Methods and its time Complexity

1) **tree2prefix**

This method use pre-order traversal of the given tree and gets all the values and store those values in the queue. It uses tree2prefixHelper to do this. And using the dequeue method of the queue data structure it store the values in the prefix string. Storing the value in queue will take the space O(n). Returns the prefix String.

Checking if tree is valid using isArithmeticExpression is O(n) which is described below.
All if condition will cost us O(1)
Using tree2prefixHelper method will cost time of O(n) described below which returns all the values in queue.
Going through all the values of queue and adding its value to string will be O(n).
Total time complexity = sum of all those method = O(n)

2) **tree2prefixHelper**

This is the helper method, which does pre-order traversal and store the value it visited in the queue and that queue. Returns the prefix String.

Let T(n) be the total time for this method then,
$T(n) = 2*T(n/2) + O(1)$
T(n/2) for visiting all the left sub-tree and T(n/2) for right sub-tree and O(1) for adding the elements in the queue. This recurrence relation will simplify to O(n).
All if condition will cost us O(1)
Total time complexity = O(n)

3) **tree2infix**

This method use pre-order traversal of the given tree and gets all the values and store those values in the queue. It uses tree2prefixHelper to do this. And using the dequeue method of the queue data structure it store the values in Stack. This stack is converted into infix string using tree2prefixHelper method. Storing the value in queue and Stack will take the space O(n) each. Returns the infix string.

Checking if tree is valid using isArithmeticExpression is O(n) which is described below.
All if condition will cost us O(1)
Using tree2prefixHelper method to get the tree values in the queue will cost us O(n).
Transferring that queue values to the stack will cost us O(n).
Using that stack to convert to infix using tree2infixHelper will cost us O(n) as described below.
Total time complexity = O(n)

4) **tree2infixHelper**

This function takes stack with all the values of trees as its parameter and return infix String value from it. Returns the infix string.

Going through all the values of the stack will cost us O(n).
All the pop and push function will cost us O(1).
Total time complexity = O(n).

5) **stringCheck**

This function checks if given string contains characters like "+", "-", "*" or can be parse into integer value without an error. Return false if those condition satisfies. It also return false if the given string is null. Return true otherwise. There are two if condition and one try and catch block in this method without any loop.
Total time complexity = O(1)

6) **intParseCheck**

This function check if the given string can be parsed into integer without any error. Return true if can be parsed without an error, false otherwise. There is only one try and catch block in this method without any loop.
Total time complexity = O(1)

7) **simplify**

This method simplifies the given binary tree. It doesn't create a new tree but modifies the tree that is given to us and doesn't use any extra data structure. This method use post-order traversal for the tree and simplifies from the leaf node of the trees and goes up. Uses simplifyHelper method to do the post-order traversal and simplification. Returns the modified tree.

Checking if tree is valid using isArithmeticExpression is O(n) which is described below.
Using simplifyHelper method will cost O(n) as described below.

All if condition will cost us O(1)
Total time complexity = O(n)

8) **simplifyHelper**

This method use post-order traversal on the tree and simplifies the tree from leaf node to up. It modifies the tree itself. Returns the modified tree.

Let T(n) be the total time for this method then,
T(n) = 2*T(n/2) + O(1)
T(n/2) for visiting all the left sub-tree and T(n/2) for right sub-tree and O(1) to update the tree (i.e. tree.set() and tree.remove()) if left and child of the operation can be parsed into integer which is equal to O(n).
All if condition will cost us O(1)
Total time complexity = O(n)

9) **simplifyFancy**

This method is more advance version of simplify. This can simplify even if String can't be parsed into integers, which follows some set of rules. This method uses the help of simplifyFancyHelper that uses post- traversal for the tree and simplifies from bottom up. It doesn't create a new tree but modifies the tree that is given to us but in some of the special cases it might use new binary to copy either left/right subtree of given tree which space will cost us O(n)

Checking if tree is valid using isArithmeticExpression is O(n) which is described below.
All if condition will cost us O(1)
simplifyFancyHelper will cost us O(n) which is described below
simplifyFancyCopySubTree will cost us O(n). We only need to use this to further simplify the tree.
Total time complexity = O(n)

10) **simplifyFancyHelper**

This method does the post- traversal of the given tree and simplifies from bottom up. Firstly, this method tries to simplify the tree as numeric simplification (like 2+3 =5). If it can't simplify with during that process it will further tries to simplify the tree which follows the set of rules given to us. Returns the modified tree.

