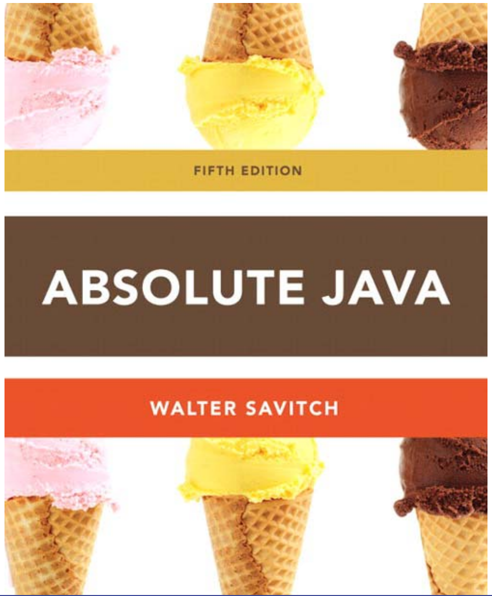



# Chapter 15

## Linked Data Structures

Slides prepared by Rose Williams, *Binghamton University*

Kenrick Mock, *University of Alaska Anchorage*

PEARSON ALWAYS LEARNING

## 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



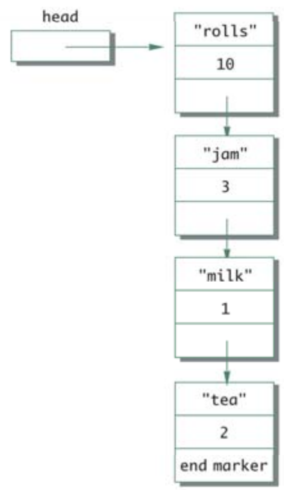

## Java Linked Lists

- 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**



# 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**

```
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**

```
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**

```
public class LinkedList1
{
    private Node1 head;

    public LinkedList1()
    {
        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(String itemName, int itemCount)
    {
        head = new Node1(itemName, itemCount, head);
    }
```

*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**

```
    /**
     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.getLink();
            return true;
        }
        else
            return false;
    }
```

(continued)





# A Linked List Class (Part 3 of 6)

Display 15.3 **A Linked List Class**

```
    /**
     Returns the number of nodes in the list.
    */
    public int size()
    {
        int count = 0;
        Node1 position = head;

```

(continued)





# A Linked List Class (Part 4 of 6)

Display 15.3 **A Linked List Class**

```
        while (position != null)
        {
            count++;
            position = position.getLink();
        }
        return count;
    }

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

*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**

```
    /**
     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 Node1 find(String target)
    {
        Node1 position = head;
        String itemAtPosition;
        while (position != null)
        {
```

(continued)

# A Linked List Class (Part 6 of 6)

Display 15.3 **A Linked List Class**

```
            itemAtPosition = position.getItem();
            if (itemAtPosition.equals(target))
                return position;
            position = position.getLink();
        }
        return null; //target was not found
    }

    public void outputList()
    {
        Node1 position = head;
        while (position != null)
        {
            System.out.println(position.getItem() + " "
                              + position.getCount());
            position = position.getLink();
        }
    }
}
```

*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



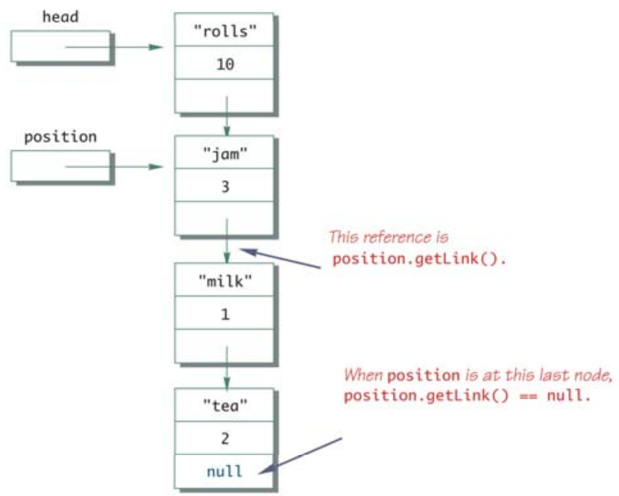

# Traversing a Linked List

- 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**



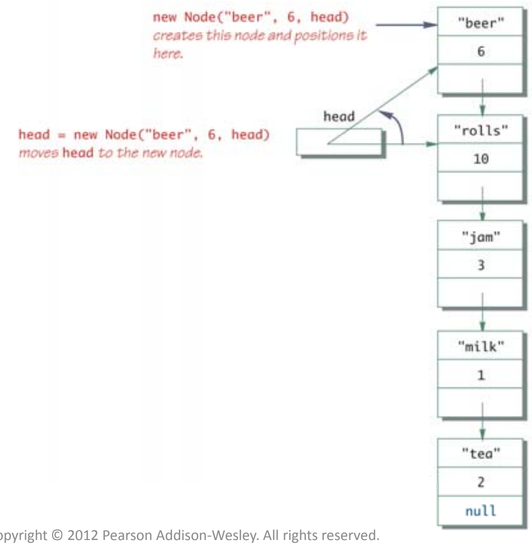
# Adding a Node to a Linked List

- 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

Display 15.5 **Adding a Node at the Start**







# 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**

