```java
import java.util.Iterator;
import java.util.NoSuchElementException;


/**
 * Stores a list of <T> elements
 *
 * @author 1828799
 *
 * @param <T> unspecified objects type T
 */
public class MyLinkedList<T> implements HansenCollection<T>,
Comparable<MyLinkedList<T>>, Iterable<T>
{
   // Instance Variables
 private  ListNode  _head  =  null;
 private  ListNode  _tail  =  null;
   private int _size = 0;


   /**
    * Creates a new MyLinkedList
    */
   public MyLinkedList(){}


   /**
    * Appends an element to the end of the list.
    *
    * @param newElement the specified element to be added
    *
    * @throws CollectionFullException (never thrown since list can never fill)
    * @throws NullPointerException if element to be added is null
    */
   @Override
   public void addElement(T newElement) throws CollectionFullException,
         NullPointerException
   {
      // Verify that the input isn't null
      if (newElement == null)
      {
         throw new NullPointerException();
      }

      // Add element using prependElement if the list is empty
      if (size() == 0)
      {
         prependElement(newElement);
      }
      else // When the list isn't empty
      {
         // Set newElement to be in the node which is the tail's next
         _tail.setNext(new ListNode<T>(newElement));

         // Update the tail
         _tail = _tail.getNext();

         // Increment the size
         _size++;

```

```
        }
    }


    /**
```

```java
    * Prepends an element to the list
    *
    * @param newElement the element to prepend to list
    *
    * @throws NullPointerException if element to be prepended is null
    */
   public void prependElement(T newElement) throws NullPointerException
   {
      // Create new node to hold newElement
      ListNode toPrepend = new ListNode<T>(newElement);

      // Verify that the input isn't null
      if (newElement == null)
      {
         throw new NullPointerException();
      }

      // Set the added nodes next to be the head
      toPrepend.setNext(_head);

      // Update the head to be the node holding newElement
      _head = toPrepend;

      // If size is zero then also set the new the new node to be the tail
      if (size() ==  0)
      {
         _tail = toPrepend;
      }

      // Increment the size
      _size++;
   }


   /**
    * Inserts the new element after the first instance of the specified element
    * in the list, if the specified element is in list.
    *
    * @param existingElement the element to insert the new element after
    * @param newObject the element to be added
    *
    * @throws NullPointerException if element to be inserted is null
    * @throws NoSuchElementException specified element is not in list
    * @throws CollectionFullException never thrown since never full
    */
   public void insertAfter(T existingObject, T newObject) throws
   NoSuchElementException, NullPointerException, CollectionFullException
   {
      ListNode nodeBefore = null;
      ListNode nodeToInsert = null;

      // Verify that the input isn't null
      if (existingObject == null || newObject == null)
      {
         throw new NullPointerException();
      }

      // Find node containing the existing object by identity (if it exists)
      nodeBefore = findNode(existingObject, false);

      // If the nodeBefore isn’t in the list throw exception
      if (nodeBefore == null)
      {
         throw new NoSuchElementException();
      }

      // Special Case: nodeBefore is the tail
      if (nodeBefore == _tail)
      {
         // Use addElement if nodeBefore is tail
         addElement(newObject);
      }
      else // Regular Case: nodeBefore is not the tail
      {
         // Create new node holding the newObject
         nodeToInsert = new ListNode<T>(newObject);

         // Set new node’s next to nodeBefore’s next
         nodeToInsert.setNext(nodeBefore.getNext());

         // Set nodeBefore’s next to new node
         nodeBefore.setNext(nodeToInsert);

         // Increment the size of list
         _size++;
      }
   }


   /**
    * Removes the first occurrence of the specified element from this list,
    * if it is present.
    * @param elementToRemove the specified element to be removed
    *
    * @throws NoSuchElementException if specified element is not in list
    */
   @Override
   public void removeElement(T elementToRemove) throws NoSuchElementException
   {
      // Set ListNode variable current to null
      ListNode current = _head;

      // Check for null input or an empty list
      if (elementToRemove == null || size() == 0)
      {
         throw new NoSuchElementException();
      }

      // Special Case: Element to remove is the head
      if (_head.getContents().equals(elementToRemove))
      {
         // Set head equal to head’s next
         _head = _head.getNext();
      }

      // Walk the list to find the node before one containing element to remove
      else // When element to remove isn’t the head
      {
         while (current.getNext() != null
               && !((current.getNext()).getContents().equals(elementToRemove)))
         {
            // Set current to current’s next
            current = current.getNext();
         }

         if (current.getNext() == null) // When element isn’t in the list
         {
            throw new NoSuchElementException();
         }
         else // When element is in the list
         {
            // Remove the element by manipulated the node before's next
            current.setNext((current.getNext()).getNext());
         }
      }

      // Decrement the size of the list
      _size--;
   }


   /**
    * Returns the element if element is in list
    *
    * @param elementSought the specified element to be found
    *
    * @return returns the element if found, returns null if element is not
    * found
    */
   @Override
   public T findElement(T elementSought)
   {
      // Set T object variable elementFound to null
      T elementFound = null;
      ListNode nodeFound = findNode(elementSought, true); // By equality

      // If the call findNode(elementSought) doesn't return null
      if (nodeFound != null)
      {
         // Set elementFound to content's of the node returned from findNode
         elementFound = (T) nodeFound.getContents();
      }

      return elementFound;
   }


   /**
    * Returns true if this list contains the specified element
    *
    * @param elementSought the specified element to be found in list
    *
    * @return true if specified element is in list
    */
   @Override
   public boolean containsElement(T elementSought)
   {
      // Use findElement to determine if elementSought is in list
      return (findElement(elementSought) != null );
   }


   /**
    * Compares the list to the specified list
    *
    * @param o the object to be compared
    *
    * @throws NullPointerException if the specified object is null
    *
    * @return Returns a negative integer, zero, or a positive integer as this
    * object is less than, equal to, or greater than specified object.
