Advanced Data Structures

Advanced Programming

ICOM 4015

Lecture 18

Reading: Java Concepts Chapter 21

Chapter Goals

- To learn about the set and map data types
- To understand the implementation of hash tables
- To be able to program hash functions
- To learn about binary trees
- To be able to use tree sets and tree maps

Continued

Chapter Goals

- To become familiar with the heap data structure
- To learn how to implement the priority queue data type
- To understand how to use heaps for sorting

Sets

- Set: unordered collection of distinct elements
- Elements can be added, located, and removed
- Sets don't have duplicates

A Set of Printers

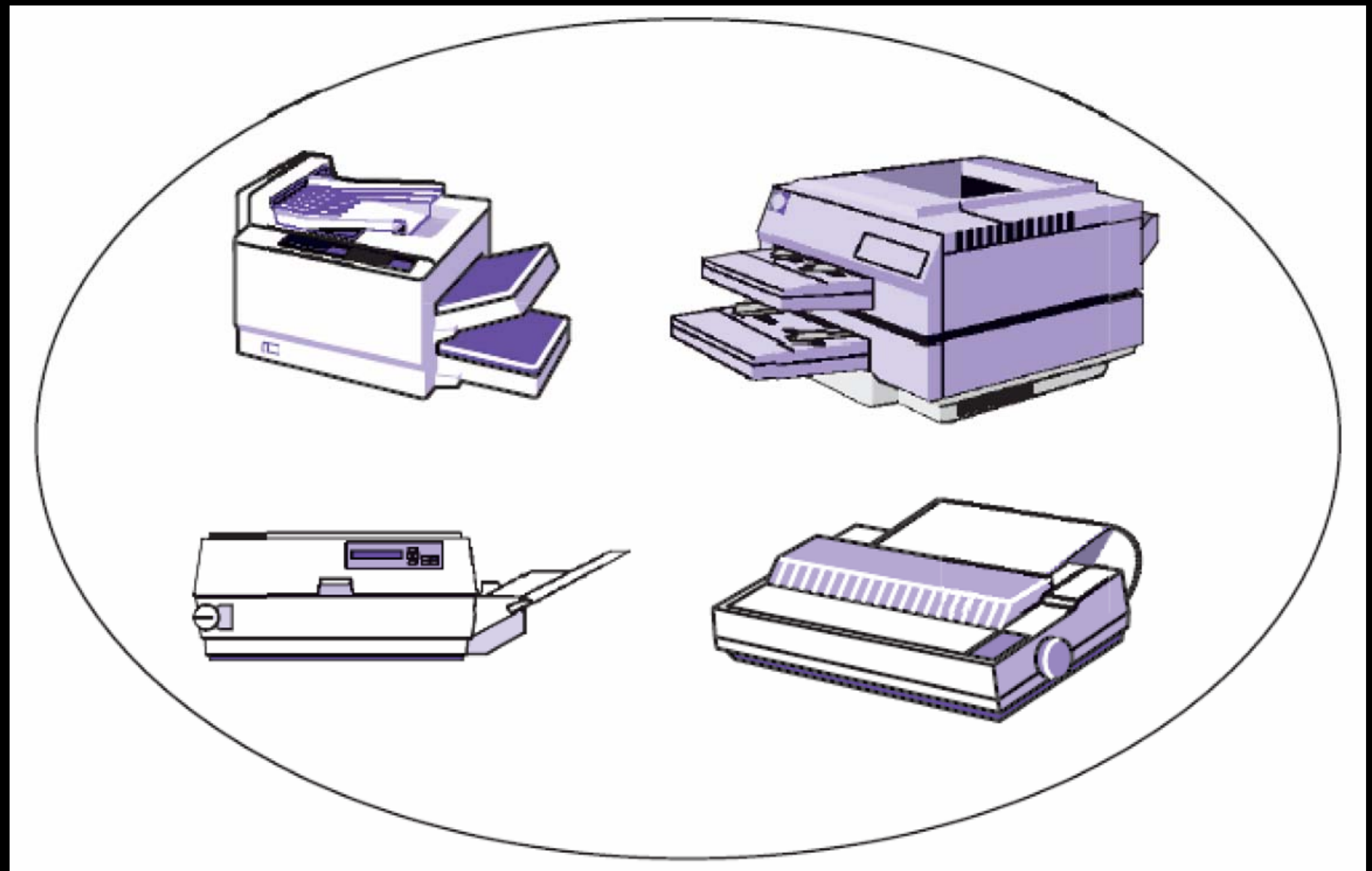

Figure 1:
A Set of Printers

Fundamental Operations on a Set

- **Adding an element**
  - Adding an element has no effect if the element is already in the set
- **Removing an element**
  - Attempting to remove an element that isn't in the set is silently ignored
- **Containment testing (does the set contain a given object?)**
- **Listing all elements (in arbitrary order)**

Sets

- **We could use a linked list to implement a set**
  - Adding, removing, and containment testing would be relatively slow
- **There are data structures that can handle these operations much more quickly**
  - Hash tables
  - Trees

Continued

Sets

- **Standard Java library provides set implementations based on both data structures**
  - HashSet
  - TreeSet
- **Both of these data structures implement the `Set` interface**

Set Classes and Interface in the Standard Library

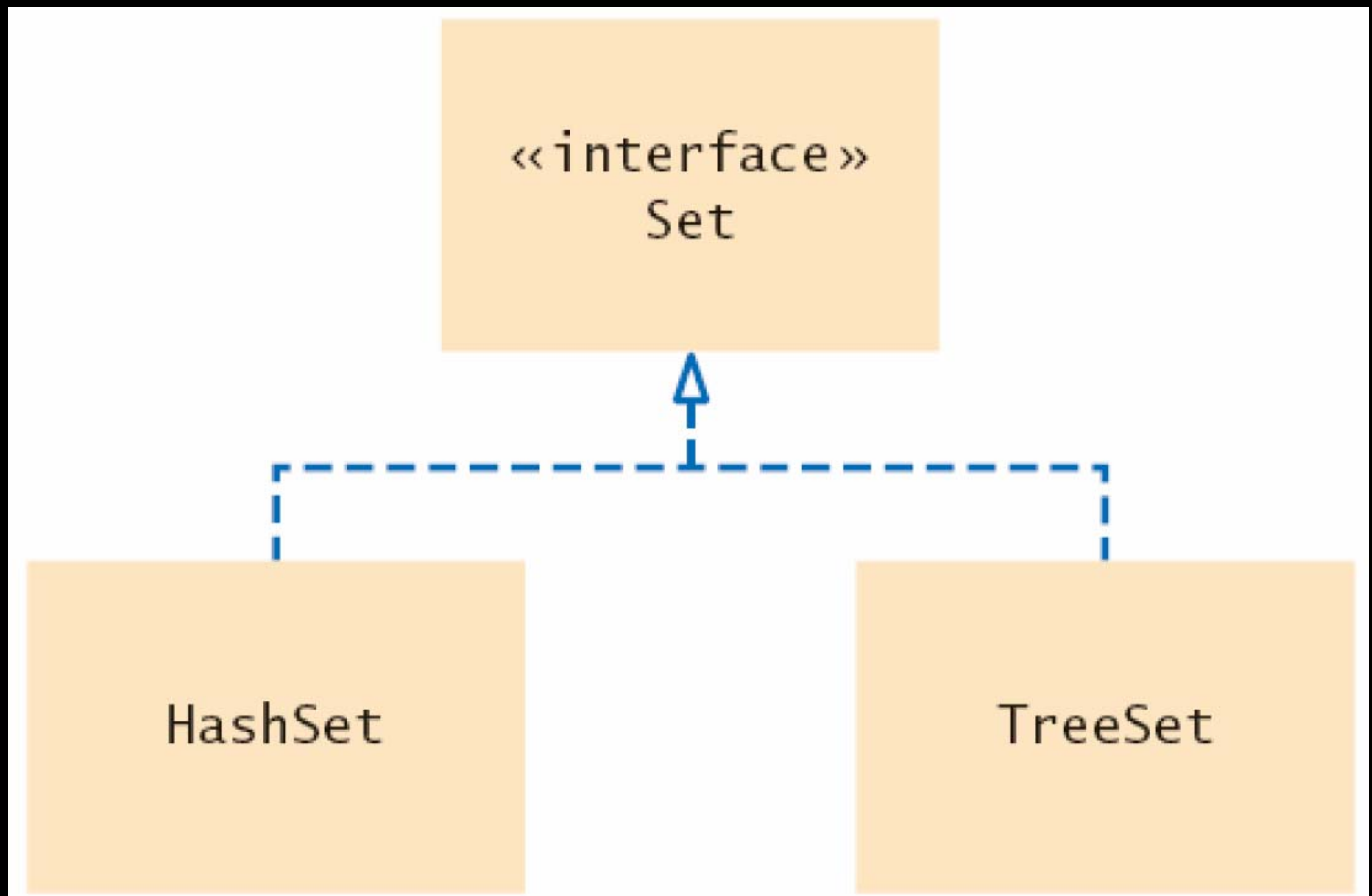

Figure 2:
Set Classes and Interfaces in the Standard Library

Iterator

- Use an iterator to visit all elements in a set
- A set iterator does not visit the elements in the order in which they were inserted
- An element can not be added to a set at an iterator position
- A set element can be removed at an iterator position

Code for Creating and Using a Hash Set

```
//Creating a hash set
Set<String> names = new HashSet<String>();
```

```
//Adding an element names.add("Romeo");
```

```
//Removing an element names.remove("Juliet");
```

```
//Is element in set
if (names.contains("Juliet") { . . .}
```

Listing All Elements with an Iterator

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   Do something with name
}


// Or, using the "for each" loop for (String name : names)
{
   Do something with name
}
```

File `SetTester.java`

```
01: import java.util.HashSet;
02: import java.util.Iterator;
03: import java.util.Scanner;
04: import java.util.Set;
05:
06:
07: /**
08:    This program demonstrates a set of strings. The user
09:    can add and remove strings.
10: */
11: public class SetTester
12: {
13:    public static void main(String[] args)
14:    {
15:       Set<String> names = new HashSet<String>();
16:       Scanner in = new Scanner(System.in);
17:
```

Continued

File SetTester.java

```
18:         boolean done = false;
19:         while (!done)
20:         {
21:            System.out.print("Add name, Q when done: ");
22:            String input = in.next();
23:            if (input.equalsIgnoreCase("Q"))
24:               done = true;
25:            else
26:            {
27:               names.add(input);
28:               print(names);
29:            }
30:         }
31:
32:         done = false;
33:         while (!done)
34:         {
```

Continued

File SetTester.java

```
35:           System.out.println("Remove name, Q when done");
36:           String input = in.next();
37:           if (input.equalsIgnoreCase("Q"))
38:              done = true;
39:           else
40:           {
41:              names.remove(input);
42:              print(names);
43:           }
44:        }
45:     }
46:
47:     /**
48:        Prints the contents of a set of strings.
49:        @param s a set of strings
50:     */
51:     private static void print(Set<String> s)
52:     {
```

Continued

File `SetTester.java`

```
53:         System.out.print("{ ");
54:         for (String element : s)
55:         {
56:            System.out.print(element);
57:            System.out.print(" ");
58:         }
59:         System.out.println("}");
60:      }
61:   }
62:
63:
```

Continued

File `SetTester.java`

- **Output**

```
Add name, Q when done: Dick
{ Dick }
Add name, Q when done: Tom
{ Tom Dick }
Add name, Q when done: Harry
{ Harry Tom Dick }
Add name, Q when done: Tom
{ Harry Tom Dick }
Add name, Q when done: Q
Remove name, Q when done: Tom
{ Harry Dick }
Remove name, Q when done: Jerry
{ Harry Dick }
Remove name, Q when done: Q
```

Self Test

1. Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?
2. Why are set iterators different from list iterators?

Answers

1. Efficient set implementations can quickly test whether a given element is a member of the set.
2. Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backwards.

Maps

- A map keeps associations between key and value objects
- Mathematically speaking, a map is a function from one set, the *key set*, to another set, the *value set*
- Every key in a map has a unique value
- A value may be associated with several keys
- Classes that implement the `Map` interface
  - HashMap
  - TreeMap

An Example of a Map

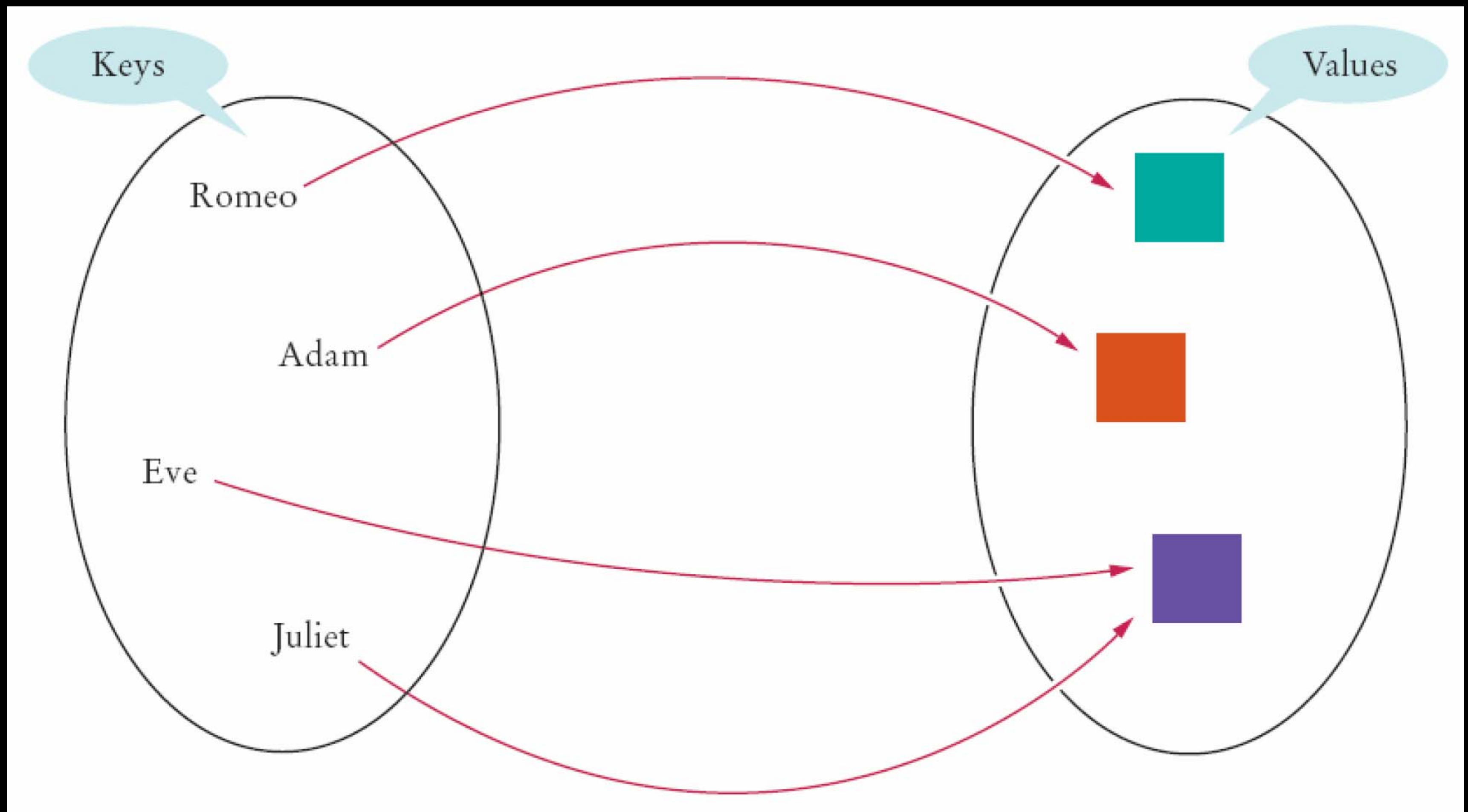

Figure 3:
An Example of a Map

Map Classes and Interfaces

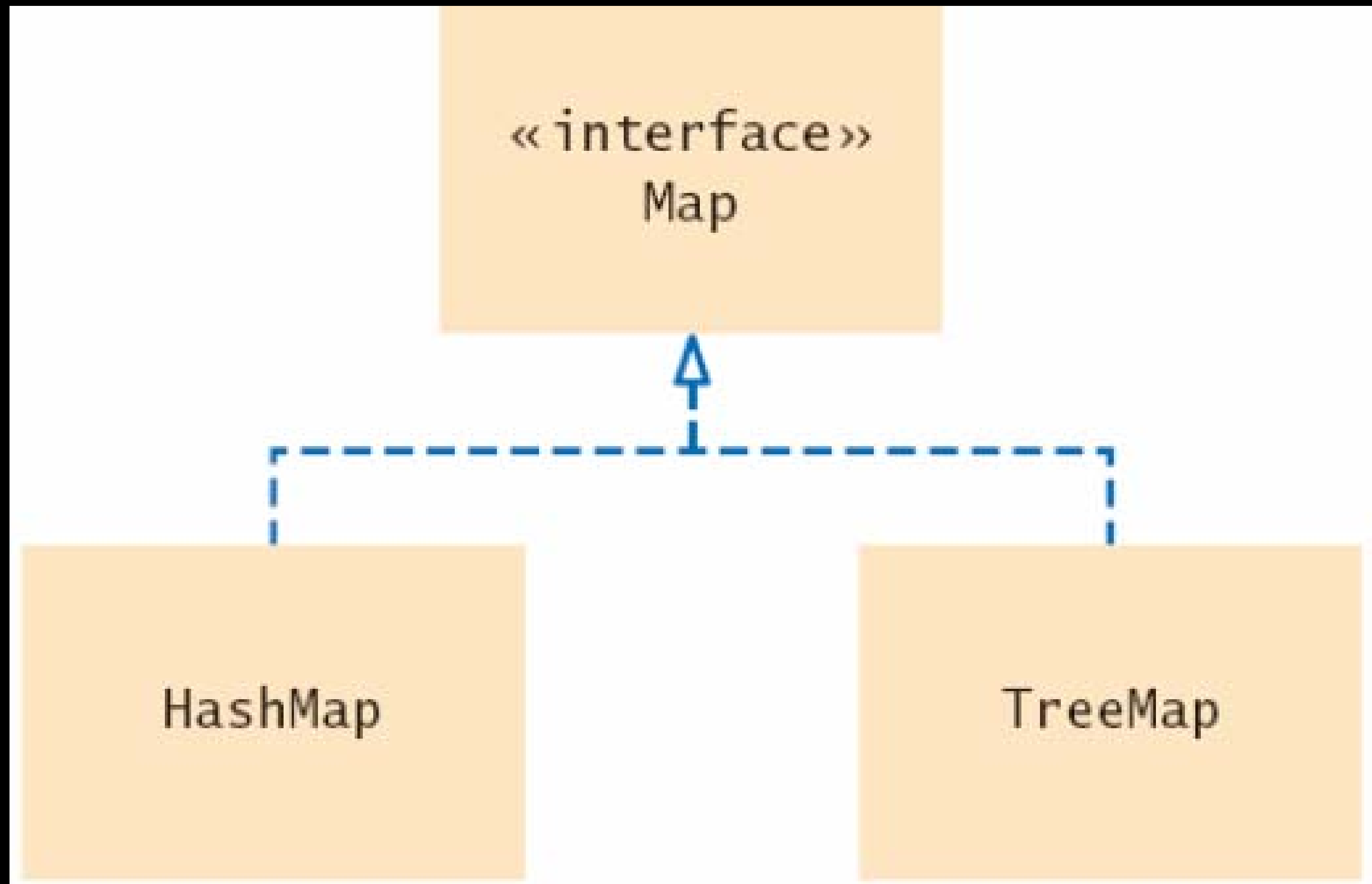

Figure 4:
Map Classes and Interfaces in the Standard Library

Code for Creating and Using a HashMap

- //Changing an existing association
  favoriteColor.put("Juliet",Color.RED);
- //Removing a key and its associated value
  favoriteColors.remove("Juliet");

Code for Creating and Using a `HashMap`

