```
using System;

namespace SimpleLList
{
    [Serializable]
    public class Node<T>
    {
        public Node(T nodeData)
        {
            Data = nodeData;
            Next = null;
        }

        public T Data { get; set; }
        public Node<T> Next { get; set; }
    }
}
```

```
/*********************************************************************/
/* Jacob Hobbie                                                      */
/* November 22, 2015                                                 */
/* Programming 2                                                     */
/*                                                                   */
/* This class library should be used to create a custome list        */
/* implementation that is a generic types and can take all sorts of  */
/* different kinds of data. Contains a driver to test.               */
/*********************************************************************/

using System;
using System.Collections;
using System.Collections.Generic;

namespace SimpleLList
{
    [Serializable]
    public class NodeList<T> where T : IEnumerable, IComparable<T>
    {
        private static Node<T> current;
        private static Node<T> previous;
        private Node<T> _head;
        private Node<T> _tail;

        // Constructs NodeList on first run
        public NodeList()
        {
            _head = null;
        }

        // Returns the number of nodes that the list contains
        public int Count
        {
            get
            {
                int index = 1;
                current = _head;

                if (current == null)
                {
                    index = 0;
                }
                else
                {
                    while (current.Next != null)
                    {
                        current = current.Next;
                        index++;
                    }
                }

                return index;
            }
        }

        // Allows an index to be used to access any of the nodes in
        // the chain
        public T this[int i]
        {
            get
            {
                current = _head;

                for (int index = 0; index < i; index++)
```

```
        {
            current = current.Next;
        }

        return current.Data;
    }
}

// Deletes node in list by simply skipping over it
public void Delete(T target)
{
    try
    {
        if (_head != _tail)
        {
            current = _head;
            previous = null;

            while (target.CompareTo(current.Data) != 0)
            {
                previous = current;
                current = current.Next;
            }

            if (current.Next == null)
            {
                previous.Next = null;
            }
            else
            {
                if (previous == null)
                {
                    _head = current.Next;
                }
                else
                {
                    previous.Next = current.Next;
                }
            }
        }
        else
        {
            _head = null;
            _tail = null;
        }
    }
    catch (NullReferenceException)
    {
        throw new NullReferenceException();
    }
}

// Returns the pointer node containing data of some type
public Node<T> FindNode(T record)
{
    try
    {
        current = _head;

        while (record.CompareTo(current.Data) != 0)
        {
            previous = current;
            current = current.Next;
        }
    }
```

```
    catch (NullReferenceException)
    {
        current = null;
    }

    return current;
}

// Allows this list to Enumerate and use foreach
public IEnumerator<T> GetEnumerator()
{
    current = _head;

    while (current != null)
    {
        yield return current.Data;
        current = current.Next;
    }
}

// Inserts a new data record received from the program in the single
// list
public void Insert(T newRecord)
{
    current = _head;
    previous = null;

    if (current == null)
    {
        _head = new Node<T>(newRecord);
        _tail = new Node<T>(newRecord);
    }
    else
    {
        if (current.Next == null && previous == null)
        {
            _head.Next = new Node<T>(newRecord);
            _tail = _head.Next;
        }
        else
        {
            while (newRecord.CompareTo(current.Data) >= 0)
            {
                if (current.Next == null)
                {
                    break;
                }
                previous = current;
                current = current.Next;
            }

            if (current.Next == null)
            {
                current.Next = new Node<T>(newRecord);
            }
            else
            {
                if (previous == null)
                {
                    _head = new Node<T>(newRecord);
                    _head.Next = current;
                }
                else
                {
                    previous.Next = new Node<T>(newRecord);
```

```
                        previous.Next.Next = current;
                    }
                }
            }
        }
    }
}
```