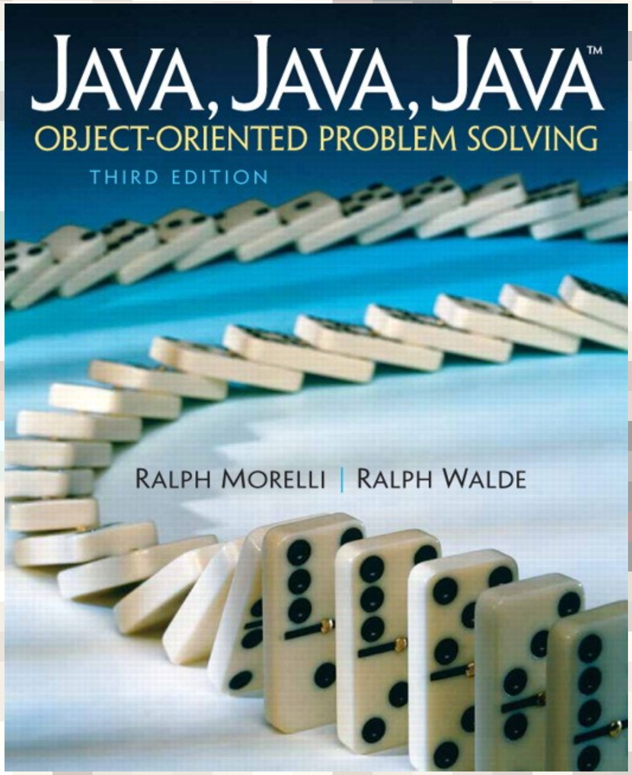


presentation slides for

Object-Oriented Problem Solving

JAVA, JAVA, JAVA

Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

Java, Java, Java

Object Oriented Problem Solving

Lecture 9: Abstract Data Structures: Queues, Sets, Tree Maps

Objectives

- Understand the concepts of a dynamic data structure and an Abstract Data Type (ADT).
- Understand the stack, queue, set, and map ADTs.
- Be able to use inheritance to define extensible data structures.
- Know how to use the TreeSet, TreeMap, HashSet, and HashMap library classes.

Outline

- Static Methods in Lists and Collections
- The Queue ADT
- The Queue<E> and PriorityQueue<E> classes
- Using the Set and Map interfaces
- The Binary Search Tree Data Structure

Static Methods in Lists and Collections

java.util.Collections	
+sort(list: List): void	Sort the specified list
+sort(list: List, c: Comparator): void	Sorts the specified list with the comparator.
+binarySearch(list: List, key: Object): int	Searches the key in the sorted list using binary search.
+binarySearch(list: List, key: Object, c: Comparator): int	Searches the key in the sorted list using binary search with the comparator.
+reverse(list: List): void	Reverses the specified list.
+reverseOrder(): Comparator	Returns a comparator with the reverse ordering.
+shuffle(list: List): void	Shuffles the specified list randomly.
+shuffle(list: List, rmd: Random): void	Shuffles the specified list with a random object.

Static Methods in Lists and Collections (cont.)