- ```
  //Creating a HashMap
  Map<String, Color> favoriteColors
     = new HashMap<String, Color>();
  ```

- ```
  //Adding an association
  favoriteColors.put("Juliet", Color.PINK);
  ```

- ```
  //Changing an existing association
  favoriteColor.put("Juliet",Color.RED);
  ```

Continued
  ```

Code for Creating and Using a `HashMap`

- ```
  //Getting the value associated with a key
  Color julietsFavoriteColor
     = favoriteColors.get("Juliet");
  ```

- ```
  //Removing a key and its associated value
  favoriteColors.remove("Juliet");
  ```

Printing Key/Value Pairs

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + "->" + value);
}
```

File `MapTester.java`

```
01: import java.awt.Color;
02: import java.util.HashMap;
03: import java.util.Iterator;
04: import java.util.Map;
05: import java.util.Set;
06:
07: /**
08:    This program tests a map that maps names to colors.
09: */
10: public class MapTester
11: {
12:    public static void main(String[] args)
13:    {
14:       Map<String, Color> favoriteColors
15:             = new HashMap<String, Color>();
16:       favoriteColors.put("Juliet", Color.pink);
17:       favoriteColors.put("Romeo", Color.green);
```

Continued

File `MapTester.java`

```
18:         favoriteColors.put("Adam", Color.blue);
19:         favoriteColors.put("Eve", Color.pink);
20:
21:         Set<String> keySet = favoriteColors.keySet();
22:         for (String key : keySet)
23:         {
24:            Color value = favoriteColors.get(key);
25:            System.out.println(key + "->" + value);
26:         }
27:      }
28: }
```

Continued

File `MapTester.java`

- **Output**

```
Romeo->java.awt.Color[r=0,g=255,b=0]
Eve->java.awt.Color[r=255,g=175,b=175]
Adam->java.awt.Color[r=0,g=0,b=255]
Juliet->java.awt.Color[r=255,g=175,b=175]
```

Self Check

1. What is the difference between a set and a map?
2. Why is the collection of the keys of a map a set?

Answers

1. A set stores elements. A map stores associations between keys and values.
2. The ordering does not matter, and you cannot have duplicates.

Hash Tables

- Hashing can be used to find elements in a data structure quickly without making a linear search
- A *hash table* can be used to implement sets and maps
- A *hash function* computes an integer value (called the *hash code*) from an object

Continued

Hash Tables

- **A good hash function minimizes *collisions*– identical hash codes for different objects**
- **To compute the hash code of object `x`:**

```
int h = x.hashCode();
```

Sample Strings and Their Hash Codes

String	Hash Code
"Adam"	2035631
"Eve"	70068
"Harry"	69496448
"Jim"	74478
"Joe"	74676
"Juliet"	–2065036585
"Katherine"	2079199209
"Sue"	83491

Simplistic Implementation of a Hash Table

- **To implement**
  - Generate hash codes for objects
  - Make an array
  - Insert each object at the location of its hash code
- **To test if an object is contained in the set**
  - Compute its hash code
  - Check if the array position with that hash code is already occupied

Simplistic Implementation of a Hash Table

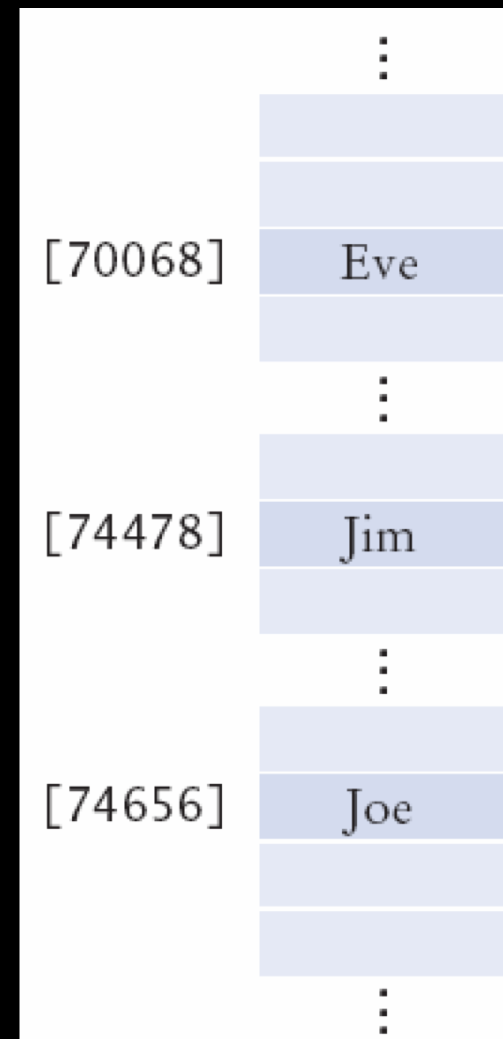

Figure 5:
A Simplistic Implementation of a Hash Table

Problems with Simplistic Implementation

- It is not possible to allocate an array that is large enough to hold all possible integer index positions
- It is possible for two different objects to have the same hash code

Solutions

- **Pick a reasonable array size and reduce the hash codes to fall inside the array**

```
int h = x.hashCode();
if (h < 0) h = -h;
h = h % size;
```

- **When elements have the same hash code:**
  - Use a node sequence to store multiple objects in the same array position
  - These node sequences are called *buckets*

Hash Table with Buckets to Store Elements with Same Hash Code

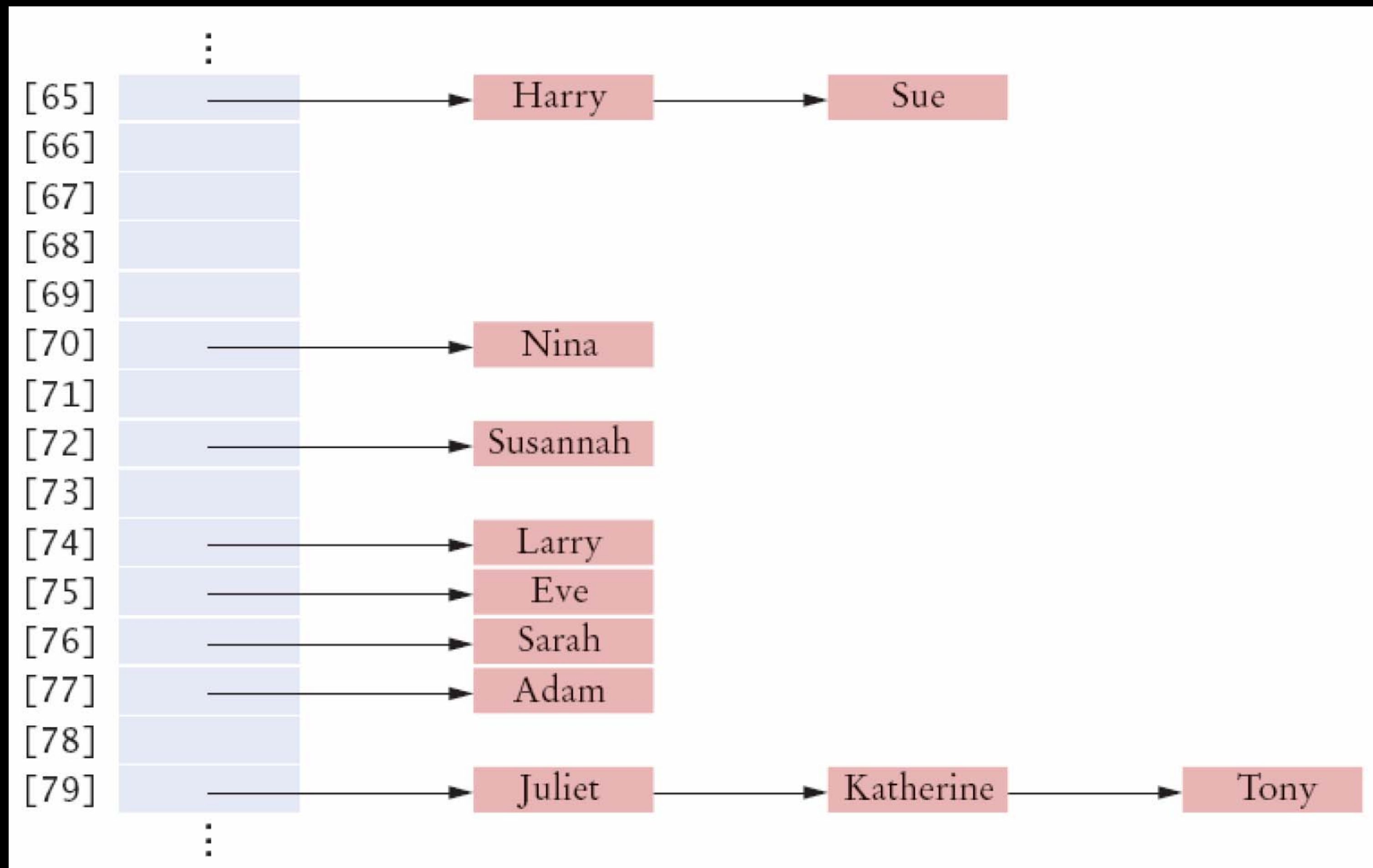

Figure 6:
A Hash Table with Buckets to Store Elements with Same Hash Code

Algorithm for Finding an Object `x` in a Hash Table

- **Get the index `h` into the hash table**
  - Compute the hash code
  - Reduce it modulo the table size
- **Iterate through the elements of the bucket at position `h`**
  - For each element of the bucket, check whether it is equal to x
- **If a match is found among the elements of that bucket, then `x` is in the set**
  - Otherwise, `x` is not in the set

Hash Tables

- **A hash table can be implemented as an array of buckets**
- **Buckets are sequences of nodes that hold elements with the same hash code**
- **If there are few collisions, then adding, locating, and removing hash table elements takes constant time**
  - Big-Oh notation: $O(1)$

Continued

Hash Tables

- **For this algorithm to be effective, the bucket sizes must be small**
- **The table size should be a prime number larger than the expected number of elements**
  - An excess capacity of 30% is typically recommended

Hash Tables

- **Adding an element: simple extension of the algorithm for finding an object**
  - Compute the hash code to locate the bucket in which the element should be inserted
  - Try finding the object in that bucket
  - If it is already present, do nothing; otherwise, insert it

Continued

Hash Tables

- **Removing an element is equally simple**
  - Compute the hash code to locate the bucket in which the element should be inserted
  - Try finding the object in that bucket
  - If it is present, remove it; otherwise, do nothing
- **If there are few collisions, adding or removing takes *O*(1) time**

File `HashSet.java`

```
001: import java.util.AbstractSet;
002: import java.util.Iterator;
003: import java.util.NoSuchElementException;
004:
005: /**
006:    A hash set stores an unordered collection of objects, using
007:    a hash table.
008: */
009: public class HashSet extends AbstractSet
010: {
011:    /**
012:       Constructs a hash table.
013:       @param bucketsLength the length of the buckets array
014:    */
015:    public HashSet(int bucketsLength)
016:    {
```

Continued

File `HashSet.java`

```
017:          buckets = new Node[bucketsLength];
018:          size = 0;
019:       }
020:
021:       /**
022:          Tests for set membership.
023:          @param x an object
024:          @return true if x is an element of this set
025:       */
026:       public boolean contains(Object x)
027:       {
028:          int h = x.hashCode();
029:          if (h < 0) h = -h;
030:          h = h % buckets.length;
031:
032:          Node current = buckets[h];
033:          while (current != null)
034:          {
```

Continued

File `HashSet.java`

```
035:            if (current.data.equals(x)) return true;
036:            current = current.next;
037:         }
038:         return false;
039:      }
040:
041:      /**
042:         Adds an element to this set.
043:         @param x an object
044:         @return true if x is a new object, false if x was
045:         already in the set
046:      */
047:      public boolean add(Object x)
048:      {
049:         int h = x.hashCode();
050:         if (h < 0) h = -h;
051:         h = h % buckets.length;
052:
```

Continued

File HashSet.java

```
053:         Node current = buckets[h];
054:         while (current != null)
055:         {
056:            if (current.data.equals(x))
057:               return false; // Already in the set
058:            current = current.next;
059:         }
060:         Node newNode = new Node();
061:         newNode.data = x;
062:         newNode.next = buckets[h];
063:         buckets[h] = newNode;
064:         size++;
065:         return true;
066:      }
067:
```

Continued

File HashSet.java

```
068:     /**
069:        Removes an object from this set.
070:        @param x an object
071:        @return true if x was removed from this set, false
072:        if x was not an element of this set
073:     */
074:     public boolean remove(Object x)
075:     {
076:        int h = x.hashCode();
077:        if (h < 0) h = -h;
078:        h = h % buckets.length;
079:
080:        Node current = buckets[h];
081:        Node previous = null;
082:        while (current != null)
083:        {
084:           if (current.data.equals(x))
085:           {
```

Continued

File HashSet.java

```
086:                if (previous == null) buckets[h] = current.next;
087:                else previous.next = current.next;
088:                size--;
089:                return true;
090:             }
091:             previous = current;
092:             current = current.next;
093:          }
094:          return false;
095:       }
096:
097:       /**
098:          Returns an iterator that traverses the elements
             of this set.
099:          @param a hash set iterator
100:       */
101:       public Iterator iterator()
102:       {
103:          return new HashSetIterator();
104:       }
```

Continued

File HashSet.java

```
105:
106:     /**
107:        Gets the number of elements in this set.
108:        @return the number of elements
109:     */
110:     public int size()
111:     {
112:        return size;
113:     }
114:
115:     private Node[] buckets;
116:     private int size;
117:
118:     private class Node
119:     {
120:        public Object data;
121:        public Node next;
122:     }
123:
```

Continued

File HashSet.java

```
124:    private class HashSetIterator implements Iterator
125:    {
126:       /**
127:          Constructs a hash set iterator that points to the
128:          first element of the hash set.
129:       */
130:       public HashSetIterator()
131:       {
132:          current = null;
133:          bucket = -1;
134:          previous = null;
135:          previousBucket = -1;
136:       }
137:
138:       public boolean hasNext()
139:       {
140:          if (current != null && current.next != null)
141:             return true;
```

Continued

File HashSet.java

```
142:            for (int b = bucket + 1; b < buckets.length; b++)
143:               if (buckets[b] != null) return true;
144:            return false;
145:         }
146:
147:         public Object next()
148:         {
149:            previous = current;
150:            previousBucket = bucket;
151:            if (current == null || current.next == null)
152:            {
153:               // Move to next bucket
154:               bucket++;
155:
156:               while (bucket < buckets.length
157:                     && buckets[bucket] == null)
158:                  bucket++;
```

Continued

File HashSet.java

```
159:                 if (bucket < buckets.length)
160:                    current = buckets[bucket];
161:                 else
162:                    throw new NoSuchElementException();
163:              }
164:              else // Move to next element in bucket
165:                 current = current.next;
166:              return current.data;
167:           }
168:
169:           public void remove()
170:           {
171:              if (previous != null && previous.next == current)
172:                 previous.next = current.next;
173:              else if (previousBucket < bucket)
174:                 buckets[bucket] = current.next;
175:              else
176:                 throw new IllegalStateException();
```

Continued

File `HashSet.java`

```
177:               current = previous;
178:               bucket = previousBucket;
179:           }
180:
181:           private int bucket;
182:           private Node current;
183:           private int previousBucket;
184:           private Node previous;
185:       }
186: }
```

File SetTester.java

```
01: import java.util.Iterator;
02: import java.util.Set;
03:
04: /**
05:    This program tests the hash set class.
06: */
07: public class SetTester
08: {
09:    public static void main(String[] args)
10:    {
11:       HashSet names = new HashSet(101); // 101 is a prime
12:
13:       names.add("Sue");
14:       names.add("Harry");
15:       names.add("Nina");
16:       names.add("Susannah");
17:       names.add("Larry");
18:       names.add("Eve");
```

Continued

File SetTester.java

```
19:         names.add("Sarah");
20:         names.add("Adam");
21:         names.add("Tony");
22:         names.add("Katherine");
23:         names.add("Juliet");
24:         names.add("Romeo");
25:         names.remove("Romeo");
26:         names.remove("George");
27: 
28:         Iterator iter = names.iterator();
29:         while (iter.hasNext())
30:            System.out.println(iter.next());
31:      }
32: }
```

Continued

File `SetTester.java`

- **Output**

```
Harry
Sue
Nina
Susannah
Larry
Eve
Sarah
Adam
Juliet
Katherine
Tony
```

Self Check

1. If a hash function returns 0 for all values, will the `HashSet` work correctly?
2. What does the `hasNext` method of the `HashSetIterator` do when it has reached the end of a bucket?

Answers

1. Yes, the hash set will work correctly. All elements will be inserted into a single bucket.
2. It locates the next bucket in the bucket array and points to its first element.

Computing Hash Codes

- A hash function computes an integer hash code from an object
- Choose a hash function so that different objects are likely to have different hash codes.

Continued

Computing Hash Codes

- **Bad choice for hash function for a string**
  - Adding the unicode values of the characters in the string

```
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = h + s.charAt(i);
```

  - Because permutations ("eat" and "tea") would have the same hash code

Computing Hash Codes

- Hash function for a string `s` from standard library

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = HASH_MULTIPLIER * h + s.charAt(i)
```

- For example, the hash code of "eat" is

```
31 * (31 * 'e' + 'a') + 't' = 100184
```

- The hash code of "tea" is quite different, namely

```
31 * (31 * 't' + 'e') + 'a' = 114704
```

A `hashCode` Method for the `Coin` Class

- **There are two instance fields: `String` coin name and double coin value**
- **Use `String's hashCode` method to get a hash code for the name**
- **To compute a hash code for a floating-point number:**
  - Wrap the number into a `Double` object
  - Then use `Double's hashCode` method
- **Combine the two hash codes using a prime number as the `HASH_MULTIPLIER`**

A `hashCode` Method for the `Coin` Class

```
class Coin
{
   public int hashCode()
   {
      int h1 = name.hashCode();
      int h2 = new Double(value).hashCode();
      final int HASH_MULTIPLIER = 29;
      int h = HASH_MULTIPLIER * h1 + h2: return h
   }
   . . .
}
```

Creating Hash Codes for your Classes

- Use a prime number as the **`HASH_MULTIPLIER`**
- Compute the hash codes of each instance field
- For an integer instance field just use the field value
- Combine the hash codes

```
int h = HASH_MULTIPLIER * h1 +h2;
h = HASH_MULTIPLIER * h + h3;
h = HASH_MULTIPLIER *h + h4;
 . . .
return h;
```

Creating Hash Codes for your Classes

- Your `hashCode` method must be compatible with the `equals` method
  - if `x.equals(y)` then `x.hashCode() == y.hashCode()`

Continued

Creating Hash Codes for your Classes

- **You get into trouble if your `class` defines an `equals` method but not a `hashCode` method**
  - If we forget to define `hashCode` method for `Coin` it inherits the method from `Object` superclass
  - That method computes a hash code from the memory location of the object

Creating Hash Codes for your Classes

- Effect: any two objects are very likely to have a different hash code

```
Coin coin1 = new Coin(0.25, "quarter");
Coin coin2 = new Coin(0.25, "quarter");
```

- **In general, define either both `hashCode` and `equals` methods or neither**

Hash Maps

- In a hash map, only the keys are hashed
- The keys need compatible `hashCode` and `equals` method

File Coin.java

```
01: /**
02:    A coin with a monetary value.
03: */
04: public class Coin
05: {
06:    /**
07:       Constructs a coin.
08:       @param aValue the monetary value of the coin.
09:       @param aName the name of the coin
10:    */
11:    public Coin(double aValue, String aName)
12:    {
13:       value = aValue;
14:       name = aName;
15:    }
16:
```

Continued

File Coin.java

```
17:     /**
18:        Gets the coin value.
19:        @return the value
20:     */
21:     public double getValue()
22:     {
23:        return value;
24:     }
25: 
26:     /**
27:        Gets the coin name.
28:        @return the name
29:     */
30:     public String getName()
31:     {
32:        return name;
33:     }
34: 
```

Continued

File Coin.java

```
35:     public boolean equals(Object otherObject)
36:     {
37:        if (otherObject == null) return false;
38:        if (getClass() != otherObject.getClass()) return false;
39:        Coin other = (Coin) otherObject;
40:        return value == other.value && name.equals(other.name);
41:     }
42:
43:     public int hashCode()
44:     {
45:        int h1 = name.hashCode();
46:        int h2 = new Double(value).hashCode();
47:        final int HASH_MULTIPLIER = 29;
48:        int h = HASH_MULTIPLIER * h1 + h2;
49:        return h;
50:     }
51:
```

Continued

File Coin.java

```
52:     public String toString()
53:     {
54:        return "Coin[value=" + value + ",name=" + name + "]";
55:     }
56:
57:     private double value;
58:     private String name;
59: }
```

File HashCodeTester.java

```
01: import java.util.HashSet;
02: import java.util.Iterator;
03: import java.util.Set;
04:
05: /**
06:    A program to test hash codes of coins.
07: */
08: public class HashCodeTester
09: {
10:    public static void main(String[] args)
11:    {
12:       Coin coin1 = new Coin(0.25, "quarter");
13:       Coin coin2 = new Coin(0.25, "quarter");
14:       Coin coin3 = new Coin(0.05, "nickel");
15:
```

Continued

File HashCodeTester.java

```
16:         System.out.println("hash code of coin1="
17:               + coin1.hashCode());
18:         System.out.println("hash code of coin2="
19:               + coin2.hashCode());
20:         System.out.println("hash code of coin3="
21:               + coin3.hashCode());
22:
23:         Set<Coin> coins = new HashSet<Coin>();
24:         coins.add(coin1);
25:         coins.add(coin2);
26:         coins.add(coin3);
27:
28:         for (Coin c : coins)
29:            System.out.println(c);
30:      }
31: }
```

Continued

File `HashCodeTester.java`

- **Output**

```
hash code of coin1=-1513525892
hash code of coin2=-1513525892
hash code of coin3=-1768365211
Coin[value=0.25,name=quarter]
Coin[value=0.05,name=nickel]
```

Self Check

1. What is the hash code of the string "to"?
2. What is the hash code of new `Integer(13)`?

Answers

1. $31 \times 116 + 111 = 3707$
2. 13.