```
public class LinkedList1Demo
{
    public static void main(String[] args)
    {
        LinkedList1 list = new LinkedList1();
        list.addToStart("Apples", 1);
        list.addToStart("Bananas", 2);
        list.addToStart("Cantaloupe", 3);
        System.out.println("List has " + list.size()
                            + " nodes.");
        list.outputList();
        if (list.contains("Cantaloupe"))
            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**

```
        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.deleteHeadNode())   <-- Empties the list. There is no loop
            ; //Empty loop body             body because the method
                                            deleteHeadNode both performs
        System.out.println("Start of list:");   an action on the list and returns
        list.outputList();                      a Boolean value.
        System.out.println("End of list.");
    }
}
```

(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**

```
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;
        }
    }//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)





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

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

```
    private Node head;

    public LinkedList2()
    {
        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(String itemName)
    {
        head = new Node(itemName, head);
    }

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

*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)





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

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

```
    public boolean deleteHeadNode()
    {
        if (head != null)
        {
            head = head.link;
            return true;
        }
        else
            return false;
    }

    /**
     Returns the number of nodes in the list.
    */
    public int size()
    {
        int count = 0;
        Node position = head;
```

(continued)





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

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

```
        while (position != null)
        {
            count++;
            position = position.link;
        }
        return count;
    }

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

*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**

```
    /**
     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 find(String target)
    {
        Node position = head;
        String itemAtPosition;
        while (position != null)
        {
            itemAtPosition = position.item;
            if (itemAtPosition.equals(target))
                return position;
            position = position.link;
        }
        return null; //target was not found
    }
```

(continued)




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

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

```
    public void outputList()
    {
        Node position = head;
        while (position != null)
        {
            System.out.println(position.item );
            position = position.link;
        }
    }

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

    public void clear()
    {
        head = null;
    }
}
```




# 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**

```
public class LinkedList3<T>
{
    private class Node<T>
    {
        private T data;
        private Node<T> link;

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

        public Node(T newData, Node<T> linkValue)
        {
            data = newData;
            link = linkValue;
        }
    }//End of Node<T> inner class
```

*This linked list holds objects of type T. The type T should have well-defined `equals` and `toString` methods.*

(continued)

# A Generic Linked List Class (Part 2 of 9)

Display 15.8 **A Generic Linked List Class**

```
    private Node<T> head;

    public LinkedList3()
    {
        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)

Display 15.8 **A Generic Linked List Class**

```
    /**
     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)

Display 15.8 **A Generic Linked List Class**

```
    /**
     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)

Display 15.8 **A Generic Linked List Class**

```
    /**
     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;
        T itemAtPosition;
        while (position != null)
        {
            itemAtPosition = position.data;
            if (itemAtPosition.equals(target))
                return position;
            position = position.link;
        }
        return null; //target was not found
    }
```

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

(continued)

# A Generic Linked List Class (Part 6 of 9)

Display 15.8 **A Generic Linked List Class**

```
    /**
     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.
    */
    public T findData(T target)
    {
        return find(target).data;
    }

    public void outputList()
    {
        Node<T> position = head;
        while (position != null)
        {
            System.out.println(position.data);
            position = position.link;
        }
    }
```

*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**

```
    public boolean isEmpty()
    {
        return (head == null);
    }

    public void clear()
    {
        head = null;
    }
```

(continued)





# A Generic Linked List Class (Part 8 of 9)

Display 15.8 **A Generic Linked List Class**

```
    /*
     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.
    */
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            LinkedList3<T> otherList = (LinkedList3<T>)otherObject;
```

(continued)





# A Generic Linked List Class (Part 9 of 9)

Display 15.8 **A Generic Linked List Class**

```
            if (size() != otherList.size())
                return false;
            Node<T> position = head;
            Node<T> otherPosition = otherList.head;
            while (position != null)
            {
                if (!(position.data.equals(otherPosition.data)))
                    return false;
                position = position.link;
                otherPosition = otherPosition.link;
            }
            return true; //no mismatch was not found
        }
    }
}
```

# 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**

```
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)

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

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

Display 15.10 **A Generic Linked List Demonstration**

```
public class GenericLinkedListDemo
{
    public static void main(String[] args)
    {
        LinkedList3<Entry> list = new LinkedList3<Entry>();

        Entry entry1 = new Entry("Apples", 1);
        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)

Display 15.10 **A Generic Linked List Demonstration**

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

**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>**, 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)

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

```
/*
 For two lists to be equal they must contain the same data items in
 the same order.
*/
public boolean equals(Object otherObject)
 {
     if (otherObject == null)
         return false;
     else if (getClass() != otherObject.getClass())
         return false;
     else
     {
         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**

