ABSOLUTE JAVA™

SIXTH EDITION
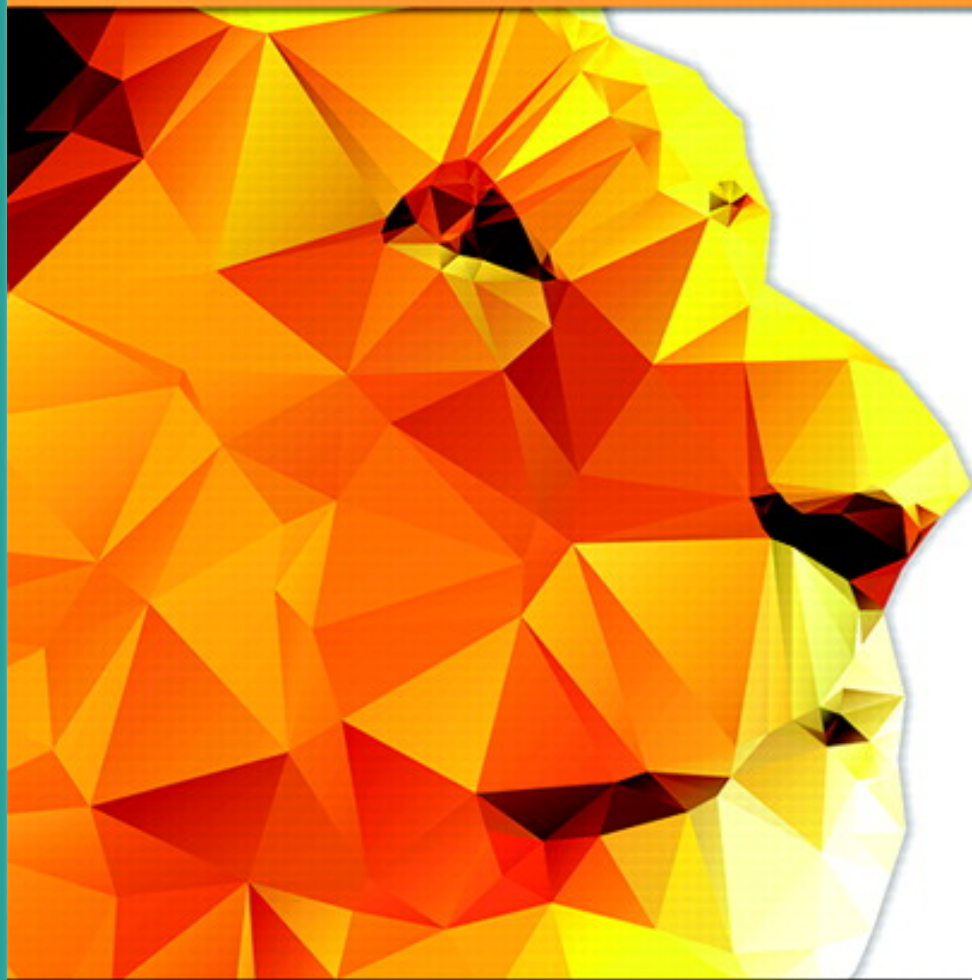
Walter Savitch

Chapter 15

Linked Data Structures

PEARSON

# 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**

head

"rolls"

10

"jam"

3

"milk"

1

"tea"

2

end marker
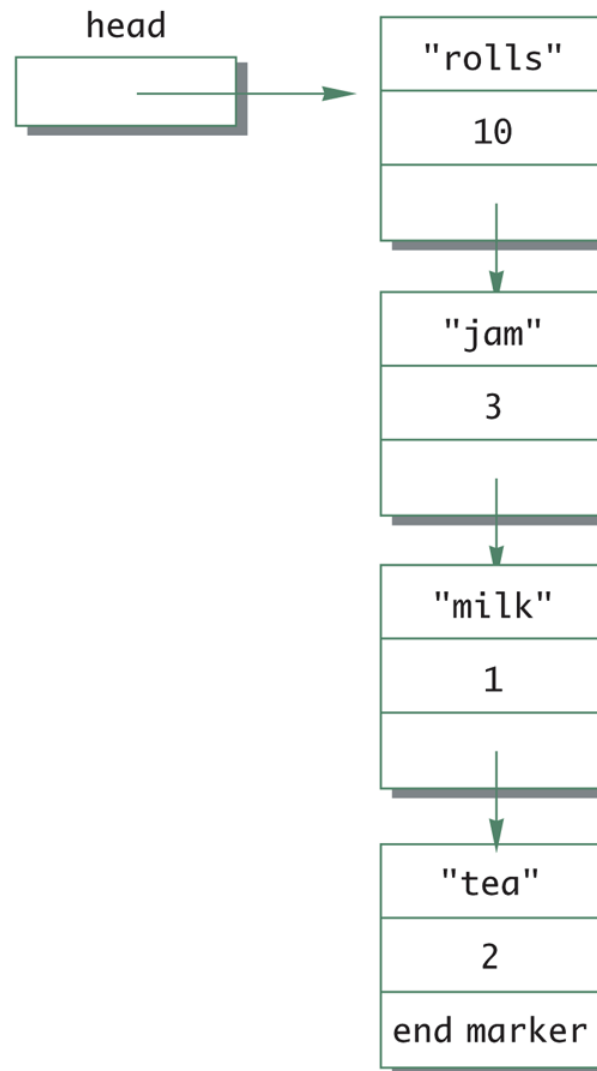
A Simple Linked List Class

- 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)

Display 15.2   A Node Class

```java
public class Node1
{
    private String item;
    private int count;
    private Node1 link;

    public Node1()
    {
        link = null;
        item = null;
        count = 0;
    }

    public Node1(String newItem, int newCount, Node1 linkValue)
    {
        setData(newItem, newCount);
        link = linkValue;
    }
```

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

We will define a number of node classes so we numbered the names, as in Node1.

(continued)

A Node Class (Part 2 of 3)

Display 15.2   A Node Class

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

public void setLink(Node1 newLink)
{
    link = newLink;
}
```

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

(continued)

A Node Class (Part 3 of 3)

Display 15.2   A Node Class

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

public int getCount()
{
    return count;
}

public Node1 getLink()
{
    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

A Simple Linked List Class

- A linked list object contains the variable **head** as an instance variable of the class
- A linked list object does not contain all the nodes in the linked list directly
  - Rather, it uses the instance variable **head** to locate the head node of the list
  - The head node and every node of the list contain a link instance variable that provides a reference to the next node in the list
  - Therefore, once the head node can be reached, then every other node in the list can be reached

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)

Display 15.3   A Linked List Class

```
1    public class LinkedList1
2    {
3        private Node1 head;
4
5        public LinkedList1()
6        {
7            head = null;
8        }
9        /**
10        Adds a node at the start of the list with the specified data.
11        The added node will be the first node in the list.
12        */
13        public void addToStart(String itemName, int itemCount)
14        {
15            head = new Node1(itemName, itemCount, head);
16        }
```

We will define a better linked list class later in this chapter.

(continued)

A Linked List Class (Part 2 of 6)

Display 15.3   A Linked List Class

```
17        /**
18          Removes the head node and returns true if the list contained at least
19          one node. Returns false if the list was empty.
20        */
21        public boolean deleteHeadNode()
22        {
23            if (head != null)
24            {
25                head = head.getLink();
26                return true;
27            }
28            else
29                return false;
30        }
```

(continued)

15-13

A Linked List Class (Part 3 of 6)

Display 15.3   **A Linked List Class**

```
31        /**
32          Returns the number of nodes in the list.
33        */
34        public int size()
35        {
36            int count = 0;
37            Node1 position = head;
38
```

(continued)

A Linked List Class (Part 4 of 6)

Display 15.3   A Linked List Class

```
39              while (position != null)
40              {
41                  count++;
42                  position = position.getLink();
43              }
44              return count;
45          }

46      public boolean contains(String item)
47      {
48          return (find(item) != null);
49      }
```

The last node is indicated by the `link` *field being equal to* `null`.

(continued)

A Linked List Class (Part 5 of 6)

Display 15.3   A Linked List Class

```
50        /**
51          Finds the first node containing the target item, and returns a
52          reference to that node. If target is not in the list, null is returned.
53        */
54        private Node1 find(String target)
55        {
56            Node1 position = head;
57            String itemAtPosition;
58            while (position != null)
59            {
```
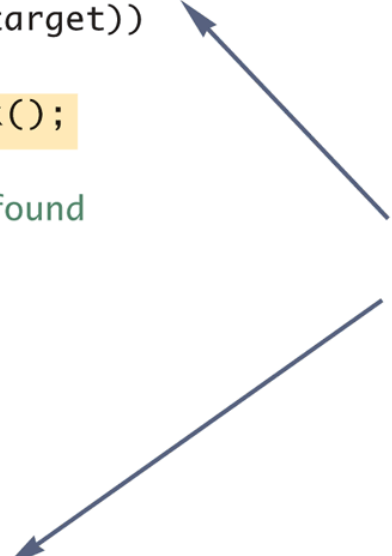
(continued)

A Linked List Class (Part 6 of 6)

Display 15.3   **A Linked List Class**

```
60                  itemAtPosition = position.getItem();
61                  if (itemAtPosition.equals(target))
62                      return position;
63                  position = position.getLink();
64          }
65          return null; //target was not found
66      }

