98). If u is not the root, insert median to parentOfu in the next iteration (lines 102-105).

<Side Remark: insert23>
The insert23(e, rightChildOfe, node) method inserts e along with the reference to its right child into the node (lines 112-124). The method first invokes locate(e, node) (line 114) to locate an insertion point, then insert e into the node (line 115). If rightChildOfe is not null, it is inserted into the child list of the node (line 117).

<Side Remark: split>
The split(e, rightChildOfe, u, v) method split a 4-node u (lines 121-141). This is accomplished as follows: (1) Move the last element from u to v and remove the median element from u (lines 124-125); (2) Move the last two child links from u to v (lines 129-132) if u is a non-leaf node; (3) If e < median, insert e into u; otherwise, insert e into v (lines 135-138); (4) return median (line 140).

<Side Remark: path>
The path(e) method returns an ArrayList of nodes searched from the root in order to locate e (lines 144-159). If e is in the tree, the last node in the pat contains e. Otherwise the last node is where e should be inserted.

<Side Remark: delete(e)>
The delete(E e) method deletes an element from the tree (lines 162-176). The method first locates the node that contains e and invokes delete(e, node) to delete e from the node (line 167). If the element is not in the tree, return false (line 175).

<Side Remark: delete(e, node)>
The delete(e, node) method deletes an element from node u (lines 179-213). If the node a leaf node, obtain the path that leads to e (line 182), delete e (line 184), set root to null if the tree becomes empty (lines 186-190), and invoke validate to apply transfer and fusion operation on empty nodes (line 192). If the node a non-leaf node, locate the rightmost element (lines 196-202), obtain the path that leads to e (line 205), replace e with the rightmost element (line 208-209), and invoke validate to apply transfer and fusion operation on empty nodes (line 211).

<Side Remark: validate>
The validate(e, u, path) method ensures that the tree is a valid 2-4 tree (lines 216-261). The for loop terminates when u's not empty (line 218). 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

224). Otherwise, if a right sibling with more than one element exists, perform a transfer on u with the left sibling (line 228). Otherwise, if a left sibling exists, perform a fusion on u with the left sibling (lines 232-241), and validate parentOfu in the next loop iteration (line 243). Otherwise, perform a fusion on u with the right sibling

<Side Remark: locate>
The locate(e, node) method locates the index of e in the node (lines 264-272).

<Side Remark: transfer>
<Side Remark: fusion>
The leftSiblingTransder(k, u, parentOfu) method performs a transfer on u with its left sibling (lines 275-289). The rightSiblingTransder(k, u, parentOfu) method performs a transfer on u with its right sibling (lines 292-304). The leftSiblingFusion(k, leftNode, u, parentOfu) method performs a fusion on u with its left sibling leftNode (lines 307-318). The rightSiblingFusion(k, rightNode, u, parentOfu) method performs a fusion on u with its right sibling rightNode (lines 321-332).

<Side Remark: preorder>
The preorder() method displays all the elements in the tree in preorder (lines 340-352).

<Side Remark: Tree24Node>
The inner class Tree24Node defines a class for a node in the tree (lines 365-380).

9.8 Testing the Tree24 Class

Listing 9.4 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 9.d TestTree24.java

******PD: Please add line numbers in the following code******
******Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.***
<Side Remark line 4: create a Tree24>
<Side Remark line 6: insert 34>
<Side Remark line 7: insert 3>
<Side Remark line 8: insert 50>
<Side Remark line 15: insert 24>
<Side Remark line 21: insert 70>
<Side Remark line 24: delete 34>

