Blue Tree Systems Technical Test

Please find a zip attached.

The code has been built in C#, with .NET 4.5 in Visual Studio 2013.

You should be able to view the .cs file at least in order to see the code. I’ve also added some snippets to this document.

**1) Write the following function in either java or c#:**

bool ExpressionMatches(string text, string pattern)

where:

* 'text' is an ASCII string
* 'pattern' is an ASCII string which can include 0 or more instances of the wildcard "\*", which represents an arbitrary substring of any length (0 or more characters)

return value: the function returns true if "text" matches the pattern represented by "pattern"

For example, the following is expected (text, pattern, return value):

"This is a test string", "T\*test\*string", true

"This is a test string", "\*test\*", true

"This is a test string", "\*", true

"This is a test string", "\*This is \*", true

"This is a test string", "A\*string", false

"This is a test string", "This is a test string", true

"This is a test string", "This is a test", false

"This is a test string", "This is\*\*\*", true

"test testing", "\*testin\*", true

*See StringTests.QuestionOne and its corresponding unit tests, StringsTests.Tests.QuestionOneTests.*

public class QuestionOne

   {

       public bool ExpressionMatches(string text, string pattern)

       {

           bool result = true;

           if (text == null)

               return false;

           if (pattern != null)

           {

               var isAscii = ValidateString(text, "String to match must be ASCII.");

               isAscii = ValidateString(pattern, "Pattern must be ASCII.");

               string patternWithWildCard = pattern.Replace("\*", ".\*?");

               Regex regex = new Regex(patternWithWildCard);

               result = regex.IsMatch(text);

           }

           return result;

       }

       private static bool ValidateString(string value, string message)

       {

           var isAscii = Encoding.UTF8.GetByteCount(value) == value.Length;

           if (!isAscii)

               throw new Exception(message);

           return isAscii;

       }

   }

UNIT TESTS

//where:

   //• 'text' is an ASCII string

   //• 'pattern' is an ASCII string which can include 0 or more instances of the wildcard "\*", which represents an arbitrary substring of any length (0 or more characters)

   //return value: the function returns true if "text" matches the pattern represented by "pattern"

   //For example, the following is expected (text, pattern, return value):

   // A:  "This is a test string", "T\*test\*string", true

   // B:  "This is a test string", "\*test\*", true

   // C: "This is a test string", "\*", true

   // D:  "This is a test string", "\*This is \*", true

   // E:  "This is a test string", "A\*string", false

   // F:  "This is a test string", "This is a test string", true

   // G:  "This is a test string", "This is a test", false

   // H:  "This is a test string", "This is\*\*\*", true

   // I:  "test testing", "\*testin\*", true

   [TestClass()]

   public class QuestionOneTests

   {

       [TestMethod()]

       [ExpectedException(typeof(Exception), "String to match must be ASCII.")]

       public void ExpressionMatches\_TextASCII\_Test()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           string text = "abc志";

           string pattern = "a";

           //Act

           bool actual = first.ExpressionMatches(text, pattern);

           //Assert

       }

       [TestMethod()]

       [ExpectedException(typeof(Exception), "Pattern must be ASCII.")]

       public void ExpressionMatches\_PatternASCII\_Test()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           string text = "abc";

           string pattern = "a志";

           //Act

           bool actual = first.ExpressionMatches(text, pattern);

           //Assert

       }

       [TestMethod()]

       public void ExpressionMatches\_SimplePassTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "abc";

           string pattern = "a";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_SimpleFailTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = false;

           bool actual = false;

           string text = "abc";

           string pattern = "d";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_NullTextTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = false;

           bool actual = false;

           string text = null;

           string pattern = "d";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_NullPatternTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "";

           string pattern = null;

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_EmptyTextTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = string.Empty;

           string pattern = "";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_EmptyPatternTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "abc";

           string pattern = string.Empty;

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_WildCardPassTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "abc";

           string pattern = "a\*";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_WildCardFailTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = false;

           bool actual = false;

           string text = "abc";

           string pattern = "d\*";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       /// <summary>

       /// A:  "This is a test string", "T\*test\*string", true

       /// </summary>

       [TestMethod()]

       public void ExpressionMatches\_ATest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "This is a test string";

           string pattern = "T\*test\*string";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       /// <summary>