67      public void outputList()
68      {
69          Node1 position = head;
70          while (position != null)
71          {
72              System.out.println(position.getItem() + " "
73                                      + position.getCount());
74              position = position.getLink();
75          }
76      }
77  }
```

This is the way you traverse an entire linked list.
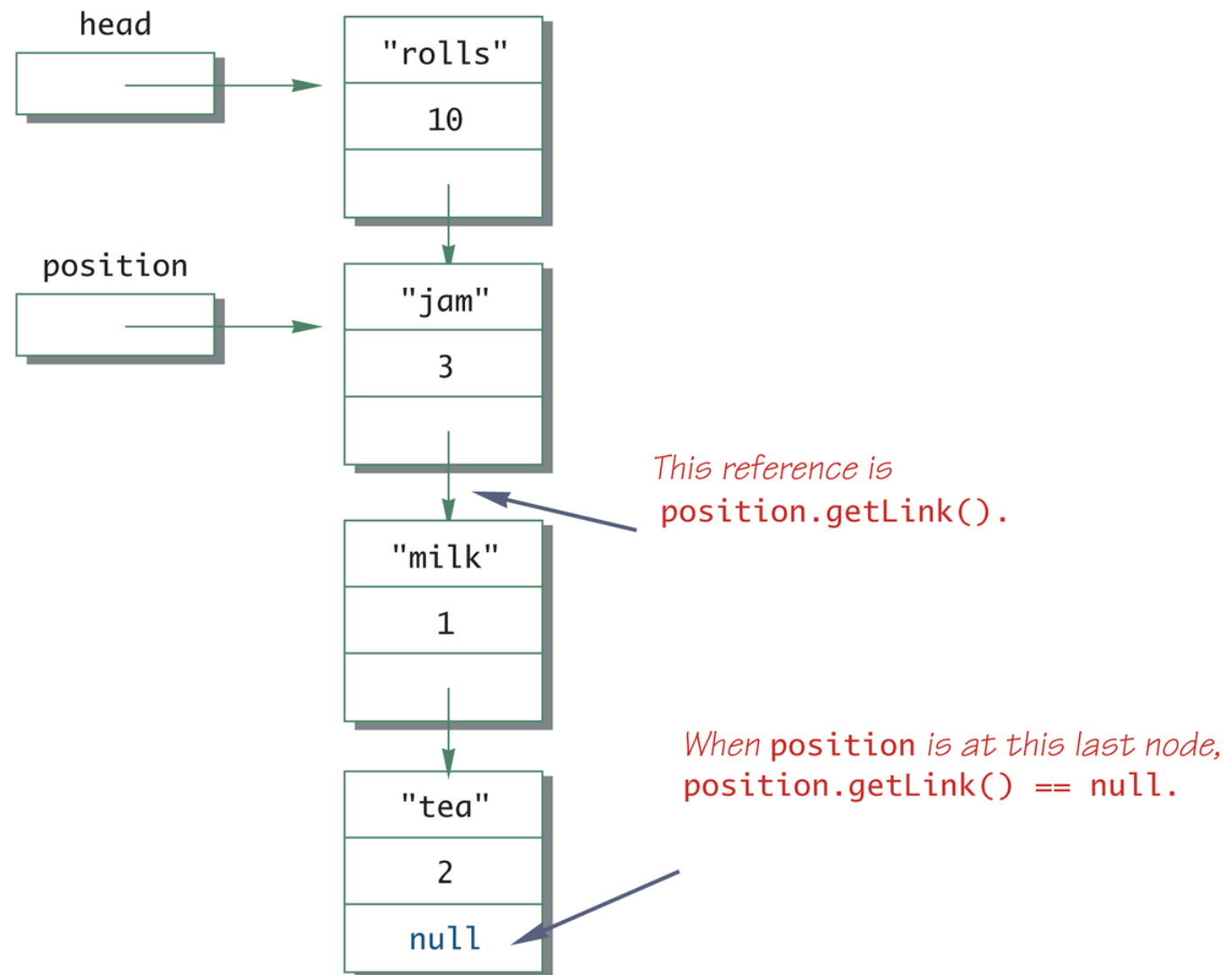
Indicating the End of a Linked List

- 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

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`

Traversing a Linked List

Display 15.4   **Traversing a Linked List**

head

"rolls"

10

position

"jam"

3

This reference is
`position.getLink().`

"milk"

1

When **position** *is at this last node,*
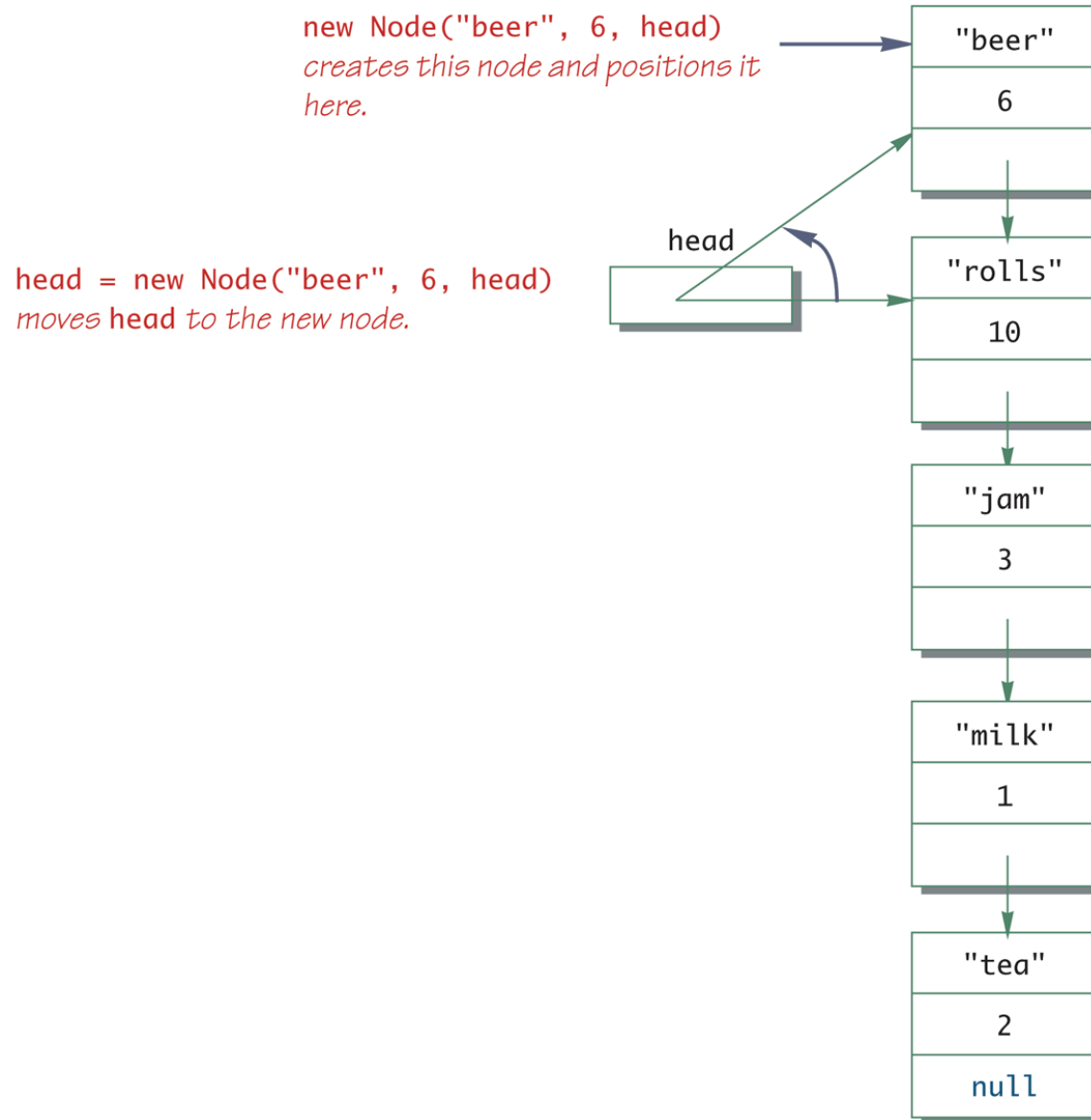`position.getLink() == null.`

"tea"

2

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

Adding a Node at the Start

new Node("beer", 6, head)
creates this node and positions it here.

"beer"

6

head

head = new Node("beer", 6, head)
moves **head** *to the new node.*

"rolls"

10

"jam"

3

"milk"

1

"tea"

2

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 1 of 3)

Display 15.6   A Linked List Demonstration

```
1    public class LinkedList1Demo
2    {
3        public static void main(String[] args)
4        {
5            LinkedList1 list = new LinkedList1();
6            list.addToStart("Apples", 1);
7            list.addToStart("Bananas", 2);
8            list.addToStart("Cantaloupe", 3);
9            System.out.println("List has " + list.size()
10                                     + " nodes.");
11           list.outputList();
12           if (list.contains("Cantaloupe"))
13               System.out.println("Cantaloupe is on list.");
```

Cantaloupe is now in the head node.

(continued)

A Linked List Demonstration (Part 2 of 3)

Display 15.6   A Linked List Demonstration

```
14          else
15               System.out.println("Cantaloupe is NOT on list.");

16          list.deleteHeadNode();              Removes the head node.

17          if (list.contains("Cantaloupe"))
18               System.out.println("Cantaloupe is on list.");
19          else
20               System.out.println("Cantaloupe is NOT on list.");

21          while (list.deleteHeadNode())       Empties the list. There is no loop
22               ; //Empty loop body             body because the method
                                                 deleteHeadNode both performs
23          System.out.println("Start of list:");   an action on the list and returns
24          list.outputList();                      a Boolean value.
25          System.out.println("End of list.");
26     }
27 }
```

(continued)

A Linked List Demonstration
(Part 3 of 3)

Display 15.6   **A Linked List Demonstration**

SAMPLE DIALOGUE

```
List has 3 entries.
Cantaloupe 3
Bananas 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)

Display 15.7 **A Linked List Class with a Node Inner Class**

```
1    public class LinkedList2
2    {
3        private class Node
4        {
5            private String item;
6            private Node link;
7
8            public Node()
9            {
10               item = null;
11               link = null;
12           }
13
14           public Node(String newItem, Node linkValue)
15           {
16               item = newItem;
17               link = linkValue;
18           }
19       }//End of Node inner class
```

It makes no difference whether we make the instance variables of Node public or private.

An inner class for the node class

(continued)

Display 15.7   A Linked List Class with a Node Inner Class

```
18        private Node head;

19        public LinkedList2()
20        {
21            head = null;
22        }
23        /**
24         Adds a node at the start of the list with the specified data.
25         The added node will be the first node in the list.
26        */
27        public void addToStart(String itemName)
28        {
29            head = new Node(itemName, head);
30        }

31        /**
32         Removes the head node and returns true if the list contained at least
33         one node. Returns false if the list was empty.
34        */
```

We have simplified this class and the previous linked list class to keep them relatively short. Among other things, these classes should have a copy constructor, an **equals** *method, and a* **clone** *method. Our next linked list example includes these items.*

(continued)

Display 15.7   **A Linked List Class with a Node Inner Class**

```
35        public boolean deleteHeadNode()
36        {
37            if (head != null)
38            {
39                head = head.link;
40                return true;
41            }
42            else
43                return false;
44        }

45        /**
46         Returns the number of nodes in the list.
47        */
48        public int size()
49        {
50            int count = 0;
51            Node position = head;
```

(continued)

Display 15.7   A Linked List Class with a Node Inner Class

```
52              while (position != null)
53              {
54                  count++;
55                  position = position.link;
56              }
57              return count;
58          }

59      public boolean contains(String item)
60      {
61          return (find(item) != null);
62      }
```

Note that the outer class has direct access to the inner class's instance variables, such as `link`.

