University of Puerto Rico – Mayagüez
Department of Computer Science and Engineering

CIIC 4020 / ICOM 4035 – Data Structures
Prof. Juan O. López Gerena
Spring 2019-2020
Laboratory #8 – Trees and Binary Trees

Introduction

Remember that the tree, as seen in lectures, is an ADT consisting of a hierarchical structure of nodes. Each node holds reference to an element of the tree, reference to the list of children of that node in the tree, as sometimes also a reference to its parent node in the tree. For simplicity, we'll be using simple elements, instead of a key/value entry.

Unlike lists, stacks or queues, there's no explicit predetermined order in trees, which is why we have different traversals. Hence, it seems natural to work directly with the nodes of the tree, so that the user may traverse the tree in whatever order he/she wants, gaining access to the data they have stored in the tree structure. However, this poses a security problem, since by having access to the node, the user could potentially change the left/right child pointers of the node, thus destroying the tree structure. Enter the `Position` ADT, an *abstraction* of the `Node` class with only one method: **`getElement()`**. Positions allow us to internally continue to use nodes the way we are familiar with, while limiting the user's access to only obtaining the value stored. Each position corresponds to a node 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; 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:

`AbstractTree<E>`:

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

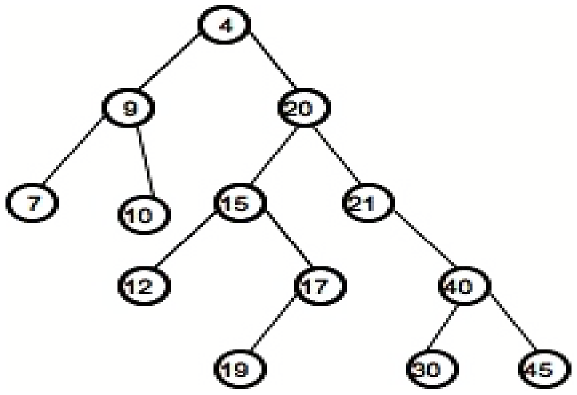
`LinkedTree<E>`:

- **`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 the method returns a reference to the position of that new child. *Recall that, unlike binary trees, nodes in general trees can have multiple children.*

- **`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 a reference to that new position. See the specification of the same method for the binary tree.
- **`E remove(Position<E> p) throws IllegalArgumentException`** - if valid, removes position p from the tree and returns its current element. If 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 discussed, except for methods that are added (as those described above) for the purpose of learning more about these important data structures.

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



Exercise 1: `buildExampleTreeAsLinkedTree`

Complete the method **`buildExampleTreeAsLinkedTree`** inside class `Utils` (see package `labUtils`). That method should build the above tree as a `LinkedTree` object. Test your code by executing the tester program `ExampleBuilder1`; the tree should be created and displayed as shown in Figure 1.

Remember that when you invoke the **`addChild`** method, it returns a reference to the new position created. You will need to store that reference, at least temporarily, if that new position will also have its own children. Once you have added all of the children for a particular position, you may discard that reference. See the `TreeTester1` class for an example.

```
__(4)
  |
  |__(9)
  |  |
  |  |__(7)
  |  |
  |  |__(10)
  |
  |__(20)
     |
     |__(15)
     |  |
     |  |__(12)
     |  |
     |  |__(17)
     |     |
     |     |__(19)
     |
     |__(21)
        |
        |__(40)
           |
           |__(30)
           |
           |__(45)
```

Figure 1

Exercise 2: `buildExampleTreeAsLinkedBinaryTree`

Now complete the method **`buildExampleTreeAsLinkedBinaryTree`** also inside class `Utils`. That method should build the above tree as a `LinkedBinaryTree` object. Test your code by executing the tester program `ExampleBuilder2`; the output should be the same as in the previous exercise.

Exercise 3: Identify your children

In the above sketch for the binary tree, 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. 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, so no changes need to be made to the `AbstractTree` class.
   HINT: Copy the code from method **`recDisplay`** in class `AbstractTree` and adjust as necessary.
   <u>Think</u>: 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. The output should be as in Figure 1.
   <u>Think</u>: Why didn't the output change?
4. Now run the program `TreeTester3`, which will print two versions of the same binary tree.
   <u>Think</u>: Why do the 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

Exercise 4: Finish the `remove` method

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 so, 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 need to discover (and fix).

```
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. Also, you should run *additional* tests to ensure that the root can be successfully removed when it is valid to do so…

Once you fix the error, you'll notice from the output that a clone of the tree was created. That was possible because the 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.

Exercise 5: Traversals

Run class program `TreeTester4` and study the result. Notice that this is displaying elements as visited by a pre-order traversal. To see how it is done, study the following 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 make any other change and run the tester again. What type of traversal is being used to print the elements now? Think why. The power of inheritance and polymorphism in action.

Exercise 6: A glimpse of things to come

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 `TreeTester5` (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.

Exercise 7: Fix the `clone` method

Write a test program to test the **`clone()`** method of `LinkedTree`. Test the case of cloning an empty tree; an error should occur. Discover the error and correct it. Make sure the test now passes and that it also works for non-empty trees. You can copy and/or modify one of the testers already included.

This highlights the importance of testing thoroughly, particularly the edge cases.

Exercise 8: Make `LinkedBinaryTree` Cloneable

Write the necessary code to make `LinkedBinaryTree` class Cloneable. Create your own testers for this.

End