       /// B:  "This is a test string", "\*test\*", true

       /// </summary>

       [TestMethod()]

       public void ExpressionMatches\_BTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "This is a test string";

           string pattern = "\*test\*";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       /// <summary>

       /// C: "This is a test string", "\*", true

       /// </summary>

       [TestMethod()]

       public void ExpressionMatches\_CTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "This is a test string";

           string pattern = "\*";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       /// <summary>

       /// D:  "This is a test string", "\*This is \*", true

       /// </summary>

       [TestMethod()]

       public void ExpressionMatches\_DTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "This is a test string";

           string pattern = "\*This is \*";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       /// <summary>

       /// E:  "This is a test string", "A\*string", false

       /// </summary>

       [TestMethod()]

       public void ExpressionMatches\_ETest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = false;

           bool actual = false;

           string text = "This is a test string";

           string pattern = "A\*string";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       /// <summary>

       /// F:  "This is a test string", "This is a test string", true

       /// </summary>

       [TestMethod()]

       public void ExpressionMatches\_FTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "This is a test string";

           string pattern = "This is a test string";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       /// <summary>

       /// G:  "This is a test string", "This is a test", false

       /// </summary>

       [TestMethod()]

       public void ExpressionMatches\_GTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = false;

           bool actual = false;

           //string text = "This is a test string";

           //string pattern = "This is a test";

           //Act

           //actual = first.ExpressionMatches(text, pattern);//todo fix.

           //Assert

           Assert.AreEqual(expected, actual);

       }

       /// <summary>

       /// H:  "This is a test string", "This is\*\*\*", true

       /// </summary>

       [TestMethod()]

       public void ExpressionMatches\_HTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "This is a test string";

           string pattern = "This is\*\*\*";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       /// <summary>

       /// I:  "test testing", "\*testin\*", true

       /// </summary>

       [TestMethod()]

       public void ExpressionMatches\_ITest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "test testing";

           string pattern = "\*testin\*";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

   }

**2) Are there any additional test cases that should be considered?**

*Text is null.*

*Pattern is null.*

*Text is empty string.*

*Pattern is empty string.*

*See testcases StringsTests.Tests.QuestionOneTests.. Assumptions made.*

*Also unsure how to handle non ASCII strings. I have assumed they are invalid and an Exception is thrown. This could also be construed that the string should be converted to ASCII as best possible.*

 [TestMethod()]

       public void ExpressionMatches\_NullTextTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = false;

           bool actual = false;

           string text = null;

           string pattern = "d";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_NullPatternTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "";

           string pattern = null;

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_EmptyTextTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = string.Empty;

           string pattern = "";

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void ExpressionMatches\_EmptyPatternTest()

       {

           //Arange

           QuestionOne first = new QuestionOne();

           bool expected = true;

           bool actual = false;

           string text = "abc";

           string pattern = string.Empty;

           //Act

           actual = first.ExpressionMatches(text, pattern);

           //Assert

           Assert.AreEqual(expected, actual);

       }

**3) String permutations**

Write a function that prints (to standard output) all possible permutations of an input string. For example, permute("abc") should print (not necessarily in this order):

abc acb bac bca cab cba

*See answer StringTests.QuestionThree.Permute(string).*

public class QuestionThree

    {

        /// <summary>

        /// Write a function that prints (to standard output) all possible permutations of an input string.

        /// For example, permute("abc") should print (not necessarily in this order): abc acb bac bca cab cba

        /// </summary>

        /// <param name="value"></param>

        public static List<string> Permute(string value)

        {

            if (value == null)

                return null;

            var result = GetPermutation(value.ToCharArray());

            foreach (var s in result)

            {

                Console.Write(s + " ");

            }

            return result.ToList();

        }

        private static void Swap(ref char a, ref char b)

        {

            if (a == b) return;

            a ^= b;

            b ^= a;

            a ^= b;

        }

        private static List<string> GetPermutation(char[] list)

        {

            int x = list.Length - 1;

            List<string> result = new List<string>();

            return GetPermutation(list, 0, x, result);

        }

        private static List<string> GetPermutation(char[] list, int index, int count, List<string> result)

        {

            if (index == count)

            {

                result.Add( new string(list));

            }

            else

                for (int i = index; i <= count; i++)

                {

                    Swap(ref list[index], ref list[i]);

                    GetPermutation(list, index + 1, count, result);

                    Swap(ref list[index], ref list[i]);

                }

            return result;

        }

    }

UNIT TESTS

/// <summary>

    /// Write a function that prints (to standard output) all possible permutations of an input string.