Binary Search Trees

- Binary search trees allow for fast insertion and removal of elements
- They are specially designed for fast searching
- A binary tree consists of two nodes, each of which has two child nodes

Continued

Binary Search Trees

- **All nodes in a binary search tree fulfill the property that:**
  - Descendants to the left have smaller data values than the node data value
  - Descendants to the right have larger data values than the node data value

A Binary Search Tree

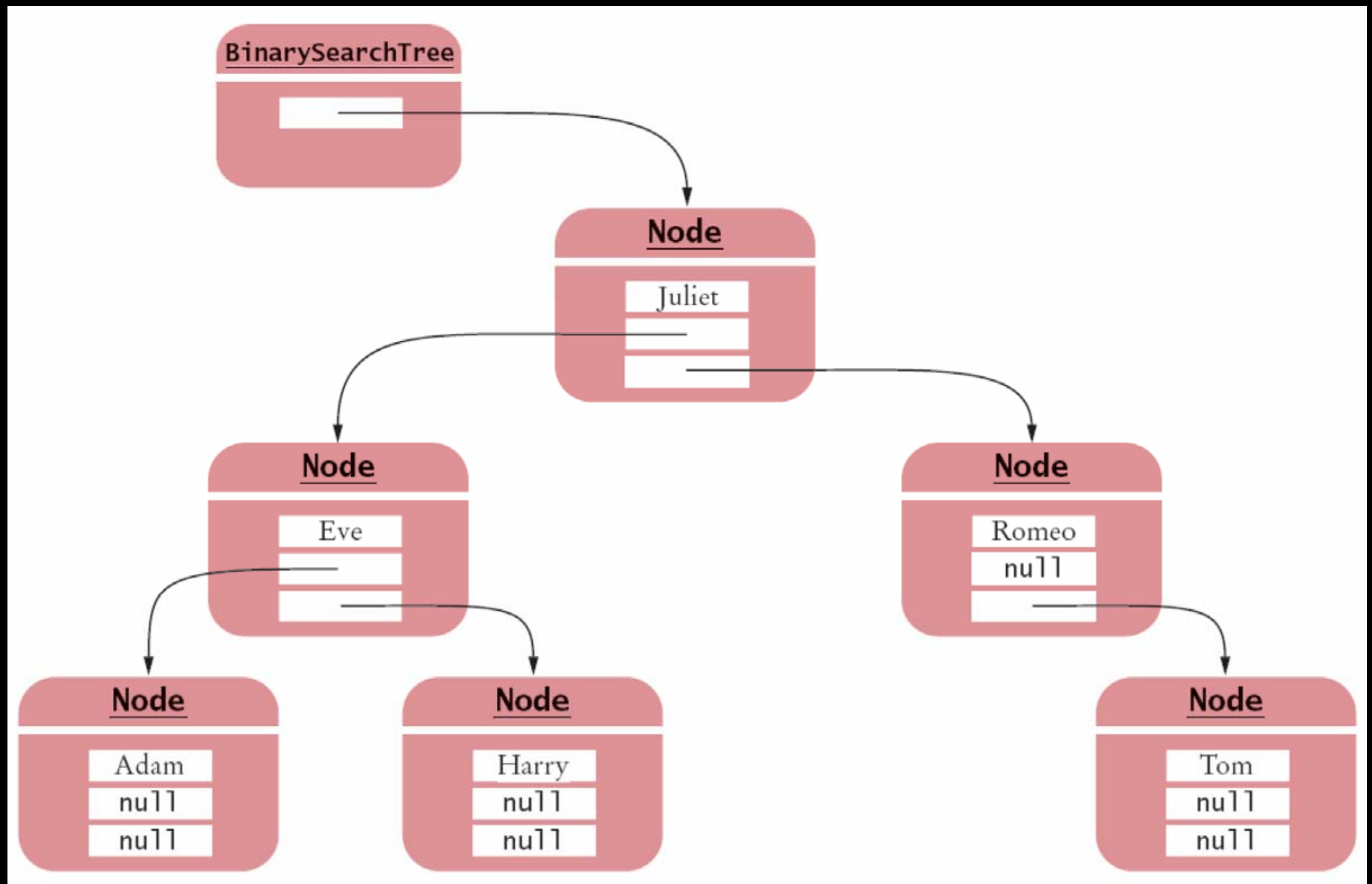

Figure 7:
A Binary Search Tree

A Binary Tree That Is Not a Binary Search Tree

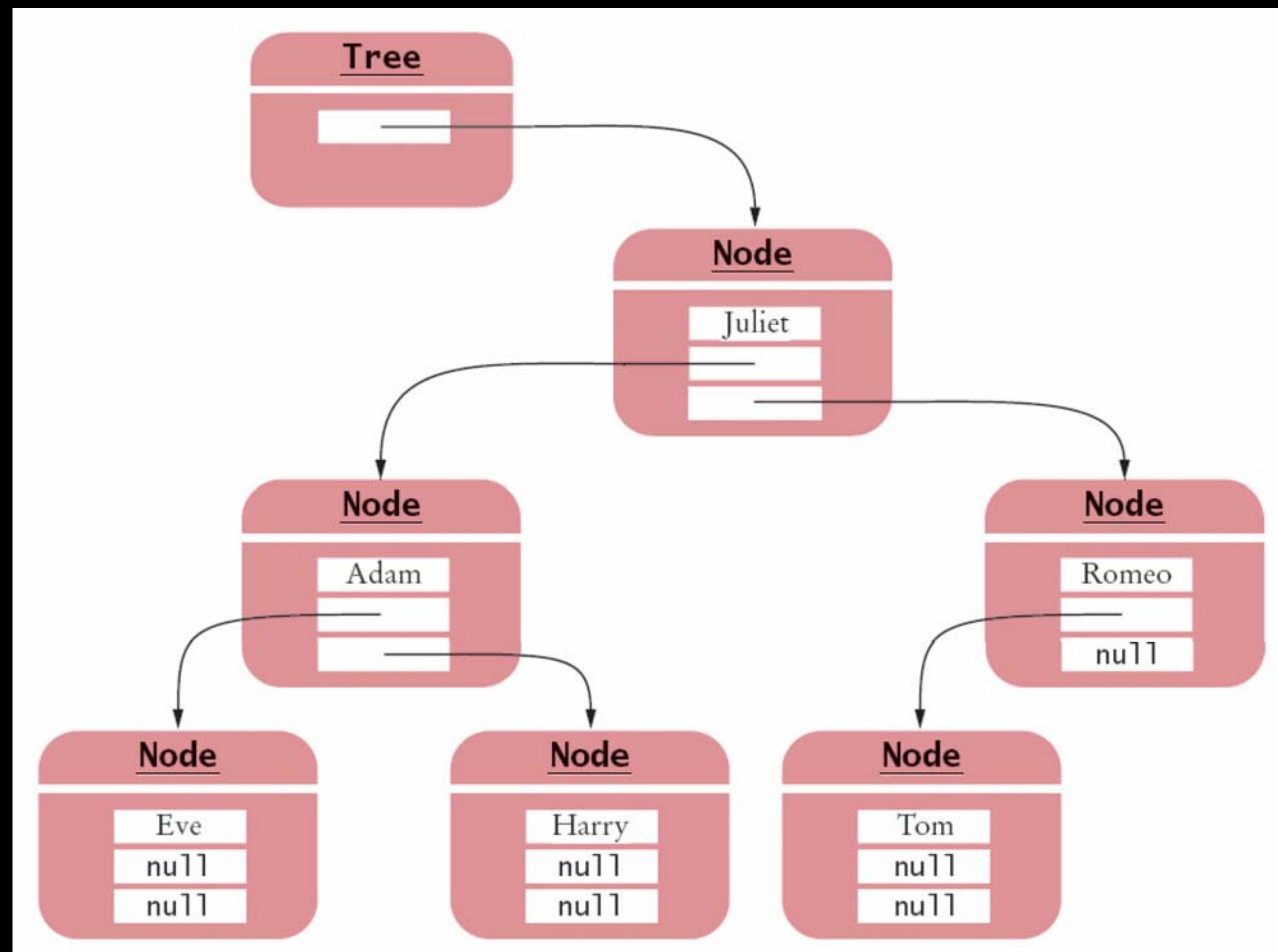

Figure 8:
A Binary Tree That Is Not a Binary Search Tree

Implementing a Binary Search Tree

- **Implement a class for the tree containing a reference to the root node**
- **Implement a class for the nodes**
  - A node contains two references (to left and right child nodes)
  - A node contains a data field
  - The data field has type `Comparable`, so that you can compare the values in order to place them in the correct position in the binary search tree

Implementing a Binary Search Tree

```
public class BinarySearchTree
{
   public BinarySearchTree() {. . .}
   public void add(Comparable obj) {. . .}
   . . .
   private Node root;
   private class Node
   {
      public void addNode(Node newNode) { . . . }
      . . .
      public Comparable data;
      public Node left;
      public Node right;
   }
}
```

Insertion Algorithm

- **If you encounter a non-`null` node reference, look at its `data` value**
  - If the `data` value of that node is larger than the one you want to insert,
    continue the process with the left subtree
  - If the existing `data` value is smaller,
    continue the process with the right subtree
- **If you encounter a null node pointer, replace it with the new node**

Example

```
BinarySearchTree tree = new BinarySearchTree();
```

```
tree.add("Juliet");
```

```
tree.add("Tom");
```

```
tree.add("Dick");
```

```
tree.add("Harry");
```

Example

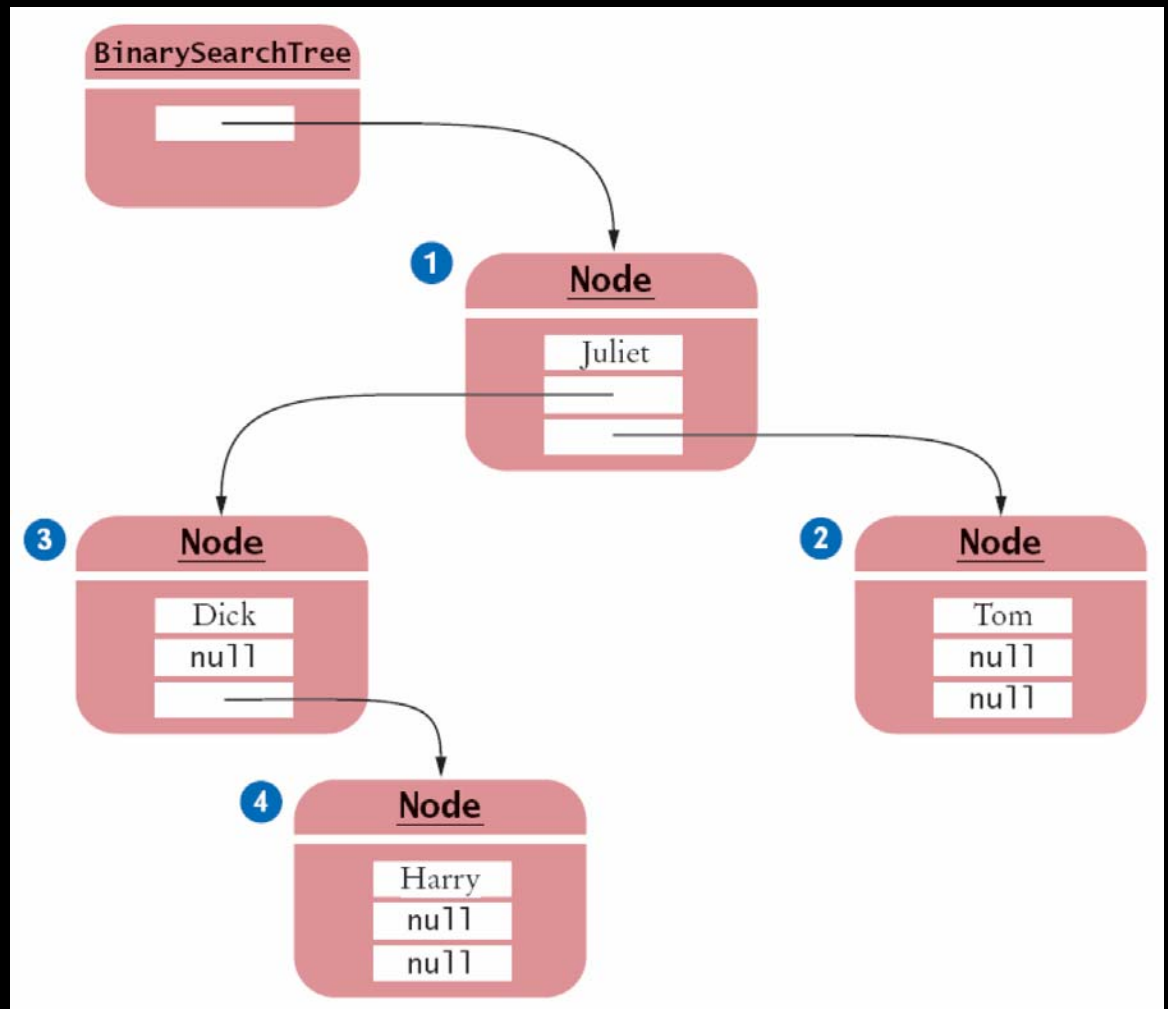

Figure 9:
Binary Search Trees After Four Insertions

Example Continued

`Tree: Add Romeo` 5

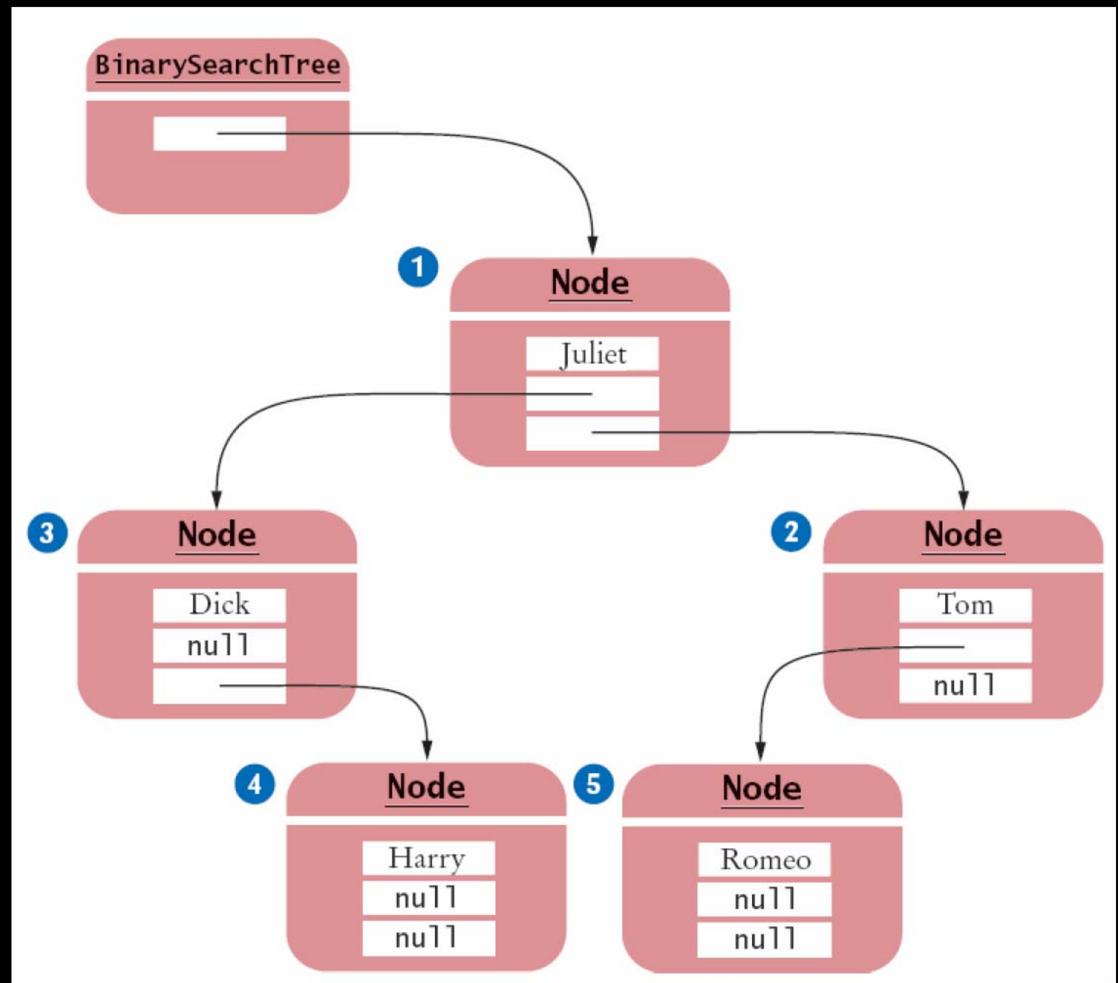

Figure 10:
Binary Search Trees After Five Insertions

Insertion Algorithm: BinarySearchTree Class

```
public class BinarySearchTree
{
   . . .
   public void add(Comparable obj)
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.left = null;
      newNode.right = null;
      if (root == null) root = newNode;
      else root.addNode(newNode);
   }
   . . .
}
```

