Chapter 15

Linked Data Structures

Introduction to Linked Data Structures

- A linked data structure consists of capsules of data known as nodes that are connected via links
  - Links can be viewed as arrows and thought of as one way passages from one node to another
- In Java, nodes are realized as objects of a node class
- The data in a node is stored via instance variables
- The links are realized as references
  - A reference is a memory address, and is stored in a variable of a class type
  - Therefore, a link is an instance variable of the node class type itself

Java Linked Lists

- The simplest kind of linked data structure is a linked list
- A linked list consists of a single chain of nodes, each connected to the next by a link
  - The first node is called the head node
  - The last node serves as a kind of end marker

Nodes and Links in a Linked List

Display 15.1 Nodes and Links in a Linked List

- The data in a node is stored via instance variables
- The links are realized as references
  - A reference is a memory address, and is stored in a variable of a class type
  - Therefore, a link is an instance variable of the node class type itself
A Simple Linked List Class

• In a linked list, each node is an object of a node class
  – Note that each node is typically illustrated as a box containing one or
    more pieces of data
• Each node contains data and a link to another node
  – A piece of data is stored as an instance variable of the node
  – Data is represented as information contained within the node "box"
  – Links are implemented as references to a node stored in an instance
    variable of the node type
  – Links are typically illustrated as arrows that point to the node to which
    they "link"

A Node Class (Part 1 of 3)

```java
public class Node {
    private String item;
    private int count;
    private Node link;

    public Node() {
        link = null;
        item = null;
        count = 0;
    }

    public Node(String newItem, int newCount, Node linkValue) {
        setData(newItem, newCount);
        link = linkValue;
    }
}
```

A node contains a reference to another node. That reference is the link to the next
node.

A Node Class (Part 2 of 3)

```java
public void setData(String newItem, int newCount) {
    item = newItem;
    count = newCount;
}

public void setLink(Node newLink) {
    link = newLink;
}
```

We will give a better definition of a node class later in this chapter.

A Node Class (Part 3 of 3)

```java
public String getItem() {
    return item;
}

public int getCount() {
    return count;
}

public Node getNext() {
    return link;
}
```
A Simple Linked List Class

- The first node, or start node in a linked list is called the head node
  - The entire linked list can be traversed by starting at the head node and visiting each node exactly once
- There is typically a variable of the node type (e.g., head) that contains a reference to the first node in the linked list
  - However, it is not the head node, nor is it even a node
  - It simply contains a reference to the head node

An Empty List Is Indicated by null

- The head instance variable contains a reference to the first node in the linked list
  - If the list is empty, this instance variable is set to null
  - Note: This is tested using ==, not the equals method
- The linked list constructor sets the head instance variable to null
  - This indicates that the newly created linked list is empty

A Linked List Class (Part 1 of 6)

```java
public class LinkedList {
    private Node head;

    public LinkedList() {
        head = null;
    }

    // Adds a node at the start of the list with the specified data.
    // The added node will be the first node in the list.
    public void addFirst(String itemName, int itemCount) {
        head = new Node(itemName, itemCount, head);
    }
}
```

(continued)
A Linked List Class (Part 2 of 6)

```java
17     // Removes the head node and returns true if the list contained at least one node. Returns false if the list was empty.
18     */
19     public boolean deleteHeadNode() {
20         if (head != null) {
21             head = head.getLink();
22             return true;
23         } else {
24             return false;
25         }
26     }
27     // (continued)
```

A Linked List Class (Part 3 of 6)

```java
31     // Returns the number of nodes in the list.
32     */
33     public int size() {
34         int count = 0;
35         Node position = head;
36         // (continued)
```

A Linked List Class (Part 4 of 6)

```java
39     while (position != null) {  // The last node is indicated by the Link field being equal to null
40         count++;
41         position = position.getLink();
42     }
43     return count;
44     // (continued)
```

A Linked List Class (Part 5 of 6)

```java
50     // Finds the first node containing the target item, and returns a reference to that node. If target is not in the list, null is returned.
51     */
52     private Node1 find(String target) {
53         Node1 position = head;
54         String itemAtPosition = null;
55         while (position != null) {
56             // (continued)
```
A Linked List Class (Part 6 of 6)

Traversing a Linked List

- If a linked list already contains nodes, it can be traversed as follows:
  - Set a local variable equal to the value stored by the head node (its reference)
  - This will provides the location of the first node
  - After accessing the first node, the accessor method for the link instance variable will provide the location of the next node
  - Repeat this until the location of the next node is equal to null

Indicating the End of a Linked List

- The last node in a linked list should have its link instance variable set to null
  - That way the code can test whether or not a node is the last node
  - Note: This is tested using ==, not the equals method
Adding a Node to a Linked List

• The method `add` adds a node to the start of the linked list
  – This makes the new node become the first node on the list
• The variable `head` gives the location of the current first node of the list
  – Therefore, when the new node is created, its `link` field is set equal to `head`
  – Then `head` is set equal to the new node

Deleting the Head Node from a Linked List

• The method `deleteHeadNode` removes the first node from the linked list
  – It leaves the `head` variable pointing to (i.e., containing a reference to) the old second node in the linked list
• The deleted node will automatically be collected and its memory recycled, along with any other nodes that are no longer accessible
  – In Java, this process is called automatic garbage collection
A Linked List Demonstration (Part 2 of 3)

```java
else
    System.out.println("Cantaloupe is NOT on list.");
    list.deleteHeadNode(); // Removes the head node.
if (list.contains("Cantaloupe"))
    System.out.println("Cantaloupe is on list.");
else
    System.out.println("Cantaloupe is NOT on list.");
while (list.deleteLastNode())
    // Empty loop body
    System.out.println("Start of list:");
    list.outputList();
    System.out.println("End of list.");
```

A Linked List Demonstration (Part 3 of 3)

```
Sample Dialogue
List has 3 entries.
Cantaloupe 3
Bonuses 2
Apples 1
Cantaloupe is on list.
Cantaloupe is NOT on list.
Start of list:
End of list.
```

Node Inner Classes

- Note that the linked list class discussed so far is dependent on an external node class
- A linked list or similar data structure can be made self-contained by making the node class an inner class
- A node inner class so defined should be made private, unless used elsewhere
  - This can simplify the definition of the node class by eliminating the need for accessor and mutator methods
  - Since the instance variables are private, they can be accessed directly from methods of the outer class without causing a privacy leak

