ASSIGNMENT 4

This assignment is due on April 12th, 2021 by 11:55pm.
See the bottom of this document for submission details.

Learning Outcomes

To gain experience with

- Solving problems with the tree data type
- Recursive and iterative applications
- The design of algorithms in pseudocode and their implementations in Java

Do not import any libraries unless specifically asked to do so. Any additional imports will be heavily penalized (-10 or more).

Introduction

In lecture, we learn about trees and primarily focus on binary trees – those in which each node can have at most 2 children. In this assignment, we are exploring the **n**-ary tree or general trees in which nodes can have multiple children rather than a strict limit of 2.

Since binary trees are limited to at most 2 children from each node, the implementation is very simple by having 2 node instance variables: left and right. In the general tree, we will use an array to contain the children from each node since we don't know how many children will be coming from each node so it doesn't make sense to have individual node variables the way we did in the binary tree.

Once the "LinkedNaryTree" (general tree) class is completed, the second part of the assignment is to create a class called ShapeTree. The ShapeTree is a tree that will store objects of the custom class Shape (provided to you) which will represent geometric shapes that have a colour and a number of lines based on its geometric properties (i.e. a triangle has 3 sides, a square has 4 sides, etc.). The interesting and challenging feature of the ShapeTree is that when a new node is added to the tree, there are specific rules that must be followed:

1. A node can have up to x children, where x is the number of lines in that node's contained Shape, i.e. a node containing a circle can only have 1 child, a node containing a triangle can have up to 3 children, etc. (Note: for the purposes of this assignment, we will consider a Star to have 5 lines)

2. A node cannot have a child whose Shape is the same colour as the node's Shape (i.e. a node containing a blue shape cannot have a child that containing a blue shape).
3. There cannot be sibling nodes (nodes of the same parent) that contain Shapes of the same colour, i.e. a node containing a red shape cannot have a sibling node that also contains a red node, etc.)
4. Nodes should be added as high (close to the root) as possible without breaking the above rules. After that, they should be added as far to the left as possible without breaking the above rules.

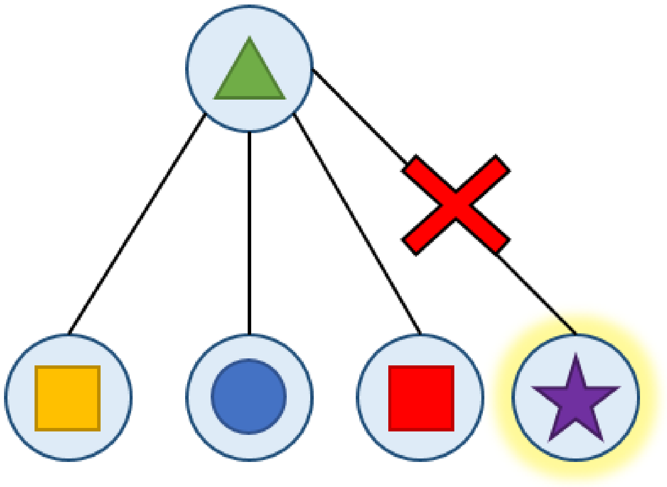

Figure 1. An example of Rule #1 being broken. The new node (with the purple star) cannot be added here because the root node (with the green triangle) already has 3 children which is the maximum since its shape is a triangle.

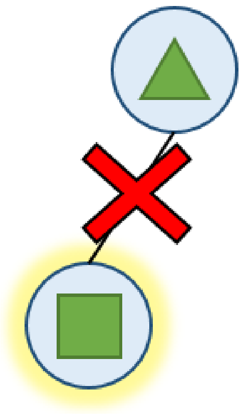

Figure 2. An example of Rule #2 being broken. The new node (with the green square) cannot be added here because the root node contains a shape that is green.

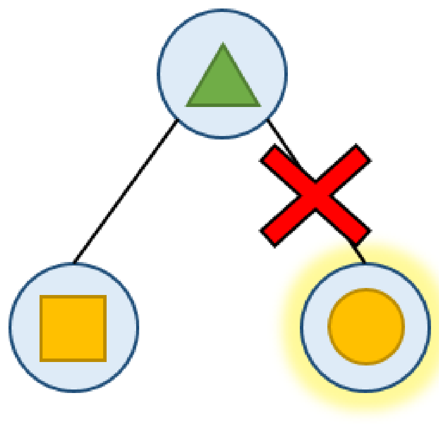

Figure 3. An example of Rule #3 being broken. The new node (with the yellow circle) cannot be added here because the root node already has a child node containing a yellow shape (square).

