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

```java
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
}
```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Scanner;
import java.util.Set;

/**
 * This program demonstrates a set of strings. The user can add and remove strings.
 */
public class SetTester {
    public static void main(String[] args) {
        Set<String> names = new HashSet<String>();
        Scanner in = new Scanner(System.in);
        Continued
boolean done = false;

while (!done)
{
    System.out.print("Add name, Q when done: ");
    String input = in.next();
    if (input.equalsIgnoreCase("Q"))
        done = true;
    else
    {
        names.add(input);
        print(names);
    }
}

done = false;

while (!done)
{
    // Continued
}
System.out.println("Remove name, Q when done");
String input = in.next();
if (input.equalsIgnoreCase("Q"))
    done = true;
else
{
    names.remove(input);
    print(names);
}

/**
 * Prints the contents of a set of strings.
 * @param s a set of strings
 */
private static void print(Set<String> s)
```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:
```
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 \textit{key set}, to another set, the \textit{value set}
- Every key in a map has a unique value
- A value may be associated with several keys
- Classes that implement the \texttt{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

```java
Set<String> keySet = m.keySet();
for (String key : keySet)
{
    Color value = m.get(key);
    System.out.println(key + "->" + value);
}
```
```java
import java.awt.Color;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

/**
 * This program tests a map that maps names to colors.
 */
public class MapTester
{
    public static void main(String[] args)
    {
        Map<String, Color> favoriteColors = new HashMap<String, Color>();
        favoriteColors.put("Juliet", Color.pink);
        favoriteColors.put("Romeo", Color.green);
    }
}
```
favoriteColors.put("Adam", Color.blue);
favoriteColors.put("Eve", Color.pink);

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

  ```
  int h = x.hashCode();
  ```
# Sample Strings and Their Hash Codes

<table>
<thead>
<tr>
<th>String</th>
<th>Hash Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
</tr>
<tr>
<td>&quot;Eve&quot;</td>
<td>70068</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
</tr>
<tr>
<td>&quot;Jim&quot;</td>
<td>74478</td>
</tr>
<tr>
<td>&quot;Joe&quot;</td>
<td>74676</td>
</tr>
<tr>
<td>&quot;Juliet&quot;</td>
<td>-2065036585</td>
</tr>
<tr>
<td>&quot;Katherine&quot;</td>
<td>2079199209</td>
</tr>
<tr>
<td>&quot;Sue&quot;</td>
<td>83491</td>
</tr>
</tbody>
</table>
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

```java
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: $O(1)$
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**
A hash set stores an unordered collection of objects, using a hash table.

```java
import java.util.AbstractSet;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class HashSet extends AbstractSet {

    /**
     * Constructs a hash table.
     * @param bucketsLength the length of the buckets array
     */
    public HashSet(int bucketsLength) {
        // Continued
    }
}
```
```java
/**
 * Tests for set membership.
 * @param x an object
 * @return true if x is an element of this set
 */
public boolean contains(Object x) {
    int h = x.hashCode();
    if (h < 0) h = -h;
    h = h % buckets.length;
    Node current = buckets[h];
    while (current != null) {
        // Continued
    }
```
if (current.data.equals(x)) return true;
current = current.next;
}
return false;
}

/**
 * Adds an element to this set.
 * @param x an object
 * @return true if x is a new object, false if x was already in the set
 */
public boolean add(Object x)
{
    int h = x.hashCode();
    if (h < 0) h = -h;
    h = h % buckets.length;
Node current = buckets[h];
while (current != null)
{
    if (current.data.equals(x))
        return false; // Already in the set
    current = current.next;
}
Node newNode = new Node();
newNode.data = x;
newNode.next = buckets[h];
buckets[h] = newNode;
size++;
```java
/**
 * Removes an object from this set.
 * @param x an object
 * @return true if x was removed from this set, false if x was not an element of this set
 */

public boolean remove(Object x) {
    int h = x.hashCode();
    if (h < 0) h = -h;
    h = h % buckets.length;