Insertion Algorithm: Node Class

```
private class Node
{
   . . .
   public void addNode(Node newNode)
   {
      int comp = newNode.data.compareTo(data);
      if (comp < 0)
      {
         if (left == null) left = newNode;
         else left.addNode(newNode);
      }
      else if (comp > 0)
      {
         if (right == null) right = newNode;
         else right.addNode(newNode);
      }
   }
   . . .
}
```

Binary Search Trees

- When removing a node with only one child, the child replaces the node to be removed
- When removing a node with two children, replace it with the smallest node of the right subtree

Removing a Node with One Child

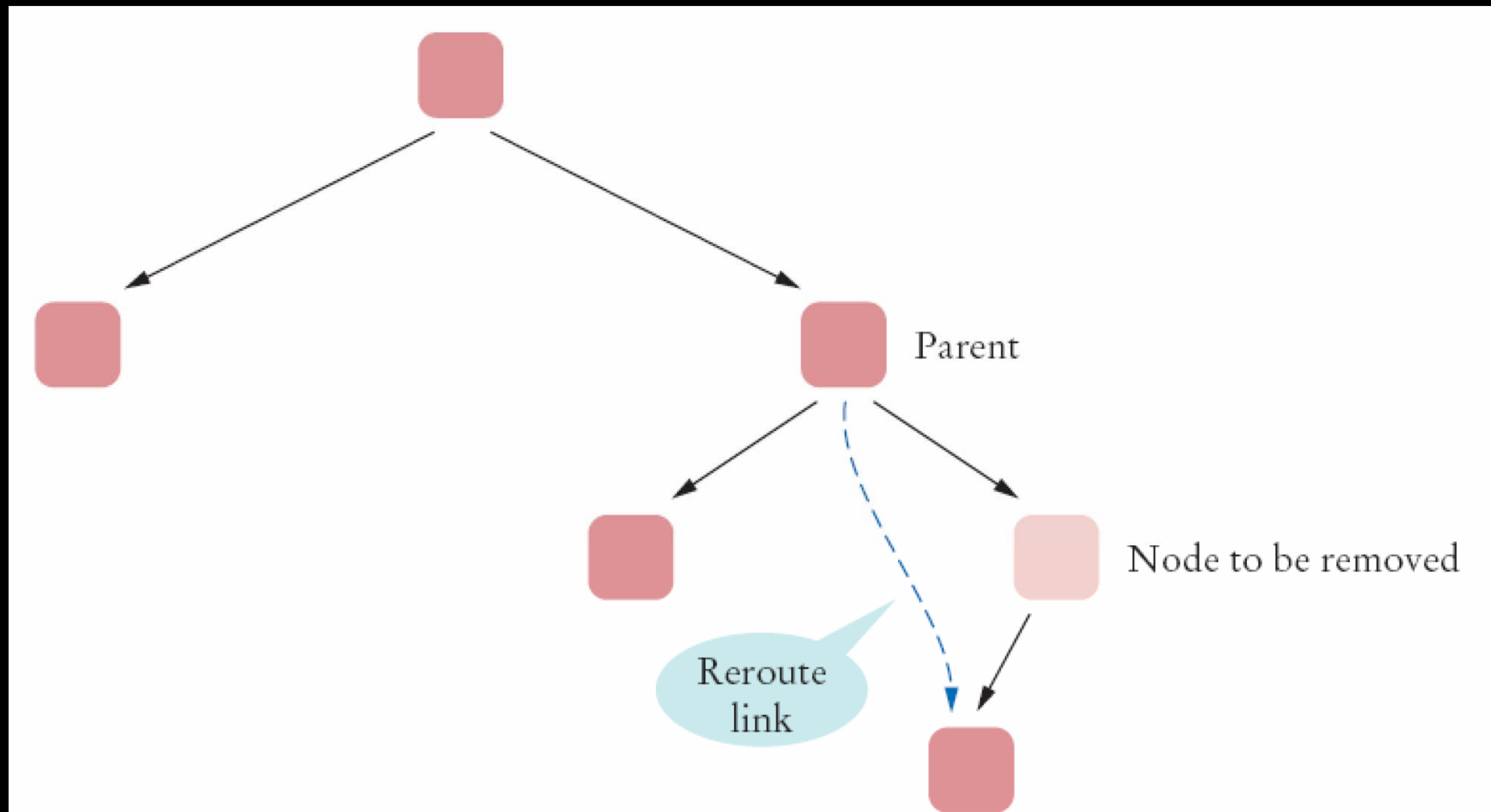

Figure 11:
Removing a Node with One Child

Removing a Node with Two Children

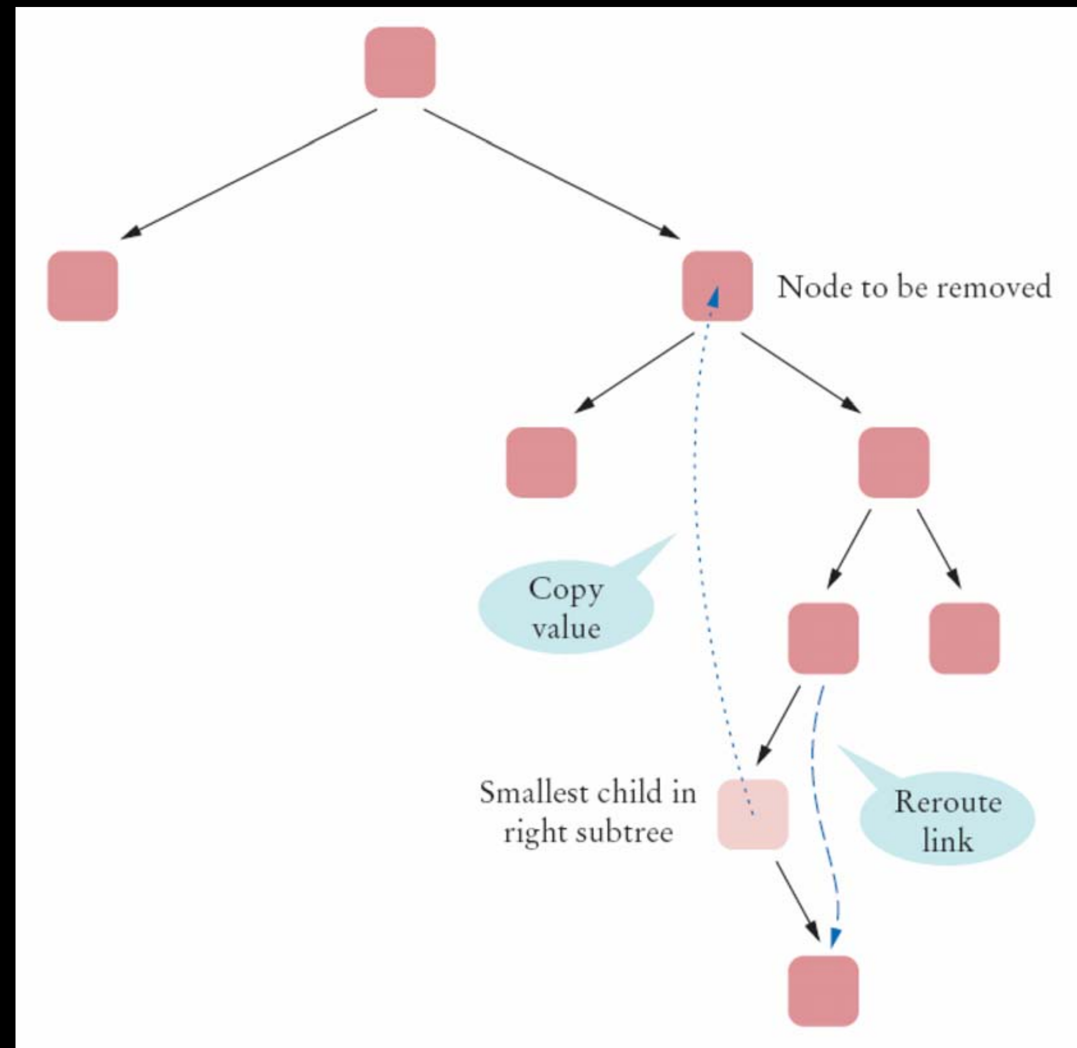

Figure 12: Removing a Node with Two Children

Binary Search Trees

- **Balanced tree: each node has approximately as many descendants on the left as on the right**
- **If a binary search tree is balanced, then adding an element takes $O(\log(n))$ time**
- **If the tree is unbalanced, insertion can be slow**
  - Perhaps as slow as insertion into a linked list

An Unbalanced Binary Search Tree

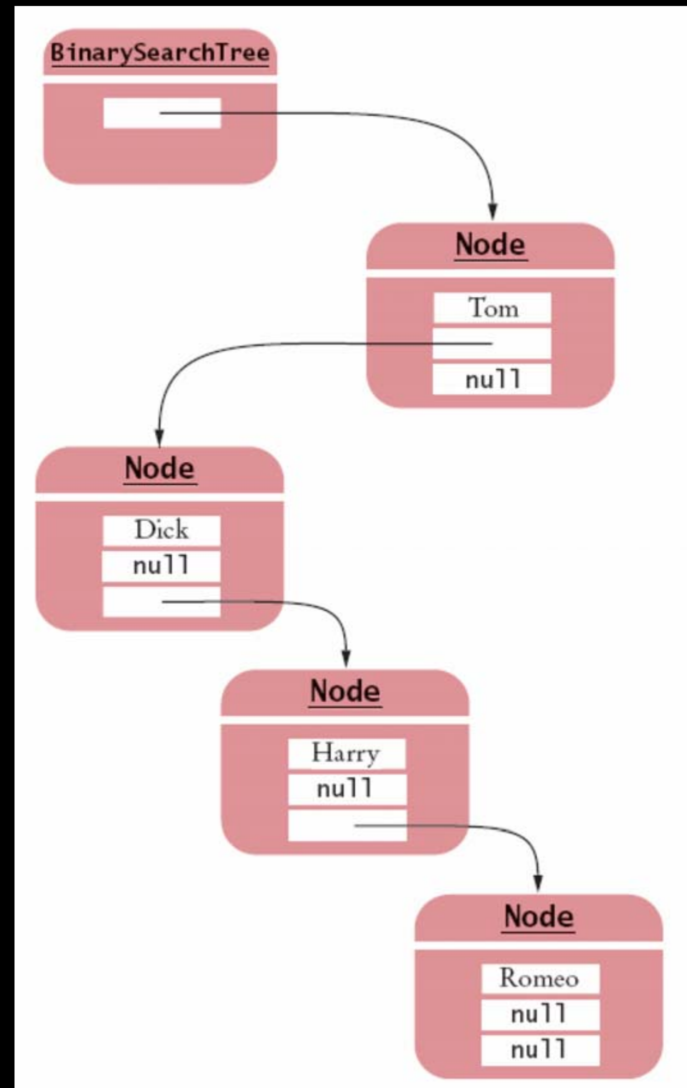

Figure 13:
An Unbalanced Binary Search Tree

File BinarySearchTree.java

```
001: /**
002:    This class implements a binary search tree whose
003:    nodes hold objects that implement the Comparable
004:    interface.
005: */
006: public class BinarySearchTree
007: {
008:    /**
009:       Constructs an empty tree.
010:    */
011:    public BinarySearchTree()
012:    {
013:       root = null;
014:    }
015:
```

Continued

File BinarySearchTree.java

```
016:     /**
017:        Inserts a new node into the tree.
018:        @param obj the object to insert
019:     */
020:     public void add(Comparable obj)
021:     {
022:        Node newNode = new Node();
023:        newNode.data = obj;
024:        newNode.left = null;
025:        newNode.right = null;
026:        if (root == null) root = newNode;
027:        else root.addNode(newNode);
028:     }
029:
```

Continued

File BinarySearchTree.java

```
030:    /**
031:       Tries to find an object in the tree.
032:       @param obj the object to find
033:       @return true if the object is contained in the tree
034:    */
035:    public boolean find(Comparable obj)
036:    {
037:       Node current = root;
038:       while (current != null)
039:       {
040:          int d = current.data.compareTo(obj);
041:          if (d == 0) return true;
042:          else if (d > 0) current = current.left;
043:          else current = current.right;
044:       }
045:       return false;
046:    }
047:
```

Continued

File BinarySearchTree.java

```
048:    /**
049:       Tries to remove an object from the tree. Does nothing
050:       if the object is not contained in the tree.
051:       @param obj the object to remove
052:    */
053:    public void remove(Comparable obj)
054:    {
055:       // Find node to be removed
056:
057:       Node toBeRemoved = root;
058:       Node parent = null;
059:       boolean found = false;
060:       while (!found && toBeRemoved != null)
061:       {
062:          int d = toBeRemoved.data.compareTo(obj);
063:          if (d == 0) found = true;
064:          else
065:          {
```

Continued

File BinarySearchTree.java

```
066:                 parent = toBeRemoved;
067:                 if (d > 0) toBeRemoved = toBeRemoved.left;
068:                 else toBeRemoved = toBeRemoved.right;
069:              }
070:           }
071:
072:           if (!found) return;
073:
074:           // toBeRemoved contains obj
075:
076:           // If one of the children is empty, use the other
077:
078:           if (toBeRemoved.left == null
                  || toBeRemoved.right == null)
079:           {
080:              Node newChild;
081:              if (toBeRemoved.left == null)
082:                 newChild = toBeRemoved.right;
```

Continued

File BinarySearchTree.java

```
083:            else
084:               newChild = toBeRemoved.left;
085:
086:            if (parent == null) // Found in root
087:               root = newChild;
088:            else if (parent.left == toBeRemoved)
089:               parent.left = newChild;
090:            else
091:               parent.right = newChild;
092:            return;
093:         }
094:
095:         // Neither subtree is empty
096:
097:         // Find smallest element of the right subtree
098:
```

Continued

File `BinarySearchTree.java`

```
099:        Node smallestParent = toBeRemoved;
100:        Node smallest = toBeRemoved.right;
101:        while (smallest.left != null)
102:        {
103:           smallestParent = smallest;
104:           smallest = smallest.left;
105:        }
106:
107:        // smallest contains smallest child in right subtree
108:
109:        // Move contents, unlink child
110:
111:        toBeRemoved.data = smallest.data;
112:        smallestParent.left = smallest.right;
113:     }
114:
```

Continued

File BinarySearchTree.java

```
115:     /**
116:        Prints the contents of the tree in sorted order.
117:     */
118:     public void print()
119:     {
120:        if (root != null)
121:           root.printNodes();
122:     }
123:
124:     private Node root;
125:
126:     /**
127:        A node of a tree stores a data item and references
128:        of the child nodes to the left and to the right.
129:     */
130:     private class Node
131:     {
```

Continued

File BinarySearchTree.java

```
132:        /**
133:           Inserts a new node as a descendant of this node.
134:           @param newNode the node to insert
135:        */
136:        public void addNode(Node newNode)
137:        {
138:           if (newNode.data.compareTo(data) < 0)
139:           {
140:              if (left == null) left = newNode;
141:              else left.addNode(newNode);
142:           }
143:           else
144:           {
145:              if (right == null) right = newNode;
146:              else right.addNode(newNode);
147:           }
148:        }
149:
```

Continued

File BinarySearchTree.java

```
150:        /**
151:           Prints this node and all of its descendants
152:           in sorted order.
153:        */
154:        public void printNodes()
155:        {
156:           if (left != null)
157:              left.printNodes();
158:           System.out.println(data);
159:           if (right != null)
160:              right.printNodes();
161:        }
162:
163:        public Comparable data;
164:        public Node left;
165:        public Node right;
166:     }
167: }
```

Continued

File `BinarySearchTree.java`

```
168:
169:
170:
```

Self Check

1. What is the difference between a tree, a binary tree, and a balanced binary tree?
2. Give an example of a string that, when inserted into the tree of Figure 10, becomes a right child of `Romeo`.

Answers

1. In a tree, each node can have any number of children. In a binary tree, a node has at most two children. In a balanced binary tree, all nodes have approximately as many descendants to the left as to the right.
2. For example, `Sarah`. Any string between `Romeo` and `Tom` will do.

Tree Traversal

- **Print the tree elements in sorted order:**
  - Print the left subtree
  - Print the data
  - Print the right subtree

Continued

Example

- **Let's try this out with the tree in Figure 10. The algorithm tells us to**
  1. Print the left subtree of `Juliet`; that is, `Dick` and descendants
  2. Print `Juliet`
  3. Print the right subtree of `Juliet`; that is, `Tom` and descendants

Continued

Example

- **How do you print the subtree starting at `Dick`?**

1. Print the left subtree of `Dick`. There is nothing to print
2. Print `Dick`
3. Print the right subtree of `Dick`, that is, `Harry`

Example

- Algorithm goes on as above
- Output:

```
Dick
Harry
Juliet
Romeo
Tom
```

- The tree is printed in sorted order

BinarySearchTree Class print Method

```
public class BinarySearchTree
{
    . . .
    public void print()
    {
        if (root != null)
            root.printNodes();
    }
    . . .
}
```

Node Class `printNodes` Method

```
private class Node
{
   . . .
   public void printNodes()
   {
      if (left != null)
         left.printNodes();
      System.out.println(data);
      if (right != null)
         right.printNodes();
   }
   . . .
}
```

Tree Traversal

- **Tree traversal schemes include**
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal

Preorder Traversal

- Visit the root
- Visit the left subtree
- Visit the right subtree

Inorder Traversal

- Visit the left subtree
- Visit the root
- Visit the right subtree

Postorder Traversal

- Visit the left subtree
- Visit the right subtree
- Visit the root

Tree Traversal

- **Postorder traversal of an expression tree yields the instructions for evaluating the expression on a stack-based calculator**

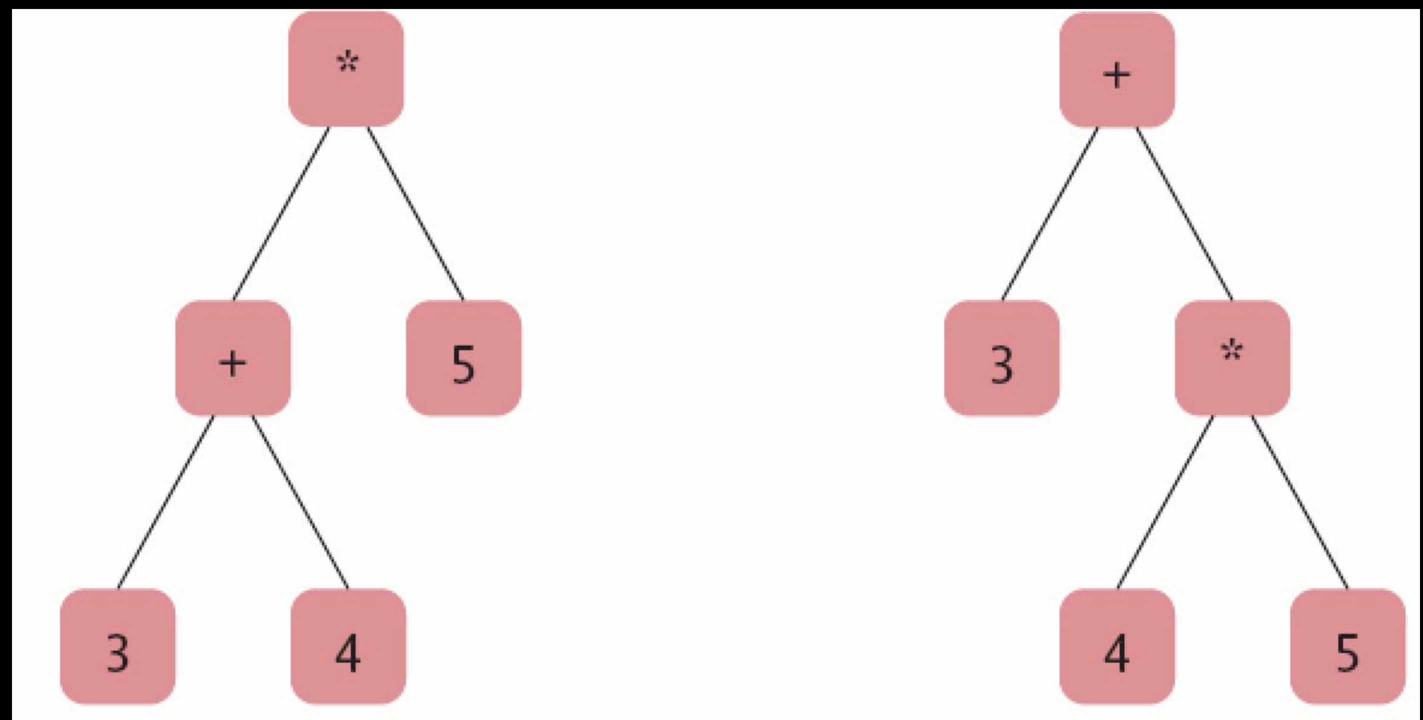

Figure 14: Expression Trees

Continued

Tree Traversal

- The first tree `((3 + 4) * 5)` yields

  `3 4 + 5 *`

- Whereas the second tree `(3 + 4 * 5)` yields

  `3 4 5 * +`

A Stack-Based Calculator

- **A number means:**
  - Push the number on the stack
- **An operator means:**
  - Pop the top two numbers off the stack
  - Apply the operator to these two numbers
  - Push the result back on the stack