    /// For example, permute("abc") should print (not necessarily in this order): abc acb bac bca cab cba

    /// </summary>

    [TestClass()]

    public class QuestionThreeTests

    {

        [TestMethod()]

        public void PermuteTest()

        {

            //Arrange

            List<string> expected = new List<string>() { "abc", "acb", "bac", "bca", "cab", "cba" };

            List<string> actual = null;

            //Act

            actual = QuestionThree.Permute("abc");

            //Assert

            CollectionAssert.AreEquivalent(expected, actual);

        }

        [TestMethod()]

        public void PermuteNullTest()

        {

            //Arrange

            List<string> expected = null;

            List<string> actual = null;

            //Act

            actual = QuestionThree.Permute(null);

            //Assert

            CollectionAssert.AreEquivalent(expected, actual);

        }

    }

**3) Database Schema**

A logging device (logger) can be installed on a truck, trailer, van, etc (vehicle). A logger has a unique serial number and records periodic GPS locations as (timestamp, latitude, longitude). A vehicle has a unique registration number.

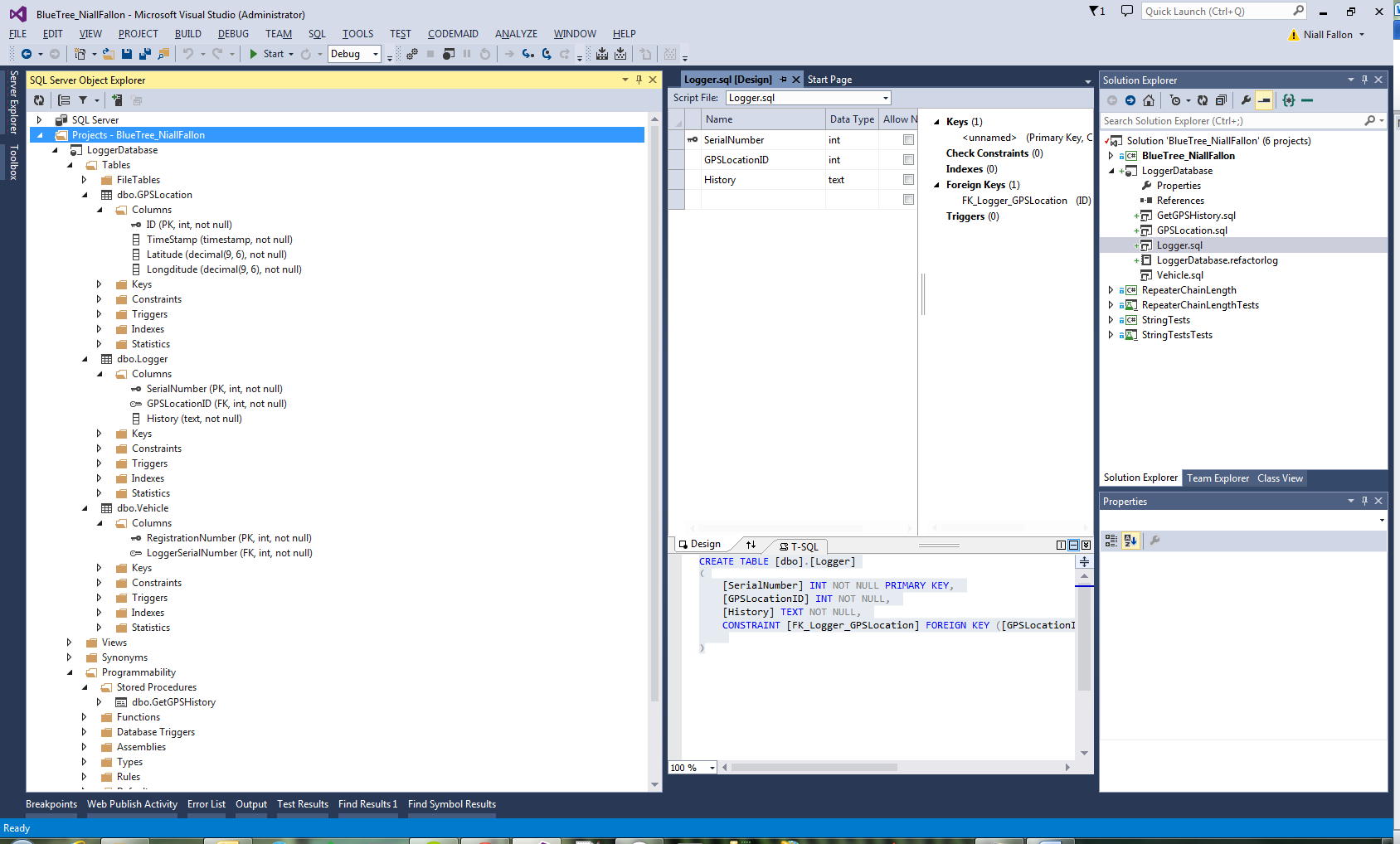
If a logger is damaged, it will be removed from the vehicle and a new logger (with a different serial number) fitted. The broken logger will then be repaired and eventually fitted on a different vehicle.