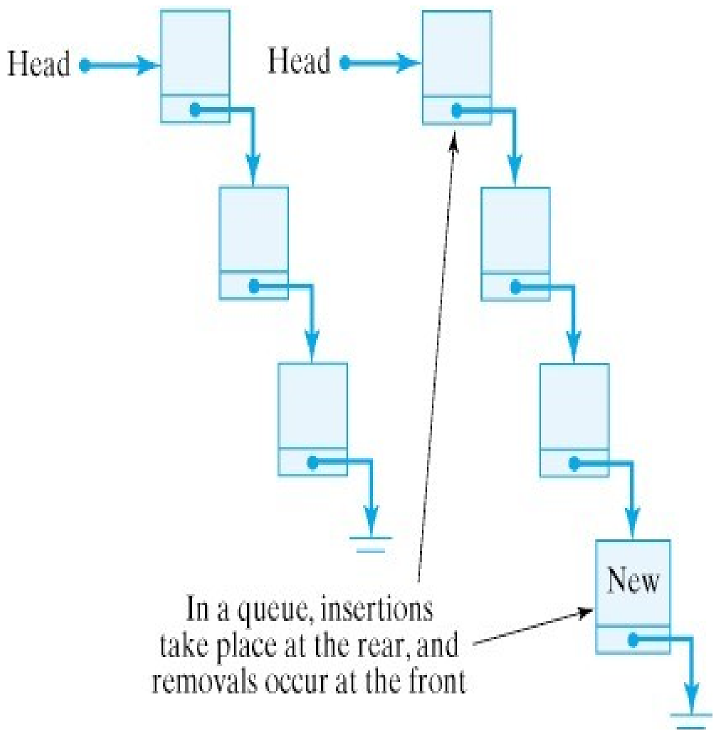
java.util.Collections	
+copy(des: List, src: List): void	Copies from the source list to the destination list.
+nCopies(n: int, o: Object): List	Returns a list consisting of *n* copies of the object.
+fill(list: List, o: Object): void	Fills the list with the object.
+max(c: Collection): Object	Returns the max object in the collection.
+max(c: Collection, c: Comparator): Object	Returns the max object using the comparator.
+min(c: Collection): Object	Returns the min object in the collection.
+min(c: Collection, c: Comparator): Object	Returns the min object using the comparator.
+disjoint(c1: Collection, c2: Collection): boolean	Returns true if c1 and c2 have no elements in common.
+frequency(c: Collection, o: Object): int	Returns the number of occurrences of the specified element in the collection.

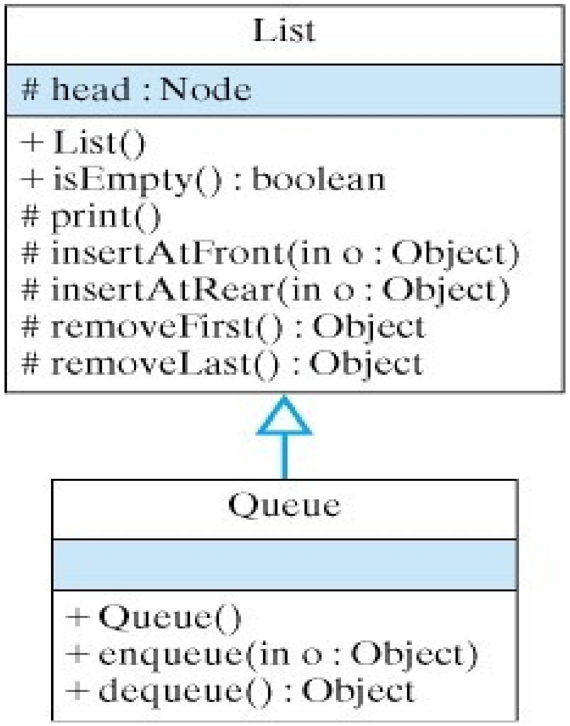
The Queue ADT

- A *Queue* is a special type of List, which extends List.
- It only allows insertions at the end of the list and removals at the front of the list.
- Thus, it enforces *first-in/first-out (FIFO)* behavior.
- A queue insert operation is called *enqueue*.
- A queue remove operation is called *dequeue*.
- Queues behave like a line at a salad bar.
- Queues are used in several computing tasks like scheduling a computer's CPU.

The Queue class

- The *Queue* class is a subclass of List.
- The dequeue() method uses removeFirst().
- The enqueue() method uses insertAtRear().