A Stack-Based Calculator

- **For evaluating arithmetic expressions**
  1. Turn the expression into a tree
  2. Carry out a postorder traversal of the expression tree
  3. Apply the operations in the given order
- **The result is the value of the expression**

A Stack-Based Calculator

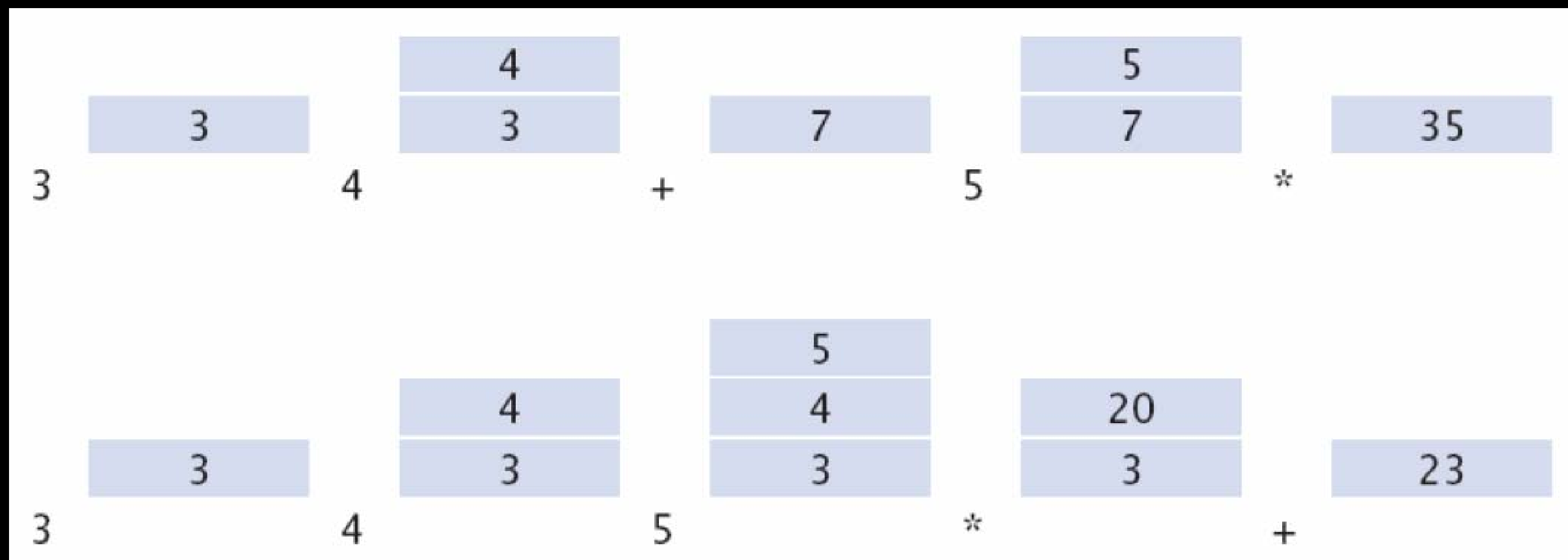

Figure 15:
A Stack-Based Calculator

Self Check

1. What are the inorder traversals of the two trees in Figure 14?
2. Are the trees in Figure 14 binary search trees?

Answers

1. For both trees, the inorder traversal is `3 + 4 * 5`.
2. No–for example, consider the children of `+`. Even without looking up the Unicode codes for `3`, `4`, and `+`, it is obvious that `+` isn't between `3` and `4`.

Reverse Polish Notation

Using Tree Sets and Tree Maps

- `HashSet` and `TreeSet` both implement the `Set` interface
- With a good hash function, hashing is generally faster than tree-based algorithms
- `TreeSet`'s balanced tree guarantees reasonable performance
- `TreeSet`'s iterator visits the elements in sorted order rather than the `HashSet`'s random order

To Use a `TreeSet`

- Either your objects must implement `Comparable` interface
- Or you must provide a `Comparator` object

To Use a `TreeMap`

- Either the keys must implement the `Comparable` interface
- Or you must provide a `Comparator` object for the keys
- There is no requirement for the values

File TreeSetTester.java

```
01: import java.util.Comparator;
02: import java.util.Iterator;
03: import java.util.Set;
04: import java.util.TreeSet;
05:
06: /**
07:    A program to test hash codes of coins.
08: */
09: public class TreeSetTester
10: {
11:    public static void main(String[] args)
12:    {
13:       Coin coin1 = new Coin(0.25, "quarter");
14:       Coin coin2 = new Coin(0.25, "quarter");
15:       Coin coin3 = new Coin(0.01, "penny");
16:       Coin coin4 = new Coin(0.05, "nickel");
17:
```

Continued

File `TreeSetTester.java`

```
18:         class CoinComparator implements Comparator<Coin>
19:         {
20:            public int compare(Coin first, Coin second)
21:            {
22:               if (first.getValue()
                        < second.getValue()) return -1;
23:               if (first.getValue()
                        == second.getValue()) return 0;
24:               return 1;
25:            }
26:         }
27:
28:         Comparator<Coin> comp = new CoinComparator();
29:         Set<Coin> coins = new TreeSet<Coin>(comp);
30:         coins.add(coin1);
31:         coins.add(coin2);
32:         coins.add(coin3);
33:         coins.add(coin4);
```

Continued

File `TreeSetTester.java`

```
34:
35:         for (Coin c : coins)
36:             System.out.println(c);
37:     }
38: }
```

File `TreeSetTester.java`

- **Output:**

```
Coin[value=0.01,name=penny]
Coin[value=0.05,name=nickel]
Coin[value=0.25,name=quarter]
```

Self Check

1. When would you choose a tree set over a hash set?
2. Suppose we define a coin comparator whose `compare` method always returns 0. Would the `TreeSet` function correctly?

Answers

1. When it is desirable to visit the set elements in sorted order.
2. No–it would never be able to tell two coins apart. Thus, it would think that all coins are duplicates of the first.

Priority Queues

- **A priority queue collects elements, each of which has a priority**
- **Example: collection of work requests, some of which may be more urgent than others**
- **When removing an element, element with highest priority is retrieved**
  - Customary to give low values to high priorities, with priority 1 denoting the highest priority

Continued

Priority Queues

- Standard Java library supplies a `PriorityQueue` class
- A data structure called *heap* is very suitable for implementing priority queues