Pitfall: Privacy Leaks

- The original node and linked list classes examined so far have a dangerous flaw
  - The node class accessor method returns a reference to a node
  - Recall that if a method returns a reference to an instance variable of a mutable class type, then the `private` restriction on the instance variables can be easily defeated
  - The easiest way to fix this problem would be to make the node class a private inner class in the linked list class
A Linked List Class with a Node Inner Class (Part 1 of 6)

```
public class LinkedList2
{
    private class Node
    {
        private String item;
        private Node link;
        public Node()
        {
            item = null;
            link = null;
        }
        public Node(String newItem, Node linkValue)
        {
            item = newItem;
            link = linkValue;
        }
    }
    private Node head;
    public LinkedList2()
    {
        head = null;
    }
    public void addToStart(String item)
    {
        head = new Node(item, head);
    }
    /*
     * Removes the head node and returns true if the list contained at least
     * one node. Returns false if the list was empty.
     */
    public boolean deleteHeadNode()
    {
        if (head != null)
        {
            head = head.link;
            return true;
        }
        else
            return false;
    }
    public int size()
    {
        Node position = head;
        int count = 0;
        while (position != null)
        {
            position = position.link;
            count++;
        }
        return count;
    }
    public boolean contains(String item)
    {
        return (find(item) != null);
    }
}
```

A Linked List Class with a Node Inner Class (Part 2 of 6)

A Linked List Class with a Node Inner Class (Part 3 of 6)

A Linked List Class with a Node Inner Class (Part 4 of 6)
A Generic Linked List

- A linked list can be created whose **Node** class has a **type parameter** T for the type of data stored in the node
  - Therefore, it can hold objects of any class type, including types that contain multiple instance variables
  - The type of the actual object is plugged in for the type parameter T
- For the most part, this class can have the same methods, coded in basically the same way, as the previous linked list example
  - The only difference is that a type parameter is used instead of an actual type for the data in the node
- Other useful methods can be added as well
A Generic Linked List Class
(Part 2 of 9)

```java
private Node<T> head;

public LinkedList()
{
    head = null;
}

/**
 * Adds a node at the start of the list with the specified data.
 * The added node will be the first node in the list.
 */
public void addToStart(T itemData)
{
    head = new Node<T>(itemData, head);
}

(continued)
```

A Generic Linked List Class
(Part 3 of 9)

```java
/**
 * Removes the head node and returns true if the list contained
 * at least one node. Returns false if the list was empty.
 */
public boolean deleteHeadNode()
{
    if (head != null)
    {
        head = head.link;
        return true;
    }
    else
    {
        return false;
    }
}

(continued)
```

A Generic Linked List Class
(Part 4 of 9)

```java
/**
 * Returns the number of nodes in the list.
 */
public int size()
{
    int count = 0;
    Node<T> position = head;
    while (position != null)
    {
        count++;
        position = position.link;
    }
    return count;
}

public boolean contains(T item)
{
    return (find(item) != null);
}

(continued)
```

A Generic Linked List Class
(Part 5 of 9)

```java
/**
 * Finds the first node containing the target item, and returns a
 * reference to that node. If target is not in the list, null is
 * returned.
 */
private Node<T> find(T target)
{
    Node<T> position = head;
    while (position != null)
    {
        if (position.data.equals(target))
        {
            return position;
        }
        position = position.link;
    }
    return null; // target was not found
}

(continued)
```

Type T must have a well-defined equals method for this to work.
A Generic Linked List Class
(Part 6 of 9)

```java
/** A Generic Linked List Class */
80
81  // Finds the first node containing the target and returns a reference to the data in that node. If target is not in the list, null is returned.
82  /**
83  * public T findData(T target)
84  * 
85  * return find(target).data;
86  */
87
88  // public void outputList()
89  {
90  // Node<T> position = head;
91  // while (position != null)
92  {
93  // System.out.println(position.data);
94  // position = position.link;
95  }
96  }
(continued)
```

A Generic Linked List Class
(Part 7 of 9)

```java
97  public boolean isEmpty()
98  {
99        return (head == null);
100    }
101  
102  public void clear()
103  {
104        head = null;
105    }
```

A Generic Linked List Class
(Part 8 of 9)

```java
105  // For two lists to be equal they must contain the same data items in the same order. The equals method of T is used to compare data items.
106  /**
107  * public boolean equals(Object otherObject)
108  *
109  * return false;
110  */
111
112  if (otherObject == null)
113  return false;
114  else if (getClass() != otherObject.getClass())
115  return false;
116  else
117  {
118      LinkedList<T> otherList = (LinkedList<T>)otherObject;
(continued)
```

A Generic Linked List Class
(Part 9 of 9)

```java
118  if (size() != otherList.size())
119  return false;
120  Node<T> position = head;
121  Node<T> otherPosition = otherList.head;
122  while (position != null)
123  {
124      if (!((position.data.equals(otherPosition.data))))
125          return false;
126      position = position.link;
127      otherPosition = otherPosition.link;
128  }
129  return true; // no mismatch was found
130  }
131  }
132  }
```
A Sample Class for the Data in a Generic Linked List (Part 1 of 2)

```java
public class Entry {
    private String item;
    private int count;
    public Entry(String itemData, int countData) {
        item = itemData;
        count = countData;
    }
    public String toString() {
        return (item + " *, " + count);
    }
}
```

(continued)

A Sample Class for the Data in a Generic Linked List (Part 2 of 2)

```java
public boolean equals(Object otherObject) {
    if (otherObject == null)
        return false;
    else if (getClass() != otherObject.getClass())
        return false;
    else {
        Entry otherEntry = (Entry)otherObject;
        return items.equals(otherEntry.items) &&
                (count == otherEntry.count);
    }
}
```

There should be other constructors and methods, including accessor and mutator methods, but we do not use them in this demonstration.

A Generic Linked List Demonstration (Part 1 of 2)

```java
public class GenericLinkedListDemo {
    public static void main(String[] args) {
        LinkedList<Entry> list = new LinkedList<Entry>();
        Entry entry1 = new Entry("Apples", 3);
        list.addToStart(entry1);
        Entry entry2 = new Entry("Bananas", 2);
        list.addToStart(entry2);
        Entry entry3 = new Entry("Cantaloupe", 3);
        list.addToStart(entry3);
        System.out.println("List has "+ list.size()
                + " nodes.");
    }
}
```

(continued)

A Generic Linked List Demonstration (Part 2 of 2)

```java
list.outputList();
System.out.println("End of list.");
}
```

SAMPLE DIALOGUE
List has 3 nodes.
Count of 3
Bananas 2
Apples 1
End of list.
Pitfall: Using **Node** instead of **Node<T>**

- **Note:** This pitfall is explained by example – any names can be substituted for the node **Node** and its parameter **<T>**
- When defining the **LinkedList3<T>** class, the type for a node is **Node<T>**, **not** **Node**
  - If the **<T>** is omitted, this is an error for which the compiler may or may not issue an error message (depending on the details of the code), and even if it does, the error message may be quite strange
  - Look for a missing **<T>** when a program that uses nodes with type parameters gets a strange error message or doesn't run correctly

A Generic Linked List: the **equals** Method

- Like other classes, a linked list class should normally have an **equals** method
- The **equals** method can be defined in a number of reasonable ways
  - Different definitions may be appropriate for different situations
- Two such possibilities are the following:
  1. They contain the same data entries (possibly in different orders)
  2. They contain the same data entries in the same order
- Of course, the type plugged in for **T** must also have redefined the **equals** method