The logged GPS locations in each logger are periodically downloaded and stored in a database.

Design a database schema that allows us to query the GPS location history of a vehicle, regardless of any replacement of loggers on vehicles.

*There a Database project in the solution, see the LoggerDatabase project.*

*I haven’t figured out how to export the Schema from Visual Studio 2013 so here is a screendump.*



**4) Repeater chain length**

In one type of simple radio network, transceivers can operate on one of 20 fixed frequencies, which we will call channels 0 to 19. Each transceiver can only cover a limited distance, beyond which the signal is too weak to be received.

The "central" transceiver always operates on channel 0. To extend the network coverage, repeaters are installed which receive a signal on one channel and re-transmit the signal on another channel. Each repeater can be described as a pair of channel numbers, the "upstream" channel number (which is "closer" to the central node) and the "downstream" channel number (which is "further" from the central node). No repeater can have a downstream channel of 0 (since this is used by the central node) and no two repeaters can have the same downstream channel (since their transmissions would interfere).

**Questions:**

1. Given a list of repeater channel number pairs, determine the longest chain of repeaters in the network, and identify any repeaters that are "orphaned" because there is no chain that connects them back to the central node.

For example, for this set or repeaters (upstream/downstream):

0/1, 1/2, 3/4, 0/5

the longest chain is two repeaters and repeater 3/4 is orphaned.

*See answer at RepeaterChainLength.SimpleRadioStation. With corresponding tests at RepeaterChainLengthTests.SimpleRadioStationTests*

/// <summary>

    /// 4) Repeater chain length

    /// In one type of simple radio network, transceivers can operate on one of 20 fixed frequencies, which we will call channels 0 to 19.

    /// Each transceiver can only cover a limited distance, beyond which the signal is too weak to be received.

    /// The "central" transceiver always operates on channel 0.

    /// To extend the network coverage, repeaters are installed which receive a signal on one channel and re-transmit the signal on another channel.

    /// Each repeater can be described as a pair of channel numbers, the "upstream" channel number (which is "closer" to the central node) and the "downstream" channel number

    /// (which is "further" from the central node).

    /// No repeater can have a downstream channel of 0 (since this is used by the central node) and no two repeaters can have the same downstream channel (since their transmissions would interfere).

    /// Questions:

    /// 1. Given a list of repeater channel number pairs, determine the longest chain of repeaters in the network,

    /// and identify any repeaters that are "orphaned" because there is no chain that connects them back to the central node.

    /// For example, for this set or repeaters (upstream/downstream):

    /// 0/1, 1/2, 3/4, 0/5

    /// the longest chain is two repeaters and repeater 3/4 is orphaned.