Example

- Consider this sample code:

```
PriorityQueue<WorkOrder> q = new PriorityQueue<WorkOrder>;
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix overflowing sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

- When calling `q.remove()` for the first time, the work order with priority 1 is removed
- Next call to `q.remove()` removes the order with priority 2

Heaps

- **A *heap* (or, a *min-heap*) is a binary tree with two special properties**
  1. It is almost complete
     - All nodes are filled in, except the last level may have some nodes missing toward the right
  2. The tree fulfills the heap property
     - All nodes store values that are at most as large as the values stored in their descendants
- **Heap property ensures that the smallest element is stored in the root**

An Almost Complete Tree

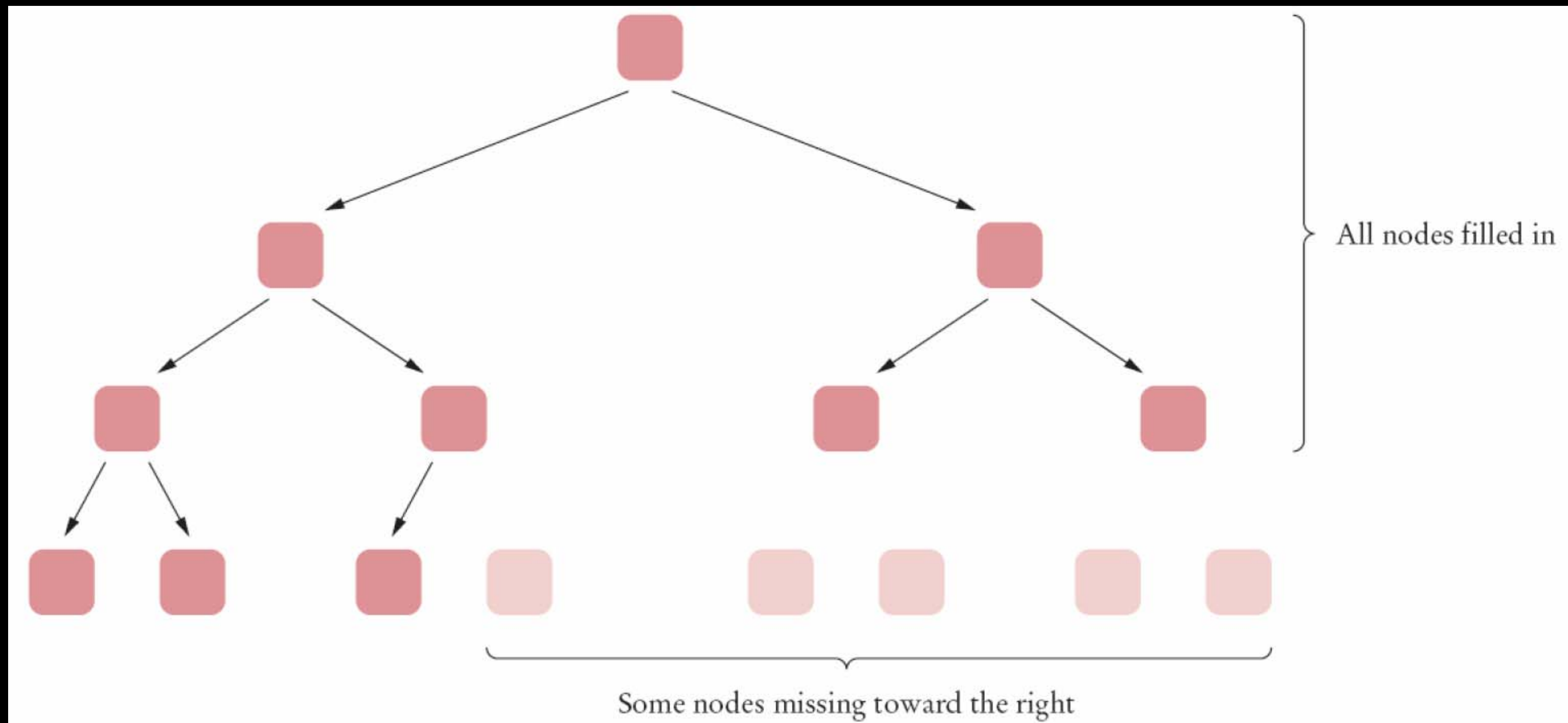

Figure 16:
An Almost Complete Tree

A Heap

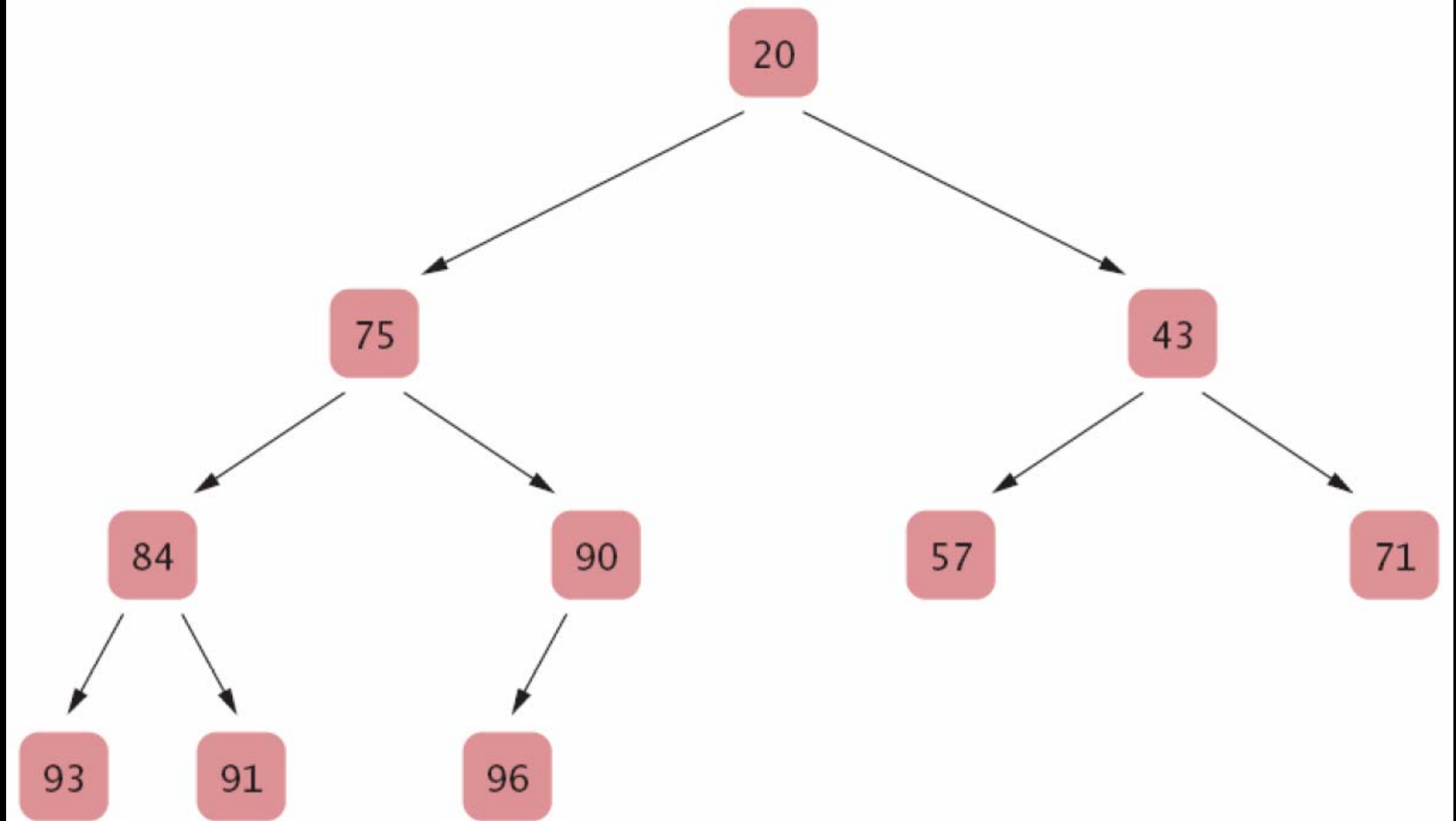

Figure 17:
A Heap

Differences of a Heap with a Binary Search Tree

1. **The shape of a heap is very regular**
   - Binary search trees can have arbitrary shapes
2. **In a heap, the left and right subtrees both store elements that are larger than the root element**
   - In a binary search tree, smaller elements are stored in the left subtree and larger elements are stored in the right subtree

Inserting a New Element in a Heap

1. Add a vacant slot to the end of the tree

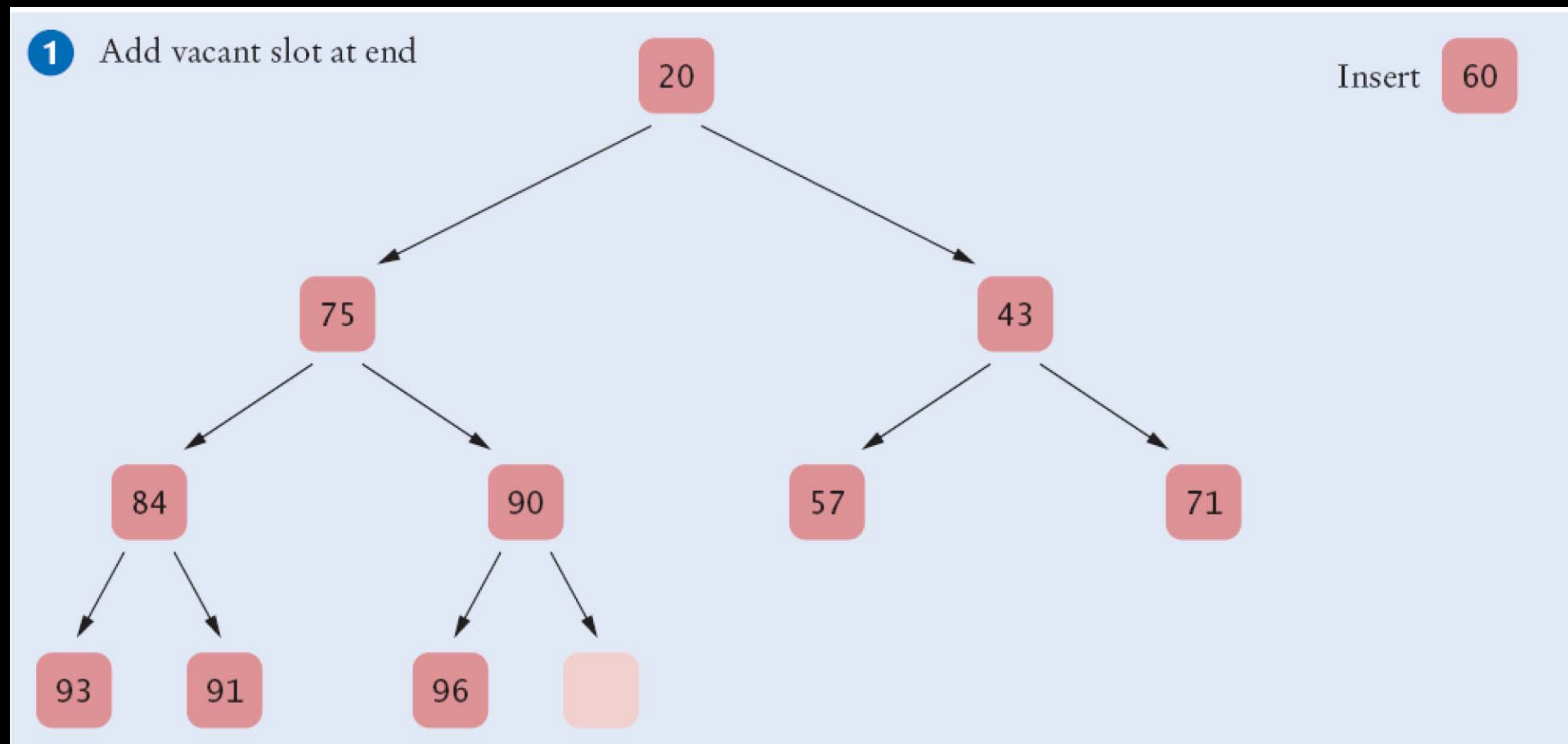

Figure 18:
Inserting a New Element in a Heap

Inserting a New Element in a Heap

1. **Demote the parent of the empty slot if it is larger than the element to be inserted**
   - Move the parent value into the vacant slot, and move the vacant slot up
   - Repeat this demotion as long as the parent of the vacant slot is larger than the element to be inserted

Continued

Inserting a New Element in a Heap

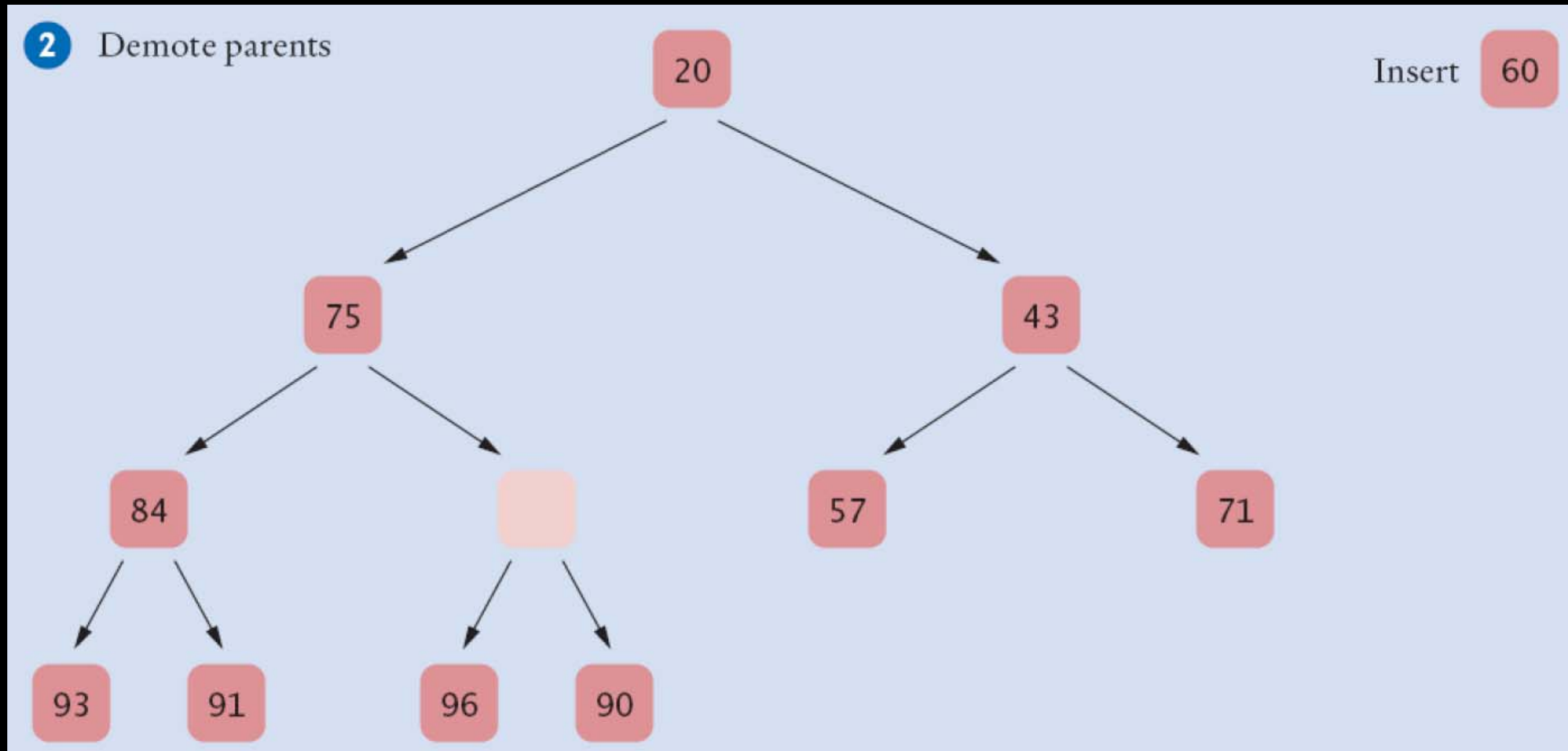

Figure 18 (continued):
Inserting a New Element in a Heap

Inserting a New Element in a Heap

1. **Demote the parent of the empty slot if it is larger than the element to be inserted**
   - Move the parent value into the vacant slot, and move the vacant slot up
   - Repeat this demotion as long as the parent of the vacant slot is larger than the element to be inserted

Continued

Inserting a New Element in a Heap

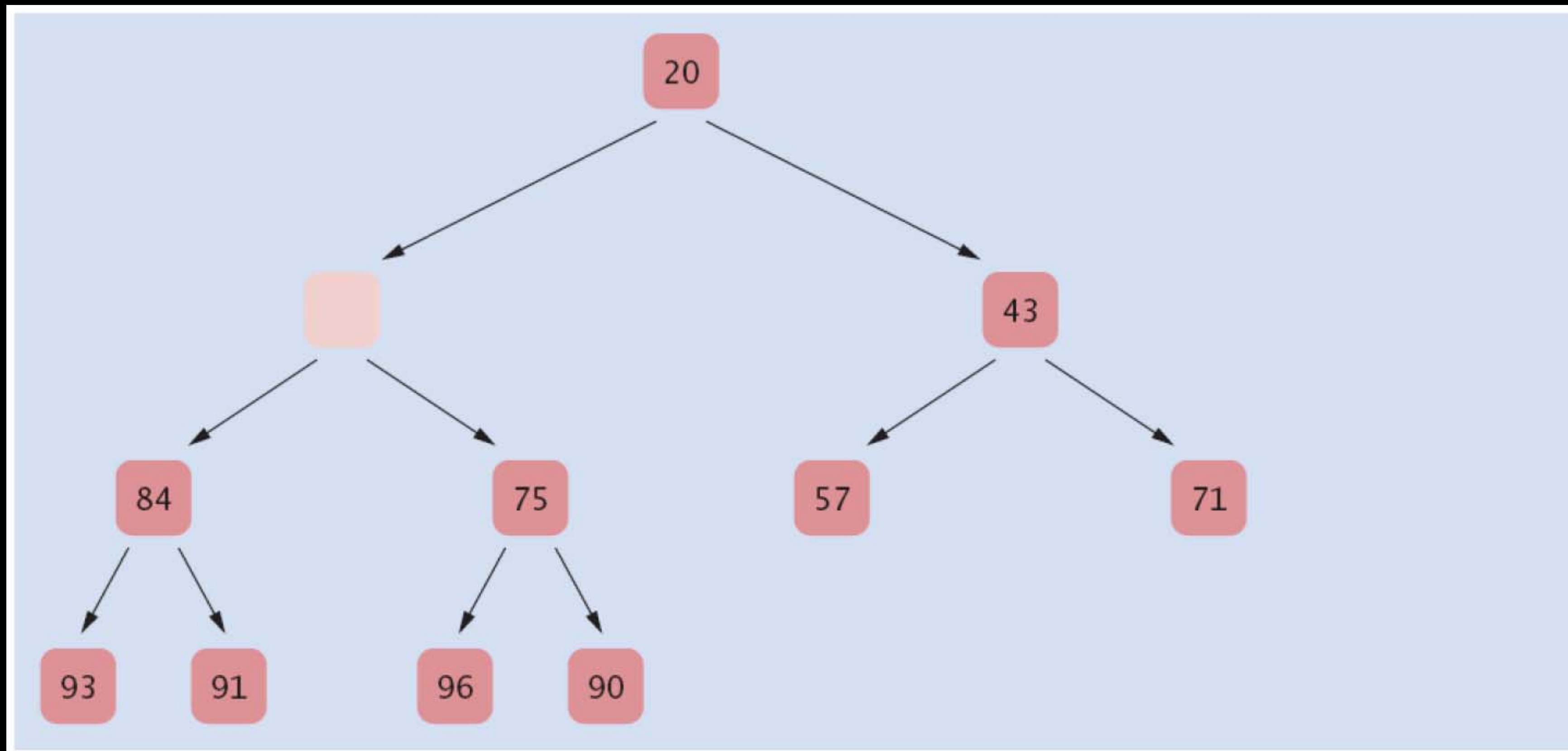

Figure 18 (continued):
Inserting a New Element in a Heap

Inserting a New Element in a Heap

1. At this point, either the vacant slot is at the root, or the parent of the vacant slot is smaller than the element to be inserted. Insert the element into the vacant slot

Continued

Inserting a New Element in a Heap

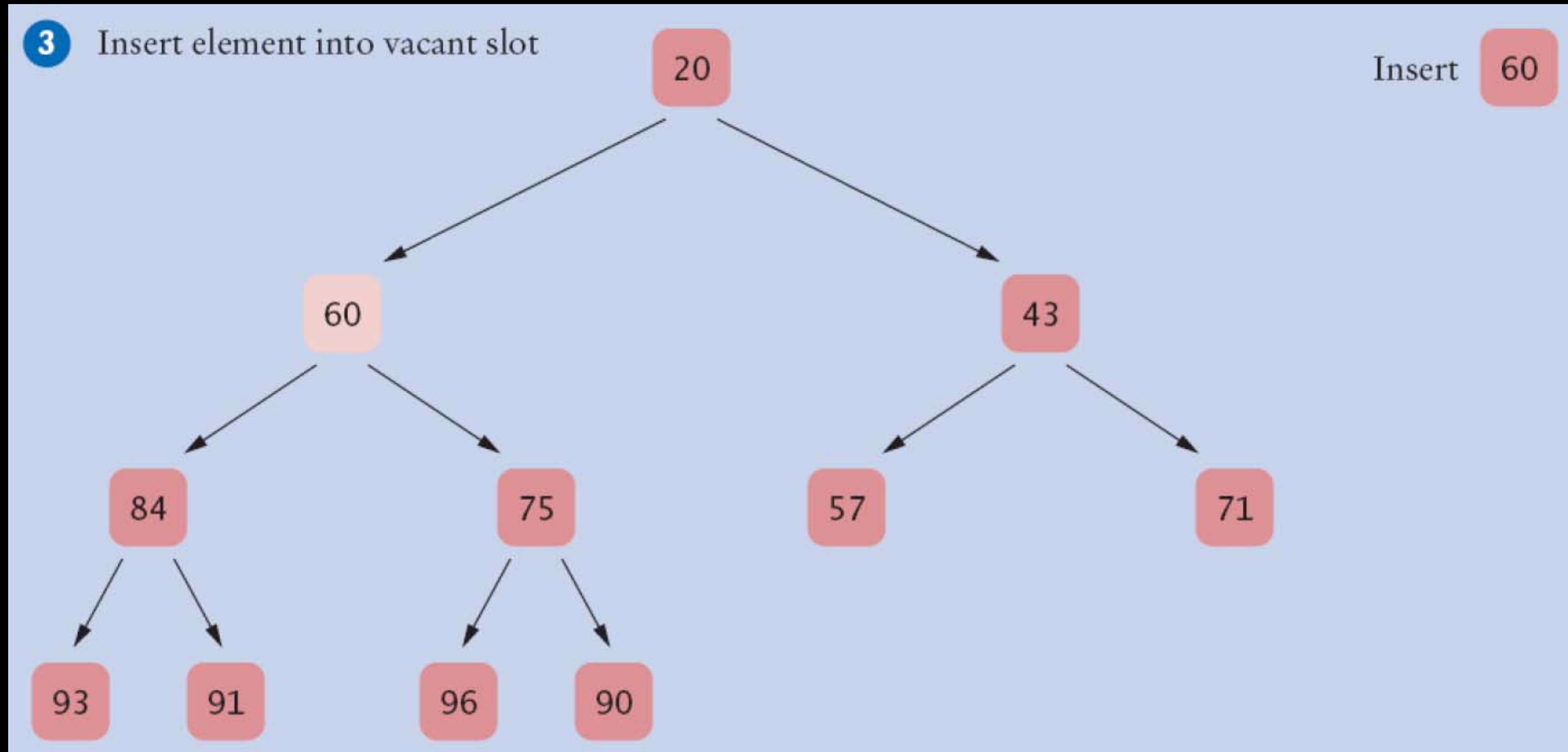

Figure 18 (continued):
Inserting a New Element in a Heap

Removing an Arbitrary Node from a Heap

1. Extract the root node value

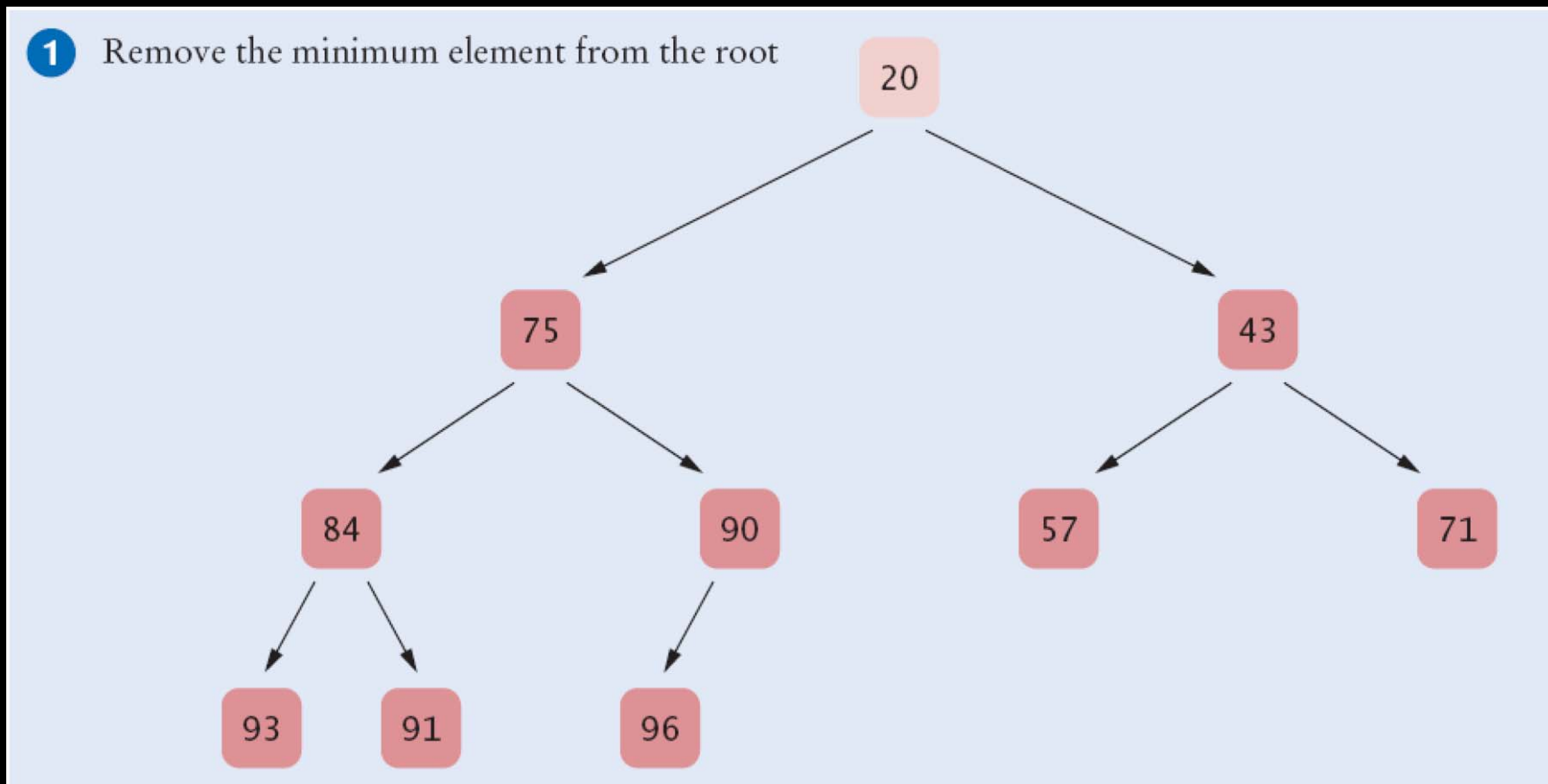

Figure 19:
Removing the Minimum Value from a Heap

Removing an Arbitrary Node from a Heap

1. **Move the value of the last node of the heap into the root node, and remove the last node. Hep property may be violated for root node (one or both of its children may be smaller).**

Continued

Removing an Arbitrary Node from a Heap

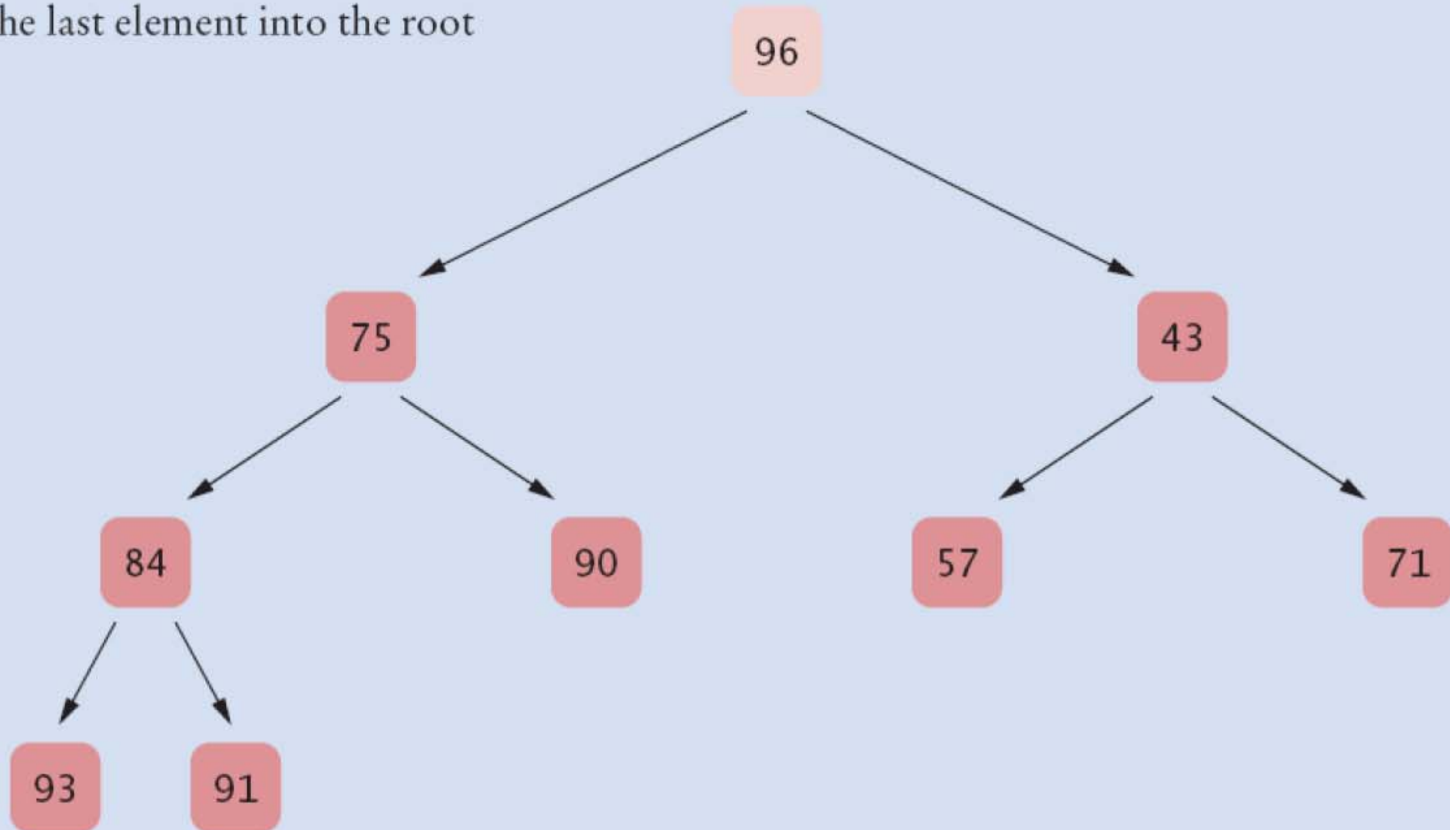

Figure 19 (continued):
Removing the Minimum Value from a Heap

Removing an Arbitrary Node from a Heap

1. Promote the smaller child of the root node. Root node again fulfills the heap property. Repeat process with demoted child. Continue until demoted child has no smaller children. Heap property is now fulfilled again.
   This process is called "fixing the heap".

Removing an Arbitrary Node from a Heap

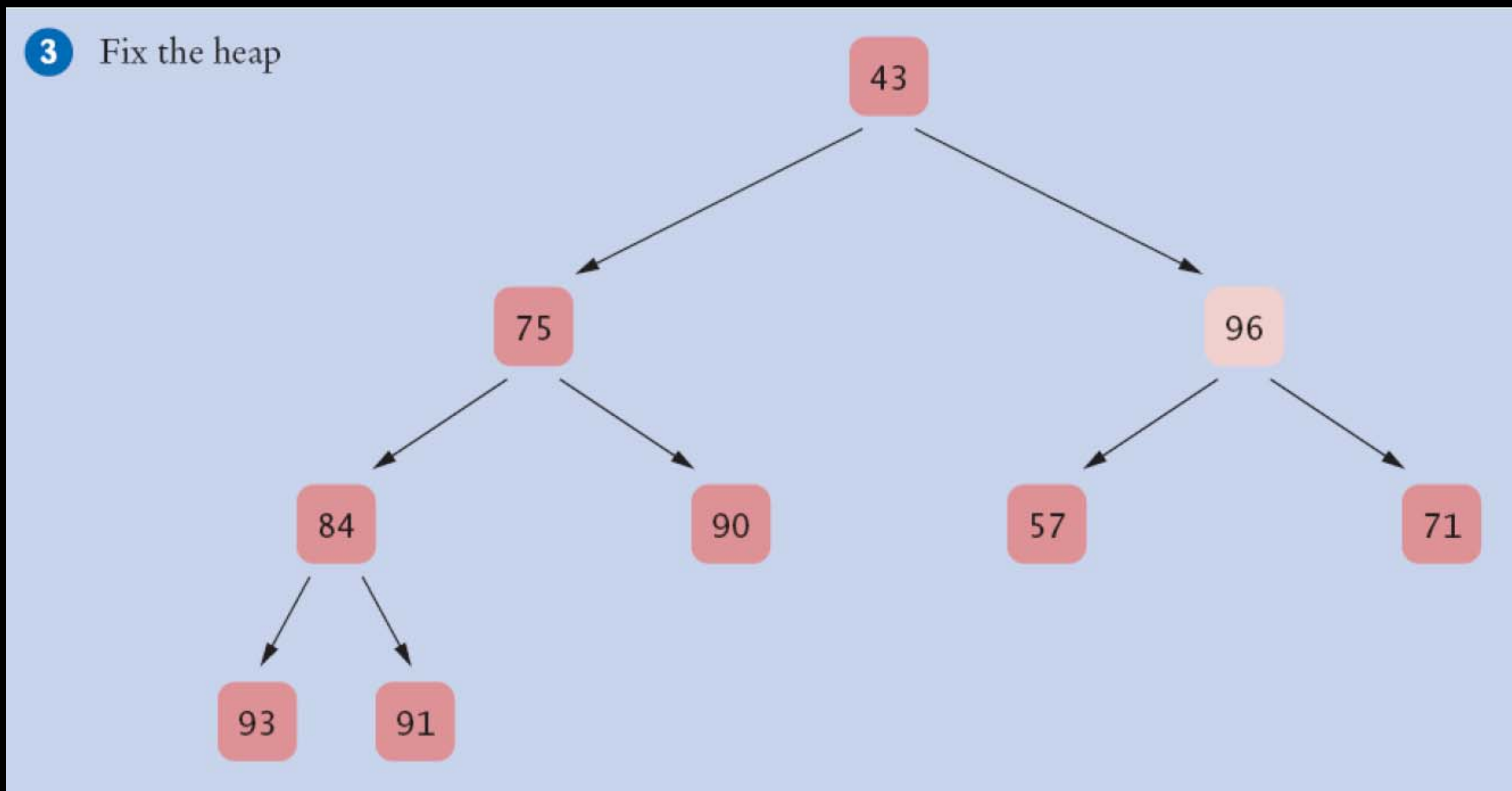

Figure 19 (continued):
Removing the Minimum Value from a Heap

Removing an Arbitrary Node from a Heap

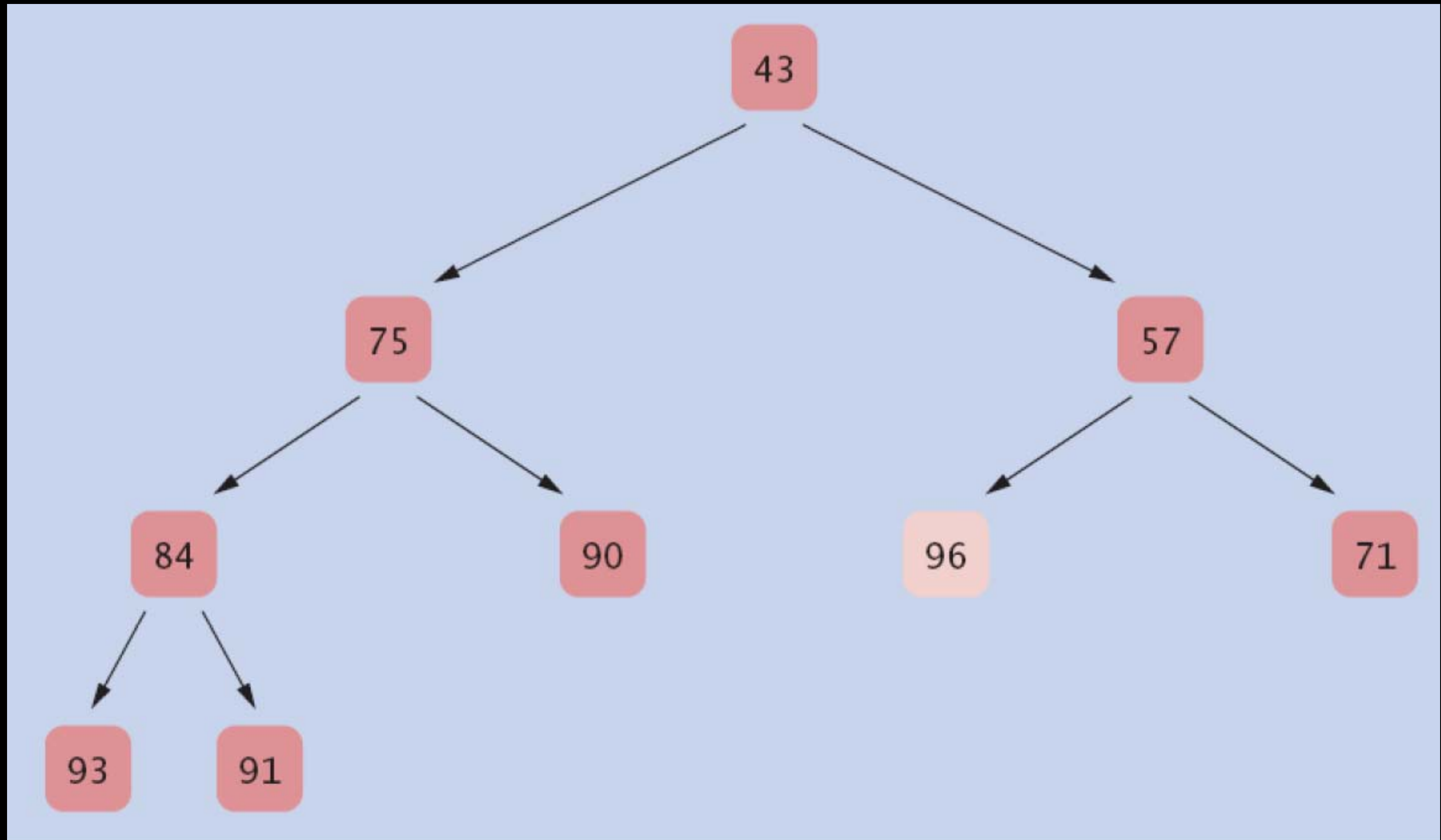

Figure 19 (continued):
Removing the Minimum Value from a Heap

Heap Efficiency

- Insertion and removal operations visit at most *h* nodes
- *h*: Height of the tree
- If *n* is the number of elements, then

$$2^{h-1} \le n < 2^h$$

or

$$h - 1 \le \log_2(n) < h$$

Continued

Heap Efficiency

- Thus, insertion and removal operations take $O(\log(n))$ steps
- Heap's regular layout makes it possible to store heap nodes efficiently in an array

Storing a Heap in an Array

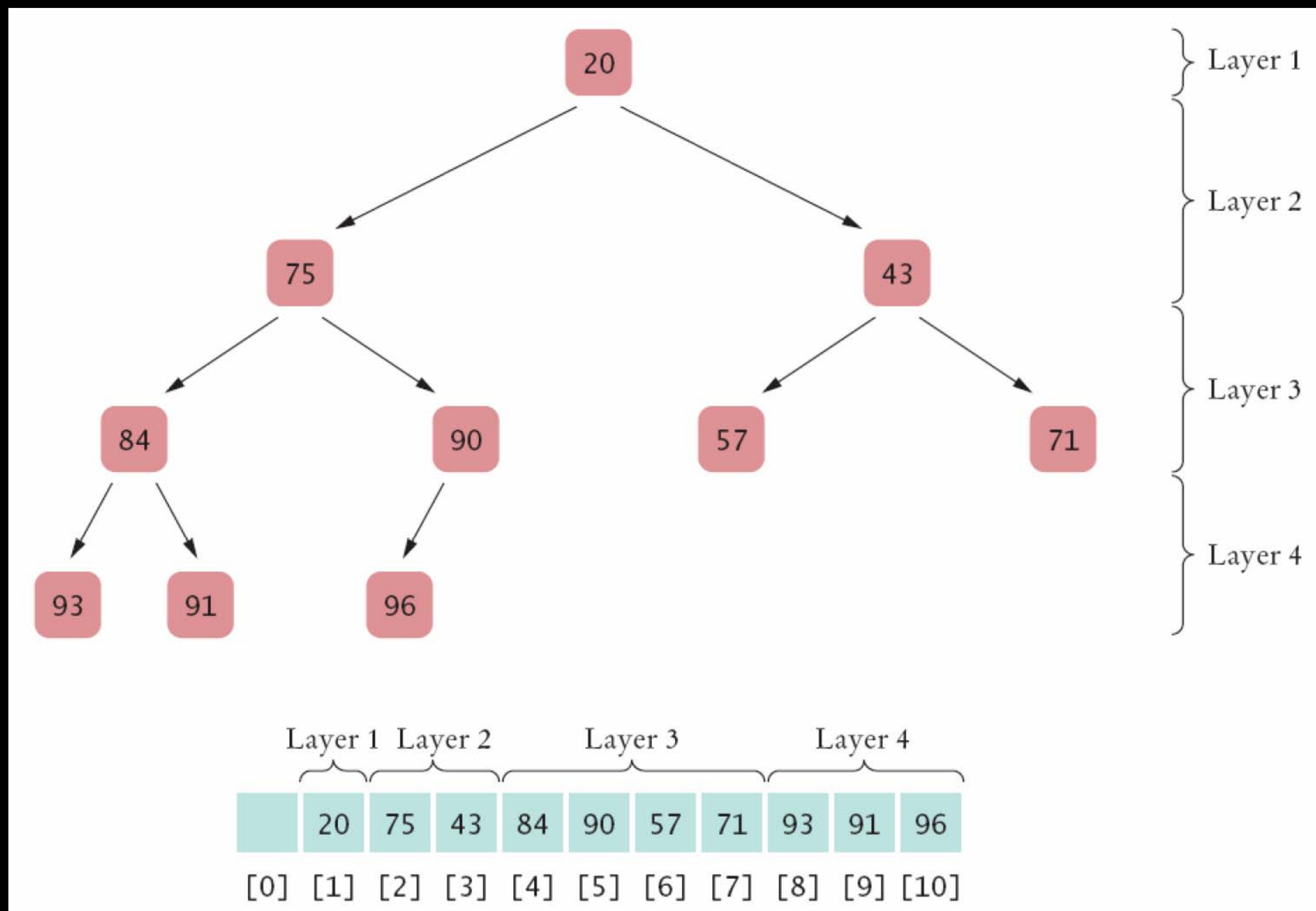

Figure 20:
Storing a Heap in an Array

File `MinHeap.java`