Head
Head
New
In a queue, insertions take place at the rear, and removals occur at the front



Implementation of the Queue class

- Queue class methods simply call List methods insertAtRear() and removeFirst().

```
public class Queue extends List {

    public Queue() {
        super();            // Initialize the list
    }

    public void enqueue(Object obj) {
        insertAtRear(obj);
    }

    public Object dequeue() {
        return removeFirst();
    }

// Queue
```

Testing the Queue class

- This main() is similar to the one for stack.
- This time the letters are not reversed.

```
public static void main(String argv[]) {
       Queue queue = new Queue();
       String string = "Hello this is a test string";
       System.out.println("String: " + string);
       for (int k = 0; k < string.length(); k++)
           queue.enqueue( new Character(string.charAt(k)));
       System.out.println("The current queue:");
       queue.print();

       Object o = null;
       System.out.println("Dequeuing:");
       while (!queue.isEmpty()) {
           o  = queue.dequeue();
           System.out.print( o.toString() );
       }
   } // main()
```

PriorityQueue

- The *PriorityQueue* class is a subclass of *Queue*.
- Elements are assigned priorities.
- Element with the highest priority is removed first.

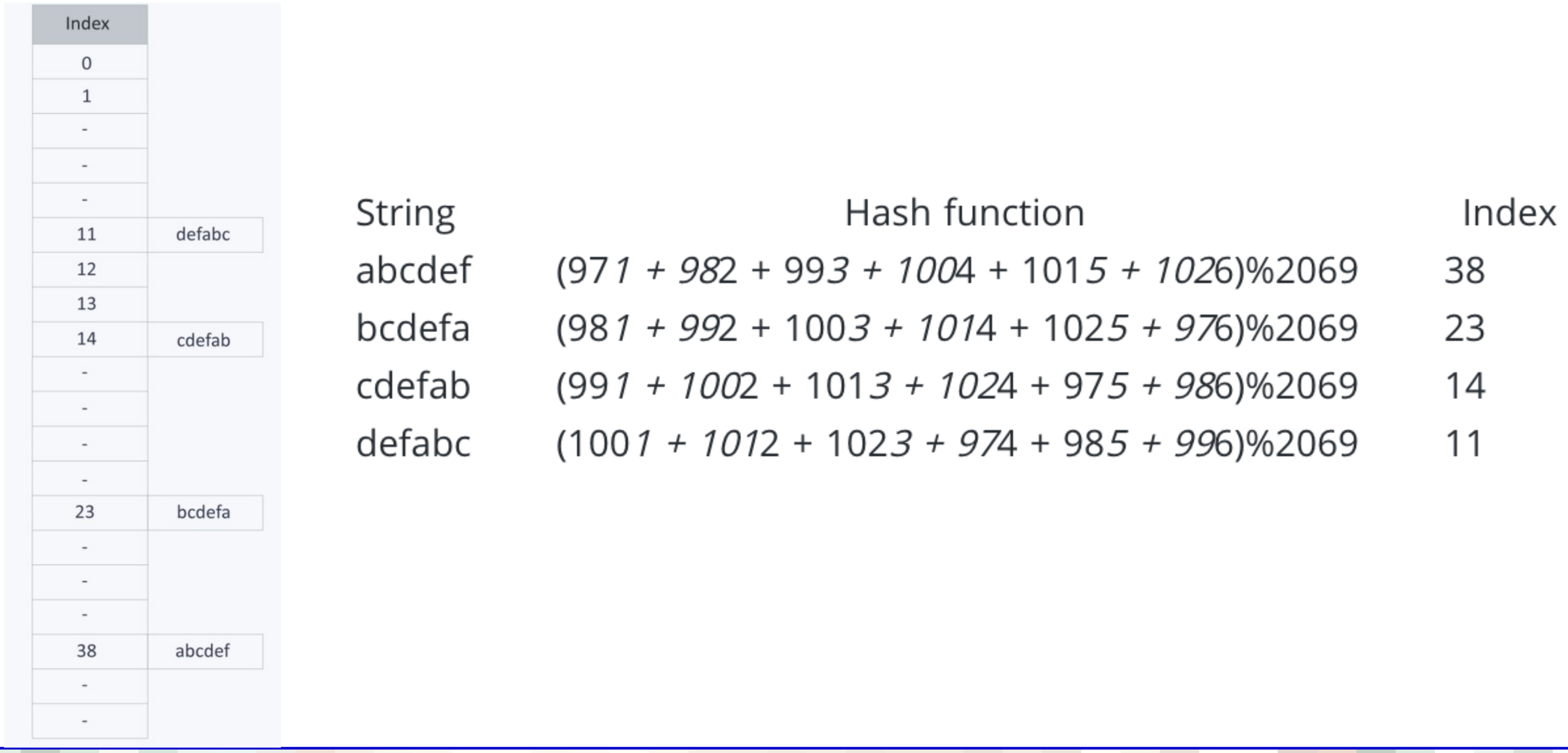
PriorityQueue<E>	
+PriorityQueue()	Creates a default priority queue with initial capacity 11.
+PriorityQueue(initialCapacity: int)	Creates a default priority queue with the specified initial capacity.
+PriorityQueue(c: Collection<? extends E>)	Creates a priority queue with the specified collection.
+PriorityQueue(initialCapacity: int, comparator: Comparator<? super E>)	Creates a priority queue with the specified initial capacity and the comparator.