(continued)

A Linked List Class with a Node Inner Class (Part 5 of 6)

Display 15.7   A Linked List Class with a Node Inner Class

```
63        /**
64          Finds the first node containing the target item, and returns a
65          reference to that node. If target is not in the list, null is returned.
66        */
67        private Node find(String target)
68        {
69            Node position = head;
70            String itemAtPosition;
71            while (position != null)
72            {
73                itemAtPosition = position.item;
74                if (itemAtPosition.equals(target))
75                    return position;
76                position = position.link;
77            }
78            return null; //target was not found
79        }
```

(continued)

Display 15.7   **A Linked List Class with a Node Inner Class**

```
80          public void outputList()
81          {
82              Node position = head;
83              while (position != null)
84              {
85                  System.out.println(position.item );
86                  position = position.link;
87              }
88          }

89          public boolean isEmpty()
90          {
91              return (head == null);
92          }

93          public void clear()
94          {
95              head = null;
96          }
97      }
```

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 variable
  - 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 1 of 9)

Display 15.8   A Generic Linked List Class

```
1   public class LinkedList3<T>
2   {
3       private class Node<T>
4       {
5           private T data;                    This linked list holds objects of type T.
6           private Node<T> link;              The type T should have well-defined
                                               equals and toString methods.
7           public Node()
8           {
9               data = null;
10              link = null;
11          }

12          public Node(T newData, Node<T> linkValue)
13          {
14              data = newData;
15              link = linkValue;
16          }
17      }//End of Node<T> inner class
```

(continued)

A Generic Linked List Class
(Part 2 of 9)

Display 15.8   A Generic Linked List Class

```
18        private Node<T> head;

19        public LinkedList3()
20        {
21            head = null;
22        }

23        /**
24         Adds a node at the start of the list with the specified data.
25         The added node will be the first node in the list.
26        */
27        public void addToStart(T itemData)
28        {
29            head = new Node<T>(itemData, head);
30        }
```

(continued)

A Generic Linked List Class
(Part 3 of 9)

Display 15.8   **A Generic Linked List Class**

```
31        /**
32         Removes the head node and returns true if the list contained at least
33         one node. Returns false if the list was empty.
34        */
35        public boolean deleteHeadNode()
36        {
37            if (head != null)
38            {
39                head = head.link;
40                return true;
41            }
42            else
43                return false;
44        }
```

(continued)

Display 15.8   A Generic Linked List Class

```java
45        /**
46          Returns the number of nodes in the list.
47        */
48        public int size()
49        {
50            int count = 0;
51            Node<T> position = head;
52            while (position != null)
53            {
54                count++;
55                position = position.link;
56            }
57            return count;
58        }

59        public boolean contains(T item)
60        {
61            return (find(item) != null);
62        }
```
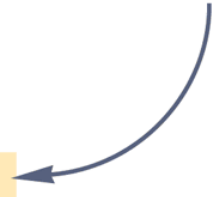
(continued)

A Generic Linked List Class
(Part 5 of 9)

Display 15.8   A Generic Linked List Class

```
63        /**
64         Finds the first node containing the target item, and returns a
65         reference to that node.  If target is not in the list, null is returned.
66        */
67        private Node<T> find(T target)
68        {
69            Node<T> position = head;
70            T itemAtPosition;
71            while (position != null)
72            {
73                itemAtPosition = position.data;
74                if (itemAtPosition.equals(target))
75                    return position;
76                position = position.link;
77            }
78            return null; //target was not found
79        }
```

Type T must have a well-defined **equals** *method for this to work.*

(continued)

15-40

A Generic Linked List Class (Part 6 of 9)

Display 15.8   A Generic Linked List Class

```
80        /**
81         Finds the first node containing the target and returns a reference
82          to the data in that node. If target is not in the list, null is returned.
83        */
84        public T findData(T target)
85        {
86            return find(target).data;
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        }
```

Type T must have a well-defined toString method for this to work.

(continued)

A Generic Linked List Class
(Part 7 of 9)

Display 15.8   A Generic Linked List Class

```
97      public boolean isEmpty()
98      {
99          return (head == null);
100     }

101     public void clear()
102     {
103         head = null;
104     }
```

(continued)

A Generic Linked List Class
(Part 8 of 9)

Display 15.8   A Generic Linked List Class

```
105       /*
106        For two lists to be equal they must contain the same data items in
107        the same order. The equals method of T is used to compare data items.
108       */
109       public boolean equals(Object otherObject)
110        {
111            if (otherObject == null)
112                return false;
113            else if (getClass() != otherObject.getClass())
114                return false;
115            else
116            {
117                LinkedList3<T> otherList = (LinkedList3<T>)otherObject;
```

(continued)

A Generic Linked List Class
(Part 9 of 9)

Display 15.8   A Generic Linked List Class

```
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 not found
130             }
131         }
132 }
```

A Sample Class for the Data in a Generic Linked List (Part 1 of 2)

Display 15.9 A Sample Class for the Data in a Generic Linked List

```
1    public class Entry
2    {
3        private String item;
4        private int count;

5        public Entry(String itemData, int countData)
6        {
7            item = itemData;
8            count = countData;
9        }

10       public String toString()
11       {
12           return (item + " " + count);
13       }
```

(continued)

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

Display 15.9   A Sample Class for the Data in a Generic Linked List

```
14        public boolean equals(Object otherObject)
15        {
16            if (otherObject == null)
17                return false;
18            else if (getClass() != otherObject.getClass())
19                return false;
20            else
21            {
22                Entry otherEntry = (Entry)otherObject;
23                return (item.equals(otherEntry.item)
24                        && (count == otherEntry.count));
25            }
26        }

        <There should be other constructors and methods, including accessor and
          mutator methods, but we do not use them in this demonstration.>
27  }
```

A Generic Linked List Demonstration (Part 1 of 2)

Display 15.10   **A Generic Linked List Demonstration**

```
 1    public class GenericLinkedListDemo
 2    {
 3        public static void main(String[] args)
 4        {
 5            LinkedList3<Entry> list = new LinkedList3<Entry>();
 6
 7            Entry entry1 = new Entry("Apples", 1);
 8            list.addToStart(entry1);
 9            Entry entry2 = new Entry("Bananas", 2);
10            list.addToStart(entry2);
11            Entry entry3 = new Entry("Cantaloupe", 3);
12            list.addToStart(entry3);
13            System.out.println("List has " + list.size()
14                                         + " nodes.");
```

(continued)

A Generic Linked List Demonstration (Part 2 of 2)

Display 15.10   A Generic Linked List Demonstration

```
15              list.outputList();
16              System.out.println("End of list.");
17          }
18  }
```

SAMPLE DIALOGUE

```
List has 3 nodes.
Cantaloupe 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>`, <u>not</u> `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)

Display 15.11 **An equals Method for the Linked List in Display 15.7**

```
1      /*
2       For two lists to be equal they must contain the same data items in
3       the same order.
4      */
5      public boolean equals(Object otherObject)
6        {
7            if (otherObject == null)
8                return false;
9            else if (getClass() != otherObject.getClass())
10               return false;
11           else
12           {
13               LinkedList2 otherList = (LinkedList2)otherObject;
```

(continued)

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

Display 15.11   An equals Method for the Linked List in Display 15.7

```
14          if (size() != otherList.size())
15              return false;
16          Node position = head;
17          Node otherPosition = otherList.head;
18          while (position != null)
19          {
20              if ( (!(position.item.equals(otherPosition.item))))
21                  return false;
22              position = position.link;
23              otherPosition = otherPosition.link;
24          }
25          return true; //A mismatch was not found
26      }
27  }
```

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)

Display 15.17   A Linked List with an Iterator

```
1    import java.util.NoSuchElementException;

2    public class LinkedList2
3    {
4        private class Node
5        {
6            private String item;
7            private Node link;
```

*This is the same as the class in Displays 15.7 and 15.11 except that the **List2Iterator** inner class and the **iterator()** method have been added.*

<The rest of the definition of the Node inner class is given in Display 15.7.>

```
8        }//End of Node inner class
```

(continued)

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

Display 15.17   A Linked List with an Iterator

```
9     /**
10     If the list is altered any iterators should invoke restart or
11     the iterator's behavior may not be as desired.
12    */
13    public class List2Iterator              An inner class for iterators for LinkedList2.
14    {
15        private Node position;
16        private Node previous;//previous value of position

17        public List2Iterator()
18        {
19            position = head; //Instance variable head of outer class.
20            previous = null;
21        }

22        public void restart()
23        {
24            position = head; //Instance variable head of outer class.
25            previous = null;
26        }
```

(continued)

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

Display 15.17   A Linked List with an Iterator

```
27      public String next()
28      {
29          if (!hasNext())
30              throw new NoSuchElementException();

31          String toReturn = position.item;
32          previous = position;
33          position = position.link;
34          return toReturn;
35      }

36      public boolean hasNext()
37      {
38          return (position != null);
39      }
```

(continued)

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

Display 15.17   A Linked List with an Iterator

```
40              /**
41               Returns the next value to be returned by next().
42               Throws an IllegalStateExpression if hasNext() is false.
43              */
44              public String peek()
45              {
46                      if (!hasNext())
47                          throw new IllegalStateException();
48                      return position.item;
49              }
```

(continued)

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

Display 15.17   A Linked List with an Iterator

```
50        /**
51         Adds a node before the node at location position.
52         previous is placed at the new node. If hasNext() is
53         false, then the node is added to the end of the list.
54         If the list is empty, inserts node as the only node.
55        */
56        public void addHere(String newData)
```
          <The rest of the method addHere is Self-Test Exercise 11.>

```
57        /**
58         Changes the String in the node at location position.
59         Throws an IllegalStateException if position is not at a node,
60        */
61        public void changeHere(String newData)
```
          <The rest of the method addHere is Self-Test Exercise 13.>

(continued)

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

Display 15.17 **A Linked List with an Iterator**

```
62              /**
63                Deletes the node at location position and
64                moves position to the "next" node.
65                Throws an IllegalStateException if the list is empty.
66              */
67              public void delete()
                    <The rest of the method delete is Self-Test Exercise 12.>
68          }//End of List2Iterator inner class

69          private Node head;

70          public List2Iterator iterator()
71          {
72              return new List2Iterator();
73          }
            <The other methods and constructors are identical to those in Displays 15.7 and 15.11.>
74  }
```

If list is an object of the class LinkedList2, then list.iterator() returns an iterator for list.

Using an Iterator (Part 1 of 6)

Display 15.18   Using an Iterator

```
1   public class IteratorDemo
2   {
3       public static void main(String[] args)
4       {
5           LinkedList2 list = new LinkedList2();
6           LinkedList2.List2Iterator i = list.iterator();

7           list.addToStart("shoes");
8           list.addToStart("orange juice");
9           list.addToStart("coat");
```

(continued)

Using an Iterator (Part 2 of 6)

Display 15.18   **Using an Iterator**

```
10          System.out.println("List contains:");
11          i.restart();
12          while(i.hasNext())
13              System.out.println(i.next());
14          System.out.println();

15          i.restart();
16          i.next();
17          System.out.println("Will delete the node for " + i.peek());
18          i.delete();
```

(continued)

Using an Iterator (Part 3 of 6)

Display 15.18   **Using an Iterator**

```
19          System.out.println("List now contains:");
20          i.restart();
21          while(i.hasNext())
22              System.out.println(i.next());
23          System.out.println();

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)

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();
46      }
47  }
```

(continued)

Using an Iterator (Part 5 of 6)

Display 15.18   Using an Iterator

SAMPLE DIALOGUE

```
List contains:
coat
orange juice
shoes

Will delete the node for orange juice
List now contains:
coat
shoes
```

(continued)

Using an Iterator (Part 6 of 6)

Display 15.18   Using an Iterator

```
Will add one node before shoes
List now contains:
coat
socks
shoes

Changing all items to credit card.