```
001: import java.util.*;
002:
003: /**
004:    This class implements a heap.
005: */
006: public class MinHeap
007: {
008:    /**
009:       Constructs an empty heap.
010:    */
011:    public MinHeap()
012:    {
013:       elements = new ArrayList<Comparable>();
014:       elements.add(null);
015:    }
016:
```

Continued

File MinHeap.java

```
017:     /**
018:        Adds a new element to this heap.
019:        @param newElement the element to add
020:     */
021:     public void add(Comparable newElement)
022:     {
023:        // Add a new leaf
024:        elements.add(null);
025:        int index = elements.size() - 1;
026:
027:        // Demote parents that are larger than the new element
028:        while (index > 1
029:              && getParent(index).compareTo(newElement) > 0)
030:        {
031:           elements.set(index, getParent(index));
032:           index = getParentIndex(index);
033:        }
```

Continued

File `MinHeap.java`

```
034:
035:        // Store the new element into the vacant slot
036:        elements.set(index, newElement);
037:    }
038:
039:    /**
040:        Gets the minimum element stored in this heap.
041:        @return the minimum element
042:    */
043:    public Comparable peek()
044:    {
045:        return elements.get(1);
046:    }
047:
048:    /**
049:        Removes the minimum element from this heap.
050:        @return the minimum element
051:    */
```

Continued

File MinHeap.java

```
052:     public Comparable remove()
053:     {
054:        Comparable minimum = elements.get(1);
055:
056:        // Remove last element
057:        int lastIndex = elements.size() - 1;
058:        Comparable last = elements.remove(lastIndex);
059:
060:        if (lastIndex > 1)
061:        {
062:           elements.set(1, last);
063:           fixHeap();
064:        }
065:
066:        return minimum;
067:     }
068:
```

Continued

File MinHeap.java

```
069:     /**
070:        Turns the tree back into a heap, provided only the
071:        root node violates the heap condition.
072:     */
073:     private void fixHeap()
074:     {
075:        Comparable root = elements.get(1);
076:
077:        int lastIndex = elements.size() - 1;
078:        // Promote children of removed root while
              they are larger than last
079:
080:        int index = 1;
081:        boolean more = true;
082:        while (more)
083:        {
084:           int childIndex = getLeftChildIndex(index);
085:           if (childIndex <= lastIndex)
086:           {
```

Continued

File `MinHeap.java`

```
087:              // Get smaller child
088:
089:              // Get left child first
090:              Comparable child = getLeftChild(index);
091:
092:              // Use right child instead if it is smaller
093:              if (getRightChildIndex(index) <= lastIndex
094:                    && getRightChild(index).compareTo(child) < 0)
095:              {
096:                 childIndex = getRightChildIndex(index);
097:                 child = getRightChild(index);
098:              }
099:
100:              // Check if larger child is smaller than root
101:              if (child.compareTo(root) < 0)
102:              {
103:                 // Promote child
```

Continued

File `MinHeap.java`

```
104:                 elements.set(index, child);
105:                 index = childIndex;
106:              }
107:              else
108:              {
109:                 // Root is smaller than both children
110:                 more = false;
111:              }
112:           }
113:           else
114:           {
115:              // No children
116:              more = false;
117:           }
118:        }
119: 
120:        // Store root element in vacant slot
121:        elements.set(index, root);
122:     }
```

Continued

File MinHeap.java

```
123:
124:    /**
125:       Returns the number of elements in this heap.
126:    */
127:    public int size()
128:    {
129:       return elements.size() - 1;
130:    }
131:
132:    /**
133:       Returns the index of the left child.
134:       @param index the index of a node in this heap
135:       @return the index of the left child of the given node
136:    */
137:    private static int getLeftChildIndex(int index)
138:    {
139:       return 2 * index;
140:    }
```

Continued

File MinHeap.java

```
141:
142:    /**
143:       Returns the index of the right child.
144:       @param index the index of a node in this heap
145:       @return the index of the right child of the given node
146:    */
147:    private static int getRightChildIndex(int index)
148:    {
149:       return 2 * index + 1;
150:    }
151:
152:    /**
153:       Returns the index of the parent.
154:       @param index the index of a node in this heap
155:       @return the index of the parent of the given node
156:    */
```

Continued

File MinHeap.java

```
157:     private static int getParentIndex(int index)
158:     {
159:        return index / 2;
160:     }
161: 
162:     /**
163:        Returns the value of the left child.
164:        @param index the index of a node in this heap
165:        @return the value of the left child of the given node
166:     */
167:     private Comparable getLeftChild(int index)
168:     {
169:        return elements.get(2 * index);
170:     }
171: 
172:     /**
173:        Returns the value of the right child.
174:        @param index the index of a node in this heap
```

Continued

File MinHeap.java

```
175:         @return the value of the right child of the given node
176:      */
177:      private Comparable getRightChild(int index)
178:      {
179:         return elements.get(2 * index + 1);
180:      }
181: 
182:      /**
183:         Returns the value of the parent.
184:         @param index the index of a node in this heap
185:         @return the value of the parent of the given node
186:      */
187:      private Comparable getParent(int index)
188:      {
189:         return elements.get(index / 2);
190:      }
191: 
192:      private ArrayList<Comparable> elements;
193: }
```

File HeapTester.java

```
01: /**
02:    This program demonstrates the use of a heap as a
      priority queue.
03: */
04: public class HeapTester
05: {
06:    public static void main(String[] args)
07:    {
08:       MinHeap q = new MinHeap();
09:       q.add(new WorkOrder(3, "Shampoo carpets"));
10:       q.add(new WorkOrder(7, "Empty trash"));
11:       q.add(new WorkOrder(8, "Water plants"));
12:       q.add(new WorkOrder(10, "Remove pencil sharpener
             shavings"));
13:       q.add(new WorkOrder(6, "Replace light bulb"));
14:       q.add(new WorkOrder(1, "Fix broken sink"));
15:       q.add(new WorkOrder(9, "Clean coffee maker"));
16:       q.add(new WorkOrder(2, "Order cleaning supplies"));
17:
```

Continued

File HeapTester.java

```
18:         while (q.size() > 0)
19:             System.out.println(q.remove());
20:     }
21: }
```

File WorkOrder.java

```
01: /**
02:    This class encapsulates a work order with a priority.
03: */
04: public class WorkOrder implements Comparable
05: {
06:    /**
07:       Constructs a work order with a given priority and
            // description.
08:       @param aPriority the priority of this work order
09:       @param aDescription the description of this work order
10:    */
11:    public WorkOrder(int aPriority, String aDescription)
12:    {
13:       priority = aPriority;
14:       description = aDescription;
15:    }
16:
```

Continued

File WorkOrder.java

```
17:     public String toString()
18:     {
19:        return "priority=" + priority + ", description="
              + description;
20:     }
21:
22:     public int compareTo(Object otherObject)
23:     {
24:        WorkOrder other = (WorkOrder) otherObject;
25:        if (priority < other.priority) return -1;
26:        if (priority > other.priority) return 1;
27:        return 0;
28:     }
29:
30:     private int priority;
31:     private String description;
32: }
```

File `WorkOrder.java`

- **Output:**