    Node current = buckets[h];
    Node previous = null;
    while (current != null) {
        if (current.data.equals(x)) {
            if (current.next != null) {
                previous.next = current.next;
            } else {
                previous.next = null;
            }
            return true;
        }
        previous = current;
        current = current.next;
    }
    return false;
}
```

Continued
if (previous == null) buckets[h] = current.next;
else previous.next = current.next;
size--;
return true;
}

previous = current;
current = current.next;
}
return false;
}

/**
 * Returns an iterator that traverses the elements of this set.
 *
 * @param a hash set iterator
 */
public Iterator iterator()
{
    return new HashSetIterator();
}
/**
 * Gets the number of elements in this set.
 * @return the number of elements
 */

public int size()
{
    return size;
}

private Node[] buckets;
private int size;

private class Node
{
    public Object data;
    public Node next;
}

Continued
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;
for (int b = bucket + 1; b < buckets.length; b++)
    if (buckets[b] != null) return true;
return false;
}

public Object next()
{
    previous = current;
    previousBucket = bucket;
    if (current == null || current.next == null)
    {
        // Move to next bucket
        bucket++;
    }
    while (bucket < buckets.length
            && buckets[bucket] == null)
        bucket++;
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();
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: }
import java.util.Iterator;
import java.util.Set;

/**
 * This program tests the hash set class.
 */
public class SetTester
{
    public static void main(String[] args)
    {
        public static void main(String[] args) {
            HashSet names = new HashSet(101); // 101 is a prime
            names.add("Sue");
            names.add("Harry");
            names.add("Nina");
            names.add("Susannah");
            names.add("Larry");
            names.add("Eve");
            // Continued
names.add("Sarah");
names.add("Adam");
names.add("Tony");
names.add("Katherine");
names.add("Juliet");
names.add("Romeo");
names.remove("Romeo");
names.remove("George");

Iterator iter = names.iterator();
while (iter.hasNext())
  System.out.println(iter.next());
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.
Computing Hash Codes

• Bad choice for hash function for a string
  ▪ Adding the unicode values of the characters in the string

```java
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

```java
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

```java
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()`
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

```java
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
/**
 * A coin with a monetary value.
 */

public class Coin
{
    /**
     * Constructs a coin.
     * @param aValue the monetary value of the coin.
     * @param aName the name of the coin
     */
    public Coin(double aValue, String aName)
    {
        value = aValue;
        name = aName;
    }
}
/**
 * Gets the coin value.
 * @return the value
 */
public double getValue()
{
    return value;
}

/**
 * Gets the coin name.
 * @return the name
 */
public String getName()
{
    return name;
}
public boolean equals(Object otherObject) {
    if (otherObject == null) return false;
    if (getClass() != otherObject.getClass()) return false;
    Coin other = (Coin) otherObject;
    return value == other.value && name.equals(other.name);
}

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;
}

Continued
public String toString() {
    return "Coin[value=" + value + ",name=" + name + "]";
}

private double value;
private String name;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

/**
 * A program to test hash codes of coins.
 */
public class HashCodeTester {
    public static void main(String[] args) {
        Coin coin1 = new Coin(0.25, "quarter");
        Coin coin2 = new Coin(0.25, "quarter");
        Coin coin3 = new Coin(0.05, "nickel");
    }
}
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: }
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
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
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
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