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

    tree.insert(34);
    tree.insert(3);
    tree.insert(50);
    tree.insert(20);
```

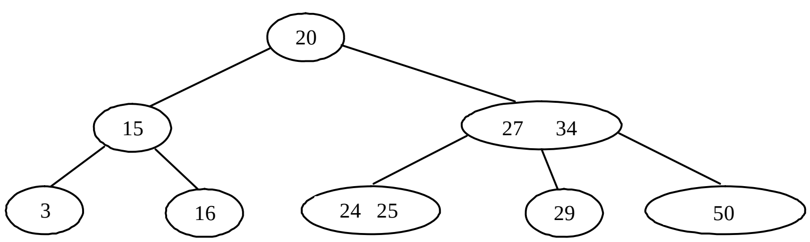
```
    tree.insert(15);
    tree.insert(16);
    tree.insert(25);
    tree.insert(27);
    tree.insert(29);
    tree.insert(24);
    System.out.print("\nAfter inserting 24:");
    printTree(tree);
    tree.insert(23);
    tree.insert(22);
    tree.insert(60);
    tree.insert(70);
    printTree(tree);

    tree.delete(34);
    System.out.print("\nAfter deleting 34:");
    printTree(tree);

    tree.delete(25);
    System.out.print("\nAfter deleting 25:");
    printTree(tree);

    tree.delete(50);
    System.out.print("\nAfter deleting 50:");
    printTree(tree);

    tree.delete(16);
    System.out.print("\nAfter deleting 16:");
    printTree(tree);

    tree.delete(3);
    System.out.print("\nAfter deleting 3:");
    printTree(tree);

    tree.delete(15);
    System.out.print("\nAfter deleting 15:");
    printTree(tree);
  }

  public static void printTree(Tree tree) {
    // Traverse tree
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

<Output>

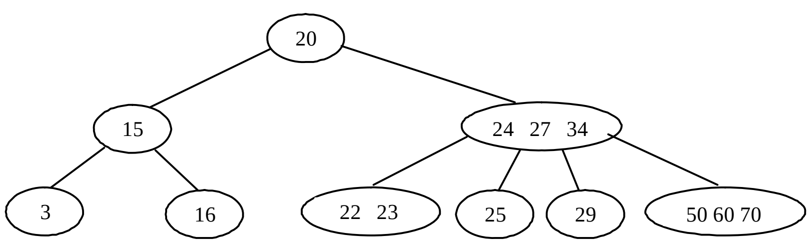
```
After inserting 24:
Preorder: 20 15 3 16 27 34 24 25 29 50
The number of nodes is 10

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
```
<End Output>

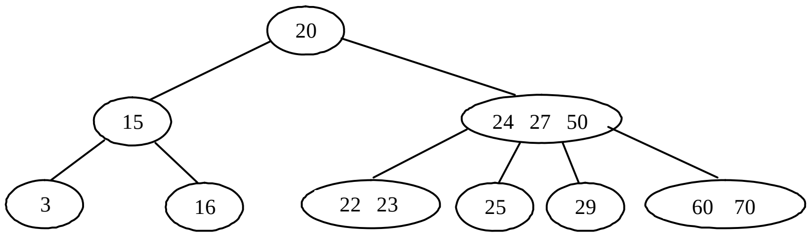
Figure 9.15 shows how the tree evolves as elements are added to the tree. After 34, 3, 50, 20, 15, 16, 25, 27, 29, and 24 are added to the tree, the tree is shown in Figure 9.15(a). After inserting 23, 22, 60, and 70, the tree is shown in Figure 9.15(b). After inserting 23, 22, 60, and 70, the tree is shown in Figure 9.15(b). After deleting 34, the tree is shown in Figure 9.15(c). After deleting 25, the tree is shown in Figure 9.15(d). After deleting 50, the tree is shown in Figure 9.15(e). After deleting 16, the tree is shown in Figure 9.15(f). After deleting 3, the tree is shown in Figure 9.15(g). After deleting 15, the tree is shown in Figure 9.15(h).