Set<E> Interface and TreeSet<E> Class

- The **TreeSet<E>** and **HashSet<E>** classes both implement the **Set<E>** interface.
- The Set<E> methods are for locating data elements using an ordering of the data when the searching needs to be done more quickly than can be done with a list.
- Note: Set is not allowed to store duplicated values.

Set<E> Interface and TreeSet<E> Class

- TreeSet uses Tree data structure for storage.
- HashSet use HashTable data structure for storage.
- Hash function is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called hash values.

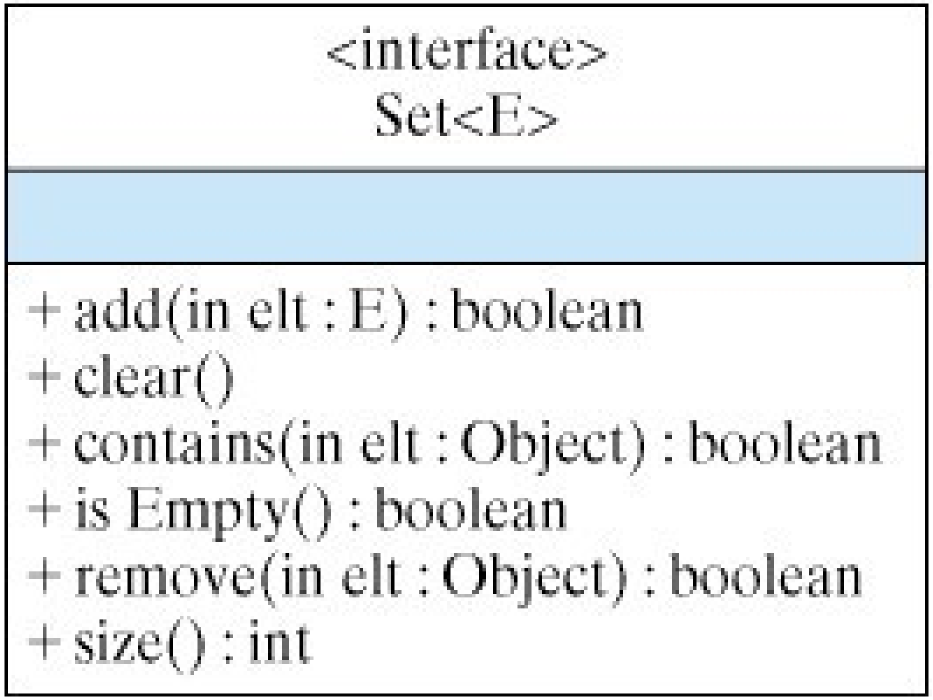


Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

String	Hash function	Index
abcdef	(971 + 982 + 993 + 1004 + 1015 + 1026)%2069	38
bcdefa	(981 + 992 + 1003 + 1014 + 1025 + 976)%2069	23
cdefab	(991 + 1002 + 1013 + 1024 + 975 + 986)%2069	14
defabc	(1001 + 1012 + 1023 + 974 + 985 + 996)%2069	11