```java
BinarySearchTree tree = new BinarySearchTree();
tree.add("Juliet");  // 1
tree.add("Tom");  // 2
tree.add("Dick");  // 3
tree.add("Harry");  // 4
```
Example

Figure 9: Binary Search Trees After Four Insertions
Example Continued

Tree: Add Romeo

Figure 10: Binary Search Trees After Five Insertions
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);
    }
    ...
}
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
/**
 * This class implements a binary search tree whose nodes hold objects that implement the Comparable interface.
 */

public class BinarySearchTree {

/**
 * Constructs an empty tree.
 */

public BinarySearchTree() {
    root = null;
}

Continued
/**
 * Inserts a new node into the tree.
 * @param obj the object to insert
 */

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);
}
/**
 * Tries to find an object in the tree.
 * @param obj the object to find
 * @return true if the object is contained in the tree
 */

public boolean find(Comparable obj)
{
    Node current = root;
    while (current != null)
    {
        int d = current.data.compareTo(obj);
        if (d == 0) return true;
        else if (d > 0) current = current.left;
        else current = current.right;
    }
    return false;
}
/**
 * Tries to remove an object from the tree. Does nothing if the object is not contained in the tree.
 * @param obj the object to remove
 */

public void remove(Comparable obj) {
    // Find node to be removed
    Node toBeRemoved = root;
    Node parent = null;
    boolean found = false;
    while (!found && toBeRemoved != null) {
        int d = toBeRemoved.data.compareTo(obj);
        if (d == 0) found = true;
        else {
            if (d == 0) found = true;
            else {

        Continued
```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
079:         || toBeRemoved.right == null)
080:     {
081:         Node newChild;
082:         if (toBeRemoved.left == null)
083:             newChild = toBeRemoved.right;
084:         newChild = toBeRemoved.right;
```

Continued
else
    newChild = toBeRemoved.left;

if (parent == null) // Found in root
    root = newChild;
else if (parent.left == toBeRemoved)
    parent.left = newChild;
else
    parent.right = newChild;

return;

// Neither subtree is empty
// Find smallest element of the right subtree
Node smallestParent = toBeRemoved;
Node smallest = toBeRemoved.right;
while (smallest.left != null) {
    smallestParent = smallest;
    smallest = smallest.left;
}
// smallest contains smallest child in right subtree
// Move contents, unlink child
toBeRemoved.data = smallest.data;
smallestParent.left = smallest.right;
/**
 * Prints the contents of the tree in sorted order.
 */

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

private Node root;

/**
 * A node of a tree stores a data item and references of the child nodes to the left and to the right.
 */

private class Node {
    
Continued
/**
   Inserts a new node as a descendant of this node.
   @param newNode the node to insert
   */
public void addNode(Node newNode)
{
    if (newNode.data.compareTo(data) < 0)
    {
        if (left == null) left = newNode;
        else left.addNode(newNode);
    }
    else
    {
        if (right == null) right = newNode;
        else right.addNode(newNode);
    }
}
/**
 * Prints this node and all of its descendants in sorted order.
 */

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

public Comparable data;
public Node left;
public Node right;
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.
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
public class BinarySearchTree
{
    . . .
    public void print()
    {
        if (root != null)
            root.printNodes();
    }
    . . .
}
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 \(((3 + 4) \times 5)\) yields
  \[ 3 \ 4 \ + \ 5 \ \times \]

• Whereas the second tree \((3 + 4 \times 5)\) yields
  \[ 3 \ 4 \ 5 \ \times \ + \]
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 $+$ is not 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
import java.util.Comparator;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

/**
   A program to test hash codes of coins.
*/

public class TreeSetTester
{
    public static void main(String[] args)
    {
        Coin coin1 = new Coin(0.25, "quarter");
        Coin coin2 = new Coin(0.25, "quarter");
        Coin coin3 = new Coin(0.01, "penny");
        Coin coin4 = new Coin(0.05, "nickel");
        
        // Continued
```java
class CoinComparator implements Comparator<Coin> {
    public int compare(Coin first, Coin second) {
        if (first.getValue() < second.getValue()) return -1;
        if (first.getValue() == second.getValue()) return 0;
        return 1;
    }
}

Comparator<Coin> comp = new CoinComparator();
Set<Coin> coins = new TreeSet<Coin>(comp);
coins.add(coin1);
coins.add(coin2);
coins.add(coin3);
coins.add(coin4);
```
for (Coin c : coins)
    System.out.println(c);
}
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
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:

```java
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
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.
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. Heap 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

\[
\frac{2^b - 1}{2} \leq n < 2^b \\
\text{or} \\
\frac{b - 1}{2} \leq \log_2(n) < b
\]
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
import java.util.*;

/**
 * This class implements a heap.
 */

public class MinHeap
{

/**
 * Constructs an empty heap.
 */

public MinHeap()
{
    elements = new ArrayList<Comparable>();
    elements.add(null);
}
/**
 * Adds a new element to this heap.
 * @param newElement the element to add
 */

public void add(Comparable newElement) {
    // Add a new leaf
    elements.add(null);
    int index = elements.size() - 1;

    // Demote parents that are larger than the new element
    while (index > 1 && getParent(index).compareTo(newElement) > 0) {
        elements.set(index, getParent(index));
        index = getParentIndex(index);
    }
    }
034:     // Store the new element into the vacant slot
035:     elements.set(index, newElement);
036: }
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:    */
public Comparable remove()
{
    Comparable minimum = elements.get(1);

    // Remove last element
    int lastIndex = elements.size() - 1;
    Comparable last = elements.remove(lastIndex);

    if (lastIndex > 1)
    {
        elements.set(1, last);
        fixHeap();
    }

    return minimum;
}
/**
 * Turns the tree back into a heap, provided only the
 * root node violates the heap condition.
 */

private void fixHeap()
{
    Comparable root = elements.get(1);
    int lastIndex = elements.size() - 1;
    // Promote children of removed root while
    // they are larger than last
    int index = 1;
    boolean more = true;
    while (more)
    {
        int childIndex = getLeftChildIndex(index);
        if (childIndex <= lastIndex)
        {
            if (childIndex <= lastIndex)
```java
// Get smaller child

// Get left child first
Comparable child = getLeftChild(index);

// Use right child instead if it is smaller
if (getRightChildIndex(index) <= lastIndex && getRightChild(index).compareTo(child) < 0)
{
    childIndex = getRightChildIndex(index);
    child = getRightChild(index);
}

// Check if larger child is smaller than root
if (child.compareTo(root) < 0)
{
    // Promote child
    Continued
```
elements.set(index, child);
index = childIndex;
}
else {
    // Root is smaller than both children
    more = false;
}
else {
    // No children
    more = false;
}
}
else {
    // Store root element in vacant slot
    elements.set(index, root);
}
Returns the number of elements in this heap.

```
/**
 * Returns the number of elements in this heap.
 */

public int size()
{
    return elements.size() - 1;
}
```