Let T(n) be the total time for this method then,
T(n) = 2*T(n/2) + O(1)
T(n/2) for visiting all the left sub-tree and T(n/2) for right sub-tree and O(1) to update the tree (i.e. tree.set() and tree.remove())which is equal to O(n).
All if condition will cost us O(1)
Total time complexity = O(n)

12) **simplifyFancyCopySubTree**

This method copy either left sub-tree or right sub-tree of the given tree. Only used if for the tree whose root is "*" and either left child or right child of that root is 1. As we know that 1 time anything is just anything. This method. This method pre- traversal tree and copies the tree.

Let T(n) be the total time for this method then,
T(n) = 2*T(n/2) + O(1)
T(n/2) for visiting all the left sub-tree and T(n/2) for right sub-tree and O(1) to add elements in new tree (i.e. tree.addLeft() and tree.addRight()) which is equal to O(n).
All if condition will cost us O(1)
Total time complexity = O(n)

13) **substitute** (normal substitute with string and value)

This method substitutes the tree's operand values to new values, which are given to us. It changes all the instances of that operand value. Doesn't use any extra data structure and modifies the given tree. Uses the substituteHelper method to substitute the values. Returns the modified tree.

Checking if tree is valid using isArithmeticExpression is O(n) which is described below.
All if condition will cost us O(1)
substituteHelper will cost O(n) as described below
Total time complexity = O(n)

14) **substituteHelper** (normal substitute with string and value)
This method does the post- traversal of the tree and check if the tree check if the value matches to the operand value of tree and changes to new one if matches. Returns the modified tree.

Let T(n) be the total time for this method then,
T(n) = 2*T(n/2) + O(1)
T(n/2) for visiting all the left sub-tree and T(n/2) for right sub-tree and O(1) to set the value to new one(using build in tree.set() function) which is equal to O(n).
All if condition will cost us O(1)
Total time complexity = O(n)

15) **substitute** (with HashMap key value)
This method check if the HashMap key matches the operands in tree and changes it to new value as in value the key is associated with. Uses Set to store all the operands of tree which will take space of O(n). At first it check if map tries to store a null values in tree and throw exception if it does. Then uses substituteHelper method to substitute the values. Returns the modified tree.

Getting the prefix string for given tree will cost O(n)
Checking if HashMap is trying to substitute will take O(n). (HashMap containsKey() is a amortized O(1) time)
All if condition will cost us O(1) (HashMap containsKey() is a amortized O(1) time)
substituteHelperFunctio will cost us O(n) as described below
Total time complexity = O(n)

16) substituteHelper (with HashMap key value)
This method does the post- traversal of the given tree and check for the operand values matches the key and change is values to the new value that is corresponding to the hashmap's value. Returns the modified tree.

Let T(n) be the total time for this method then,
T(n) = 2*T(n/2) + O(1)
T(n/2) for visiting all the left sub-tree and T(n/2) for right sub-tree and O(1) to set the value to new one (using build in tree.set() function) which is equal to O(n).
All if condition will cost us O(1) (HashMap containsKey() is a amortized O(1) time)
Total time complexity = O(n)

17) **isArithmeticExpression**
This method checks if the given tree is a valid arithmetic expressed binary tree. Uses the isArithmeticExpressionHelper method to check if tree are valid. This helper method uses pre- traversal of the tree. Return true if the tree is valid arithmetic expression binary tree, false otherwise.

isArithmeticExpressionHelper will cost us O(n) as described below
All if condition will cost us O(1)
Total time complexity = O(n)

18) **isArithmeticExpressionHelper**
This method does the pre-traversal of the tree and checks if the operation has left and right child not null for tree to be valid unless tree has only a not null root which is a valid tree. Return true if the tree is valid arithmetic expression binary tree, false otherwise.