An **equals** Method for the Linked List in Display 15.7 (Part 1 of 2)

```
1 /*
2     For two lists to be equal they must contain the same data items in the same order.
3 */
4 public boolean equals(Object otherObject)
5 {
6       if (otherObject == null)
7           return false;
8       else if (getClass() != otherObject.getClass())
9           return false;
10       else
11           (continued)
```

An **equals** Method for the Linked List in Display 15.7 (Part 2 of 2)

```
14       if (size() != otherList.size())
15           return false;
16           Node otherPosition = otherList.head;
17           while (position != null)
18             {
19               if (!position.item.equals(otherPosition.item))
20                 return false;
21               position = position.link;
22               otherPosition = otherPosition.link;
23             }
24           return true; // a mismatch was not found
25       }
```
Simple Copy Constructors and clone Methods

- There is a simple way to define copy constructors and the clone method for data structures such as linked lists
  - Unfortunately, this approach produces only shallow copies
- The private helping method copyOf is used by both the copy constructor and the clone method
- The copy constructor uses copyOf to create a copy of the list of nodes
- The clone method first invokes its superclass clone method, and then uses copyOf to create a clone of the list of nodes

A Generic Linked List: the private method copyOf

- The private helping method copyOf takes an argument that is a reference to a head node of a linked list, and returns a reference to the head node of a copy of that list
  - It goes down the argument list one node at a time and makes a copy of each node
  - The new nodes are added to the end of the linked list being built
- However, although this produces a new linked list with all new nodes, the new list is not truly independent because the data object is not cloned

A Copy Constructor and clone Method for a Generic Linked List (Part 1 of 6)

```java
public class LinkedList<T> implements Cloneable {
    private class Node {
        private T data;
        private Node next;

        public Node() {
            data = null;
            next = null;
        }

        // Copy constructor
        public Node(T data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    public Node() {
        head = null;
        // Other methods...
    }

    // Clone method
    public Node clone() {
        if (head == null) {
            return null;
        }

        Node cloneHead = null;
        for (Node node = head; node != null; node = node.next) {
            cloneHead = new Node(node.data, cloneHead);
        }
        return cloneHead;
    }
}
```

(continued)
A Copy Constructor and `clone` Method for a Generic Linked List (Part 3 of 6)

```java
/** A Copy Constructor and clone method for a Generic Linked List */
19 20 21 22
23 public LinkedList<T> cloneList(LinkedList<T> otherList) {
24 25 26 27
28 29 30 31
32 33
34 (continued)
```

A Copy Constructor and `clone` Method for a Generic Linked List (Part 4 of 6)

```java
34 35 36
37 public LinkedList<T> clone() {
38 39 40
41 42
43 44
45 46
47 (continued)
```

A Copy Constructor and `clone` Method for a Generic Linked List (Part 5 of 6)

```java
51 52 53
54 55
56 private Node<T> copyOf(Node<T> otherHead) {
57 58
59 60
61 (continued)
```

A Copy Constructor and `clone` Method for a Generic Linked List (Part 6 of 6)

```java
61 62 63
64 65
66 67
68 69 70
71 72
73 74 75
```

Pitfall: The clone Method Is Protected in Object

- It would have been preferable to clone the data belonging to the list being copied in the `copyOf` method as follows:
  ```java
  nodeReference = new Node((T)(position.data).clone(), null);
  ```
- However, this is not allowed, and this code will not compile
  - The error message generated will state that `clone` is protected in `Object`.
  - Although the type used is `T`, not `Object`, any class can be plugged in for `T`.
  - When the class is compiled, all that Java knows is that `T` is a descendent class of `Object`.

Exceptions

- A generic data structure is likely to have methods that throw exceptions.
- Situations such as a `null` argument to the copy constructor may be handled differently in different situations.
  - If this happens, it is best to throw a `NullPointerException`, and let the programmer who is using the linked list handle the exception, rather than take some arbitrary action.
  - A `NullPointerException` is an *unchecked* exception: it need not be caught or declared in a throws clause.

Tip: Use a Type Parameter Bound for a Better clone

- One solution to this problem is to place a bound on the type parameter `T` so that it must satisfy a suitable interface.
  - Although there is no standard interface that does this, it is easy to define one.
- For example, a `PubliclyCloneable` interface could be defined.

Tip: Use a Type Parameter Bound for a Better clone

- Any class that implements the `PubliclyCloneable` interface would have these three properties:
  1. It would implement the `Cloneable` interface because `PubliclyCloneable` extends `Cloneable`.
  2. It would have to implement a public `clone` method.
  3. Its `clone` method would have to make a deep copy.
The PubliclyCloneable Interface

Display 15-13

```java
/*
1 The programmer who defines a class implementing this interface
2 has the responsibility to define clone so it makes a deep copy
3 (in the officially sectioned way.)
*/

6 public interface PubliclyCloneable extends Cloneable
7 {
8   public Object clone();
9 }

Any class that implements PubliclyCloneable must have a public clone method.
Any class that implements PubliclyCloneable automatically implements Cloneable.
```

A Generic Linked List with a Deep Copy clone Method (Part 1 of 8)

Display 15-14

```java
1 public class LinkedList<T extends PubliclyCloneable>
2   implements PubliclyCloneable
3 {
4   private class Node<T>
5   {
6     private T data;
7     private Node<T> link;
8     public Node() 9   {} 10     data = null;
11     link = null;
12   }
```

(continued)

A Generic Linked List with a Deep Copy clone Method (Part 2 of 8)

Display 15-14

```java
13   public Node<T> newNode(T newData, Node<T> linkValue) 14   {
15     data = newData;
16     link = linkValue;
17     } 18   //End of Node<T> inner class
19   private Node<T> head;
20   public LinkedList()
21   {
22     head = null;
23   }
```

(continued)

A Generic Linked List with a Deep Copy clone Method (Part 3 of 8)

Display 15-14

```java
24   // Produces a new linked list, but it is not a true deep copy.
25   // Throws a Nullninterception if other is null
26   //
27   public LinkedList(LinkedList<T> otherList) 28   {
29     if (otherList == null) 30       throw new NullPointerExce
31       if (otherList.head == null) 32         head = null;
33       else 34         head = copyOf(otherList.head);
35       } 36   }
```

(continued)
A Generic Linked List with a Deep Copy clone Method (Part 4 of 8)

```java
public LinkedList<T> clone()
{
    try {
        LinkedList<T> copy = (LinkedList<T>) super.clone();
        if (head == null)
            copy.head = null;
        else
            copy.head = copyOf(head);
        return copy;
    } catch(CloneNotSupportedException e) {
        // This should not happen.
        return null; // To keep the compiler happy.
    }
}
```

(continued)

