1. Description and justification of tests:
   * 1. testTree2PrefixIllegal():

Purpose**:** This is a series of tests, they focus on the mainly 3 illegal input types, to see if the program can handle the illegal input correctly.

Input: null tree, illegal tree has only operators, illegal tree has only operands.

Expected output: throw IllegalArgumentException();

* + 1. testTree2PrefixExample():

Purpose: this test is to see if the program can convert the legal tree to its corresponding prefix formula correctly.

Input: a simple, legal tree. With prefix “+ 1 – 2 3”

Expected output: the prefix formula corresponding to the tree.

* + 1. testTree2InfixIllegal1():

Purpose: this series of test is meant to test how the Tree2Infix method handle illegal inputs, note that it also tests how the IsArithmetric() method work, as this method is called to check the legality of the tree.

Input: null tree, illegal tree

Expected output: throw IllegalArgumentException();

* + 1. testTree2infixExample():

Purpose: this test is to check if the program can convert the legal tree to its corresponding infix formula correctly.

Input: a simple, legal tree. With prefix “+ 1 – 2 3”

Expected output: the infix formula corresponding to the tree.

* + 1. testSimplifyExample():

Purpose: this test is designed to check if the simplify method can function correctly. The input tree is a simple tree that has nodes can be simplified along with variables, the result should be the simplified form of the tree, with variables unchanged.

Input: a simple, legal tree, with numbers, operators, and variables. With prefix “+ - 2 2 c”

Expected output: the simplified tree, whose infix form should be “+ 0 c”

* + 1. testSimplifyFancyExample():

Purpose: this test is designed to check if the simplify fancy method can function correctly. The input tree is same as the one used in simplify example, the difference is that this tree can be further simplified if the fancy rules are applied.

Input: a simple legal tree, with numbers, operators, and variables. With prefix “+ - 2 2 c”

Expected output: tree with one root node “c”

* + 1. testSimplifyFancyBigTree()

Purpose: this test is designed to check if the simplify fancy method can function correctly, on a tree that fancy rules can be applied on a bigger, ‘branch’ level. The input tree is somewhat different from the one used in previous example.

Input: a simple legal tree, with numbers, operators, and variables. With prefix”+ 0 - c 1”

Expected output: tree with prefix “- c 1”

* + 1. testFancyMultiVariable():

Purpose: this test is designed to check if the simplify fancy method can handle multi variable ( eg. 0 + x + y + z ) correctly.

Input: a tree with numbers that can be simplified and multi variables. With prefix “+ y \* x 0”

Expected output: correctly simplified tree, with each variable simplified by fancy rules. The prefix should be “y”

* + 1. testSubstituteNull():

Purpose: to test if the substitute method can handle null input correctly.

Input: null string value

Expected output: throw IllegalArgumentException();

* + 1. testSubstitute():

Purpose: to test if the substitute method can function normally. This is not only for the replacement of element, but also the element should be usable for further simplify calculation.

Input: a simple tree with prefix “+ 0 2 2 c”, variable for substitute: “c”, value for substitute: “0”.

Expected output: the tree with variable substituted. With prefix “+ 0 2 2 0”

* + 1. testSubstituteMap():

purpose: to test if the substitute method can function normally when a map of variables& values are given.

Input: a simple tree with multiple variables,with prefix “+ c – a b” ;

a map with multiple variables-value pairs, are:<a,1>, <b,5>, <c,3>

Expected output: the tree with all variables substituted. With prefix “+ 3 – 1 5”

* + 1. testSubstitudeNullMap():

purpose: to test if the method can handle null map input correctly.

Input: a simple tree with multiple variables, a null map.

Expected output: throw IllegalArgumentException();

* + 1. testSubstitudeNullElement():

purpose: to test if the method can handle map with null element correctly.

Input: a simple tree with multiple variables, a map with null values.

Expected output: throw IllegalArgumentException();

* + 1. testIsArithmetricNull():

purpose: to test if this method can correctly return value for the null tree.

Input: a null tree

Expected output: false.

