1. **Introduction**

Remember that the tree, as seen in lectures, is an ADT consisting of a hierarchical structure of positions. Each position holds reference to an element of the tree, reference to the list of children of that position in the tree, as well as reference to its parent position in the tree. Each position corresponds to a vertex of the tree as per the definition of the tree data structure. In the following discussion, we may use terms *position*, *vertex*, or *node* to refer to the same concept.

You have received a Java project which contains partial implementations of the Tree<E> ADT and of the BinaryTree<E> ADT as specified in the textbook. Both are linked structures. For the general tree, class LinkedTree<E>, which is also a subclass of class AbstractTree<E>, corresponds to an implementation based on a linked structure. Notice that some other methods (not part of Tree<E> interface) are implemented here. These are as follows:

In AbstractTree class:

* void display() - this method displays the tree content (as we will see in a later example)

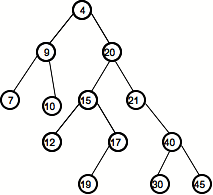
In LinkedTree class:

* Position<E> addChild(Position<E> p, E e) throws IllegalArgumentException - this method will add a new child at the end of the current list of children that position p has. If p is not valid, an error occurs. If successful, the element for that new child shall be e, and at the end returns reference to the position of that new child.
* Position<E> addRoot(E e) throws IllegalStateException - this method will add to the tree a new node holding element e, and that node will become its root. The tree must be empty; otherwise, an error occurs. If successful, it returns reference to that new position. See the specification of the same method for the binary tree as described in lecture or in textbook.
* E remove(Position<E> p) throws IllegalArgumentException - if valid, removes position p from the tree and returns its current element.. If the p is the root, then the operation is valid only if p has no more than one child, and in that case, that child becomes the new root of the tree; otherwise, if it is the root and it has more than one child, an error occurs. If p is not the root, then all its children are given to its parent position.

Review the concepts studied in lectures about trees. The partial implementation that you are receiving follows the approach used in the textbook, except for methods that are added (as those described above) for the purpose of learning more about these important data structures.

1. **Exercises**

Import the partial project received as an Eclipse project in your system. The next tree is used in some of the exercises:



1. Complete the method buildExampleTreeAsLinkedTree inside class Utils (see package labUtils). That method should build the above tree as a LinkedTree object. Then execute class program ExampleBuilder1. The tree should be created and also displayed. The output must be as shown in Figure 1.

|  |
| --- |
| \_\_(4)  |  |\_\_(9)  | |  | |\_\_(7)  | |  | |\_\_(10)  |  |\_\_(20)  |  |\_\_(15)  | |  | |\_\_(12)  | |  | |\_\_(17)  | |  | |\_\_(19)  |  |\_\_(21)  |  |\_\_(40)  |  |\_\_(30)  |  |\_\_(45)  **Figure 1** |

1. Now complete the method buildExampleTreeAsLinkedBinaryTree also inside class Utils. That method should build the above tree as a LinkedBinaryTree object. Then execute class program ExampleBuilder2. The output should be the same as in the previous exercise.
2. In the above sketch for the binary tree case, it would be nice to identify each node as either a left child or a right child. We want to include R or L before the parentheses enclosing the position’s value, depending on the type of child that the node is (if not the root). Here you are asked to do the minimum adjustments needed so that the display for a tree of type LinkedBinaryTree also includes those identification labels. For the sake of completeness, include identification for the root node -- the word ROOT before the parentheses enclosing the root’s value. HINT: see method recDisplay in class AbstractTree, which is a protected method. Proceed as follows:
   1. Override method recDisplay at the level of the AbstractBinaryTree class. Uncomment the partial implementation there, and add the necessary code. Notice that this only applies to binary trees and not to any tree in general. No changes need to be done at the level of AbstractTree class. HINT: see method recDisplay in class AbstractTree and adjust as necessary. Why is this enough?
   2. Once you have completed your changes, run again the program in class ExampleTreeBuilder2. The new output will be as shown in Figure 2.
   3. Run the program in ExampleTreeBuilder1 once more. You should see the output as in Figure 1. Why?
   4. Now run the program TreeTester3. You have two versions of the same binary tree. Why results look different?