Methods of the Set<E> Interface

- **TreeSet<E>** and **HashSet<E>** classes both implement methods of the **Set<E>** interface.
- This is a partial list of methods of Set<E>.



```
+ add(in elt : E) : boolean
+ clear()
+ contains(in elt : Object) : boolean
+ is Empty() : boolean
+ remove(in elt : Object) : boolean
+ size() : int
```

Using Set<E> and TreeSet<E>

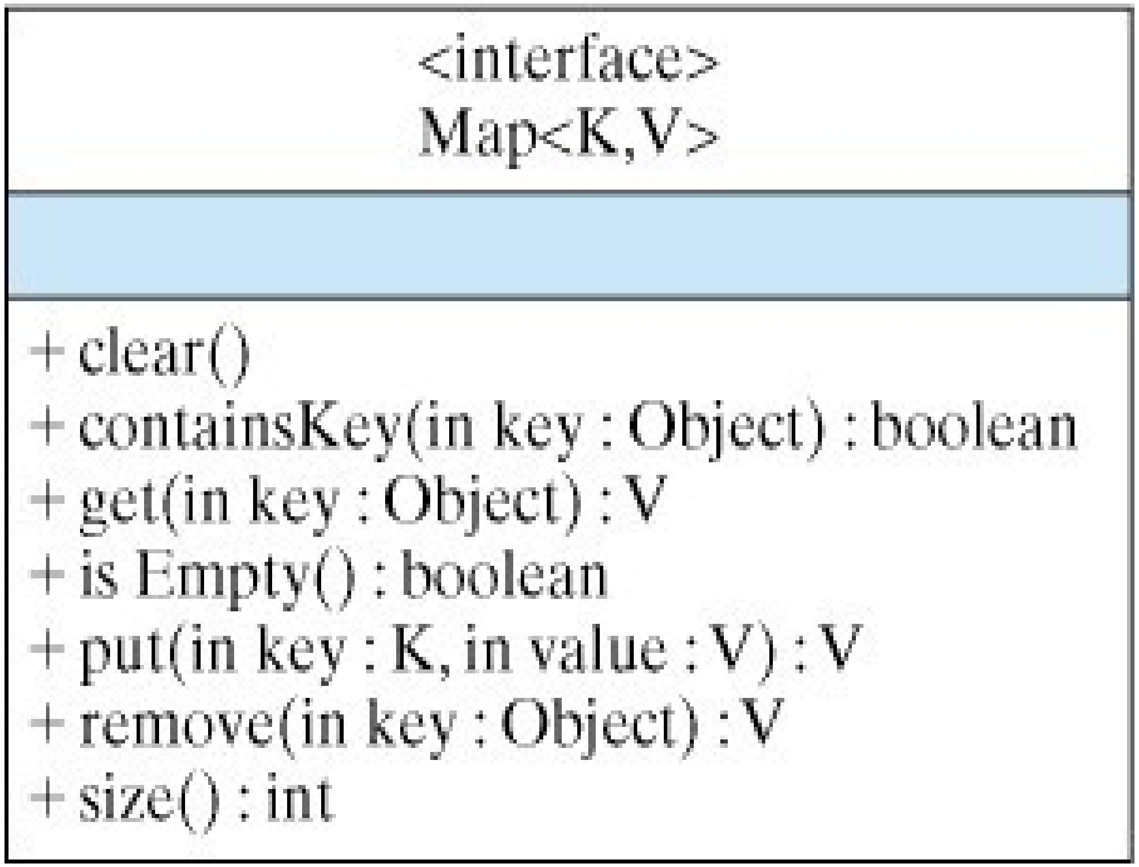
```
public static void testSet() {
  Set<PhoneRecord> theSet = new TreeSet<PhoneRecord>();
  // new HashSet<PhoneRecord>(); could also be used.
  theSet.add(new PhoneRecord("Roger M", "090-997-2918"));
  theSet.add(new PhoneRecord("Jane M", "090-997-1987"));
  theSet.add(new PhoneRecord("Stacy K", "090-997-9188"));
  theSet.add(new PhoneRecord("Gary G", "201-119-8765"));
  theSet.add(new PhoneRecord("Jane M", "090-987-6543"));
  System.out.println("Testing TreeSet and Set");
  PhoneRecord ph1 =
    new PhoneRecord("Roger M", "090-997-2918");
  PhoneRecord ph2 =
    new PhoneRecord("Mary Q", "090-242-3344");
  System.out.println("Roger M contained in theSet is");
  System.out.println(theSet.contains(ph1));
  System.out.println("Mary Q contained in theSet is");
  System.out.println(theSet.contains(ph2));
  for (PhoneRecord pr : theSet) System.out.println(pr);
} // testSet()
```