Let T(n) be the total time for this method then,
T(n) = 2*T(n/2) + O(1)
T(n/2) for visiting all the left sub-tree and T(n/2) for right sub-tree and O(1) to check if the left child and right child are nulls, which is equal to O(n).
All other if condition will cost us O(1)
Total time complexity = O(n)

Test Cases

I have added my own test cases in the **TestAssignment.java file.** The test cases that I have added that try to cover all the assignment specification for that given method. They also checks if the given methods throws the exception or not. In all of the test cases the test start by testing the examples that are shown in the assignment specification and then it check the edge cases for that method and finally there are check for all the exception. The test cases use the equal function to check if two given trees are equal or not. I have added necessary comments on the test cases to figure out what the test cases are checking.

1) **testTree2PrefixCheck**

This method uses prefix2Tree to make a valid tree and convert that tree to prefix using method tree2prefix and assertEquals with the string that we used to make tree.

```
LinkedBinaryTree<String> tree =  Assignment.prefix2tree("+ 5 10");
String prefix = Assignment.tree2prefix(tree);
assertEquals("+ 5 10" , prefix);
```

Does the illegal argument check by making a not valid tree and tries to convert to prefix

```
LinkedBinaryTree<String> tree3 = new LinkedBinaryTree<>();
tree3.addRoot("+");
thrown.expect(IllegalArgumentException.class);
prefix = Assignment.tree2prefix(tree3);
tree3.addLeft(tree3.root(), "3");
```

2) **testTree2infix**

This method uses prefix2Tree to make a valid tree and convert that tree to prefix using method tree2infix and assertEquals with the string that we used to make tree.

```
LinkedBinaryTree<String> tree = Assignment.prefix2tree("+ 5 10");
String infix = Assignment.tree2infix(tree);
assertNotEquals("(5+5)" , infix);
assertEquals("(5+10)" , infix);
```

Does the illegal argument check by making a not valid tree and tries to convert to infix.

```
LinkedBinaryTree<String> tree3 = new LinkedBinaryTree<>();
tree3.addRoot("+");
thrown.expect(IllegalArgumentException.class);
infix = Assignment.tree2infix(tree3);
```

3) **testSimplify**

This method uses prefix2Tree to make a valid tree and also use that method to make expected tree after simplification using that function. Uses equal function form Assignment class to check it those trees are equal.

```
tree = Assignment.prefix2tree("- 0 0");
tree = Assignment.simplify(tree);
expected = Assignment.prefix2tree("0");
assertTrue(Assignment.equals(tree, expected));
```

Does the illegal argument check by trying to pass the invalid arithmetic expression tree in simplify function.

```
//Illegal Argument check
LinkedBinaryTree<String> tree3 = new LinkedBinaryTree<>();
tree3.addRoot("+");
thrown.expect(IllegalArgumentException.class);
tree3 =Assignment.simplify(tree3);
```

4) **testSimplifyFancy**

This method uses prefix2Tree to make a valid tree and also use that method to make expected tree after simplification using that function. Uses equal function form Assignment class to check it those trees are equal.

```
LinkedBinaryTree<String> tree = Assignment.prefix2tree("- * 1 c + c 0");
LinkedBinaryTree<String> expected = Assignment.prefix2tree("0");
LinkedBinaryTree<String> not_expected = Assignment.prefix2tree("c");
tree = Assignment.simplifyFancy(tree);
assertTrue(Assignment.equals(tree, expected));
assertFalse(Assignment.equals(tree, not_expected));
```

Some edge cases check

```
// Checking for some special edge cases
tree = Assignment.prefix2tree("* 0 + 9 + d c");
expected = Assignment.prefix2tree("0");
tree = Assignment.simplifyFancy(tree);
assertTrue(Assignment.equals(tree, expected));

tree = Assignment.prefix2tree("* + a b 1");
expected = Assignment.prefix2tree("+ a b");
tree = Assignment.simplifyFancy(tree);
assertTrue(Assignment.equals(tree, expected));

tree = Assignment.prefix2tree("* 1 + - 4 c 3");
expected = Assignment.prefix2tree("+ - 4 c 3");
tree = Assignment.simplifyFancy(tree);
assertTrue(Assignment.equals(tree, expected));
```

