```
class recursiveLinkedLists
{

   static class Node
   {
      String data;
      Node   next;

      Node(String data, Node next)
      {
         this.data = data;
         this.next = next;
      }
   }



////////////////////////////////////////////////////////////////
// Find the length of a linked-list

   static int length(Node p)
   {
      if(p == null)
         return 0;
      else
         return 1 + length(p.next);
    }


////////////////////////////////////////////////////////////////
// Print a linked-list

   /*
    * Recursion allows us flexibility in printing out
    * a list forwards or in reverse (by exchanging the
    * order of the recursive call)
    */

   static void printList(Node p)
   {
      if (p != null)
      {
          System.out.println(p.data);
          printList(p.next);
      }
   }


   static void printReverseList(Node p)
   {
      if (p != null)
      {
          printReverseList(p.next);
          System.out.println(p.data);
      }
   }


////////////////////////////////////////////////////////////////
// Copy a list

   static Node copy(Node p)
   {
```

```
      if(p == null)
         return null;
      else
         return new Node(p.data, copy(p.next));
   }


////////////////////////////////////////////////////////////////
// Reverse a linked-list

   /*
    * Take nodes one at a time from the head of
    * the "From" linked-list and add them to the
    * "to" linked-list.
    */
   static Node reverse(Node p)
   {
      return reverse(p, null);
   }

   static Node reverse(Node p, Node ancestor)
   {
      // empty list?
      if(p == null)
         return ancestor;

      // remember who the next node is
      Node theNextNode = p.next;

      // change the current node's ".next" pointer
      // to point to the ancestor we received when
      // we were called
      p.next = ancestor;

      // now recurse to invert the next node
      // in the list, (which we remembered
      // before: theNextNode) telling that
      // node that its new ancestor is us
      return reverse(theNextNode, p);
   }


////////////////////////////////////////////////////////////////
// Inserting an item into a sorted list

   static Node insertInOrder(String key, Node p)
   {
      if(p == null || p.data.compareTo(key) >= 0)
         return new Node(key, p);

      else
      {
         p.next = insertInOrder(key, p.next);
         return p;
      }
   }


////////////////////////////////////////////////////////////////
// Deleting an item from a list

   /*
    * This algorithm deletes the first occurrence of
    * an item from a list.  A simple change enables
```

```
     * this algorithm to delete all occurrences of the
     * item, by continuing to chain down the list
     * after the item has been found. We assume that
     * the list is unordered; you can easily change
     * this to stop after finding a item beyond the
     * search item by changing the first if condition.
     */

    static Node deleteInOrder(String key, Node p)
    {
       // if the list is ordered use:  (p == null || p.data > k)
       if(p == null)
          return p;

       else if (p.data.equals(key))
                              // if you want to delete
          return p.next;      // all instances, use:
                              // return deleteItem(key, p.next);

       else
       {
          p.next = deleteInOrder(key, p.next);
          return p;
       }
    }

////////////////////////////////////////////////////////////////
// Deleting the last Node of the list

    /*
     * This is a rather messy process in the iterative case;
     * the use of recursion makes it much simpler:
     */

    static Node deleteLast(Node p)
    {
       if(p == null || p.next == null)
          return null;

       else
       {
          p.next = deleteLast(p.next);
          return p;
       }
    }

////////////////////////////////////////////////////////////////



    static public void main(String[] args)
    {
       // create a new empty linked-list:
       Node head = null;

       // insert a node or two:
       head = new Node("apple",
                 new Node("banana",
                    new Node("cherries",
                       new Node("fig",
                          new Node("grapes", null)
                       )
                    )
```

```
                )
            );


        System.out.println("\nRecursive Count");
        System.out.println("There are: "
              + length(head) + " nodes in the linked-list");

        System.out.println("\nRecursive Print:");
        printList(head);

        System.out.println("\nRecursive Print (Reversed):");
        printReverseList(head);

        System.out.println("\nRecursive Copy:");
        Node copy = copy(head);
        printList(copy);

        System.out.println("\nRecursive Reverse a Linked-List:");
        Node invert = reverse(copy);
        printList(invert);

        System.out.println("\nRecursive Insert \"date\" into "
                                              + "an Ordered Linked-List");
        head = insertInOrder("dates", head);
        printList(head);

        System.out.println("\nRecursive Delete \"date\" from "
                                              + "an Ordered Linked-List");
        head = deleteInOrder("dates", head);
        printList(head);

        System.out.println("\nRecursive Delete the last node "
                                              + "from a Linked-List");
        head = deleteLast(head);
        printList(head);

        System.out.println("\nDone!");
    }


}
```