```
         if (size() != otherList.size())
             return false;
         Node position = head;
         Node otherPosition = otherList.head;
         while (position != null)
         {
             if ( (!(position.item.equals(otherPosition.item))))
                 return false;
             position = position.link;
             otherPosition = otherPosition.link;
         }
         return true; //A mismatch was not found
     }
 }
```

## 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)

Display 15.12 **A Copy Constructor and clone Method for a Generic Linked List**

```
public class LinkedList3<T> implements Cloneable
{
    private class Node<T>
    {
        private T data;
        private Node<T> link;

        public Node()
        {
            data = null;
            link = null;
        }
```

*This copy constructor and this clone method do not make deep copies. We discuss one way to make a deep copy in the Programming Tip subsection "Use a Type Parameter Bound for a Better clone."*

(continued)





## A Copy Constructor and clone Method for a Generic Linked List (Part 2 of 6)

Display 15.12 **A Copy Constructor and clone Method for a Generic Linked List**

```
        public Node(T newData, Node<T> linkValue)
        {
            data = newData;
            link = linkValue;
        }
    }//End of Node<T> inner class

    private Node<T> head;

  <All the methods from Display 15.8 are in the class definition,
                         but they are not repeated in this display.>
```

(continued)

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

Display 15.12 **A Copy Constructor and `clone` Method for a Generic Linked List**

```java
    /**
     Produces a new linked list, but it is not a true deep copy.
     Throws a NullPointerException if other is null.
    */
    public LinkedList3(LinkedList3<T> otherList)
    {
        if (otherList == null)
            throw new NullPointerException( );
        if (otherList.head == null)
            head = null;
        else
            head = copyOf(otherList.head);
    }


```

(continued)





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

Display 15.12 **A Copy Constructor and `clone` Method for a Generic Linked List**

```java
    public LinkedList3<T> clone( )
    {
        try
        {
            LinkedList3<T> copy =
                               (LinkedList3<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 Copy Constructor and `clone` Method for a Generic Linked List (Part 5 of 6)

Display 15.12 **A Copy Constructor and `clone` Method for a Generic Linked List**

```java
    /*
     Precondition: otherHead != null
     Returns a reference to the head of a copy of the list
     headed by otherHead. Does not return 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 new growing list.
```

(continued)





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

Display 15.12 **A Copy Constructor and `clone` Method for a Generic Linked List**

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

        while (position != null)
        {//copy node at position to end of new list.
            end.link =
                new Node<T>(position.data, null);
            end = end.link;
            position = position.link;
        }

        return newHead;
    }
}
```

*Invoking* `clone` *with* `position.data` *would be illegal.* (annotation on the first `new Node<T>(position.data, null)`)

*Invoking* `clone` *with* `position.data` *would be illegal.* (annotation on the second `new Node<T>(position.data, null)`)

## 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:

```
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 The PubliclyCloneable Interface**

```java
/*
 The programmer who defines a class implementing this interface
 has the responsibility to define clone so it makes a deep copy
 (in the officially sectioned way.)
*/
public interface PubliclyCloneable extends Cloneable
{
    public Object clone();
}
```

*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 A Generic Linked List with a Deep Copy clone Method**

```java
public class LinkedList<T extends PubliclyCloneable>
                          implements PubliclyCloneable
{
    private class Node<T>
    {
        private T data;
        private Node<T> link;

        public Node()
        {
            data = null;
            link = null;
        }
```

(continued)




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

**Display 15.14 A Generic Linked List with a Deep Copy clone Method**

```java
        public Node(T newData, Node<T> linkValue)
        {
            data = newData;
            link = linkValue;
        }
    }//End of Node<T> inner class

    private Node<T> head;

    public LinkedList()
    {
        head = null;
    }
```

(continued)




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

**Display 15.14 A Generic Linked List with a Deep Copy clone Method**

```java
    /**
     Produces a new linked list, but it is not a true deep copy.
     Throws a NullPointerException if other is null.
    */
    public LinkedList(LinkedList<T> otherList)
    {
        if (otherList == null)
            throw new NullPointerException( );
        if (otherList.head == null)
            head = null;
        else
            head = copyOf(otherList.head);
    }
```

(continued)

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

Display 15.14 **A Generic Linked List with a Deep Copy `clone` Method**

```
    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)

Display 15.14 **A Generic Linked List with a Deep Copy `clone` Method**

*This definition of* **copyOf** *gives a deep copy of the linked list.*

```
    /*
     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 new growing list.