    /// </summary>

    public class SimpleRadioStation

    {

        List<Transceiver> \_tranceivers = new List<Transceiver>();

        public SimpleRadioStation()

        {

            CentralReceiver centralReceiver = new CentralReceiver();

            \_tranceivers.Add(centralReceiver);

        }

        public void AddRepeater(int upstreamFrequency, int downStreamFrequency)

        {

            var repeater = new Repeater(upstreamFrequency, downStreamFrequency);

            repeater.Index = \_tranceivers.Count + 1;

            ValidateDownStreamChannel(repeater);

            \_tranceivers.Add(repeater);

        }

        private void ValidateDownStreamChannel(Repeater repeater)

        {

            foreach(var existingRepeater in \_tranceivers)

            {

                if (existingRepeater.DownStreamFrequency == repeater.DownStreamFrequency)

                    throw new Exception(string.Format("An Transeiver with the same Downstream Frequency {0} already exists", repeater.DownStreamFrequency));

            }

        }

        public int GetLongestChainOfRepeaters()

        {

            int result = 0;

            //start with the down.

            //get its up.

            //get the next repeater with a down value matching the up.

            //increment the count.

            //repeat.

            int receiverCount = 0;

            foreach (var receiver in \_tranceivers)

            {

                var currentLength = CalculateLongest(receiver, receiverCount);

                if (currentLength > result)

                    result = currentLength;

                receiverCount++;

            }

            return result;

        }

        private int CalculateLongest(Transceiver repeater, int receiverCount)

        {

            int length = 0;

            //start with the down.

            //get its up.

            //get the next repeater with a down value matching the up.

            //increment the count.

            //repeat.

            var down = repeater.DownStreamFrequency;

            var up = repeater.UpStreamFrequency;

            foreach (var nextRepeater in \_tranceivers)

            {

                if (repeater.Index != nextRepeater.Index)//don't test self.

                {

                    var nextRepeaterDown = nextRepeater.DownStreamFrequency;

                    var nextRepeaterUP = nextRepeater.UpStreamFrequency;

                    if (down == nextRepeaterUP)

                        length++;

                }

            }

            return length;

        }

        public string GetOrphanedRepeatersAsString()

        {

            string result = null;

            StringBuilder sb = new StringBuilder();

            var orphanedRepeaters = GetOrphanedRepeaters();

            foreach (var repeater in orphanedRepeaters)

                sb.Append(repeater.ToString());

            result = sb.ToString();

            return result;

        }

        private List<Transceiver> GetOrphanedRepeaters()

        {

            List<Transceiver> result = new List<Transceiver>();

            //start with the down.

            //get its up.

            //get the next repeater with a down value matching the up.

            //increment the count.

            //repeat.

            int receiverCount = 0;

            foreach (var receiver in \_tranceivers)

            {

                var orphaned = CalculateOrphans(receiver, receiverCount);

                if (orphaned != null)

                    result.Add(orphaned);

                receiverCount++;

            }

            return result;

        }

        private Transceiver CalculateOrphans(Transceiver repeater, int receiverCount)

        {

            if (!(repeater is Repeater))

                return null;

            Transceiver result = repeater;

            //get its down.

            //traverse through ever other repeater and check it's up.

            //if there is not a single match then I am an orphan.

            var down = repeater.DownStreamFrequency;

            var up = repeater.UpStreamFrequency;

            foreach (var nextRepeater in \_tranceivers)

            {

                if (repeater.Index != nextRepeater.Index)//don't test self.

                {

                    var nextRepeaterDown = nextRepeater.DownStreamFrequency;

                    var nextRepeaterUP = nextRepeater.UpStreamFrequency;

                    if (up == nextRepeaterDown)

                        return null;

                }

            }

            return result;

        }

    }

    abstract class Transceiver

    {

        List<int> \_frequencies = new List<int>(20);

        int \_index;

        public int Index

        {

            get { return \_index; }

            set { \_index = value; }

        }

        int \_upStreamFrequency;

        /// <summary>

        /// closer to central node

        /// </summary>

        public int UpStreamFrequency

        {

            get { return \_upStreamFrequency; }

            set {

                \_upStreamFrequency = value; }

        }

        int \_downStreamFrequency;

        /// <summary>

        /// further from central node

        /// </summary>

        public virtual int DownStreamFrequency

        {

            get { return \_downStreamFrequency; }

            set

            {

                if (value > 19)

                    throw new Exception("Downstream Channel too large.");