A Generic Linked List with a Deep Copy clone Method (Part 5 of 8)

```java
// Precondition: otherHead != null
// Returns a reference to the head of a copy of the list
// headed by otherHead. Returns a true deep copy.
private Node<T> copyOf(Node<T> otherHead)
{
    Node<T> position = otherHead; // Moves down other's list.
    Node<T> newHead; // Will point to head of the copy list.
    Node<T> end = null; // Positioned at end of or now growing list.

    // Create first node:
    newHead = new Node<T>(position.data, clone(), null);
    end = newHead;
    position = position.link;

    // Copy rest of list.
    while (position != null)
    {
        end.link = new Node<T>(position.data, clone(), null);
        end = end.link;
        position = position.link;
    }

    return newHead;
}
```

(continued)

A Generic Linked List with a Deep Copy clone Method (Part 6 of 8)

```java
while (position != null)
{
    end.link = new Node<T>(position.data, clone(), null);
    end = end.link;
    position = position.link;
}

return newHead;
```

(continued)

A Generic Linked List with a Deep Copy clone Method (Part 7 of 8)

```java
public boolean equals(Object otherObject)
{
    if (otherObject == null)
        return false;
    else if (getClass() != otherObject.getClass())
        return false;
    else
    {
        LinkedList<T> otherList = (LinkedList<T>) otherObject;
        // The rest of the definition is the same as in Display 15.6. The only difference
        // between this definition of equals and the one in Display 15.6 is that we
        // have replaced the class name LinkedList<T> with LinkedList<T>.

        (continued)
```
A Generic Linked List with a Deep Copy `clone` Method (Part 8 of 8)

Some of the details of the clone method in the previous linked list class may be puzzling, since the following code would also return a deep copy:

```java
public LinkedList<T> clone() {
    return new LInkedList<T>(this);
}
```

However, because the class implements `PubliclyCloneable` which, in turn, extends `Cloneable`, it must implement the `Cloneable` interface as specified in the Java documentation.

A PubliclyCloneable Class (Part 1 of 4)

```java
public class StockItem implements PubliclyCloneable {
    private String name;
    private int number;
    public StockItem() {
        name = null;
        number = 0;
    }
    public StockItem(String nameData, int numberData) {
        name = nameData;
        number = numberData;
    }
    // (continued)
}
```

A PubliclyCloneable Class (Part 2 of 4)

```java
public void setNumber(int newNumber) {
    number = newNumber;
}
public void setName(String newName) {
    name = newName;
}
public String toString() {
    return (name + " + number);
}
```
A PubliclyCloneable Class (Part 3 of 4)

```java
27    public Object clone()
28    {
29        try
30            return super.clone();
31        } catch(CloneNotSupportedException e)
32            //This should not happen.
33            return null; //To keep compiler happy.
34    }
35
36    )
37    
38    (continued)
```

A PubliclyCloneable Class (Part 4 of 4)

```java
39    public boolean equals(Object otherObject)
40    {
41        if (otherObject == null)
42            return false;
43        else if (getClass() != otherObject.getClass())
44            return false;
45        else
46            try
47                StockItem otherItem = (StockItem) otherObject;
48                return (name.equalsIgnoreCase(otherItem.name)
49                    && color.equalsIgnoreCase(otherItem.color)
50                    && size.equalsIgnoreCase(otherItem.size));
51            }
52    
```

Demonstration of Deep Copy clone (Part 1 of 3)

```java
1    public class DeepDemo
2    {
3        public static void main(String[] args)
4        {
5            LinkedList<StockItem> originalList =
6                new LinkedList<StockItem>();
7                originalList.addFirst(new StockItem("red dress", 12));
8                originalList.addFirst(new StockItem("black shoe", 23));
9                LinkedList<StockItem> copyList = originalList.clone();
10                System.out.println("OK, Lists are equal.");
11        }
12        
13        )
14        (continued)
```

Demonstration of Deep Copy clone (Part 2 of 3)

```java
12    System.out.println("Now we change copylist.");
13    StockItem dataEntry =
14        copyList.findItem(new StockItem("red dress", 12));
15    dataEntry.setSize("orange pants");
16    System.out.println("originalList:");
17    originalList.outputList();
18    System.out.println("copylist:");
19    copyList.outputList();
20    System.out.println("Only one list is changed.");
21    System.out.println("Only one list is changed.");
22    
23    )
```

(continued)
Demonstration of Deep Copy clone (Part 3 of 3)

Sample Dialogue
- OK, lists are equal.
- Now we change copylist.
- originalList:
  - block shoe 2
  - red shoes 1
- copylist:
  - block shoe 2
  - orange pants 1
- Only new list is changed

Tip: Cloning is an "All or Nothing" Affair
- If a clone method is defined for a class, then it should follow the official Java guidelines
  - In particular, it should implement the Cloneable interface

Iterators
- A collection of objects, such as the nodes of a linked list, must often be traversed in order to perform some action on each object
  - An iterator is any object that enables a list to be traversed in this way
- A linked list class may be created that has an iterator inner class
  - If iterator variables are to be used outside the linked list class, then the iterator class would be made public
  - The linked list class would have an iterator method that returns an iterator for its calling object
  - Given a linked list named list, this can be done as follows:
    ```
    LinkedList2.List2Iterator i = list.iterator();
    ```

Iterators
- The basic methods used by an iterator are as follows:
  - restart: Resets the iterator to the beginning of the list
  - hasNext: Determines if there is another data item on the list
  - next: Produces the next data item on the list
A Linked List with an Iterator
(Part 1 of 6)

```java
import java.util.NoSuchElementException;

public class LinkedList2 { // This is the same as the class in Display 15.7 and
private class Node // TL;DR except that the List2Iterator inner class
private String item;
private Node link;

// The rest of the definition of the Node inner class is given in Display 15.7.
}

// End of Node inner class (continued)
```

A Linked List with an Iterator
(Part 2 of 6)

```java
// If the list is altered any iterators should invoke restart or
// the iterator's behavior may not be as desired.

public class List2Iterator {
    private Node position;
    private Node previous; // Previous value of position

    public List2Iterator() {
        position = head; // Instance variable head of outer class.
        previous = null;
    }

    public void restart() {
        position = head; // Instance variable head of outer class.
        previous = null;
    }

    (continued)
```

A Linked List with an Iterator
(Part 3 of 6)

```java
public String next() {
    if (!hasNext())
        throw new NoSuchElementException();

    String toReturn = position.item;
    position = position.link;
    return toReturn;

    public boolean hasNext() {
        return (position != null);
    }

    (continued)
```

A Linked List with an Iterator
(Part 4 of 6)

```java
/**
 * Returns the next value to be returned by next().
 * Throws an IllegalStateException if hasNext() is false.
 *
 * @return the next value
 */

public String peek() {
    if (!hasNext())
        throw new IllegalStateException();

    return position.item;

    (continued)
```
A Linked List with an Iterator
(Part 5 of 6)

```java
50 /**
51 * Adds a node before the node at location position. If hasNext() is false, then the node is added to the end of the list.
52 * @param newPosition the position before which to add the new node
53 */
54 public void addBefore(int newPosition)
55 {
56     // Add code to implement this method
57 }
58 
59 /*
60 * Changes the String in the node at location position. If the new String is null, changes the String to "null".
61 * @param newPosition
62 * @param newValue the new String
63 */
64 public void changeString(int newPosition, String newValue)
65 {
66     // Add code to implement this method
67 }
(continued)
```