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

(continued)




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

Display 15.14 **A Generic Linked List with a Deep Copy `clone` Method**

```
        while (position != null)
        {//copy node at position to end of new list.
            end.link =
                new Node<T>((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)

Display 15.14 **A Generic Linked List with a Deep Copy `clone` Method**

```
    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 15.8. The only difference
 between this definition of equals and the one in Display 15.8 is that we
 have replaced the class name LinkedList3<T> with LinkedList<T>.>

    }
```

(continued)

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

Display 15.14 **A Generic Linked List with a Deep Copy `clone` Method**

<All the other methods from Display 15.8 are in the class definition, but are not repeated in this display. >

```
    public String toString()
    {
        Node<T> position = head;
        String theString = "";
        while (position != null)
        {
            theString = theString + position.data + "\n";
            position = position.link;
        }
        return theString;
    }
}
```

*We added a `toString` method so `LinkedList<T>` would have all the properties we want T to have.*




# A Linked List with a Deep Copy `clone` Method

- 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:

```
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)

Display 15.15 **A `PubliclyCloneable` Class**

```
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)

Display 15.15 **A `PubliclyCloneable` Class**

```
    public void setNumber(int newNumber)
    {
        number = newNumber;
    }

    public void setName(String newName)
    {
        name = newName;
    }

    public String toString()
    {
        return (name + " " + number);
    }
```

(continued)

# A PubliclyCloneable Class (Part 3 of 4)

Display 15.15 **A PubliclyCloneable Class**

```
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch(CloneNotSupportedException e)
        {//This should not happen.
            return null; //To keep compiler happy.
        }
    }

```

(continued)




# A PubliclyCloneable Class (Part 4 of 4)

Display 15.15 **A PubliclyCloneable Class**

```
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            StockItem otherItem = (StockItem) otherObject;
            return (name.equalsIgnoreCase(otherItem.name)
                        && number == otherItem.number);
        }
    }
}
```




# Demonstration of Deep Copy clone (Part 1 of 3)

Display 15.16 **Demonstration of Deep Copy clone**

```
public class DeepDemo
{
    public static void main(String[] args)
    {
        LinkedList<StockItem> originalList =
                                new LinkedList<StockItem>();
        originalList.addToStart(new StockItem("red dress", 1));
        originalList.addToStart(new StockItem("black shoe", 2));

        LinkedList<StockItem> copyList = originalList.clone();
        if (originalList.equals(copyList))
            System.out.println("OK, Lists are equal.");
```

(continued)




# Demonstration of Deep Copy clone (Part 2 of 3)

Display 15.16 **Demonstration of Deep Copy clone**

```
        System.out.println("Now we change copyList.");
        StockItem dataEntry =
                copyList.findData(new StockItem("red dress", 1));
        dataEntry.setName("orange pants");

        System.out.println("originalList:");
        originalList.outputList();

        System.out.println("copyList:");
        copyList.outputList();

        System.out.println("Only one list is changed.");
    }
}
```

(continued)

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

Display 15.16 **Demonstration of Deep Copy clone**

SAMPLE DIALOGUE

```
OK, Lists are equal.
Now we change copyList.
originalList:
black shoe 2
red dress 1
copyList:
black shoe 2
orange pants 1
Only one 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)

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

```
import java.util.NoSuchElementException;

public class LinkedList2
{
    private class Node
    {
        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
```

*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.*

(continued)




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

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

```
    /**
     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;
        }
```

*An inner class for iterators for* **LinkedList2**.

(continued)




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

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

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

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

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

(continued)




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

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

```
        /**
         Returns the next value to be returned by next().
         Throws an IllegalStateExpression if hasNext() is false.
        */
        public String peek()
        {
            if (!hasNext())
               throw new IllegalStateException();
            return position.item;
        }
```

(continued)

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

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

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

        /**
         Changes the String in the node at location position.
         Throws an IllegalStateException if position is not at a node,
        */
        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**

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

    private Node head;

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

*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**

```
public class IteratorDemo
{
    public static void main(String[] args)
    {
        LinkedList2 list = new LinkedList2();
        LinkedList2.List2Iterator i = list.iterator();

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

(continued)




# Using an Iterator (Part 2 of 6)

**Display 15.18 Using an Iterator**

```
        System.out.println("List contains:");
        i.restart();
        while(i.hasNext())
            System.out.println(i.next());
        System.out.println();

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

(continued)

# Using an Iterator (Part 3 of 6)

Display 15.18 **Using an Iterator**

```
        System.out.println("List now contains:");
        i.restart();
        while(i.hasNext())
            System.out.println(i.next());
        System.out.println();

        i.restart();
        i.next();
        System.out.println("Will add one node before " + i.peek());
        i.addHere("socks");
        System.out.println("List now contains:");
        i.restart();
        while(i.hasNext())
            System.out.println(i.next());
```

(continued)




# Using an Iterator (Part 4 of 6)

Display 15.18 **Using an Iterator**

```
        System.out.println();
        System.out.println("Changing all items to credit card.");
        i.restart();
        while(i.hasNext())
        {
            i.changeHere("credit card");
            i.next();
        }
        System.out.println();

        System.out.println("List now contains:");
        i.restart();
        while(i.hasNext())
            System.out.println(i.next());
        System.out.println();
    }
}
```

(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`**:

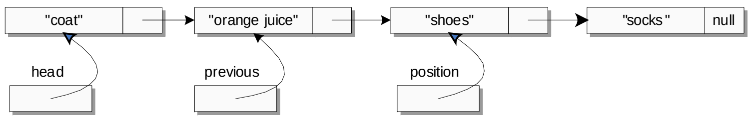
```
previous.link = position.link;
position = position.link;
```

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



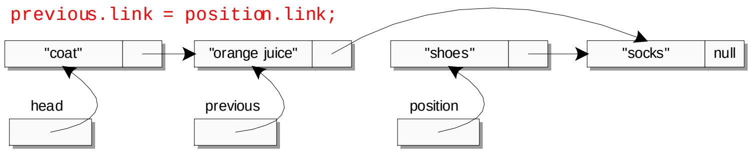

# Deleting a Node (Part 1 of 2)

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




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

```
previous.link = position.link;
```



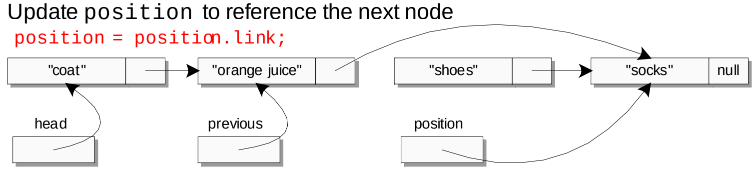




# Deleting a Node (Part 2 of 2)

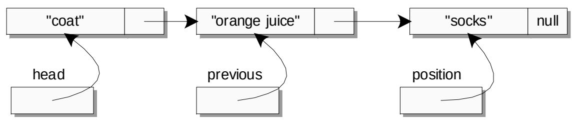
3. Update `position` to reference the next node

```
position = position.link;
```




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

4. Same picture with deleted node not shown



# Adding and Deleting Nodes

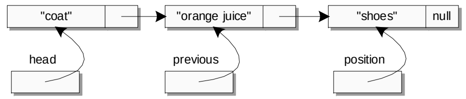
- Note that Java has *automatic garbage collection*
  - In many other languages the programmer has to keep track of deleted nodes and explicitly return their memory for recycling
  - This procedure is called *explicit memory management*
- The iterator variables **`position`** and **`previous`** can be 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;
```



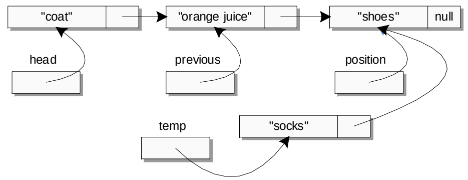

# Adding a Node between Two Nodes (Part 1 of 2)

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




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

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



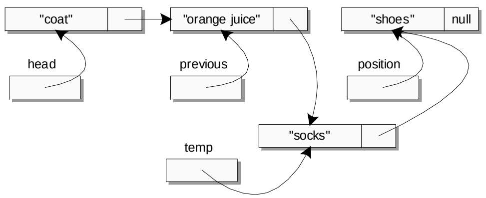




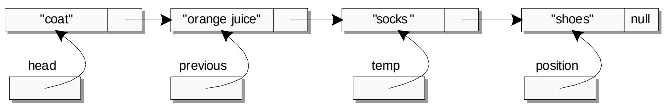
# Adding a Node between Two Nodes (Part 2 of 2)

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

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




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







# Variations on a Linked List

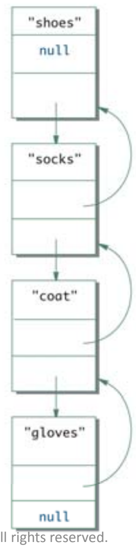
- An ordinary linked list allows movement in one direction only
- However, a doubly linked list has one link that references the next node, and one that references the previous node
- The node class for a doubly linked list can begin as follows:

```
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

Display 15.21 **A Doubly Linked List**

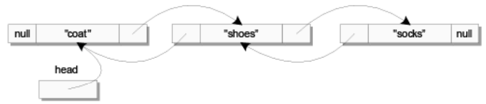






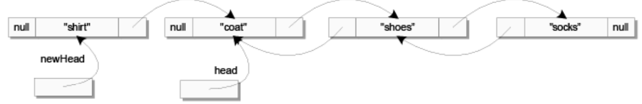

# Adding a Node to the Front of a Doubly Linked List

1. Existing list



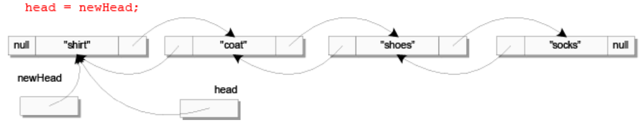

2. Create new TwoWayNode linked to "coat"

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




3. Set backward link and set new head

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

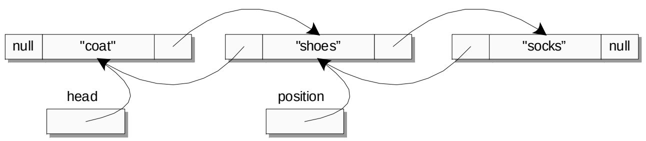






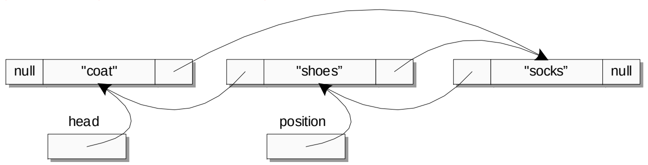

# Deleting a Node from a Doubly Linked List (1 of 2)

1. Existing list with an iterator referencing "shoes"




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

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



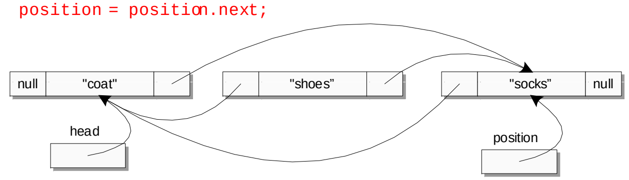




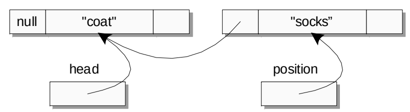

# Deleting a Node from a Doubly Linked List (2 of 2)

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;
```

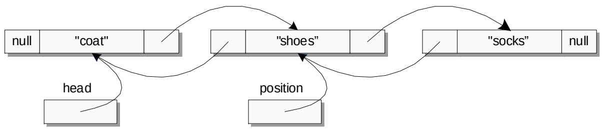



4. Picture redrawn for clarity with the "shoes" node removed since there are no longer references pointing to this node .



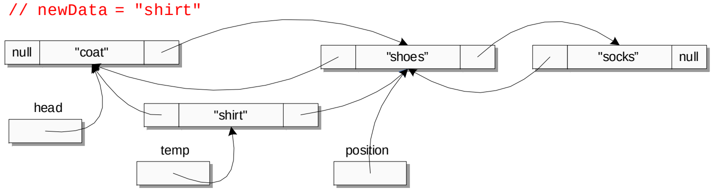
# Inserting a Node Into a Doubly Linked List (1 of 2)

1. Existing list with an iterator referencing "shoes"




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

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



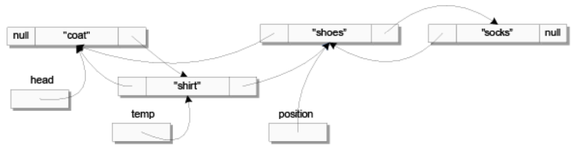





# Inserting a Node Into a Doubly Linked List (2 of 2)

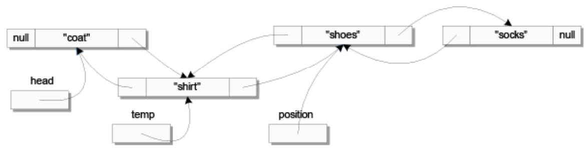
3. Set next link from "coat" to the new node of "shirt"

```
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**

```
public class Queue
{
    private class Node
    {
        private String item;
        private Node link;

        public Node()
        {
            item = null;
            link = null;
        }
```

(continued)





# A Queue Class (Part 2 of 5)

**A Queue Class**

```
        public Node(String newItem, Node linkValue)
        {
            item = newItem;
            link = linkValue;
        }
    }//End of Node inner class

    private Node front;
    private Node back;

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

(continued)





# A Queue Class (Part 3 of 5)

**A Queue Class**

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

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

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

(continued)





# A Queue Class (Part 4 of 5)

**A Queue Class**

```
    /**
     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)

**A Queue Class**

```
    /**
     Removes a String from the front of the queue.
     Returns false if the list is empty.
    */
    public boolean removeFront()
    {
        if (front != null)
        {
            front = front.link;
            return true;
        }
        else
            return false;
    }
}
```




# Demonstration of the Queue Class (Part 1 of 2)

**Demonstration of the Queue Class**

```
public class QueueDemo
{
    public static void main(String[] args)
    {
        Queue q = new Queue();

        q.addToBack("Tom");
        q.addToBack("Dick");
        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**

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

*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

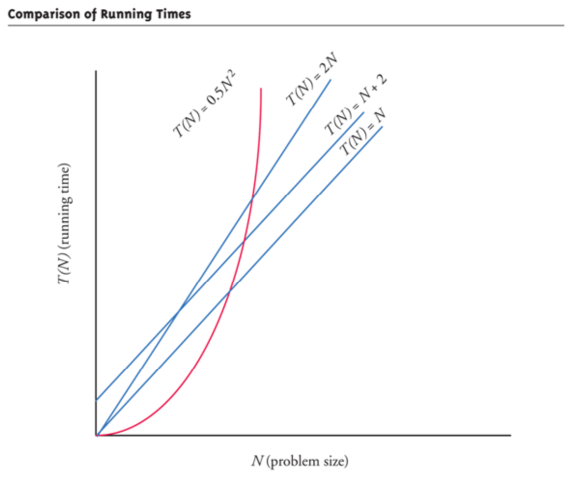



# Big-O Terminology

- 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







# 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



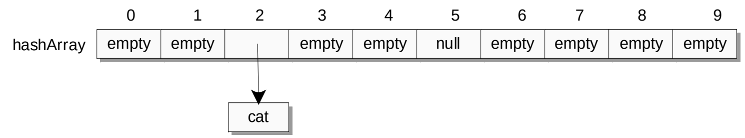

# Constructing a Hash Table (1 of 2)

1. Existing hash table initialized with ten empty linked lists