```

```
     */
    @Override
    public int compareTo(MyLinkedList<T> o)
    {

       int compareValue = 0; // The compared value to be returned
       ListNode myCurrent = _head; // Starting point of this list
       ListNode otherCurrent = o._head; // Starting point of other list

       // CHECK IF OBJECT TO BE COMPARED TO IS NULL
       // if o is null
       //    throw NullPointerException

       // Walk the list and compare at each step
       while (compareValue == 0 && myCurrent != null && otherCurrent != null)
       {
          // Set compareValue to return of comparing contents of two currents
          compareValue = ((Comparable) myCurrent.getContents()).compareTo((Comparable) otherCurrent.getContents());

          // Move each list the the next in list
          myCurrent = myCurrent.getNext();
          otherCurrent = otherCurrent.getNext();
       }

       // Check for when the lists are the same but one is longer
       if (compareValue == 0 && size() - o.size() != 0)
       {
          // When different sizes set return to be this size - other size
          compareValue = size() - o.size();
       }

       return compareValue;
    }


    /**
     * Returns false since list can never be full.
     *
     * @return always returns false since list can never be full
     */
    @Override
    public boolean isFull()
    {
       // WILL ALWAYS BE FALSE
       return false;
    }


    /**
     * Returns true if list contains no elements.
     *
     * @return true if list is empty
     */
    @Override
    public boolean isEmpty()
    {
       // Use size() to determine if list is empty
       return (size() == 0);
    }


    /**
     * Empties the list
```

```
     */
    @Override
    public void makeEmpty()
    {
        // Set head to null
```

```java
      _head = null;

      // Set tail to null
      _tail = null;

      // Set size to 0
      _size = 0;
   }


   /**
    * Returns the number of elements in list
    *
    * @return the number of objects in list
    */
   @Override
   public int size()
   {
      // Return the size that is tracked throughout
      return _size;
   }


   /**
    * Creates an iterator that can iterate over your list
    *
    * @return the new iterator
    */
   public Iterator<T> iterator()
   {
      return new Iterator<T>() {

         ListNode current = _head;

         /**
          * Returns true if the iteration has more elements
          *
          * @return  true if the iteration has more elements
          */
         public boolean hasNext()
         {
            return (current != null);
         }

         /**
          * Returns the next element in the iteration
          *
          * @return the next element in the iteration
          *
          * @throws NoSuchElementException if the iteration has no more elements
          */
         public T next() throws NoSuchElementException
         {
            if (!hasNext())
            {
               throw new NoSuchElementException();
            }

            T contents = (T) current.getContents();
            current = current.getNext();

```

```
           return contents;
        }

        /**
```

```java
          * Unsupported operation
          */
         public void remove(){throw new UnsupportedOperationException();}
      };
   }


   /**
    * Finds the node in the list that contains specified object, if such a node
    * is in the list.
    *
    * @param contentsOfNode the contents of the node to be found
    *
    * @return first node containing instance of specified object, null if no
    * node contains specified objects
    */
   private ListNode findNode(T contentsOfNode, boolean byEquality)
   {
      ListNode nodeFound = null; // Return value
      ListNode current = _head;

      // Walk the list to find the node containing specified elements
      while (byEquality && current != null && nodeFound == null)
      {
         // If current's contents equal contentsOfNode
         if (current.getContents().equals(contentsOfNode))
         {
            // Set nodeFound to be current
            nodeFound = current;
         }
         // Set current to current's next
         current = current.getNext();
      }

      while (!byEquality && current != null && nodeFound == null)
      {
         // If current's contents have same identity as contentsOfNode
         if (current.getContents() == contentsOfNode)
         {
            // Set nodeFound to be current
            nodeFound = current;
         }
         // Set current to current's next
         current = current.getNext();
      }

      return nodeFound;
   }


   /**
    * An inner-class for use by a Linked List to hold the contents of the list.
    * Note that this class and its methods are not public nor private, their
    * visibility is within the "package" and that includes any other classes
    * defined in the same file.  This class definition can appear INSIDE the
    * definition of a Linked List class.
    *
    * @author David M. Hansen
    * @version 2.0
    * @param < T > type of object contained by this node
    */
```

```
class ListNode < T >
{

   // Constructors
```

```

    /**
     * Create a new ListNode holding the given object and pointing to the
     * given node as the next node in the list
     * @param objectToHold the object to store in this node
     * @param nextNode the node this node should point to. Can be null
     */
    ListNode(T objectToHold, ListNode nextNode)
    {
       p_contents = objectToHold;
       p_next = nextNode;
    }

    /**
     * Create a new ListNode holding the given object.  No next node
     * @param objectToHold the object to store in this node
     */
    ListNode(T objectToHold)
    {
       // Use the more general constructor passing null as the next
       // node
       this(objectToHold, null);
    }


    // Accessors

    /**
     * @returns Object stored within this node
     */
    T getContents()
    {
       return p_contents;
    }

    /**
     * @return the next node
     */
    ListNode getNext()
    {
       return p_next;
    }



    // Mutators

    /**
     * Set the node this node is linked to
     * @param nextNode the node to point to as our next node. Can be null.
     */
    void
```

```
setNext(ListNode nextNode)
           {
              p_next
= nextNode;
           }


           // Private attributes
           private T p_contents; // The object held by
this node private ListNode p_next; // A reference
to the next node
        } // ListNode


}
```