A Linked List with an Iterator
(Part 6 of 6)

```java
62 /**
63 * Deletes the node at location position and moves position to the "next" node. If the list is empty;
64 * throws an IllegalStateException if the list is empty.
65 */
66 public void delete()
67 {
68     // Add code to implement this method
69 }
70 
71 private Node head; // Linked list head
72 
73 public List2Iterator iterator()
74 {
75     return new List2Iterator();
76 }
77 
78 } // End of List2Iterator inner class
(continued)
```

Using an Iterator
(Part 1 of 6)

```java
4 public class IteratorDemo
5 {
6     // Add code to implement this method
7 }
8 
9 // Linked list2 list = new LinkedList2();
10 // LinkedList2.List2Iterator i = list.iterator();
11 i.addFirst("shoes");
12 i.addFirst("orange juice");
13 i.addFirst("coat");
(continued)
```

Using an Iterator
(Part 2 of 6)

```java
18 System.out.println("list contains:");
19 i.restart();
20 while (i.hasNext())
21 System.out.println(i.next());
22 System.out.println();
23 
24 i.restart();
25 i.next();
26 System.out.println("Will delete the node for "+ i.peek());
27 i.delete();
28 
29 } // End of IteratorDemo
(continued)
```
Using an Iterator (Part 3 of 6)

```java
Display 15.18 Using an iterator

20 System.out.println("List now contains:");
21 i.restart();
22 while(i.hasNext())
23 System.out.println(i.next());
24 i.restart();
25 i.next();
26 System.out.println("will add one node before ", + i.peek());
27 i.addHere("socks");
28 System.out.println("List now contains:");
29 i.restart();
30 while(i.hasNext())
31 System.out.println(i.next());
```

(continued)

Using an Iterator (Part 4 of 6)

```java
Display 15.18 Using an iterator

32 System.out.println();
33 System.out.println("Changing all items to credit card.");
34 i.restart();
35 while(i.hasNext())
36 {
37 i.changeHere("credit card");
38 i.next();
39 }
40 System.out.println();
41 System.out.println("List now contains:");
42 i.restart();
43 while(i.hasNext())
44 System.out.println(i.next());
45 System.out.println();
47 }
```

(continued)

Using an Iterator (Part 5 of 6)

```java
SAMPLE DIALOGUE
```

```
<table>
<thead>
<tr>
<th>List contains:</th>
</tr>
</thead>
<tbody>
<tr>
<td>coat</td>
</tr>
<tr>
<td>orange juice</td>
</tr>
<tr>
<td>shoes</td>
</tr>
</tbody>
</table>

Will delete the node for orange juice
```

```
List now contains:
| coat |
| shoes|
```

(continued)

Using an Iterator (Part 6 of 6)

```java
Display 15.18 Using an iterator

Will add one node before shoes
```

```
List now contains:
| coat |
| shoes|
```

Changing all items to credit card.
```
List now contains:
| credit card |
| credit card |
```

(continued)
The Java Iterator Interface

- Java has an interface named **Iterator** that specifies how Java would like an iterator to behave
  - Although the iterators examined so far do not satisfy this interface, they could be easily redefined to do so

Adding and Deleting Nodes

- An iterator is normally used to add or delete a node in a linked list
- Given iterator variables `position` and `previous`, the following two lines of code will delete the node at location `position`:
  ```java
  previous.link = position.link;
  position = position.link;
  ```
  - Note: `previous` points to the node before `position`

Deleting a Node (Part 1 of 2)

1. Existing list with the iterator positioned at "shoes"

```
head ────> "orange juice" ────> "shoes" ────> "socks" ────> null
```

2. Bypass the node at `position` from `previous`

```
head ────> "orange juice" ────> "socks" ────> null
```

Deleting a Node (Part 2 of 2)

3. Update `position` to reference the next node

```
head ────> "orange juice" ────> "shoes" ────> "socks" ────> null
```

Since no variable references the node "shoes" Java will automatically recycle the memory allocated for it.

4. Same picture with deleted node not shown

```
head ────> "orange juice" ────> "socks" ────> null
```
Adding and Deleting Nodes

- Note that Java has automatic garbage collection
  - In many other languages the programmer has to keep track of deleted nodes and explicitly return their memory for recycling
  - This procedure is called explicit memory management
- The iterator variables position and previous can be used to add a node as well
  - previous will point to the node before the insertion point, and position will point to the node after the insertion point

```java
Node temp = new Node(newData, position);
previous.link = temp;
```

Adding a Node between Two Nodes (Part 1 of 2)

1. Existing list with the iterator positioned at “shoes”

2. Create new Node with "socks" linked to "shoes"

```java
temp = new Node(newData, position); // newData is "socks"
```

Adding a Node between Two Nodes (Part 2 of 2)

3. Make previous link to the Node temp

```java
previous.link = temp;
```

4. Picture redrawn for clarity, but structurally identical to picture 3

Variations on a Linked List

- An ordinary linked list allows movement in one direction only
- However, a doubly linked list has one link that references the next node, and one that references the previous node
- The node class for a doubly linked list can begin as follows:

```java
private class TwoWayNode
{
    private String item;
    private TwoWayNode previous;
    private TwoWayNode next;
    ...
}
```
- In addition, the constructors and methods in the doubly linked list class would be modified to accommodate the extra link
A Doubly Linked List

Adding a Node to the Front of a Doubly Linked List

1. Existing list

2. Create new TwoWayNode linked to "coat"

3. Set backward link and set new head

4. Picture redrawn for clarity with the "shoes" node removed since there are no longer references pointing to this node.

Deleting a Node from a Doubly Linked List (1 of 2)

1. Existing list with an iterator referencing "shoes"

2. Bypass the "shoes" node from the next link of the previous node

Deleting a Node from a Doubly Linked List (2 of 2)

3. Bypass the "shoes" node from the previous link of the next node and move position off the deleted node

4. Picture redrawn for clarity with the "shoes" node removed since there are no longer references pointing to this node.
Inserting a Node Into a Doubly Linked List (1 of 2)

1. Existing list with an iterator referencing "shoes"

   ![Diagram](image1)

2. Create new TwoWayNode with previous linked to "coat" and next to "shoes"

   `TwoWayNode temp = new TwoWayNode(newData, position.previous, position);`
   
   ![Diagram](image2)

Inserting a Node Into a Doubly Linked List (2 of 2)

3. Set next link from "coat" to the new node of "shirt"

   `position.previous.next = temp;`
   
   ![Diagram](image3)

4. Set previous link from "shoes" to the new node of "shirt"

   `position.previous = temp;`
   
   ![Diagram](image4)

The Stack Data Structure

- A stack data structure is not necessarily a linked data structure, but can be implemented as one
  - A stack is a data structure that removes items in the reverse order of which they were inserted (LIFO: Last In First Out)
  - A linked list that inserts and deletes only at the head of the list is a stack