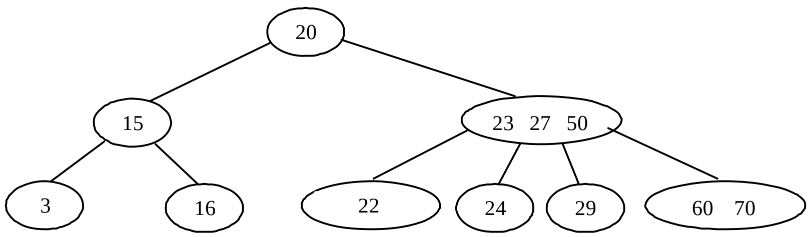
(a) After inserting 34, 3, 50, 20, 15, 16, 25, 27, 29, and 24, in this order



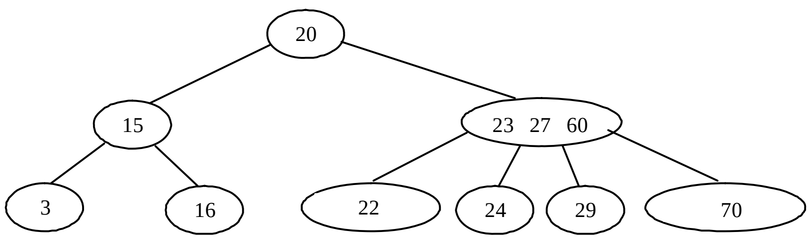
(b) After inserting 23, 22, 60, and 70



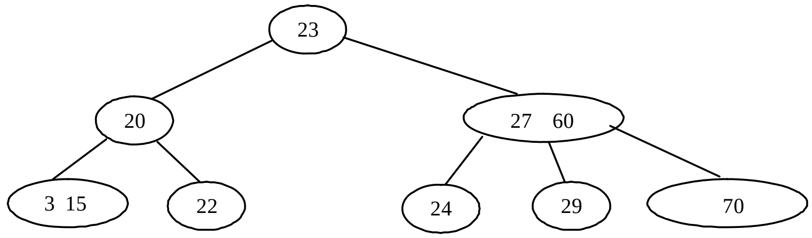
(c) After deleting 34



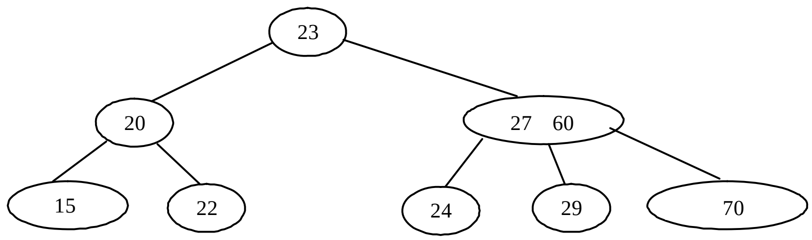
(d) After deleting 25



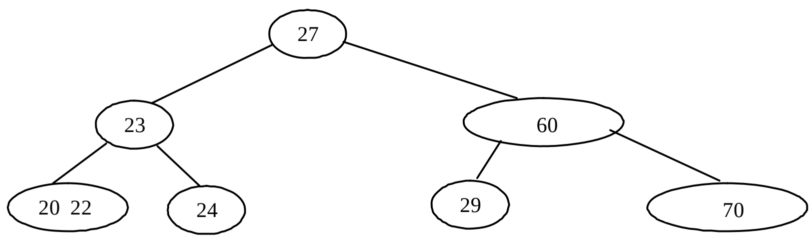
(e) After deleting 50



(f) After deleting 16



(g) After deleting 3



(h) After deleting 15

Figure 9.15
The tree evolves as elements are inserted and deleted.

9.9 Time Complexity Analysis

Since a 2-4 tree is a complete 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 is 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 is a constant time to perform a transfer and fusion operation. So, the delete method takes $O(\log n)$ time.

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

1. Each node except the root contains between $\lceil d/2 \rceil - 1$ and $d-1$ number of keys.

2. The root may contain up to $d-1$ number of keys.

3. A non-leaf node with k number of keys has $k+1$ number of children.

4. All leaf nodes have the same depth.

Figure 9.16 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.