                \_downStreamFrequency = value;

            }

        }

        public Transceiver(int upStreamFrequency, int downStreamFrequency)

        {

            UpStreamFrequency = upStreamFrequency;

            DownStreamFrequency = downStreamFrequency;

        }

        public override string ToString()

        {

            return UpStreamFrequency + "/" + DownStreamFrequency;

        }

    }

    internal class CentralReceiver : Transceiver

    {

        public CentralReceiver() : base(0, 0)

        {

        }

    }

    internal class Repeater : Transceiver

    {

        public Repeater(int upStreamFrequency, int downStreamFrequency) : base(upStreamFrequency, downStreamFrequency)

        {

        }

        /// <summary>

        /// further from central node

        /// </summary>

        public override int DownStreamFrequency

        {

            get { return base.DownStreamFrequency; }

            set

            {

                if (value <= 0)

                    throw new Exception("DownStream Frequency cannot be less than 0.");

                base.DownStreamFrequency = value;

            }

        }

    }

Unit Tests

/// <summary>

   ///  1. Given a list of repeater channel number pairs, determine the longest chain of repeaters in the network,

   /// and identify any repeaters that are "orphaned" because there is no chain that connects them back to the central node.

   /// For example, for this set or repeaters (upstream/downstream):

   /// 0/1, 1/2, 3/4, 0/5

   /// the longest chain is two repeaters and repeater 3/4 is orphaned.

   /// </summary>

   [TestClass()]

   public class SimpleRadioStationTests

   {

       [TestMethod()]

       public void GetLongestChainOfRepeatersTest()

       {

           //Arrange

           SimpleRadioStation simpleRadioStation = new SimpleRadioStation();

           int expected = 2;

           int actual = 0;

           //Act

           simpleRadioStation.AddRepeater(0, 1);

           simpleRadioStation.AddRepeater(1, 2);

           simpleRadioStation.AddRepeater(3, 4);

           simpleRadioStation.AddRepeater(0, 5);

           actual = simpleRadioStation.GetLongestChainOfRepeaters();

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [TestMethod()]

       public void GetOrphanedRepeatersTest()

       {

           //Arrange

           SimpleRadioStation simpleRadioStation = new SimpleRadioStation();

           string expected = "3/4";

           string actual = string.Empty;

           //Act

           simpleRadioStation.AddRepeater(0, 1);

           simpleRadioStation.AddRepeater(1, 2);

           simpleRadioStation.AddRepeater(3, 4);

           simpleRadioStation.AddRepeater(0, 5);

           actual = simpleRadioStation.GetOrphanedRepeatersAsString();

           //Assert

           Assert.AreEqual(expected, actual);

       }

       [ExpectedException(typeof(Exception), "An Transeiver with the same Downstream Frequency 5 already exists")]

       [TestMethod()]

       public void DuplicateDownStreamTest()

       {

           //Arrange

           SimpleRadioStation simpleRadioStation = new SimpleRadioStation();

           //Act

           simpleRadioStation.AddRepeater(0,1);

           simpleRadioStation.AddRepeater(1,2);

           simpleRadioStation.AddRepeater(0,5);

           simpleRadioStation.AddRepeater(0,5);

       }

       [ExpectedException(typeof(Exception), "DownStream Frequency cannot be less than 0.")]

       [TestMethod()]

       public void TooLowDownStreamTest()

       {

           //Arrange

           SimpleRadioStation simpleRadioStation = new SimpleRadioStation();

           //Act

           simpleRadioStation.AddRepeater(0, 1);

           simpleRadioStation.AddRepeater(1, 2);

           simpleRadioStation.AddRepeater(0, 5);

           simpleRadioStation.AddRepeater(0, 5);

       }

   }

2. What is the approximate cost of your implementation (in big-O notation)?

*My simple solution I have is not scalable and even lacks recursion. Lists are probably not efficient either. Something like a LinkedList might be more effective but would take more time to implement and design.*

3. If the number of possible channels (and hence possible repeaters) was much larger (say 100000), would you implement it differently?

*With more time I would attempt to use LinkedLists for this example. Maybe even implement with some Visitor Pattern to do the analysis of child/parent.*