The Queue Data Structure

- A queue is a data structure that handles data in a first-in/first-out fashion (FIFO) like a line at a bank
  - Customers add themselves to the end of the line and are served from the front of the line
- A queue can be implemented with a linked list
  - However, a queue needs a pointer at both the head and tail (the end) of the linked list
  - Nodes are removed from the front (head end), and are added to the back (tail end)
A Queue Class (Part 1 of 5)

```java
public class Queue
{
    private class Node
    {
        private String item;
        private Node link;
        public Node()
        {
            item = null;
            link = null;
        }
    }
}
```

A Queue Class (Part 2 of 5)

```java
public Node(String newItem, Node linkValue)
{
    item = newItem;
    link = linkValue;
}

private Node front;
private Node back;

public Queue()
{
    front = null;
    back = null;
}
```

A Queue Class (Part 3 of 5)

```java
/**
 * Adds a String to the back of the queue.
 */
public void addToBack(String itemValue)

public boolean isEmpty()
{
    return (front == null);
}

public void clear()
{
    front = null;
    back = null;
}
```

A Queue Class (Part 4 of 5)

```java
/**
 * Returns the String in the front of the queue. 
 * Returns null if queue is empty. 
 */
public String whoIsNext()
{
    if (front == null)
        return null;
    else
        return front.item;
}
```

(continued)
A Queue Class (Part 5 of 5)

```java
public boolean removeFront()
{
    if (front == null)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

Demonstration of the Queue Class (Part 1 of 2)

```java
public class QueueDemo
{
    public static void main(String[] args)
    {
        queue = new queue();
        q.addTidlock(“Tom”);
        q.removeTidlock(“Dick”);
        q.removeTidlock(“Harriet”);
    }
}
```

Demonstration of the Queue Class (Part 2 of 2)

```java
while(!q.isEmpty())
{
    System.out.println(q,whoIsNext());
    q.removeFront();
}
System.out.println(“The queue is empty.”);
```

Running Times

- How fast is program?
  - "Seconds"?
  - Consider: large input? .. small input?
- Produce "table"
  - Based on input size
  - Table called "function" in math
    - With arguments and return values!
  - Argument is input size:
    - T(10), T(10,000), ...
- Function T is called "running time"
Table for Running Time Function:

**Display 15.31 Some Values of a Running Time Function**

<table>
<thead>
<tr>
<th>INPUT SIZE</th>
<th>RUNNING TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 numbers</td>
<td>2 seconds</td>
</tr>
<tr>
<td>100 numbers</td>
<td>2.5 seconds</td>
</tr>
<tr>
<td>1,000 numbers</td>
<td>10 seconds</td>
</tr>
<tr>
<td>10,000 numbers</td>
<td>2.5 minutes</td>
</tr>
</tbody>
</table>

Consider Sorting Program

- Faster on smaller input set?
  - Perhaps
  - Might depend on "state" of set
    - "Mostly" sorted already?
- Consider worst-case running time
  - T(N) is time taken by "hardest" list
    - List that takes longest to sort

Counting Operations

- T(N) given by formula, such as:
  - T(N) = 5N + 5
    - "On inputs of size N program runs for 5N + 5 time units"
- Must be "computer-independent"
  - Doesn’t matter how "fast" computers are
  - Can’t count "time"
  - Instead count "operations"

Counting Operations Example

- int i = 0;
  Boolean found = false;
  while ((i < N) && !found)
    if (a[i] == target)
      found = true;
    else
      i++;

- 5 operations per loop iteration:
  <, &&, i[], ==, ++
- After N iterations, final three: <, &&, !
- So: 6N+5 operations when target not found
Big-O Notation

• Recall: 6N+5 operations in "worst-case"
• Expressed in "Big-O" notation
  – Some constant "c" factor where c(6N+5) is actual running time
    • c different on different systems
  – We say code runs in time O(6N+5)
  – But typically only consider "highest term"
    • Term with highest exponent
  – O(N) here

Big-O Terminology

• Linear running time:
  – O(N)—directly proportional to input size N
• Quadratic running time:
  – O(N^2)
• Logarithmic running time:
  – O(log N)
    • Typically "log base 2"
    • Very fast algorithms!

Display 15.32
Comparison of Running Times

Efficiency of Linked Lists

• Find method for linked list
  – May have to search entire list
  – On average would expect to search half of the list, or n/2
  – In big-O notation, this is O(n)
• Adding to a linked list
  – When adding to the start we only reassign some references
  – Constant time or O(1)
Hash Tables

• A hash table or hash map is a data structure that efficiently stores and retrieves data from memory
• Here we discuss a hash table that uses an array in combination with singly linked lists
• Uses a hash function
  – Maps an object to a key
  – In our example, a string to an integer

Simple Hash Function for Strings

• Sum the ASCII value of every character in the string and then compute the modulus of the sum using the size of the fixed array.

```
private int computeHash(String s)
{
    int hash = 0;
    for (int i = 0; i < s.length(); i++)
    {
        hash += s.charAt(i);
    }
    return hash % SIZE; // SIZE = 10 in example
}
```

Example: "dog" = ASCII 100, 111, 103
Hash = (100 + 111 + 103) % 10 = 4

Hash Table Idea

• Storage
  – Make an array of fixed size, say 10
  – In each array element store a linked list
  – To add an item, map (i.e. hash) it to one of the 10 array elements, then add it to the linked list at that location

• Retrieval
  – To look up an item, determine its hash code then search the linked list at the corresponding array slot for the item

Constructing a Hash Table (1 of 2)

1. Existing hash table initialized with ten empty linked lists

```
hashArray = new LinkedList[size]; // SIZE = 10
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
</tr>
</tbody>
</table>

2. After adding "cat" with hash of 2

```
hashArray | empty | empty | empty | empty | empty | empty | empty | empty | empty |
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>null</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
</tr>
</tbody>
</table>

"cat"
Constructing a Hash Table (2 of 2)

3. After adding "dog" with hash of 4 and "bird" with hash of 7

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
</tr>
<tr>
<td>cat</td>
<td>dog</td>
<td>bird</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. After adding "turtle" with hash of 2 – collision and chained to linked list with "cat"

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
<td>empty</td>
</tr>
<tr>
<td>cat</td>
<td>dog</td>
<td>bird</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A Hash Table Class (1 of 3)

```java
public class HashTable
{
    // Uses the generic LinkedList2 class from Display 15.7
    private LinkedList2[] hashArray;
    private static final int SIZE = 10;

    public HashTable()
    {
        hashArray = new LinkedList2[SIZE];
        for (int i=0; i < SIZE; i++)
            hashArray[i] = new LinkedList2();
    }

    private int computeHash(String s)
    {
        int hash = 0;
        for (int i = 0; i < s.length(); i++)
            hash += s.charAt(i);
        return hash % SIZE;
    }

    public boolean containsString(String target)
    {
        int hash = computeHash(target);
        LinkedList2 list = hashArray[hash];
        if (list.contains(target))
            return true;
        return false;
    }

    public void put(String s)
    {
        int hash = computeHash(s);
        LinkedList2 list = hashArray[hash];
        if (list.contains(s))
            return false;
        list.addToStart(s);
    }
} // End HashTable class
```

A Hash Table Class (2 of 3)