List now contains:
credit card
credit card
credit card
```

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**:
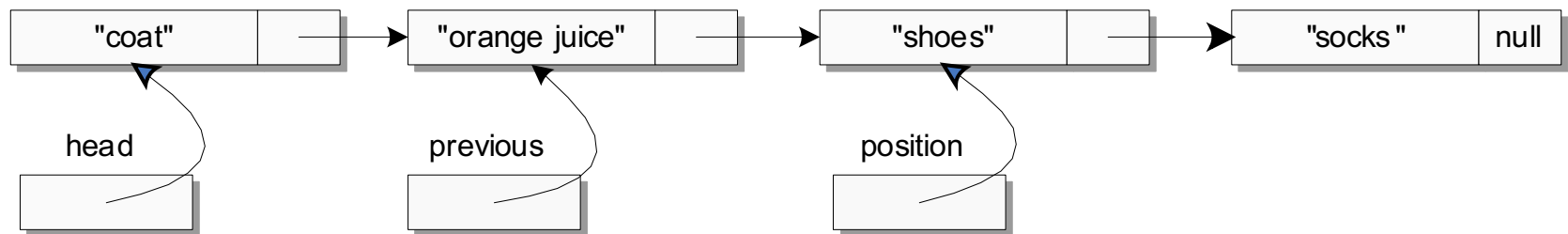
  ```
  previous.link = position.link;
  position = position.link;
  ```

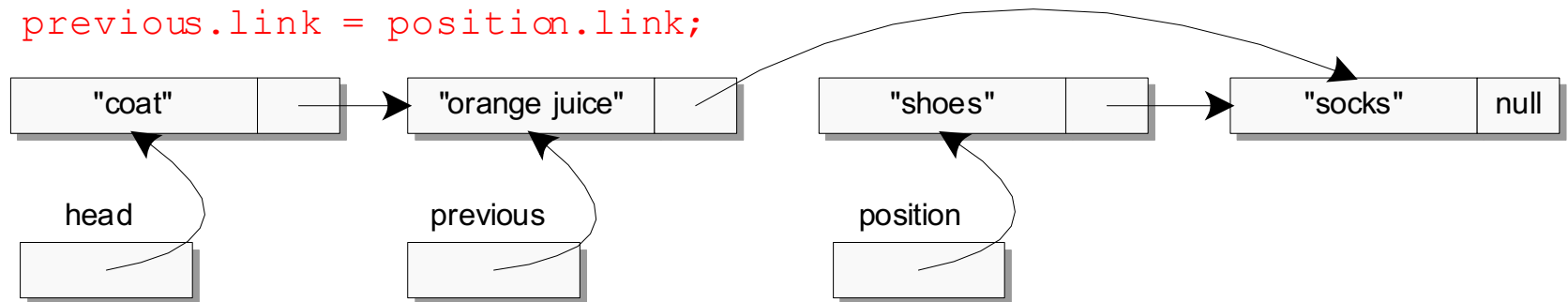 - Note: **previous** points to the node before **position**

Deleting a Node (Part 1 of 2)

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

| "coat" | | → | "orange juice" | | → | "shoes" | | → | "socks" | null |

head

previous

position

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

`previous.link = position.link;`

| "coat" | | → | "orange juice" | | | "shoes" | | → | "socks" | null |

head
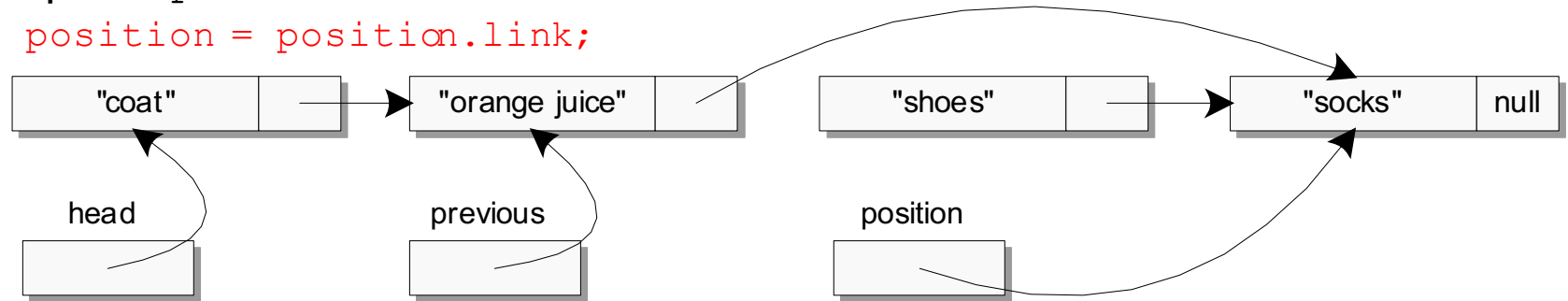
previous

position

Deleting a Node (Part 2 of 2)

3. Update `position` to reference the next node

   `position = position.link;`

   | "coat" | | → | "orange juice" | | | "shoes" | | → | "socks" | null |

   head   previous   position

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

4. Same picture with deleted node not shown

   | "coat" | | → | "orange juice" | | → | "socks" | null |

   head   previous   position
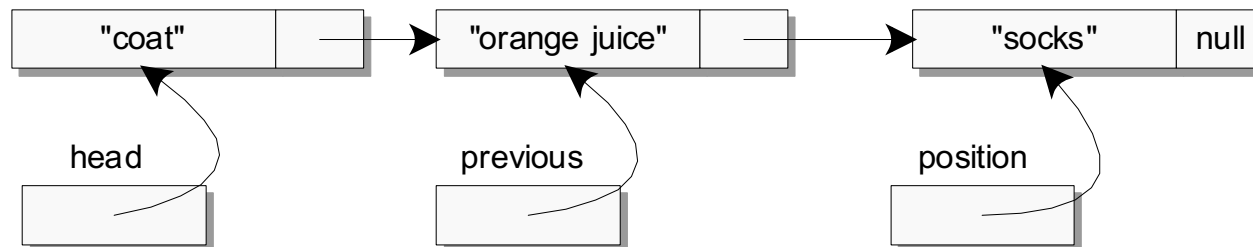
Adding and Deleting Nodes

- 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

    ```
    Node temp = new Node(newData,position);
    previous.link = temp;
    ```
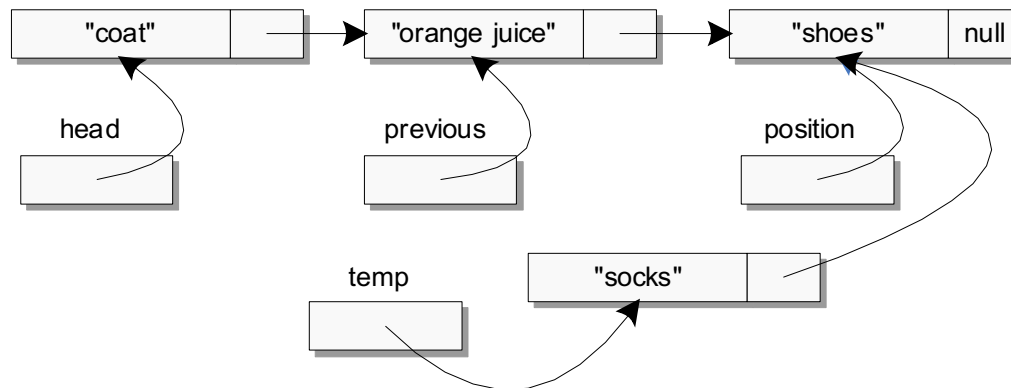
Adding a Node between Two Nodes (Part 1 of 2)

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

| "coat" | | ➝ | "orange juice" | | ➝ | "shoes" | null |

head   previous   position
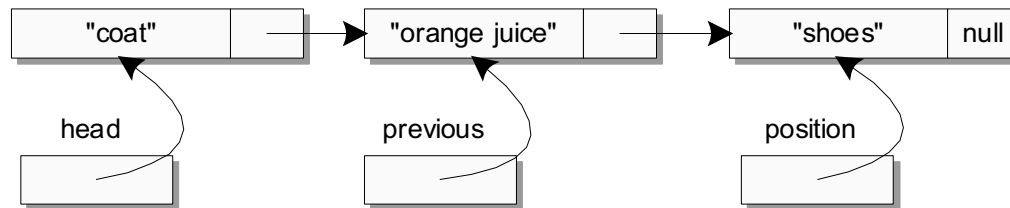
2. Create new Node with "socks" linked to "shoes"

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

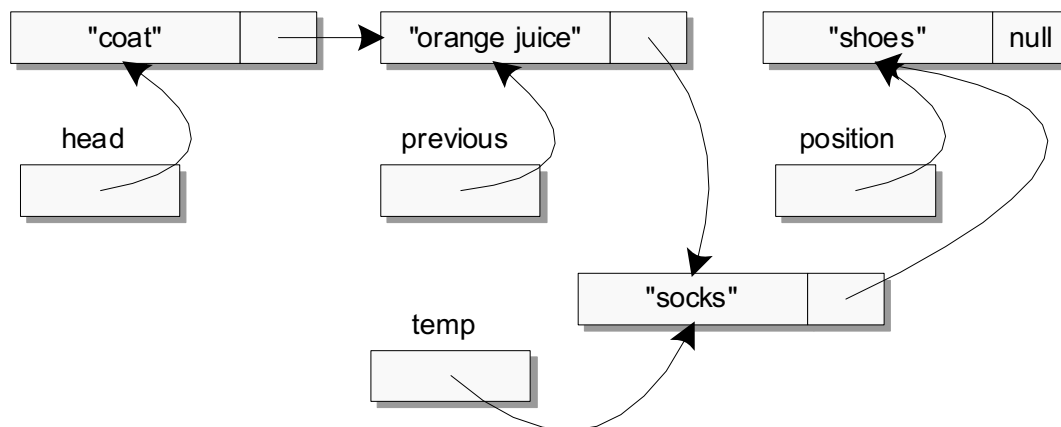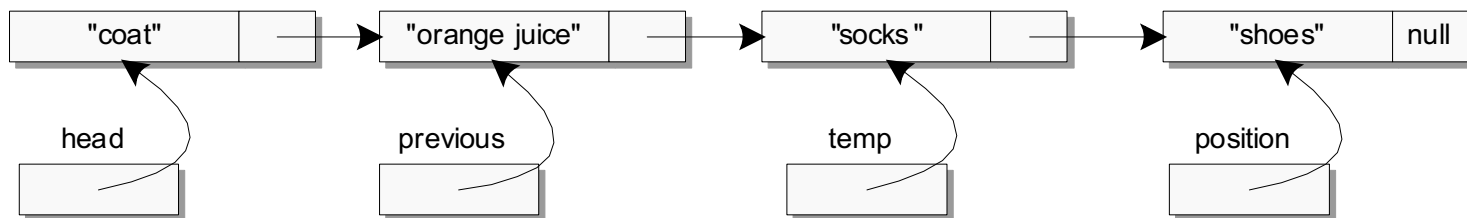| "coat" | | ➝ | "orange juice" | | ➝ | "shoes" | null |

head   previous   position

temp   "socks"

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

3. Make `previous` link to the Node `temp`

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

| "coat" | | → | "orange juice" | | | "shoes" | null |

head

previous

position

| "socks" | |

temp

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

| "coat" | | → | "orange juice" | | → | "socks " | | → | "shoes" | null |

head

previous

temp

position

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:

```
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
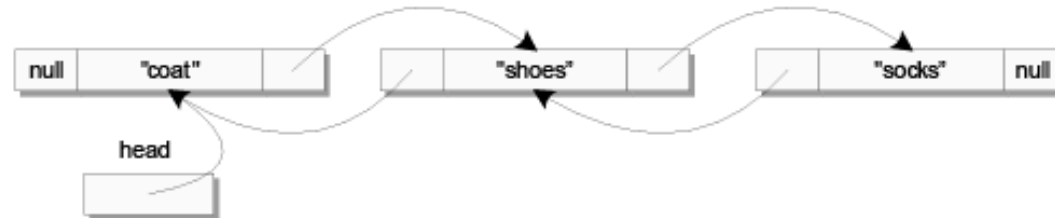
A Doubly Linked List
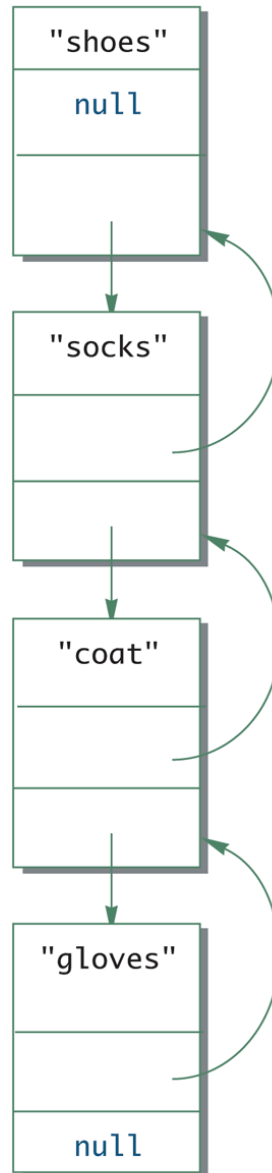
Adding a Node to the Front of a Doubly Linked List

1. Existing list

| null | "coat" | | | "shoes" | | | "socks" | null |

head

2. Create new TwoWayNode linked to "coat"

```
TwoWayNode newHead = new TwoWayNode(itemName, null, head); // itemName = "shirt"
```

| null | "shirt" | | | null | "coat" | | | "shoes" | | | "socks" | null |
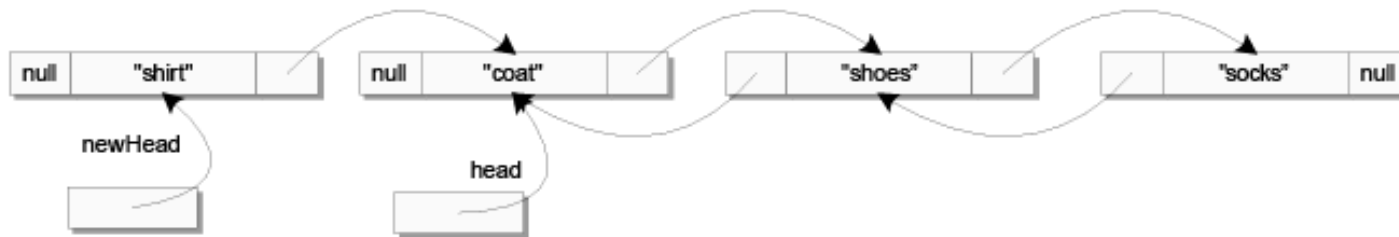
newHead

head

3. Set backward link and set new head

```
head.previous = newHead;
head = newHead;
```

| null | "shirt" | | | | "coat" | | | "shoes" | | | "socks" | null |

newHead

head

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

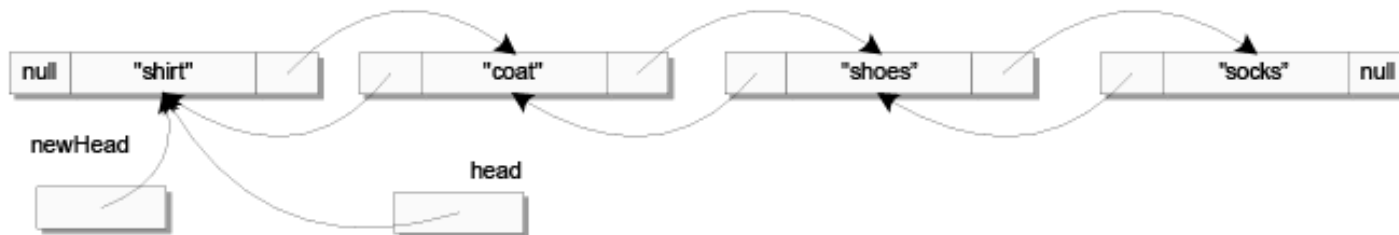1. Existing list with an iterator referencing "shoes"

| null | "coat" | | | "shoes" | | | "socks" | null |

head
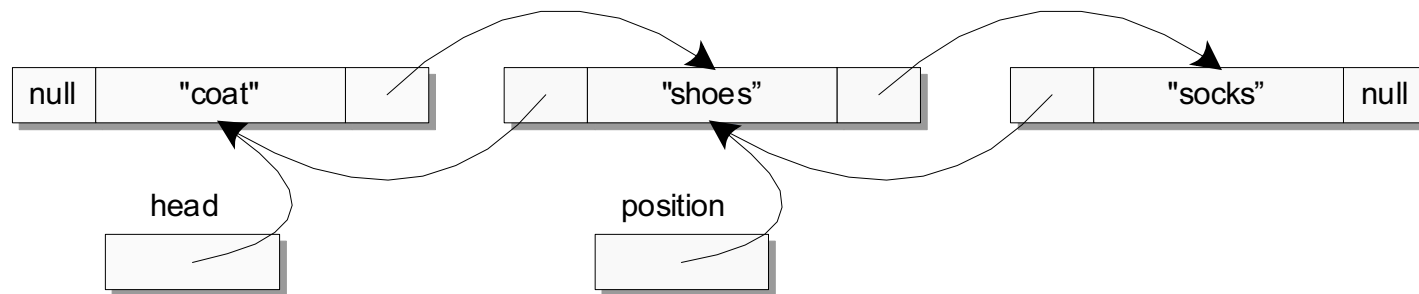
position

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

```
position.previous.next = position.next;
```

| null | "coat" | | | "shoes" | | | "socks" | null |

head
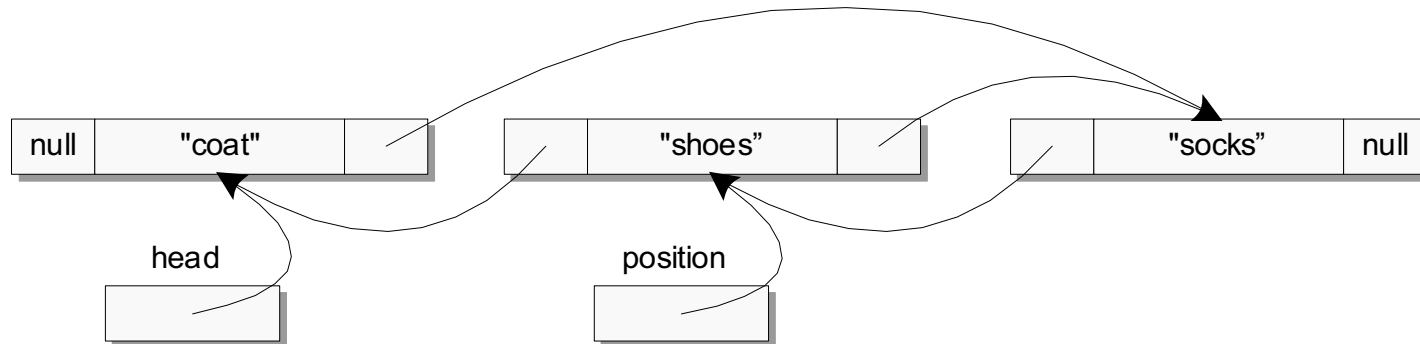
position

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

```
position.next.previous = position.previous;
position = position.next;
```


| null | "coat" | | | | "shoes" | | | | "socks" | null |

head

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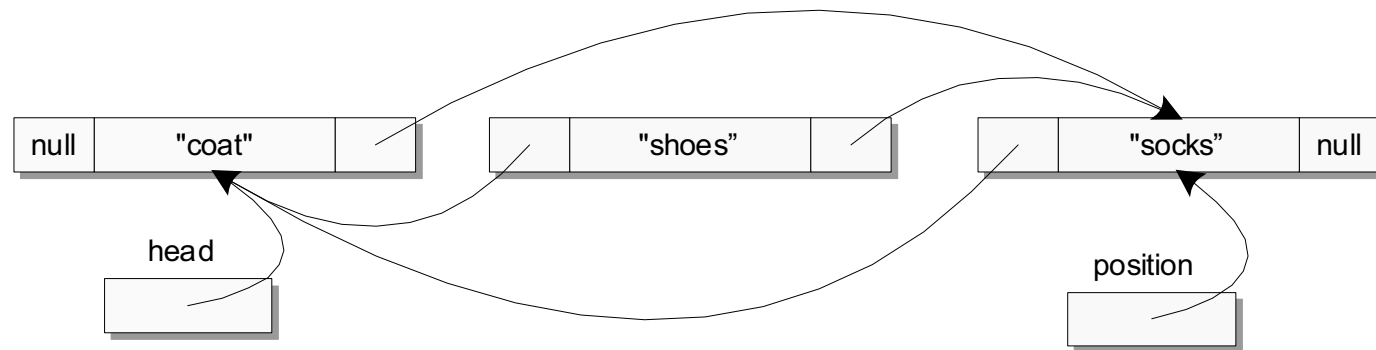
4. Picture redrawn for clarity with the "shoes" node removed since there are no longer references pointing to this node .
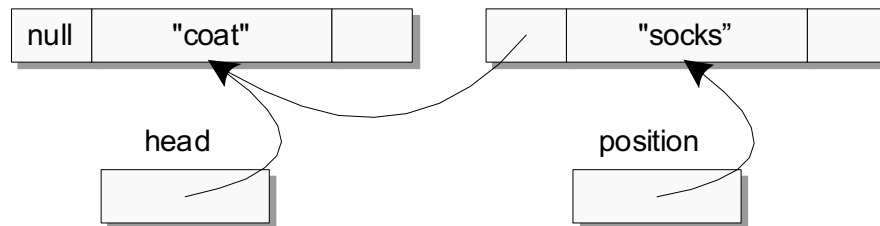
| null | "coat" | | | "socks" | |

head

position

Inserting a Node Into a Doubly Linked List (1 of 2)