```
priority=1, description=Fix broken sink
priority=2, description=Order cleaning supplies
priority=3, description=Shampoo carpets
priority=6, description=Replace light bulb
priority=7, description=Empty trash
priority=8, description=Water plants
priority=9, description=Clean coffee maker
priority=10, description=Remove pencil sharpener shavings
```

Self Check

1. The software that controls the events in a user interface keeps the events in a data structure. Whenever an event such as a mouse move or repaint request occurs, the event is added. Events are retrieved according to their importance. What abstract data type is appropriate for this application?
2. Could we store a binary search tree in an array so that we can quickly locate the children by looking at array locations `2 * index` and `2 * index + 1`?

Answers

1. A priority queue is appropriate because we want to get the important events first, even if they have been inserted later.
2. Yes, but a binary search tree isn't almost filled, so there may be holes in the array. We could indicate the missing nodes with `null` elements.

The `Heapsort` Algorithm

- **Based on inserting elements into a heap and removing them in sorted order**
- **This algorithm is an $O(n \log(n))$ algorithm:**
  - Each insertion and removal is $O(\log(n))$
  - These steps are repeated n times, once for each element in the sequence that is to be sorted

The `Heapsort` Algorithm

- **Can be made more efficient**
  - Start with a sequence of values in an array and "fixing the heap" iteratively
- **First fix small subtrees into heaps, then fix larger trees**
- **Trees of size 1 are automatically heaps**

Continued

The `Heapsort` Algorithm

- Begin the fixing procedure with the subtrees whose roots are located in the next-to-lowest level of the tree
- Generalized `fixHeap` method fixes a subtree with a given root index:

```
void fixHeap(int rootIndex, int lastIndex)
```

Turning a Tree into a Heap

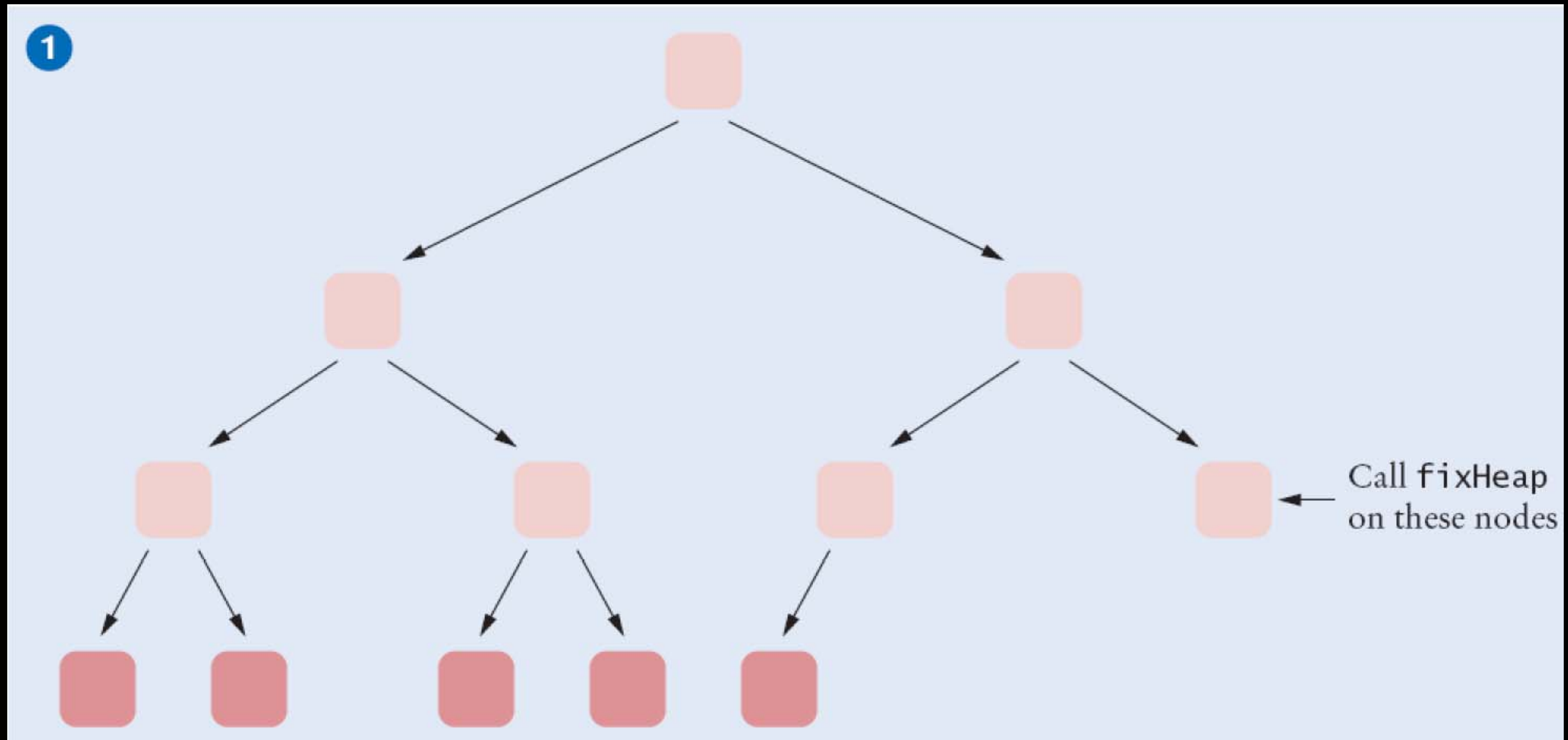

Figure 21a:
Turning a Tree into a Heap

Turning a Tree into a Heap

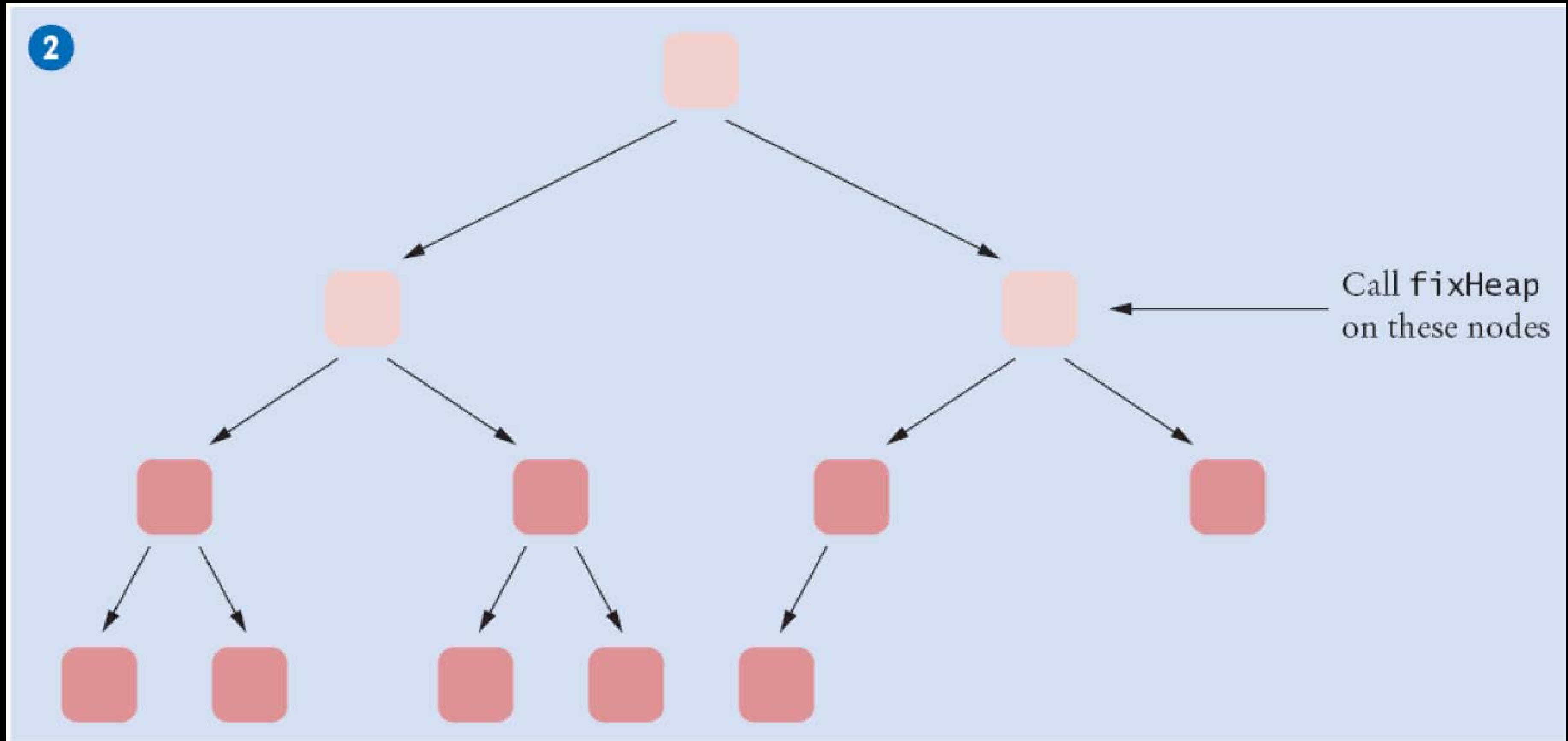

Figure 21b:
Turning a Tree into a Heap

Turning a Tree into a Heap

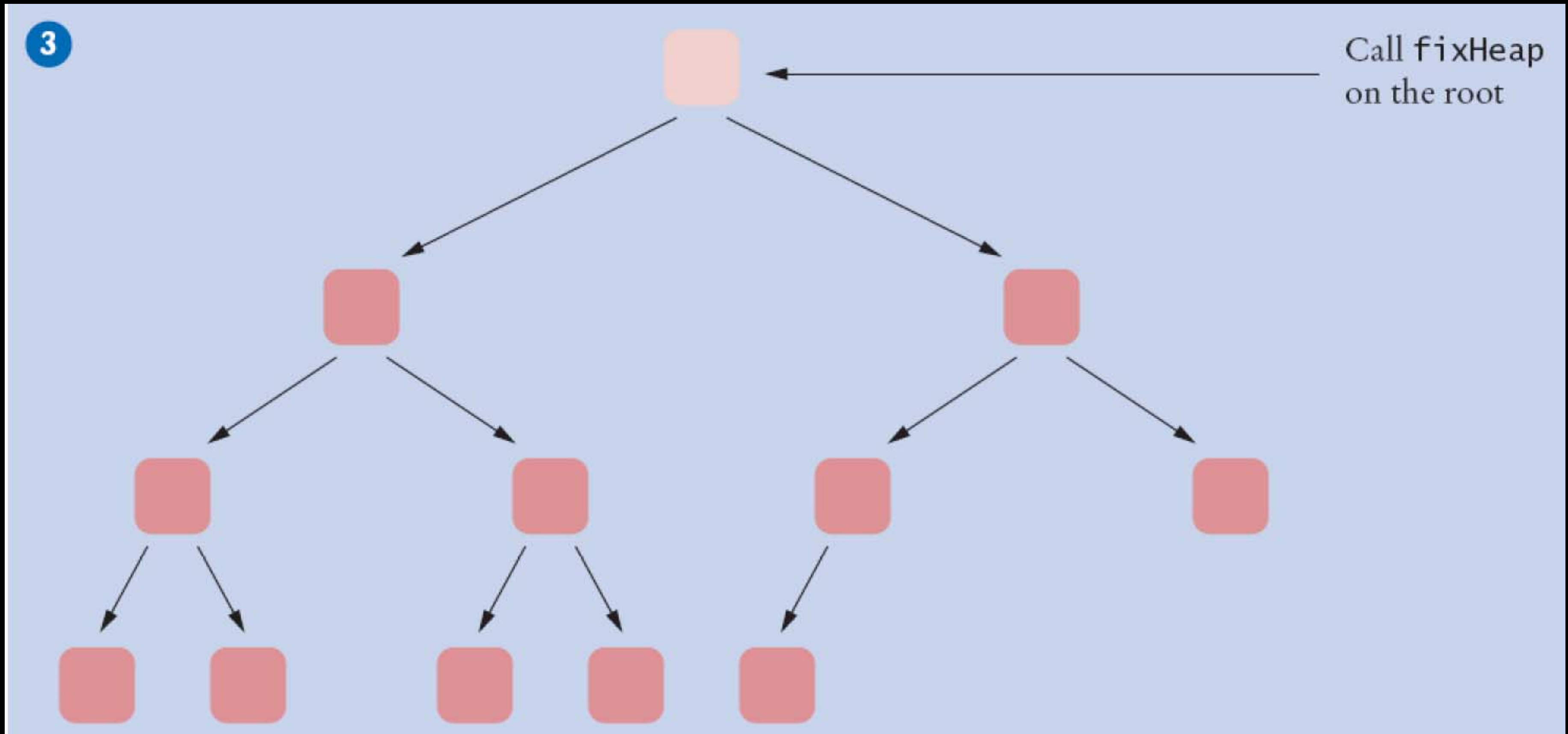

Figure 21c:
Turning a Tree into a Heap

The `Heapsort` Algorithm

- **After array has been turned into a heap, repeatedly remove the root element**
  - Swap root element with last element of the tree and then reduce the tree length
- **Removed root ends up in the last position of the array, which is no longer needed by the heap**

Continued

The `Heapsort` Algorithm

- We can use the same array both to hold the heap (which gets shorter with each step) and the sorted sequence (which gets longer with each step)
- Use a max-heap rather than a min-heap so that sorted sequence is accumulated in the correct order

Using `Heapsort` to Sort an Array

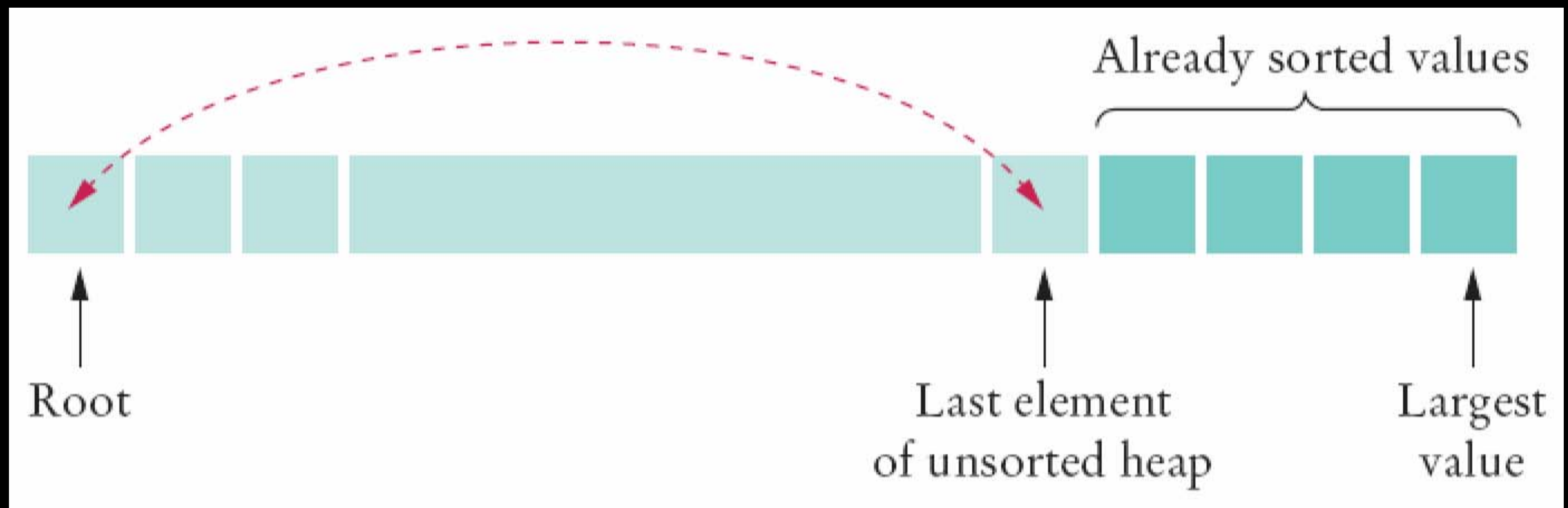

Figure 22:
Using `Heapsort` to Sort an Array

File Heapsorter.java

```
001: /**
002:    This class applies the heapsort algorithm to sort an array.
003: */
004: public class HeapSorter
005: {
006:    /**
007:       Constructs a heap sorter that sorts a given array.
008:       @param anArray an array of integers
009:    */
010:    public HeapSorter(int[] anArray)
011:    {
012:       a = anArray;
013:    }
014:
015:    /**
016:       Sorts the array managed by this heap sorter.
017:    */
```

Continued

File Heapsorter.java

```
018:    public void sort()
019:    {
020:       int n = a.length - 1;
021:       for (int i = (n - 1) / 2; i >= 0; i--)
022:          fixHeap(i, n);
023:       while (n > 0)
024:       {
025:          swap(0, n);
026:          n--;
027:          fixHeap(0, n);
028:       }
029:    }
030:
031:    /**
032:       Ensures the heap property for a subtree, provided its
033:       children already fulfill the heap property.
```

Continued

File `Heapsorter.java`

```
034:          @param rootIndex the index of the subtree to be fixed
035:          @param lastIndex the last valid index of the tree that
036:          contains the subtree to be fixed
037:       */
038:       private void fixHeap(int rootIndex, int lastIndex)
039:       {
040:          // Remove root
041:          int rootValue = a[rootIndex];
042:
043:          // Promote children while they are larger than the root
044:
045:          int index = rootIndex;
046:          boolean more = true;
047:          while (more)
048:          {
049:             int childIndex = getLeftChildIndex(index);
050:             if (childIndex <= lastIndex)
```

Continued

File Heapsorter.java

```
051:             {
052:                // Use right child instead if it is larger
053:                int rightChildIndex = getRightChildIndex(index);
054:                if (rightChildIndex <= lastIndex
055:                      && a[rightChildIndex] > a[childIndex])
056:                {
057:                   childIndex = rightChildIndex;
058:                }
059:
060:                if (a[childIndex] > rootValue)
061:                {
062:                   // Promote child
063:                   a[index] = a[childIndex];
064:                   index = childIndex;
065:                }
066:                else
067:                {
```

Continued

File `Heapsorter.java`

```
068:                     // Root value is larger than both children
069:                     more = false;
070:                 }
071:             }
072:             else
073:             {
074:                 // No children
075:                 more = false;
076:             }
077:         }
078: 
079:         // Store root value in vacant slot
080:         a[index] = rootValue;
081:     }
082: 
```

Continued

File `Heapsorter.java`

```
083:     /**
084:        Swaps two entries of the array.
085:        @param i the first position to swap
086:        @param j the second position to swap
087:     */
088:     private void swap(int i, int j)
089:     {
090:        int temp = a[i];
091:        a[i] = a[j];
092:        a[j] = temp;
093:     }
094:
095:     /**
096:        Returns the index of the left child.
097:        @param index the index of a node in this heap
098:        @return the index of the left child of the given node
099:     */
```

Continued

File Heapsorter.java

```
100:    private static int getLeftChildIndex(int index)
101:    {
102:       return 2 * index + 1;
103:    }
104:
105:    /**
106:       Returns the index of the right child.
107:       @param index the index of a node in this heap
108:       @return the index of the right child of the given node
109:    */
110:    private static int getRightChildIndex(int index)
111:    {
112:       return 2 * index + 2;
113:    }
114:
115:    private int[] a;
116: }
```

Self Check

1. Which algorithm requires less storage, heapsort or mergesort?
2. Why are the computations of the left child index and the right child index in the `HeapSorter` different than in `MinHeap`?

Answers

1. Heapsort requires less storage because it doesn't need an auxiliary array.
2. The `MinHeap` wastes the 0 entry to make the formulas more intuitive. When sorting an array, we don't want to waste the 0 entry, so we adjust the formulas instead.