```java
/**
 * Returns true if the target is in the hash table,
 * false if it is not.
 */
public boolean containsString(String target)
{
    int hash = computeHash(target);
    LinkedList2 list = hashArray[hash];
    if (!list.contains(target))
        return false;
}
```

A Hash Table Class (3 of 3)

```java
/**
 * Stores or puts string s into the hash table
 */
public void put(String s)
{
    int hash = computeHash(s); // Get hash value
    LinkedList2 list = hashArray[hash];
    if (list.contains(s))
    {
        // Only add the target if it's not already
        // on the list.
        hashArray[hash].addToStart(s);
    }
}
```

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.
Hash Table Demonstration (1 of 2)

```java
public class HashTableDemo {
    public static void main(String[] args) {
        HashTable h = new HashTable();
        System.out.println("Adding dog, cat, turtle, bird");
        h.put("dog");
        h.put("cat");
        h.put("turtle");
        h.put("bird");
        System.out.println("Contains dog? " + h.containsString("dog"));
        System.out.println("Contains cat? " + h.containsString("cat");
        System.out.println("Contains turtle? " + h.containsString("turtle");
        System.out.println("Contains bird? " + h.containsString("bird");
    }
}
```

Hash Table Demonstration (2 of 2)

```java
System.out.println("Contains fish? " + h.containsString("fish");
System.out.println("Contains cow? " + h.containsString("cow");
```

**SAMPLE DIALOGUE**

Adding dog, cat, turtle, bird
Contains dog? true
Contains cat? true
Contains turtle? true
Contains bird? true
Contains fish? false
Contains cow? false

Hash Table Efficiency

- **Worst Case**
  - Every item inserted into the table has the same hash key, the find operation may have to search through all items every time (same performance as a linked list, O(n) to find)

- **Best Case**
  - Every item inserted into the table has a different hash key, the find operation will only have to search a list of size 1, very fast, O(1) to find.

- Can decrease the chance of collisions with a better hash function

- Tradeoff: Lower chance of collision with bigger hash table, but more wasted memory space

Set Template Class

- A set is a collection of elements in which no element occurs more than once
- We can implement a simple set that uses a linked list to store the items in the set
- Fundamental set operations we will support:
  - Add
  - Contains
  - Union
  - Intersection
Sets Using Linked Lists

A Set Class (1 of 5)

```java
1 // Uses a linked list as the internal data structure
2 // to store items in a set.
3 public class Set<T>
4 {
5     // private class Node<T>
6     private class Node<T> {
7         private T data;
8         private Node<T> link;
9         public Node() {
10             data = null;
11             link = null;
12         }
13         public Node(T newData, Node<T> linkValue) {
14             data = newData;
15             link = linkValue;
16         }
17     }
18     // End of Node<T> inner class
19     private Node<T> head;
20 }
```

A Set Class (2 of 5)

```java
21 public Set()
22 {
23     head = null;
24 }  // Add a new item to the set. If the item
25     // is already in the set, false is returned,
26     // otherwise true is returned.
27     public boolean add(T newItem) {
28         if (!contains(newItem)) {  // contains
29             head = new Node<T>(newItem, head);
30             return true;
31         } else {  //target was not found
32             return false;
33         }
34     }
35 
36 public boolean contains(T item) {
37     Node<T> position = head;
38     while (position != null) {
39         if (item.equals(position.data)) {
40             return true;
41         }
42         position = position.link;
43     }
44     return false;  //target was not found
45 }
46 public void output() {
47     Node position = head;
48     while (position != null) {
49         System.out.print(position.data.toString() + " ");
50         position = position.link;
51     }
52     System.out.println();
53 }
```

A Set Class (3 of 5)

```java
59 public boolean contains(T item) {
60     Node<T> position = head;
61     while (position != null) {
62         if (item.equals(position.data)) {
63             return true;
64         }
65         position = position.link;
66     }
67     return false;  //target was not found
68 }
```
A Set Class (4 of 5)

```java
/**
 * Returns a new set that is the union of this set and the input set.
 */
public Set<T> union(Set<T> otherSet) {
    Set<T> unionSet = new Set<T>();
    // Copy this set to unionSet
    Node<T> position = head;
    while (position != null) {
        unionSet.add(position.data);
        position = position.link;
    }
    // Copy otherSet items to unionSet.
    // The add method eliminates any duplicates.
    position = otherSet.head;
    while (position != null) {
        unionSet.add(position.data);
        position = position.link;
    }
    return unionSet;
}
```

A Set Class (5 of 5)

```java
/**
 * Returns a new that is the intersection of this set and the input set.
 */
public Set<T> intersection(Set<T> otherSet) {
    Set<T> interSet = new Set<T>();
    // Copy only items in both sets
    Node<T> position = head;
    while (position != null) {
        if (otherSet.contains(position.data)) {
            interSet.add(position.data);
            position = position.link;
        }
    }
    return interSet;
}
```

A Set Class Demo (1 of 3)

```java
class SetDemo {
    public static void main(String[] args) {
        // Round things
        Set round = new Set<String>();
        // Green things
        Set green = new Set<String>();
        // Add some data to both sets
        round.add("peas");
        round.add("pie");
        round.add("grapes");
        green.add("peas");
        green.add("grapes");
        green.add("garden hose");
        green.add("grass");
        System.out.println("Contents of set round: ");
        round.output();
        System.out.println("Contents of set green: ");
        green.output();
        System.out.println("ball in set round? "+
            round.contains("ball"));
        System.out.println("ball in set green? "+
            green.contains("ball"));
        System.out.println("ball and peas in same set? "+
            ((round.contains("ball") &&
            round.contains("peas")) ||
            (green.contains("ball") &&
            green.contains("peas"))));
        System.out.println("pie and grass in same set? "+
            ((round.contains("pie") &&
            round.contains("grass")) ||
            (green.contains("pie") &&
            green.contains("grass"))));
```

A Set Class Demo (2 of 3)

```java
    System.out.println("ball in set round? " +
        round.contains("ball"));
    System.out.println("ball in set green? " +
        green.contains("ball"));
    System.out.println("ball and peas in same set? " +
        ((round.contains("ball") &&
        round.contains("peas")) ||
        (green.contains("ball") &&
        green.contains("peas"))));
    System.out.println("pie and grass in same set? " +
        ((round.contains("pie") &&
        round.contains("grass")) ||
        (green.contains("pie") &&
        green.contains("grass"))));
```
A Set Class Demo (3 of 3)