Returns the index of the left child.

```
/**
* Returns the index of the left child.
* @param index the index of a node in this heap
* @return the index of the left child of the given node
*/

private static int getLeftChildIndex(int index)
{
    return 2 * index;
}
```
>Returns the index of the right child.
@returns the index of the right child of the given node
*/

private static int getRightChildIndex(int index)
{
    return 2 * index + 1;
}

/**
Regards the index of the parent.
@returns the index of the parent of the given node
*/
private static int getParentIndex(int index) {
    return index / 2;
}

/**
   * Returns the value of the left child.
   * @param index the index of a node in this heap
   * @return the value of the left child of the given node
   */
private Comparable getLeftChild(int index) {
    return elements.get(2 * index);
}

/**
   * Returns the value of the right child.
   * @param index the index of a node in this heap
   * @return the value of the right child of the given node
   */
MinHeap.java

```java
@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: }
```
/**
 * This program demonstrates the use of a heap as a priority queue.
 */

class HeapTester
{
    public static void main(String[] args)
    {
        MinHeap q = new MinHeap();
        q.add(new WorkOrder(3, "Shampoo carpets");
        q.add(new WorkOrder(7, "Empty trash");
        q.add(new WorkOrder(8, "Water plants");
        q.add(new WorkOrder(10, "Remove pencil sharpener shavings");
        q.add(new WorkOrder(6, "Replace light bulb");
        q.add(new WorkOrder(1, "Fix broken sink");
        q.add(new WorkOrder(9, "Clean coffee maker");
        q.add(new WorkOrder(2, "Order cleaning supplies");
    }
while (q.size() > 0)
System.out.println(q.remove());
/**
 * This class encapsulates a work order with a priority.
 */

public class WorkOrder implements Comparable {

    /**
     * Constructs a work order with a given priority and description.
     * 
     * @param aPriority the priority of this work order
     * @param aDescription the description of this work order
     */
    public WorkOrder(int aPriority, String aDescription) {
        priority = aPriority;
        description = aDescription;
    }
}
public String toString()
{
    return "priority=" + priority + ", description="
        + description;
}

public int compareTo(Object otherObject)
{
    WorkOrder other = (WorkOrder) otherObject;
    if (priority < other.priority) return -1;
    if (priority > other.priority) return 1;
    return 0;
}

private int priority;
private String description;
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 $2 \times \text{index}$ and $2 \times \text{index} + 1$?
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 \texttt{fixHeap} method fixes a subtree with a given root index:

\begin{verbatim}
void fixHeap(int rootIndex, int lastIndex)
\end{verbatim}
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
/**
 * This class applies the heapsort algorithm to sort an array.
 */

public class HeapSorter {

/**
 * Constructs a heap sorter that sorts a given array.
 * @param anArray an array of integers
 */
  public HeapSorter(int[] anArray) {
    a = anArray;
  }

/**
 * Sorts the array managed by this heap sorter.
 */
  */
public void sort() {
    int n = a.length - 1;
    for (int i = (n - 1) / 2; i >= 0; i--)
        fixHeap(i, n);
    while (n > 0) {
        swap(0, n);
        n--;
        fixHeap(0, n);
    }
}

/**
 * Ensures the heap property for a subtree, provided its children already fulfill the heap property.
 */
private void fixHeap(int rootIndex, int lastIndex) {
    // Remove root
    int rootValue = a[rootIndex];
    // Promote children while they are larger than the root
    int index = rootIndex;
    boolean more = true;
    while (more) {
        int childIndex = getLeftChildIndex(index);
        if (childIndex <= lastIndex) {
            // Promote children
            // ...
```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:          {
```
// Root value is larger than both children
more = false;
}
else
{
  // No children
  more = false;

  }

}

// Store root value in vacant slot
a[index] = rootValue;
/**
 * Swaps two entries of the array.
 * @param i the first position to swap
 * @param j the second position to swap
 */
private void swap(int i, int j)
{
    int temp = a[i];
a[i] = a[j];
a[j] = temp;
}

/**
 * Returns the index of the left child.
 * @param index the index of a node in this heap
 * @return the index of the left child of the given node
 */
private static int getLeftChildIndex(int index) {
    return 2 * index + 1;
}

/**
 * Returns the index of the right child.
 * @param index the index of a node in this heap
 * @return the index of the right child of the given node
 */
private static int getRightChildIndex(int index) {
    return 2 * index + 2;
}

private int[] a;
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.