1. Existing list with an iterator referencing "shoes"

| null | "coat" | | | "shoes" | | | "socks" | null |

head
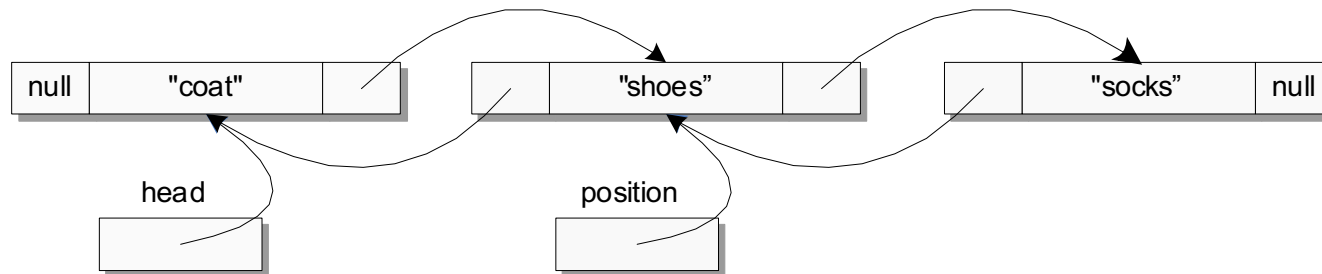
position

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

```
TwoWayNode temp = new TwoWayNode(newData, position.previous, position);
// newData = "shirt"
```

| null | "coat" | | | "shoes" | | | "socks" | null |

head

| | "shirt" | |
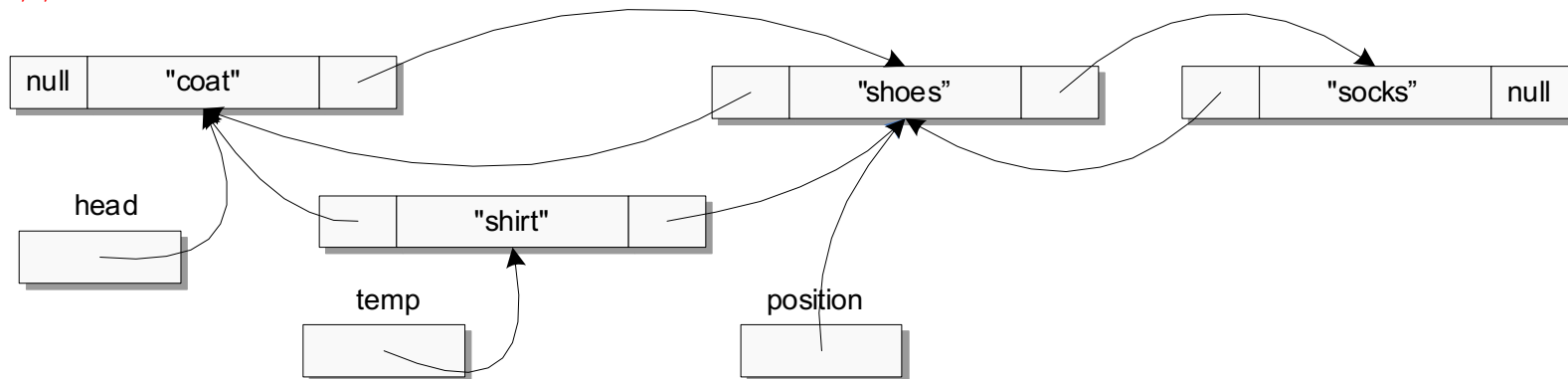
temp

position

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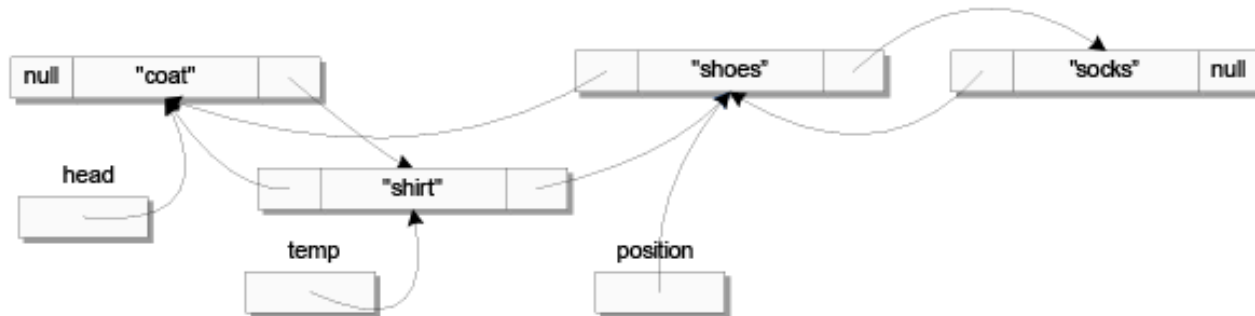
4. Set previous link from "shoes" to the new node of "shirt"

`position.previous = temp;`
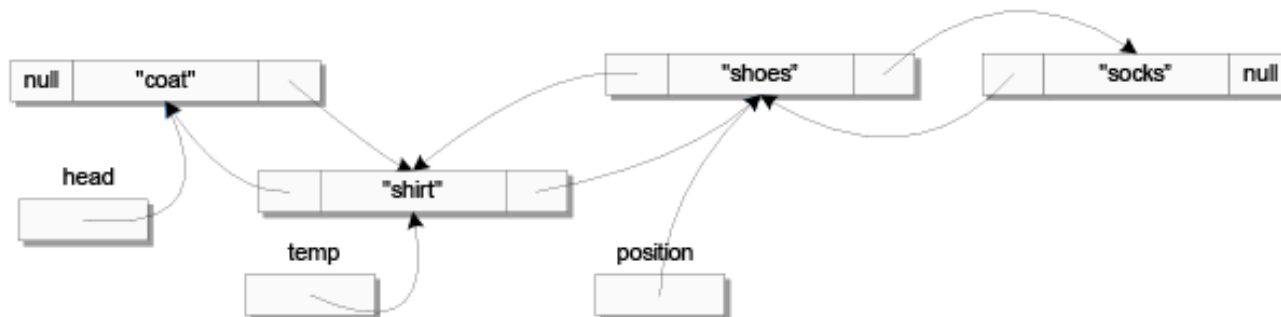
The Stack Data Structure

- 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)

A Queue Class

```java
1   public class Queue
2   {
3       private class Node
4       {
5           private String item;
6           private Node link;

7           public Node()
8           {
9               item = null;
10              link = null;
11          }
```

(continued)

A Queue Class (Part 2 of 5)

A Queue Class

```java
12              public Node(String newItem, Node linkValue)
13              {
14                  item = newItem;
15                  link = linkValue;
16              }
17        }//End of Node inner class

18      private Node front;
19      private Node back;

20      public Queue()
21      {
22          front = null;
23          back = null;
24      }
```

(continued)

A Queue Class (Part 3 of 5)

A Queue Class

```
25        /**
26          Adds a String to the back of the queue.
27        */
28        public void addToBack(String itemName)
          <The definition of this method is Self-Test Exercise 14.>

29        public boolean isEmpty()
30        {
31            return (front == null);
32        }

33        public void clear()
34        {
35            front = null;
36            back = null;
37        }
```

(continued)

A Queue Class (Part 4 of 5)

A Queue Class

```
38        /**
39          Returns the String in the front of the queue.
40          Returns null if queue is empty.
41        */
42        public String whoIsNext()
43        {
44            if (front == null)
45                return null;
46            else
47                return front.item;
48        }
49
```

(continued)

A Queue Class (Part 5 of 5)

A Queue Class

```
50        /**
51          Removes a String from the front of the queue.
52          Returns false if the list is empty.
53        */
54        public boolean removeFront()
55        {
56            if (front != null)
57            {
58                 front = front.link;
59                 return true;
60            }
61            else
62                 return false;
63        }
64    }
```

Demonstration of the Queue Class (Part 1 of 2)

Demonstration of the Queue Class

```
1    public class QueueDemo
2    {
3        public static void main(String[] args)
4        {
5            Queue q = new Queue();

6            q.addToBack("Tom");
7            q.addToBack("Dick");
8            q.addToBack("Harriet");
```

Items come out of the queue in the same order that they went into the queue.

(continued)

Demonstration of the Queue Class (Part 2 of 2)

Demonstration of the Queue Class

```
 9              while(!q.isEmpty())
10              {
11                  System.out.println(q.whoIsNext());
12                  q.removeFront();
13              }
14              System.out.println("The queue is empty.");
15          }
16  }
```

Items come out of the queue in the same order that they went into the queue.