```java
37 System.out.print("Union of green and round: ");
38 round.union(green).output();
39 System.out.print("Intersection of green and round: ");
40 round.intersection(green).output();
41 }
```

SAMPLE DIALOGUE

Contents of set round:
grapes pie ball peas
Contents of set green:
Grass garden hose grapes peas

ball in set round?  true
ball in set green?  false
ball and peas in same set?  true
pie and grass in same set?  false
Union of green and round: garden hose grass peas ball pie grapes
Intersection of green and round: peas grapes

Trees

• Trees are a very important and widely used data structure
• Like linked lists, they are a structure based on nodes and links, but are more complicated than linked lists
  – All trees have a node called the root
  – Each node in a tree can be reached by following the links from the root to the node
  – There are no cycles in a tree: Following the links will always lead to an "end"

Trees

A binary tree is the most common kind of tree
  – Each node in a binary tree has exactly two link instance variables
  – A binary tree must satisfy the Binary Search Tree Storage Rule
• The root of the tree serves a purpose similar to that of the instance variable head in a linked list
  – The node whose reference is in the root instance variable is called the root node
• The nodes at the "end" of the tree are called leaf nodes
  – Both of the link instance variables in a leaf node are null
A Binary Tree (Part 2 of 2)

```java
class IntTree {  
    public class IntTreeNode {  
        private int data;  
        private IntTreeNode leftLink;  
        private IntTreeNode rightLink;  
    }  

    private IntTreeNode root;  
    // The methods and other inner classes are not shown.
}
```

Binary Search Tree Storage Rule

1. All the values in the left subtree must be less than the value in the root node.
2. All the values in the right subtree must be greater than or equal to the value in the root node.
3. This rule is applied recursively to each of the two subtrees.
   (The base case for the recursion is an empty tree)

Tree Properties

- Note that a tree has a recursive structure
  - Each tree has two subtrees whose root nodes are the nodes pointed to by the `leftLink` and `rightLink` of the root node.
  - This makes it possible to process trees using recursive algorithms.
- If the values of a tree satisfying the Binary Search Tree Storage Rule are output using Inorder Processing, then the values will be output in order from smallest to largest.

Preorder Processing

1. Process the data in the root node.
2. Process the left subtree.
3. Process the right subtree.
Inorder Processing

1. Process the left subtree
2. Process the data in the root node
3. Process the right subtree

Postorder Processing

1. Process the left subtree
2. Process the right subtree
3. Process the data in the root node

A Binary Search Tree for Integers (Part 1 of 6)

```java
1 /**
2 * Class invariant: The tree satisfies the binary search tree storage rule.
3 */
4 public class IntTree
5 {
6     private static class IntTreeNode
7     {
8         private int data;
9         private IntTreeNode leftlink;
10        private IntTreeNode rightLink;
11     }
12     private int data;
13     private IntTreeNode leftlink;
14     private IntTreeNode rightLink;
15     private static int max = Integer.MAX_VALUE;
16     private static int min = Integer.MIN_VALUE;
17     private int root = null;
18     public IntTreeNode(int newdata, IntTreeNode newleftlink, 
19           IntTreeNode newRightLink)
20     {
21         data = newdata;
22         leftLink = newleftlink;
23         rightLink = newRightLink;
24         this.
25     }
26     public IntTree()
27     {
28         root = null;
29         // this class should have more methods. this is just
30         // a sample of possible methods.
31     }
32     public IntTree(IntTreeNode root, int newdata)
33         IntTreeNode newleftlink, 
34         IntTreeNode newRightLink)
35         root = null;
36         // this class should have more methods. this is just
37         // a sample of possible methods.
38         // (continued)
39     }
40     `
A Binary Search Tree for Integers (Part 3 of 6)

```java
25  public void add(int item)
26  {
27      root = insertInSubtree(item, root);
28  }
29  
30  public boolean contains(int item)
31  {
32      return isInSubtree(item, root);
33  }
34  
35  public void showElements()
36  {
37      showElementsInSubtree(root);
38  }
```

(continued)

A Binary Search Tree for Integers (Part 4 of 6)

```java
37  // Returns the root node of a tree that is the tree with root node
38  // subTreeRoot, but with a new node added that contains item.
39  //
40  private static IntTreeNode insertInSubtree(int item,
41  IntTreeNode subTreeRoot)
42  {
43      if (subTreeRoot == null)
44          return new IntTreeNode(item, null, null);
45      else if (item < subTreeRoot.data)
46          subTreeRoot.leftLink = insertInSubtree(item, subTreeRoot.leftLink);
47      else // item >= subTreeRoot.data
48          return subTreeRoot;
49  }  
50  
51  //Uses inorder traversal.
52  //
53  private static void showElementsInSubtree(IntTreeNode subTreeRoot)
54  {
55      if (subTreeRoot != null)
56      {
57          showElementsInSubtree(subTreeRoot.leftLink);
58          System.out.print(subTreeRoot.data + " ");
59          showElementsInSubtree(subTreeRoot.rightLink);
60      }  //else do nothing. Empty tree has nothing to display.
61  }
```

(continued)

A Binary Search Tree for Integers (Part 5 of 6)

```java
61  
62  private static boolean isInSubtree(int item, IntTreeNode subTreeRoot)
63  {
64      if (subTreeRoot == null)  
65          return false;
66      else if (subTreeRoot.data == item)
67          return true;
68      else if (item < subTreeRoot.data)
69          return isInSubtree(item, subTreeRoot.leftLink);
70      else // item >= subTreeRoot.data
71          return isInSubtree(item, subTreeRoot.rightLink);
72  }
```

(continued)

A Binary Search Tree for Integers (Part 6 of 6)
Demonstration Program for the Binary Search Tree (Part 1 of 3)

Demonstration Program for the Binary Search Tree

```java
1 import java.util.Scanner;
2 public class BinarySearchTreeDemo
3 {
4     public static void main(String[] args)
5     {
6         Scanner keyboard = new Scanner(System.in);
7         IntTree tree = new IntTree();
8         System.out.println("Enter a list of nonnegative integers.");
9         System.out.println("Place a negative integer at the end.");
10        int next = keyboard.nextInt();
11        while (next >= 0)
12        {
13            tree.add(next);
14            next = keyboard.nextInt();
15        }
16        System.out.println("In sorted order");
17        tree.showElements();
18    }
19 }
```

Efficiency of Binary Search Trees

- A binary search tree that is as short as possible can be processed most efficiently
  - A short tree is one where all paths from root to a leaf differ by at most one node
- When this is so, the search method `isSubtree` is about as efficient as the binary search on a sorted array
  - Its worst-case running time is $O(\log n)$, where $n$ is the number of nodes in the tree

---

**SAMPLE DIALOGUE**

Enter a list of nonnegative integers.
Place a negative integer at the end.
40 30 20 10 9 8 7 6 5 4 3 2 1

In sorted order:
10 11 20 22 30 33 40 44
Efficiency of Binary Search Trees

• As a tree becomes more tall and thin, this efficiency falls off
  – In the worst case, it is the same as that of searching a linked list with the same number of nodes
• Maintaining a tree so that it remains short and fat, as nodes are added, is known as balancing the tree
  – A tree maintained in this manner is called a balanced tree