Map<K,V> Interface and TreeMap<K,V> Class

- The **TreeMap<K,V>** and **HashMap<K,V>** classes both implement the **Map<K,V>** interface.
- The Map<K,V> methods are for locating a value from V given a key from K.
- Whether a TreeMap<K,V> may be more appropriate than HashMap<K,V> depends on issues connected to properties of binary search trees and hash functions.

Methods in the Map<K,V> Interface

- A partial list of methods in **Map<K,V>**.



```
<interface>
Map<K,V>

+ clear()
+ containsKey(in key : Object) : boolean
+ get(in key : Object) : V
+ is Empty() : boolean
+ put(in key : K, in value : V) : V
+ remove(in key : Object) : V
+ size() : int
```

Using Map<K,V> and TreeMap<K,V>

- The following code demonstrates how Map<String,String> can be used with TreeMap<String,String>.

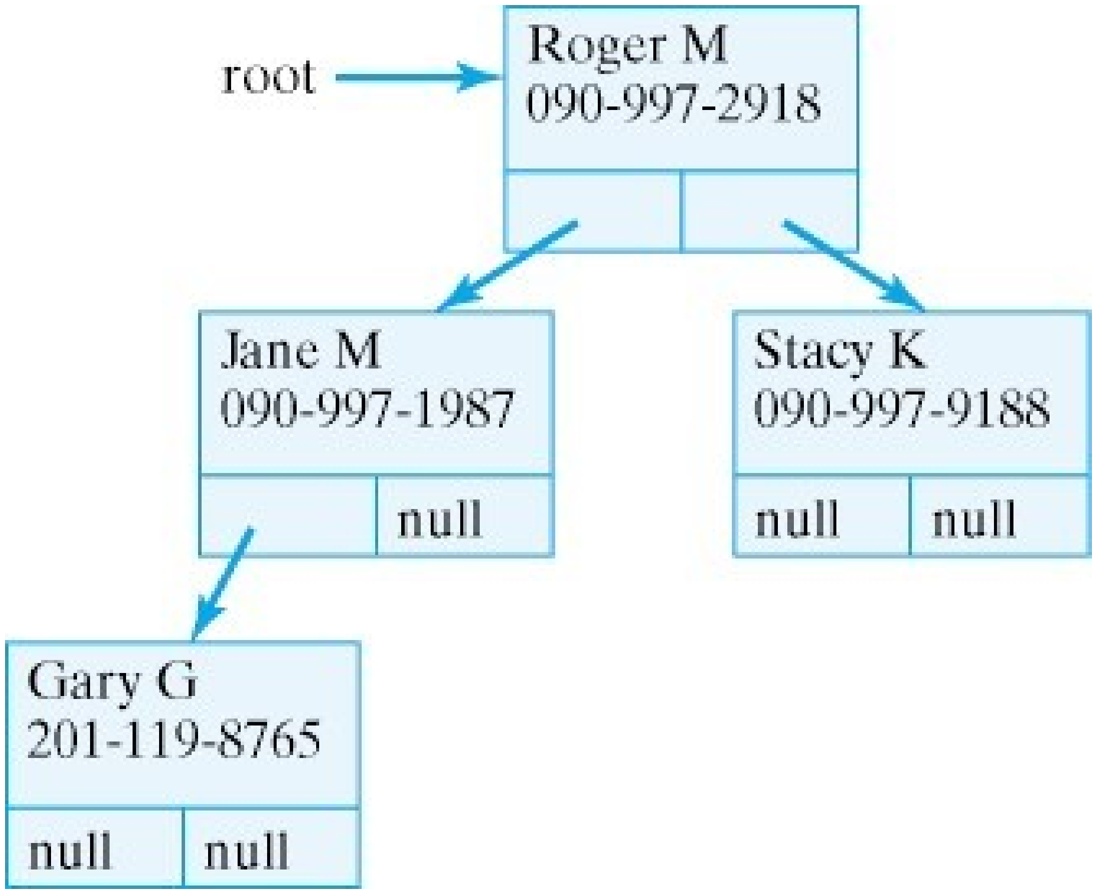
```java
public static void testMap() {
    Map<String,String> theMap = new TreeMap<String,String>();
    // new HashMap<String,String>(); could also be used.
    theMap.put("Roger M", "090-997-2918");
    theMap.put("Jane M", "090-997-1987");
    theMap.put("Stacy K", "090-997-9188");
    theMap.put("Gary G", "201-119-8765");
    theMap.put("Jane M", "090-233-0000");
    System.out.println("Testing TreeMap and Map");
    System.out.println("Stacy K has phone ");
    System.out.println(theMap.get("Stacy K"));
    System.out.println("Jane M has phone ");
    System.out.println(theMap.get("Jane M"));
} // testMap()
```

The Binary Search Tree Data Structure

- Instead of using TreeSet<E> or TreeMap<K,V>, one could implement one's own binary search tree data structure.
- A *binary search tree* is a binary tree in which the ordered data stored at any node is greater than all the data stored in its left subtree and less than all the data stored in its right subtree.
- Data in a binary search tree must be comparable. So, assume it implements the **Comparable** interface.

A Name and Phone Binary Search Tree

- Nodes have left and right references to nodes.



Design for Binary Search Tree Program

```java
public class PhoneTreeNode {
  private String name;
  private String phone;
  private PhoneTreeNode left;
  private PhoneTreeNode Right;
  public PhoneTreeNode(String nam,String pho){ }//constructor
  public void setData(String nam,String pho){ }
  public String getName(){ }
  public boolean contains(String nam,String pho){ }
  public void insert(String nam,String pho){ }
 // other methods
} // PhoneTreeNode class
```

```java
public class PhoneTree {
  private PhoneTreeNode root;
  public PhoneTree(){ }//constructor
  public boolean contains(String nam,String pho){ }
  public void insert(String nam,String pho){ }
// other methods
} // PhoneTree class
```