SAMPLE DIALOGUE

```
Tom
Dick
Harriet
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

Some Values of a Running Time Function

INPUT SIZE	RUNNING TIME
10 numbers	2 seconds
100 numbers	2.1 seconds
1,000 numbers	10 seconds
10,000 numbers	2.5 minutes

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:
  <, &&, !, [ ], ==, ++

- 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

Comparison of Running Times
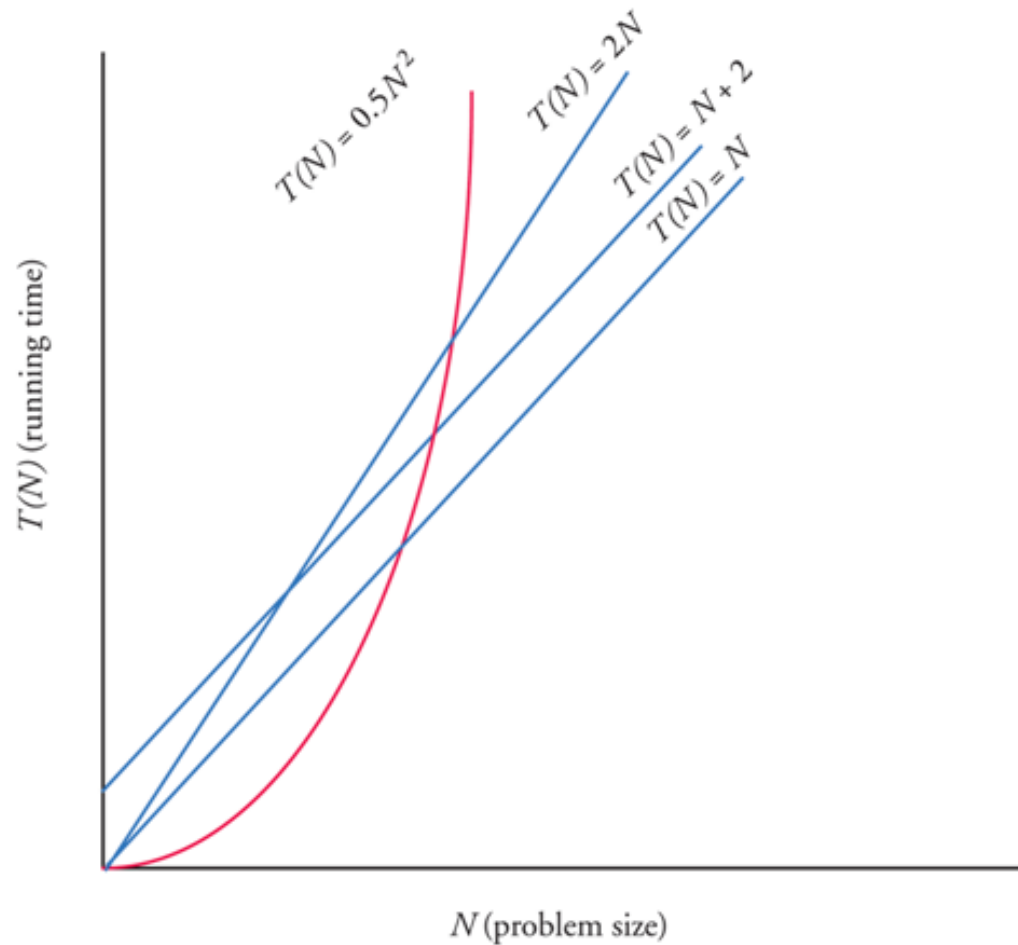
Efficiency of Linked Lists

- 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 LinkedList3[SIZE];  // SIZE = 10`

	0	1	2	3	4	5	6	7	8	9
hashArray	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty
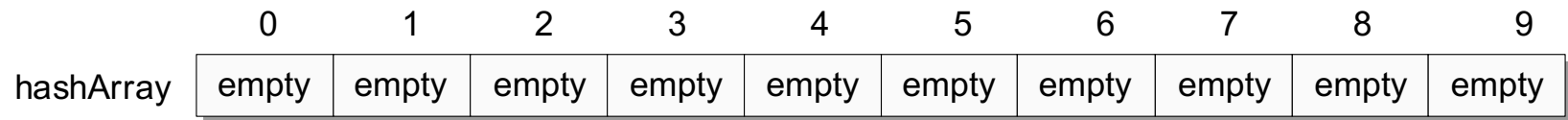
2. After adding "cat" with hash of 2

	0	1	2	3	4	5	6	7	8	9
hashArray	empty	empty		empty	empty	null	empty	empty	empty	empty

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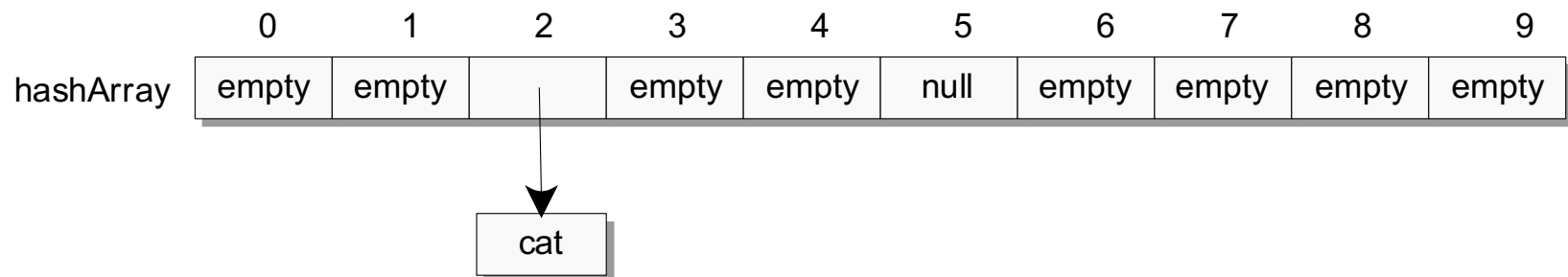
Constructing a Hash Table (2 of 2)

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

0	1	2	3	4	5	6	7	8	9
empty	empty		empty		empty	empty		empty	empty

hashArray

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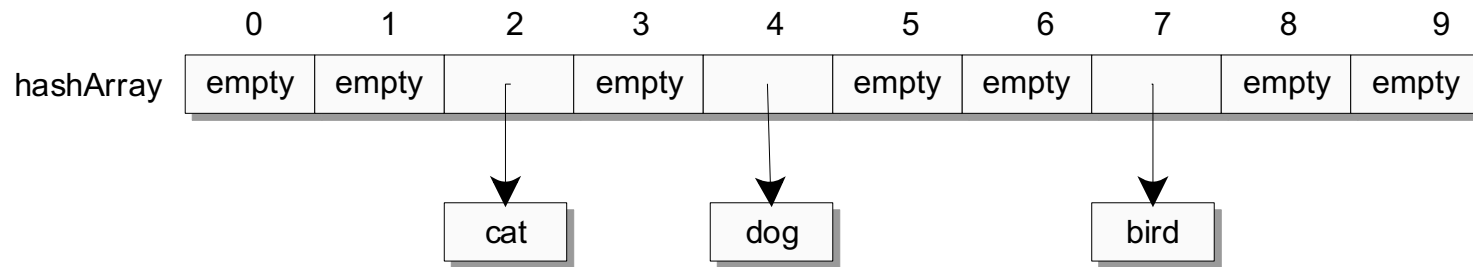
4. After adding "turtle" with hash of 2 – collision and chained to linked list with "cat"

0	1	2	3	4	5	6	7	8	9
empty	empty		empty		empty	empty		empty	empty

hashArray

turtle   dog   bird

cat
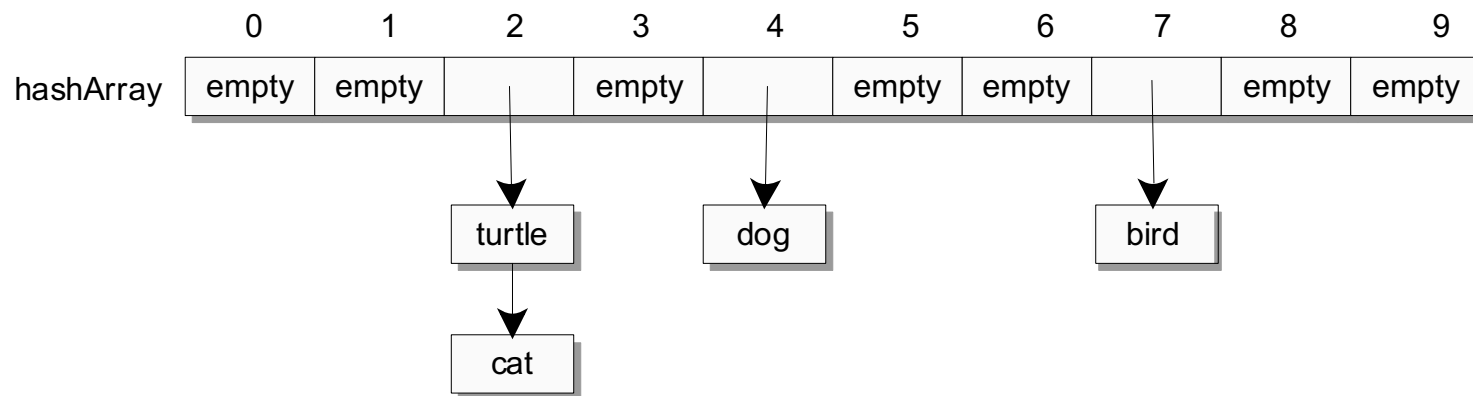
A Hash Table Class (1 of 3)

```
1    public class HashTable
2    {
3            // Uses the generic LinkedList2 class from Display 15.7
4            private LinkedList2[] hashArray;
5            private static final int SIZE = 10;

6            public HashTable()
7            {
8                    hashArray = new LinkedList2[SIZE];
9                    for (int i=0; i < SIZE; i++)
10                           hashArray[i] = new LinkedList2();
11           }

12           private int computeHash(String s)
13           {
14                   int hash = 0;
15                   for (int i = 0; i < s.length(); i++)
16                   {
17                           hash += s.charAt(i);
18                   }
19                   return hash % SIZE;
20           }
```

A Hash Table Class (2 of 3)

```
21          /**
22           Returns true if the target is in the hash table,
23           false if it is not.
24          */
25          public boolean containsString(String target)
26          {
27                  int hash = computeHash(target);
28                  LinkedList2 list = hashArray[hash];
29                  if (list.contains(target))
30                          return true;
31                  return false;
32          }
```

```
33              /**
34               Stores or puts string s into the hash table
35              */
36              public void put(String s)
37              {
38                      int hash = computeHash(s);          // Get hash value
39                      LinkedList2 list = hashArray[hash];
40                      if (!list.contains(s))
41                      {
42                              // Only add the target if it's not already
43                              // on the list.
44                              hashArray[hash].addToStart(s);
45                      }
46              }
47      } // End HashTable class
```

Hash Table Demonstration (1 of 2)

```
1        public class HashTableDemo
2        {
3                public static void main(String[] args)
4                {
5                        HashTable h = new HashTable();