1. Analysis of run-time cost.
   * 1. public static LinkedBinaryTree<String> prefix2tree(String expression)

Algorithm: recursive pre-order traversal

Worst case run time: **O(n)=O(n)[traversing through the tree] + nO(1)[giving node value]**.

As this method is actually ‘constructing’ a new tree, in the helper method, every node of the new tree will be reached, and each node will have a O(1) operation to give them value.

* + 1. public static boolean equals(LinkedBinaryTree<String> a, LinkedBinaryTree<String> b)

Algorithm: recursive pre-order traversal.

Worst case run time: **O(n)=O(n)[traversing through the tree] + nO(1)[compare element value]**.

As the method is comparing every one single node of two trees. In the worst case scenario, the two trees only has the last node different. Thus, every node is visited. Each tree has n nodes, so the total nodes visited is 2n. each 2 nodes will have an O(1) operation to compare their value.

* + 1. public static String tree2prefix(LinkedBinaryTree<String> tree)

Algorithm: recursive pre-order traversal.

Worst case run time: **O(n)=O(n)[traversing through the tree] + nO(1)[read value]**.

As this method is traversing through every single node of the tree to get all their elements.

* + 1. public static String tree2infix(LinkedBinaryTree<String> tree)

Algorithm: recursive pre-order traversal.

Worst case run time: **O(n)=O(n)[traversing through the tree] + nO(1)[read value]**.

This method is similar to tree2prefix, they are all traversing through every single node of the tree.

* + 1. Public static LinkedBinaryTree<String> simplify(LinkedBinaryTree<String> tree)

Algorithm: recursive pre-order traversal + stack push&pop + stringreplacement

Worst case run time: **O(n)=O(n)[call tree2prefix]+nO(1)[stack load element] +nO(1)[change element] + O(n)[call prefix2tree].**

This method calls tree2prefix first to convert the tree structure into string, then split the string into tokens, and load them into stack, perform simplify operation on the stacked tokens, then call prefix2tree method at last to convert the simplified prefix string into tree.

* + 1. public static LinkedBinaryTree<String> simplifyFancy(LinkedBinaryTree<String> tree)

Algorithm: recursive pre-order traversal + stack push&pop + string replacement

Worst case run time: **O(n)=O(n)[call tree2prefix]+nO(1)[stack load element] +nO(1)[change element] + O(n)[call prefix2tree].**

Similar to simplify, this method calls tree2prefix first to convert the tree structure into string, then split the string into tokens, and load them into stack, perform simplify fancy operation on the stacked tokens, then call prefix2tree method at last to convert the simplified prefix string into tree.

* + 1. public static LinkedBinaryTree<String> substitute(LinkedBinaryTree<String> tree, String variable, int value)

Algorithm: recursive pre-order traversal + string replacement

Worst case run time: **O(n)=O(n)[call tree2prefix]+O(1)[change value] + O(n)[call prefix2tree].**

This method calls tree2prefix first to convert the tree structure into string, and replace the corresponding character that represents variable into the value in parameter. At last, call prefix2tree method to convert the replaced prefix string into tree.

* + 1. public static LinkedBinaryTree<String> substitute(LinkedBinaryTree<String> tree, HashMap<String, Integer> map)

Algorithm: recursive pre-order traversal + string replacement

Worst case run time: **O(n)=O(n)[call tree2prefix]+nO(1)[change value] + O(n)[call prefix2tree].**

This method calls tree2prefix first to convert the tree structure into string, and call a ‘for each loop’ that will replace every variable in the map into its corresponding value. At last, call prefix2tree method to convert the replaced prefix string into tree.

* + 1. public static boolean isArithmeticExpression(LinkedBinaryTree<String> tree)

Algorithm: recursive pre-order traversal

Worst case run time: **O(n)=O(n)[traversed all nodes of the tree].**

This method calls tree2prefix at first as all ways that find if the tree is /isn’t an arithmetic expression is specified at that method. Then return corresponding Boolean value based on the result of that method.