```
hashArray = new LinkedList3[SIZE];  // SIZE = 10
```

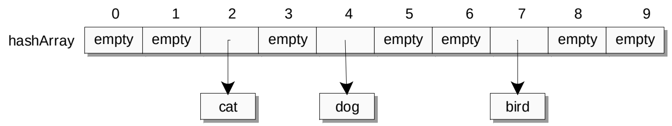
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| hashArray | empty | empty | empty | empty | empty | empty | empty | empty | empty | empty |

2. After adding "cat" with hash of 2

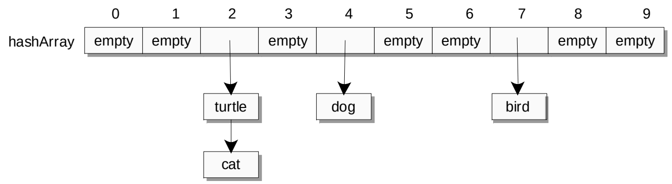


# Constructing a Hash Table (2 of 2)

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




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







# A Hash Table Class (1 of 3)

```
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;
        }
```




# A Hash Table Class (2 of 3)

```
        /**
         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 true;
                return false;
        }
```




# A Hash Table Class (3 of 3)

```
        /**
         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);
                }
        }
} // End HashTable class
```

# Hash Table Demonstration (1 of 2)

```
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)

```
                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

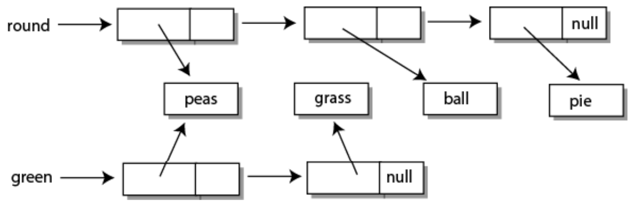
- Worst Case
  - Every item inserted into the table has the same hash key, the find operation may have to search through all items every time (same performance as a linked list, O(n) to find)
- Best Case
  - Every item inserted into the table has a different hash key, the find operation will only have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

```
      // Uses a linked list as the internal data structure
      // to store items in a set.
      public class Set<T>
      {
          private class Node<T>
          {
              private T data;
              private Node<T> link;

              public Node( )
              {
                   data = null;
                   link = null;
              }
              public Node(T newData, Node<T> linkValue)
              {
                  data = newData;
                  link = linkValue;
              }
           }//End of Node<T> inner class

          private Node<T> head;
```




# A Set Class (2 of 5)

```
        public Set()
        {
                 head = null;
        }
        /**
         Add a new item to the set.  If the item
         is already in the set, false is returned,
         otherwise true is returned.
        */
        public boolean add(T newItem)
        {
                if (!contains(newItem))
                {
                        head = new Node<T>(newItem, head);
                        return true;
                }
                return false;
        }
```




# A Set Class (3 of 5)

```
          public boolean contains(T item)
          {
              Node<T> position = head;
              T itemAtPosition;
              while (position != null)
              {
                  itemAtPosition = position.data;
                  if (itemAtPosition.equals(item))
                      return true;
                  position = position.link;
              }
              return false; //target was not found
          }

          public void output( )
          {
              Node position = head;
              while (position != null)
        {
                  System.out.print(position.data.toString() + " ");
                  position = position.link;
              }
              System.out.println();
          }
```

# A Set Class (4 of 5)

```
    /**
     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)

```
    /**
     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;
    }
}
```

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




# A Set Class Demo (1 of 3)

```
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("ball");
        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();
```




# A Set Class Demo (2 of 3)

```
        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)

```
            System.out.print("Union of green and round: ");
            round.union(green).output();
            System.out.print("Intersection of green and round: ");
            round.intersection(green).output();
        }
}
```

```
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"



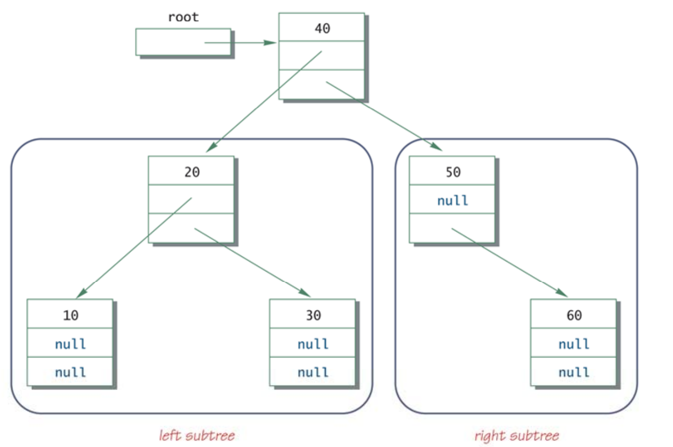

# Trees

- 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 1 of 2)