6                        System.out.println("Adding dog, cat, turtle, bird");
7                        h.put("dog");
8                        h.put("cat");
9                        h.put("turtle");
10                       h.put("bird");
11                       System.out.println("Contains dog? " +
12                               h.containsString("dog"));
13                       System.out.println("Contains cat? " +
14                               h.containsString("cat"));
15                       System.out.println("Contains turtle? " +
16                               h.containsString("turtle"));
17                       System.out.println("Contains bird? " +
18                               h.containsString("bird"));
```

```
19                          System.out.println("Contains fish? " +
20                                  h.containsString("fish"));
21                          System.out.println("Contains cow? " +
22                                  h.containsString("cow"));
23              }
24 }
```

```
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
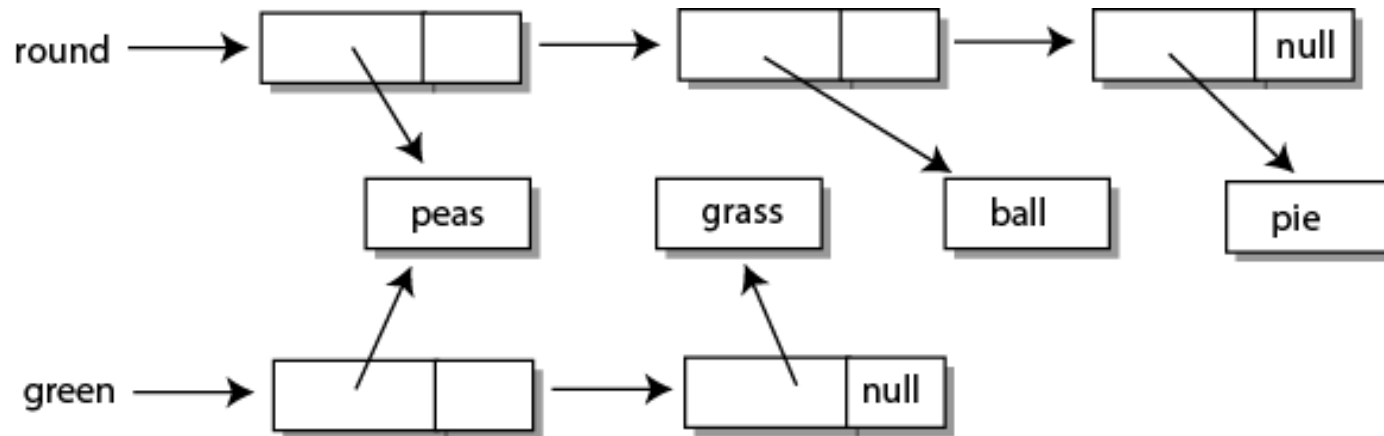
A Set Class (1 of 5)

```
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            {
7                private T data;
8                private Node<T> link;

9                public Node( )
10               {
11                    data = null;
12                    link = null;
13               }
14               public Node(T newData, Node<T> linkValue)
15               {
16                    data = newData;
17                    link = linkValue;
18               }
19          }//End of Node<T> inner class

20        private Node<T> head;
```

```
21          public Set()
22          {
23                  head = null;
24          }
25          /**
26           Add a new item to the set.  If the item
27           is already in the set, false is returned,
28           otherwise true is returned.
29          */
30          public boolean add(T newItem)
31          {
32                  if (!contains(newItem))
33                  {
34                          head = new Node<T>(newItem, head);
35                          return true;
36                  }
37                  return false;
38          }
```

A Set Class (3 of 5)

```
39      public boolean contains(T item)
40      {
41          Node<T> position = head;
42          T itemAtPosition;
43          while (position != null)
44          {
45              itemAtPosition = position.data;
46              if (itemAtPosition.equals(item))
47                  return true;
48              position = position.link;
49          }
50          return false; //target was not found
51      }

52      public void output( )
53      {
54          Node position = head;
55          while (position != null)
56  {
57              System.out.print(position.data.toString() + " ");
58              position = position.link;
59          }
60          System.out.println();
61      }
```

A Set Class (4 of 5)

```
62        /**
63         Returns a new set that is the union
64         of this set and the input set.
65        */
66        public Set<T> union(Set<T> otherSet)
67        {
68                Set<T> unionSet = new Set<T>();
69                // Copy this set to unionSet
70                Node<T> position = head;
71                while (position != null)
72                {
73                        unionSet.add(position.data);
74                        position = position.link;
75                }
76                // Copy otherSet items to unionSet.
77                // The add method eliminates any duplicates.
78                position = otherSet.head;
79                while (position != null)
80                {
81                        unionSet.add(position.data);
82                        position = position.link;
83                }
84                return unionSet;
85        }
```

A Set Class (5 of 5)

```
86                      /**
87                       Returns a new that is the intersection
88                       of this set and the input set.
89                      */
90                      public Set<T> intersection(Set<T> otherSet)
91                      {
92                              Set<T> interSet = new Set<T>();
93                              // Copy only items in both sets
94                          Node<T> position = head;
95                          while (position != null)
96                          {
97                                  if (otherSet.contains(position.data))
98                                        interSet.add(position.data);
99                                  position = position.link;
100                         }
101                         return interSet;
102                     }
103         }
```

The clear, size, and isEmpty methods are identicalto those in Display 15.8 for the LinkedList3 class.

A Set Class Demo (1 of 3)

```
1        class SetDemo
2        {
3                public static void main(String[] args)
4                {
5                        // Round things
6                        Set round = new Set<String>();
7                        // Green things
8                        Set green = new Set<String>();

9                        // Add some data to both sets
10                       round.add("peas");
11                       round.add("ball");
12                       round.add("pie");
13                       round.add("grapes");

14                       green.add("peas");
15                       green.add("grapes");
16                       green.add("garden hose");
17                       green.add("grass");

18                       System.out.println("Contents of set round: ");
19                       round.output();
20                       System.out.println("Contents of set green: ");
21                       green.output();
22                       System.out.println();
```

A Set Class Demo (2 of 3)

```
23            System.out.println("ball in set round? " +
24                    round.contains("ball"));
25            System.out.println("ball in set green? " +
26                    green.contains("ball"));

27            System.out.println("ball and peas in same set? " +
28                    ((round.contains("ball") &&
29                      (round.contains("peas"))) ||
30                   (green.contains("ball") &&
31                      (green.contains("peas")))));

32            System.out.println("pie and grass in same set? " +
33                    ((round.contains("pie") &&
34                      (round.contains("grass"))) ||
35                    (green.contains("pie") &&
36                      (green.contains("grass")))));
```

```
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                      }
42 }
```

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"

15-120

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 1 of 2)

A Binary Tree

left subtree   right subtree
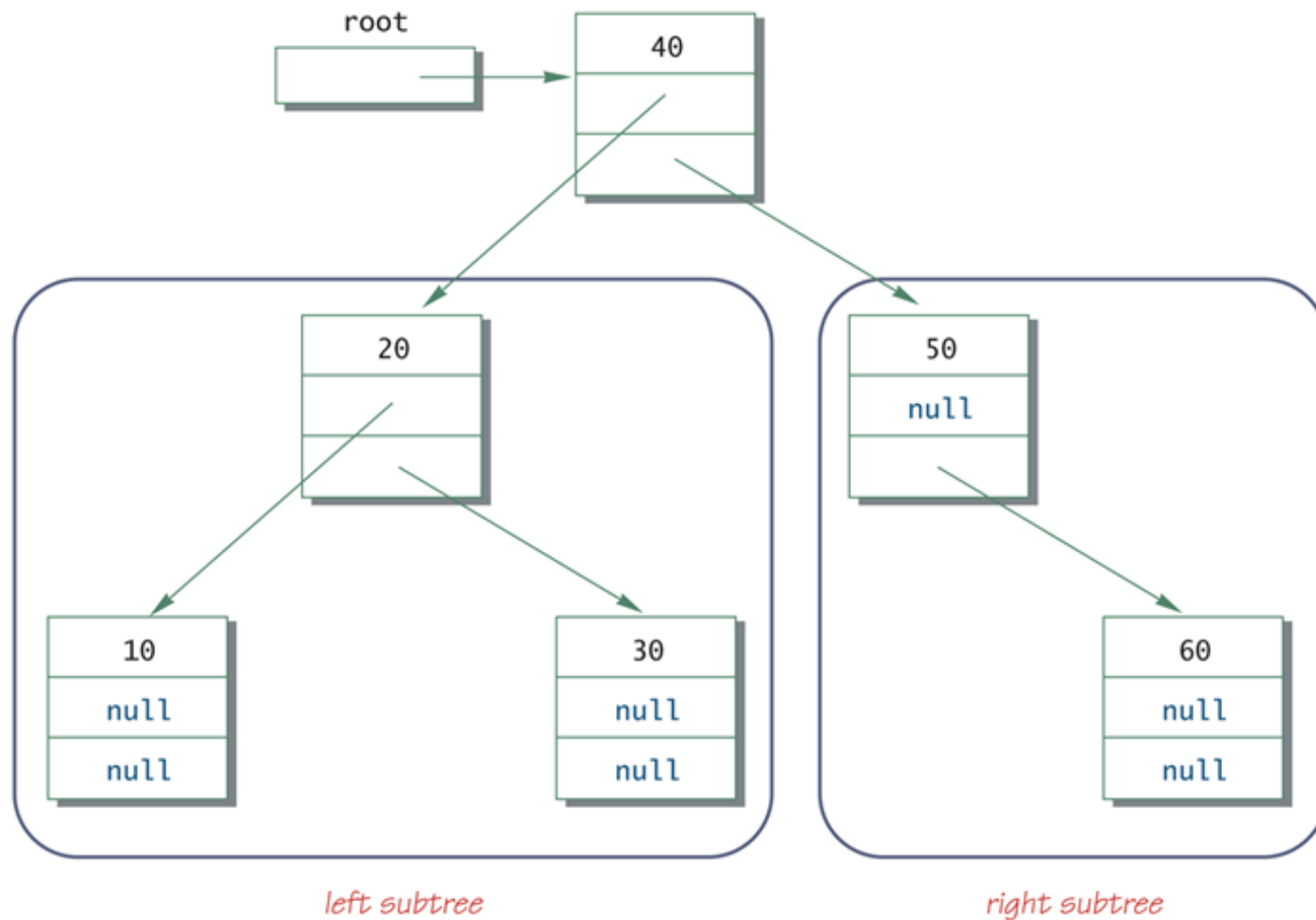
(continued)

A Binary Tree (Part 2 of 2)

A Binary Tree

```
1    public class IntTree
2    {
3        public class IntTreeNode
4        {
5            private int data;
6            private IntTreeNode leftLink;
7            private IntTreeNode rightLink;
8        } //End of IntTreeNode inner class

9        private IntTreeNode root;
         <The methods and other inner classes are not shown.>

10   }
```