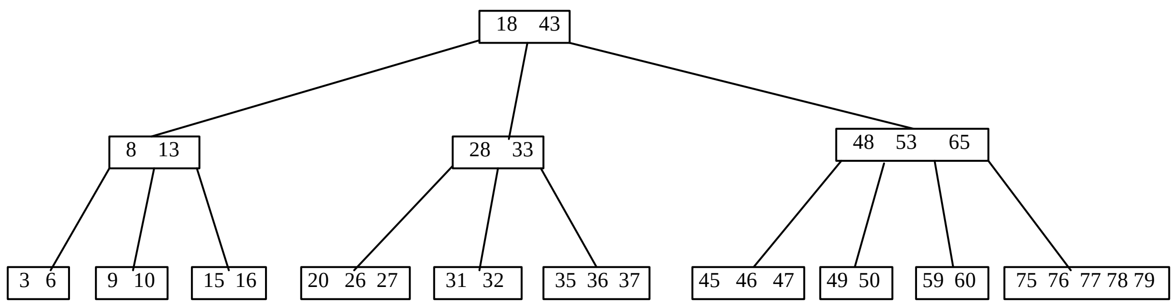


Figure 9.16
In a B-tree of order 6, each node except the root may contain between 2 and 5 keys.

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 9.17. 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.

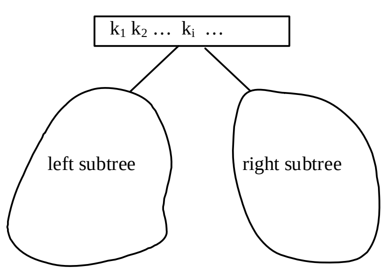


Figure 9.17
The keys in the left (right) subtree of key k_i are less than (greater than) k_i.

<Side Remark: one block per node>

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.

<Side Remark: insertion>

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 similar *split* operation 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 9.18. If inserting m into the parent node of u causes an overflow, repeat the same split process on the parent node.

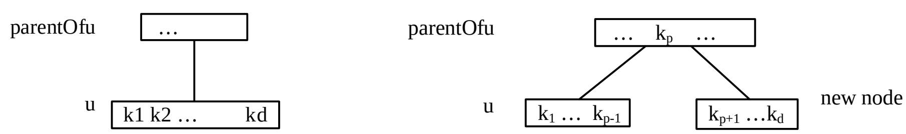


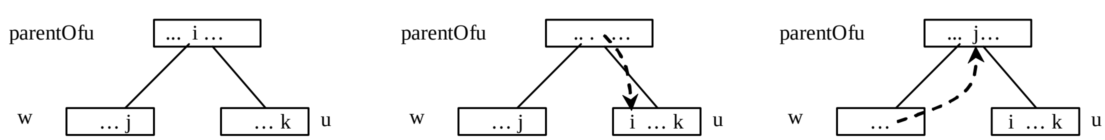
Figure 9.18

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

<Side Remark: deletion>

Deleting a key k from a B-tree can be done in the same way as for 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 w of u that has more than $\lceil d/2 \rceil - 1$ keys if such sibling exists, as shown in Figure 9.20. Otherwise perform a fusion with a sibling w of u, as shown in Figure 9.21.



(a) before a transfer is performed (b) key i moved to node u (c) key j moved to parentOfu