**A Binary Tree**




(continued)

# A Binary Tree (Part 2 of 2)

**A Binary Tree**

```
public class IntTree
{
    public class IntTreeNode
    {
        private int data;
        private IntTreeNode leftLink;
        private IntTreeNode rightLink;
    } //End of IntTreeNode inner class

    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)

**A Binary Search Tree for Integers**

```
/**
 Class invariant: The tree satisfies the binary search tree storage rule.
*/
public class IntTree
{
    private static class IntTreeNode
    {
        private int data;
        private IntTreeNode leftLink;
        private IntTreeNode rightLink;

```

*The only reason this inner class is static is that it is used in the static methods* `insertInSubtree`, `isInSubtree`, *and* `showElementsInSubtree`.

(continued)





# A Binary Search Tree for Integers (Part 2 of 6)

**A Binary Search Tree for Integers**

```
        public IntTreeNode(int newData, IntTreeNode newLeftLink,
                                        IntTreeNode newRightLink)
        {
            data = newData;
            leftLink = newLeftLink;
            rightLink = newRightLink;
        }
    } //End of IntTreeNode inner class

    private IntTreeNode root;

    public IntTree()
    {
        root = null;
    }
```

*This class should have more methods. This is just a sample of possible methods.*

(continued)

# A Binary Search Tree for Integers (Part 3 of 6)

**A Binary Search Tree for Integers**

```
    public void add(int item)
    {
        root = insertInSubtree(item, root);
    }

    public boolean contains(int item)
    {
        return isInSubtree(item, root);
    }

    public void showElements()
    {
        showElementsInSubtree(root);
    }
```

(continued)





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

**A Binary Search Tree for Integers**

```
    /**
     Returns the root node of a tree that is the tree with root node
     subTreeRoot, but with a new node added that contains item.
    */
    private static IntTreeNode insertInSubtree(int item,
                                               IntTreeNode subTreeRoot)
    {
        if (subTreeRoot == null)
            return new IntTreeNode(item, null, null);
        else if (item < subTreeRoot.data)
        {
            subTreeRoot.leftLink = insertInSubtree(item, subTreeRoot.leftLink);
            return subTreeRoot;
        }
```

(continued)





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

**A Binary Search Tree for Integers**

```
        else //item >= subTreeRoot.data
        {
            subTreeRoot.rightLink = insertInSubtree(item, subTreeRoot.rightLink);
            return subTreeRoot;
        }
    }

    private static boolean isInSubtree(int item, IntTreeNode subTreeRoot)
    {
        if (subTreeRoot == null)
            return false;
        else if (subTreeRoot.data == item)
            return true;
        else if (item < subTreeRoot.data)
            return isInSubtree(item, subTreeRoot.leftLink);
        else //item >= link.data
            return isInSubtree(item, subTreeRoot.rightLink);
    }
```

(continued)





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

**A Binary Search Tree for Integers**

```
    private static void showElementsInSubtree(IntTreeNode subTreeRoot)
    {//Uses inorder traversal.
        if (subTreeRoot != null)
        {
            showElementsInSubtree(subTreeRoot.leftLink);
            System.out.print(subTreeRoot.data + " ");
            showElementsInSubtree(subTreeRoot.rightLink);
        }//else do nothing. Empty tree has nothing to display.
    }
}
```

# Demonstration Program for the Binary Search Tree (Part 1 of 3)

**Demonstration Program for the Binary Search Tree**

```
import java.util.Scanner;

public class BinarySearchTreeDemo
{
   public static void main(String[] args)
   {
      Scanner keyboard = new Scanner(System.in);
      IntTree tree = new IntTree();
```

(continued)





# Demonstration Program for the Binary Search Tree (Part 2 of 3)

**Demonstration Program for the Binary Search Tree**

```
      System.out.println("Enter a list of nonnegative integers.");
      System.out.println("Place a negative integer at the end.");
      int next = keyboard.nextInt();
      while (next >= 0)
      {
         tree.add(next);
         next = keyboard.nextInt();
      }

      System.out.println("In sorted order:");
      tree.showElements();
   }
}
```

(continued)





# Demonstration Program for the Binary Search Tree (Part 3 of 3)

**Demonstration Program for the Binary Search Tree**

**Sample Dialogue**

```
Enter a list of nonnegative integers.
Place a negative integer at the end.
40
30
20
10
11
22
33
44
-1
In sorted order:
10 11 20 22 30 33 40 44
```





# Efficiency of Binary Search Trees

- A Binary search trees 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 `isInSubtree` 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

# 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*