contains() for PhoneTreeNode and PhoneTree

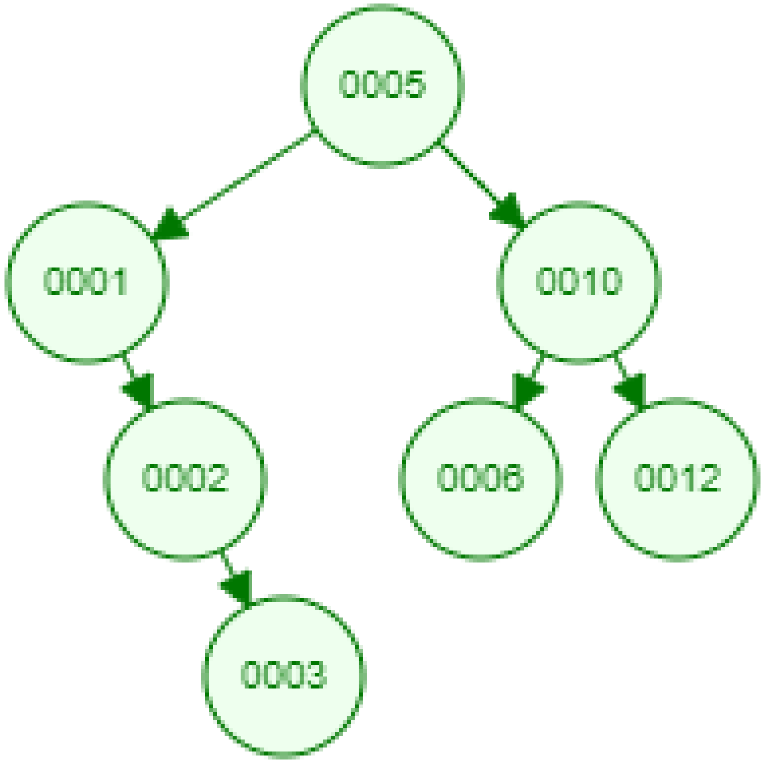
- An implementation of a single method in each class to give a flavor of the Java code.

```java
public boolean contains(String nam,String pho){//in PhoneTreeNode
  if (name.equals(nam)) return true;
  else if (name.compareTo(nam)< 0){// name < nam
    if (right == null) return false;
    else return right.contains(name,pho);
  }else {                           // name > nam
    if (left == null) return false;
    else return left.contains(name,pho);
  } //else
}// contains() in PhoneTreeNode
```

```java
public boolean contains(String nam,String pho){//in PhoneTree
  if (root == null) return false;
  else return root.contains(nam,pho);
} // contains() in PhoneTree
```

Binary Tree Visualization

https://www.cs.usfca.edu/~galles/visualization/BST.html



Technical Terms

- Abstract Data Type (ADT)
- binary search tree
- data structure
- dequeue
- dynamic structure
- enqueue
- first-in/first-out (FIFO)
- generic type
- Java collections framework
- key
- last-in/first-out (LIFO)
- link
- linked list
- list
- pop
- push
- queue
- reference
- self-referential object
- stack
- static structure
- traverse
- value

Summary Of Important Points (part 4)

- In developing list algorithms, it is important to test them thoroughly. Ideally, you should test every possible combination insertions and removals that the list supports. Practically, you should test every independent case of insertions and removals that the list supports.
- An *Abstract Data Type* (ADT) combines two elements: a collection of data, and the operations that can be performed on the data. For a list ADT, the data are the values (**Object**s or **int**s) contained in the nodes that make up the list, and the operations are insertion, removal, and tests of whether the list is empty.

Summary Of Important Points (part 5)

- In designing an ADT, it is important to provide a public interface that can be used to access the ADTs data. The ADTs implementation should not matter to the user and therefore should be hidden. A java class definition, with its **public** and **private** aspects, is perfectly suited to implement and ADT.
- A *stack* is a list that allows insertions and removals only at the front of the list. A stack insertion is called a *push*, and a stack removal is called a *pop*. The first element in a stack is usually called the top of the stack. The **Stack ADT** can easily be defined as a subclass of **List**. Stacks are used for managing method call and returns in programming languages.

Summary Of Important Points (part 6)

- A *queue* is a list that only allows insertions at the rear and removals from the front. A insertion is called an *enqueue*, and a queue removal is called a *dequeue*. The **Queue ADT** can easily be defined as a subclass of **List**. Queues are used for managing various lists of used by the CPU scheduler - such as the ready, waiting, and blocked queues.
- A *binary search tree* is a binary tree in which the ordered data stored at any node is greater than all the data stored in its left subtree and less than all the data stored in its right subtree.