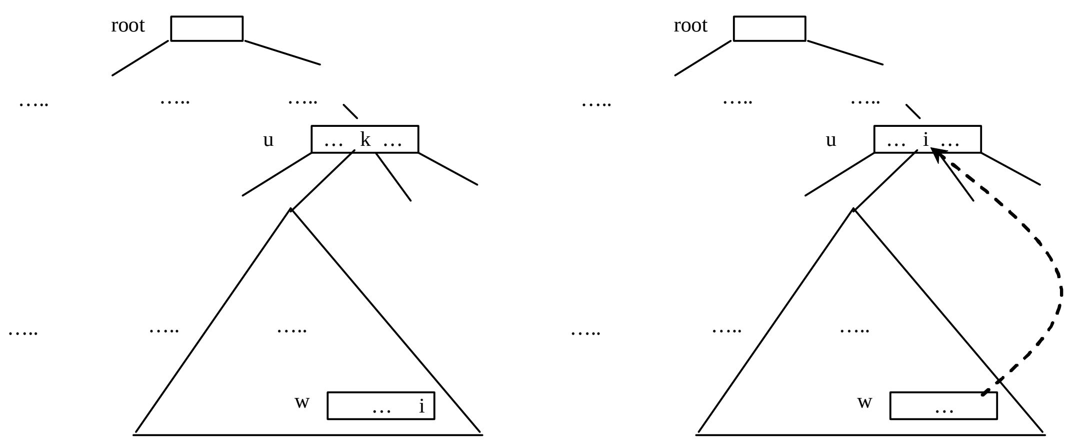
Figure 9.20

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

(a) before a fusion is performed (b) After a fusion is performed

Figure 9.21
The fusion operation moves key i from the parentOfu to w and moves all keys in u to w.

Case 2: u is a non-leaf node. Find the rightmost leaf node in the left subtree of k. Let this node be w, as shown in Figure 9.22(a). Move the last key in w to replace k in u, as shown in Figure 9.22(b). If w becomes underflow, apply a transfer or fusion operation on w.



(a) key is in u (b) Replace key k with key i

Figure 9.22
A key in the internal node is replaced by an element in a leaf node.

<Side Remark: B-tree performance>

The performance of a B-tree is dependent on number of disk IOs (i.e., the number of nodes accessed). The number of nodes accessed for search, insertion, and deletion operations is dependent 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 ten 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 9. If you use an AVL tree, the maximum number of nodes visited would be $\log_2 10{,}000{,}000 \approx 24$.

Key Terms

******PD: Please place terms in two columns same as in intro6e.***

- 2-3-4 tree
- 2-4 tree
- 2-node
- 3-node
- 4-node
- B-Tree
- fusion operation
- split operation
- transfer operation

Chapter Summary

- A 2-4 tree is a complete balanced search tree. In a 2-4 tree, a node may have one, two, or three elements.
- Searching an element in a 2-4 tree is similar to searching an element in a binary tree. The difference is that you have search an element within a node.
- 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-node or 3-node, simply insert the element into the node. If the node is a 4-node, split the node.
- The process of deleting an element from a 2-4 tree is similar to deleting an element from a binary tree. The difference is that you have to perform transfer or fusion operations for empty nodes.
- The height of an 2-4 tree is O(logn). So, the time complexity for the search, insert, and delete methods are O(logn).
- 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 $d-1$ number of 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 the data in the database systems where a large amount of data is stored on disks.

Review Questions

Sections 9.1-9.2

9.1
What is a 2-4 tree? What is a 2-node, 3-node, and 4-node?

9.2
Describe the data fields in the Tree24 class and the data fields in the Tree24Node class.

9.3
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?

Sections 9.3-9.5

9.4
How do you search an element in a 2-4 tree?

9.5
How do you insert an element into a 2-4 tree?

9.6
How do you delete an element from a 2-4 tree?

9.7
Show the change of a 2-4 tree when inserting 1, 2, 3, 4, 10, 9, 7, 5, 8, 6 into the tree, in this order.

9.8
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, 6 from the tree in this order.

9.9
Show the change of a B-Tree 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, 27 into the tree, in this order.

9.10
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, 6 from the tree in this order.

Programming Exercises

9.1*
(*Implementing inorder*) The inorder method in Tree24 is left as exercise. Implement it.

9.2
(*Implementing postorder*) The postorder method in Tree24 is left as exercise. Implement it.

9.3
(*Implementing iterator*) The iterator method in Tree24 is left as exercise. Implement it to iterate the elements using inorder.

9.4*
(*Displaying a 2-4 tree graphically*) Write an applet that displays a 2-4 tree.

9.5***
(*2-4 tree animation*) Write a Java applet that animates the 2-4 tree insert, delete, and search methods, as shown in Figure 9.4.

9.6**
(*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.

9.7***
(*The BTree class*) Design and implement a class for B-trees.