|  |
| --- |
| \_\_ROOT(4)  |  |\_\_L(9)  | |  | |\_\_L(7)  | |  | |\_\_R(10)  |  |\_\_R(20)  |  |\_\_L(15)  | |  | |\_\_L(12)  | |  | |\_\_R(17)  | |  | |\_\_L(19)  |  |\_\_R(21)  |  |\_\_R(40)  |  |\_\_L(30)  |  |\_\_R(45)  **Figure 2** |

1. If you examine the code inside class LinkedTree, in particular, inside method remove, you will see that part of it is marked as *missing*. In this exercise, you are asked to discover what is missing and correct it. You need to carefully study the other implemented parts of that method. Before doing it, run class program TreeTester2. It will end with an error from an exception, as shown in Figure 3. That error happens because of what is missing, and that you will need to discover.

|  |
| --- |
| Tree t after removing Maria    \_\_(ROOT)  |  |\_\_(Rosa)  |  |\_\_(null)  Exception in thread "main" java.lang.IllegalArgumentException: Target position is not part of a tree.  at treeClasses.LinkedTree.validate(LinkedTree.java:29)  at treeClasses.LinkedTree.numChildren(LinkedTree.java:62)  at treeClasses.AbstractTree.recDisplay(AbstractTree.java:101)  at treeClasses.AbstractTree.recDisplay(AbstractTree.java:104)  at treeClasses.AbstractTree.display(AbstractTree.java:87)  at testerClasses.TreeTester2.displayTree(TreeTester2.java:50)  at testerClasses.TreeTester2.main(TreeTester2.java:43)  **Figure 3** |

One more comment to add here. The error above would be very difficult to detect without the validation of positions required in several methods.

One more thing here. Notice that class LinkedTree now implements Cloneable. Study the method clone(), whose purpose is to create a clone of a tree. It is important that you understand how it works.

1. Run class program TreeTester4. Study the result. Notice that this is displaying elements as visited by a preorder traversal. To see how it is done, study the two methods: positions() and iterator(), as implemented in class AbstractTree. In particular, study the internal (protected) method fillIterable. Now go inside class AbstractBinaryTree and remove the /\*\* and the \*\*/ before and after method fillIterable in that class (or just add a / character at the end of the first and another / at the beginning of the last; these will convert those two lines to /\*\*/ in both cases...). Don’t do any other change. Run this tester again. What type of traversal is being used to print the elements now? Think why. The power in inheritance and polymorphism in action.
2. Study class LinkedBinaryTree2. Notice that this is a type of binary tree whose implementation is based on an internal object of type LinkedBinaryTree (instance variable t) and whose elements are *comparable*. This type of tree now has one new method that is not part of the BinaryTree ADT specification. That method is insert; it inserts a new element into the tree. Study the implementation of that method and visualize how the structure of the tree is being shaped as new elements are added. Elements are inserted into the internal binary tree (t) based on comparisons. Run the class program TreeTester6 (you should have completed *Exercise 5* before this in order to see the results in the right order). The results are shown in increasing order! See the tree structure. Change the order in which the insert operations are being done, and run again. Any difference in the structure of the tree? Any difference on how the elements are listed?

By the way, this is one very important application of binary trees, and we shall study more about this in coming lectures.

1. Write a test program to test the clone() of LinkedTree. Test the case of cloning an empty tree. An error should occur. Discover the error and correct it. Make sure the test is passed and that it also works for nonempty trees. You can use one of the testers already included.
2. Write the necessary code to make LinkedBinaryTree class Cloneable. Write your own testers for this.
3. FIN …