Does the illegal argument check by trying to pass the invalid arithmetic expression tree in simplifyFancy function.

```
LinkedBinaryTree<String> tree3 = new LinkedBinaryTree<>();
tree3.addRoot("+");
thrown.expect(IllegalArgumentException.class);
tree3 =Assignment.simplifyFancy(tree3);
```

5) **testSubstitute_Variable_and_Value:** This method uses prefix2Tree to make a valid tree and also use that method to make expected tree after simplification using that function. Uses equal function form Assignment class to check it those trees are equal. Check if the substitute methods actually modifies or not for the given tree.

```
variable = "G";
value = -7;
tree = Assignment.substitute(tree, variable, value);
expected =Assignment.prefix2tree("+ - * + 1 B + C D - E F -7");
not_expected =Assignment.prefix2tree("+ - * + A B + C D - E F G");
assertTrue(Assignment.equals(tree, expected));
assertFalse(Assignment.equals(tree, not_expected));
```

Does the illegal argument check by trying to pass the string with null values and invalid tree.

```
// Illegal Argument check
// Passing the null tree
LinkedBinaryTree<String> tree3 = new LinkedBinaryTree<>();
thrown.expect(IllegalArgumentException.class);
tree3 =Assignment.substitute(tree3,variable,value);

// Passing the null variable
tree = Assignment.prefix2tree("+ - * + A B + C D - E F G");
variable = null;
value = 0;
thrown.expect(IllegalArgumentException.class);
tree3 =Assignment.substitute(tree3,variable,value);
```

6) **testSubstitute_HashmapValue:** This method uses prefix2Tree to make a valid tree and also use that method to make expected tree after simplification using that function. Uses equal function form Assignment class to check it those trees are equal. Passes the hashmap with the key value pair to replace those values in the tree if key matches.

```
// Hashmap multiple key to replace tree's value check
LinkedBinaryTree<String> tree = Assignment.prefix2tree("+ - * + A B + C D - E F G");
HashMap<String,Integer> map = new HashMap<>();
map.put("C", new Integer(3));
map.put("B", new Integer(2));
map.put("D", new Integer(4));
tree = Assignment.substitute(tree, map);
LinkedBinaryTree<String> expected = Assignment.prefix2tree("+ - * + A 2 + 3 4 - E F G");
LinkedBinaryTree<String> not_expected = Assignment.prefix2tree("+ - * + A B + C D - E F G");
assertTrue(Assignment.equals(tree, expected));
assertFalse(Assignment.equals(tree, not_expected));
```

Does the illegal argument check by trying to pass the value for the key that matches the tree variable value and passing invalid tree.

```
//Illegal Argument check by passing invalid tree
LinkedBinaryTree<String> tree3 = new LinkedBinaryTree<>();
thrown.expect(IllegalArgumentException.class);
tree3 =Assignment.substitute(tree3,map);
tree3.addRoot("+");
thrown.expect(IllegalArgumentException.class);
tree3 =Assignment.substitute(tree3,map);

//HashMap null value check which throws IllegalArgumentException
tree3 = Assignment.prefix2tree("+ A B");
map.put("A", null);
map.put("B" , new Integer(9));
thrown.expect(IllegalArgumentException.class);
tree3 =Assignment.substitute(tree3,map);
```

7) **testArithmeticExpression:** This method uses prefix2Tree to make a valid tree and check if isArithmeticExpression from Assignment class if it returns true. Also make invalid tree and check if returns false.

```
// Valid Arithmetic Expression check
LinkedBinaryTree<String> tree = Assignment.prefix2tree("- + * B 2 * * 4 A C * 1 2");
assertTrue(Assignment.isArithmeticExpression(tree));

tree = Assignment.prefix2tree("0");
assertTrue(Assignment.isArithmeticExpression(tree));
```

```
// Invalid Arithmetic Expression check
LinkedBinaryTree<String> tree2 = new LinkedBinaryTree<>();
assertFalse(Assignment.isArithmeticExpression(tree2));

tree2.addRoot("+");
tree2.addLeft(tree2.root(), "0");
assertFalse(Assignment.isArithmeticExpression(tree2));
```