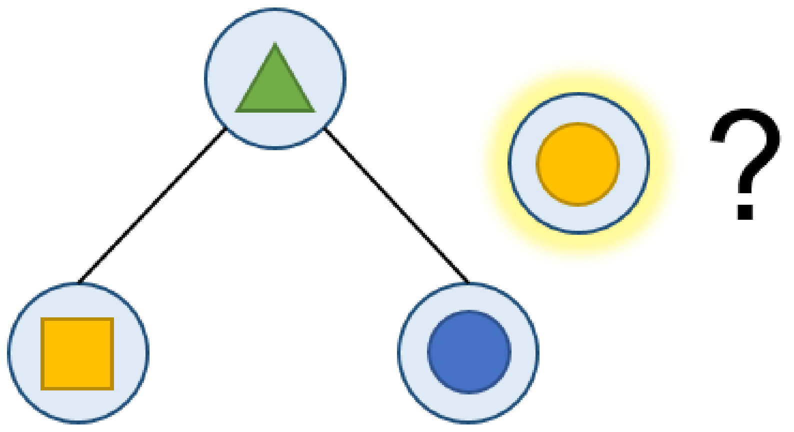

Figure 4. An example of an existing ShapeTree and a new node (with the yellow circle) that is about to be added to the tree. Examine the rules above to determine how and where this new node will be added.

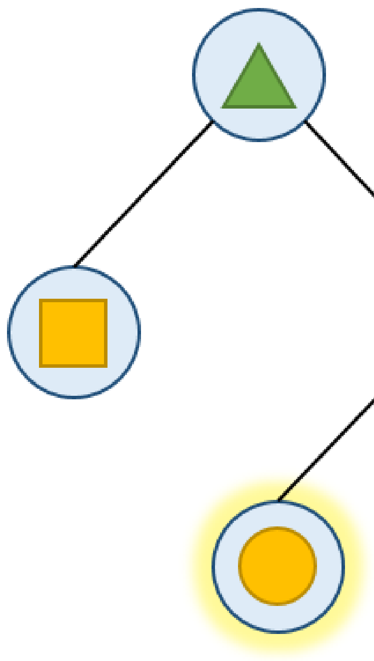

Figure 5. An example of an existing ShapeTree with the new node (with the yellow circle) added to it. This new node could not be added directly under the root node because of Rule #3. The next place it tried was under the first child of the root node (with the yellow square) but the new node could not be added there because of Rule #2. Next, it tried to add under the second child of the root node (with the blue circle) and it did not break any rules so that is where the node was added.

Provided files

The following is a list of files provided to you for this assignment. You do not need alter these files in any way and should assume we will mark them with unmodified ones. Studying these will help improve your understanding of Java.

- Classes provided:
    - *ArrayIterator.java, ArrayList.java, ArrayStack.java, ArrayUnorderedList.java, ElementNotFoundException.java, EmptyCollectionException.java, ListADT.java, Shape.java, StackADT.java, TestNaryTree.java, TestShapeTree.java,* and *UnorderedListADT.java*

Classes to implement

A description of the classes that you are required to implement for full marks in this assignment are given below.

In all these classes, you may implement more private (helper) methods as desired, but you may **not** implement more public methods. You may **not** add additional static methods, or instance variables, or static variables. Penalties will be applied if you implement these.

NaryTreeNode.java

This class represents a single node in an n-ary (general) tree. The class must work with the generic type (T).

This class must have the following *private* variables:

- data (T)
- numChildren (int)
- children (NaryTreeNode<T>[ ])

The class must have the following *public* methods:

- NaryTreeNode (constructor) – takes in a parameter of type T and sets that into the data variable. The children variable should be initialized to null here.
- addChild – takes in a parameter of type NaryTreeNode<T> representing a child node to add to this node. If children is currently null, then initialize it with 3 slots. If the children array is full, then call the expandCapacity() method. Add the parameter child node to the next available slot in the children array and increment numChildren
- expandCapacity – expands the capacity of the children array by adding 3 more spaces and transfers the existing children nodes into the new, larger array
- getNumChildren – returns the numChildren value
- getChild – takes in an int index and returns the child stored in that index
- getData – returns the data item stored in this node
- toString() – returns a string that says "Node containing " followed by the data item stored in this node

LinkedNaryTree.java

You should import java.util.Iterator; in this class

This class represents the n-ary (general) tree itself, which is composed of the NaryTreeNode objects. This class must have the following *private* variable:

- root (NaryTreeNode<T>)

The class must have the following *public* methods:

- LinkedNaryTree (constructor) – no parameters. Set the root to null.
- LinkedNaryTree (constructor) – takes in an NaryTreeNode<T> parameter. Set the root to this input parameter.
- addNode – takes in two parameters both of type NaryTreeNode<T>. The first is the parent and the second is the child. Call the addChild() method on the parent node to add the child node to it.
- getRoot – return the root
- getRootElement – return the data element within the root
- isEmpty – return a boolean value indicating whether or not the tree is empty (hint: check if root has been initialized or not)
- size – take in an NaryTreeNode<T> parameter and calculate the size (number of elements) in the tree from that parameter node. Use recursion to accomplish this.
- preorder – takes in an NaryTreeNode<T> parameter and an ArrayUnorderedList<T> parameter. Perform a preorder traversal of the tree using recursion and adding each element to the list as you visit that node (hint: use the preorder method from a binary tree as a base but adapt it to work with any number of children nodes)
- iteratorPreorder – create and initialize an ArrayUnorderedList. Call the preorder method with the root node and the list you just created. Return the iterator of this list.
- toString – returns a string with each of the preorder elements (use the iterator from above to help with this) with a newline character after each element. If the tree is empty, simply return "Tree is empty." Instead.

ShapeTree.java

This class represents the tree that contains Shape objects in the nodes. It must follow the rules outlined and illustrated in the Introduction section of this document. The class must have the following *private* variable:

- tree (LinkedNaryTree<Shape>)

The class must have the following *public* methods:

- getTree – returns tree
- getRoot – returns the root of the tree
- addShapeNode – takes in a Shape parameter. In the method, create a new NaryTreeNode<Shape> containing the shape parameter. If the tree is empty, set the new node as the root. Otherwise, call the addShapeNodeHelper method (see below) to

determine if/where the new node can be added in the tree. If there is a suitable place for it, then add the new node as a child to that node.

- addShapeNodeHelper – takes in a Shape parameter. Perform a Stack-based preorder traversal on the tree, like the one shown in lectures on a binary tree, but it will have to be adapted to work with any number of children. The "visiting" of each node, v, in this method will require you to call the checkNode method (see below) to see if the shape parameter can be added on node v. If it does find such a node v on which the new shape can be added, then it should immediately return that node and end the function. If no nodes are found suitable, then it should return null after the traversal is complete.
- checkNode – takes in an NaryTreeNode<Shape> parameter and a Shape parameter. This method must check the first 3 rules of the ShapeTree as explained in the Introduction section of this document, and return a boolean value indicating whether or not the given shape parameter can be added on the given node parameter.
- toString – returns the same string as the variable tree's toString()

Marking notes

Marking categories

- Functional specifications
    - Does the program behave according to specifications?
    - Does it produce the correct output and pass all tests?
    - Are the class implemented properly?
    - Are you using appropriate data structures?
- Non-functional specifications
    - Are there comments throughout the code?
    - Are the variables and methods given appropriate, meaningful names?
    - Is the code clean and readable with proper indenting and white-space?
    - Is the code consistent regarding formatting and naming conventions?
- Penalties
    - Lateness: 10% per day
    - Submission error (i.e. missing files, too many files, ZIP, etc.): 5%
    - "package" line at the top of a file: 5%
    - Additional global scope variables, static methods or variables: 25%
    - Importing unnecessary or unasked for dependencies: 50%
    - Uploading unnecessary classes (only submit the 3 listed on the next page)

Remember you must do all the work on your own. Do not copy or even look at the work of another student. All submitted code will be run through cheating-detection software.

ASSIGNMENT 4

Submission (due April 12th, 2021 at 11:55pm ET)

Rules

- Please only submit the files specified below. Do not attach other files even if they were part of the assignment.
- Submit the assignment on time. Late submissions will receive a penalty of 10% per day.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope
- Submitting the files in an incorrect submission page or website will receive a penalty.
- Assignment files are NOT to be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.
- You may re-submit code if your previous submission was not complete or correct, however, re-submissions after the regular assignment deadline will receive a penalty.

Files to submit

- NaryTreeNode.java
- LinkedNaryTree.java
- ShapeTree.java

Grading Criteria

- Total Marks: [20]
- Functional Specifications:
  - [2] NaryTreeNode.java
  - [3] LinkedNaryTree.java
  - [3] ShapeTree.java
  - [10] Passing Tests
- Non-Functional Specifications:
  - [0.5] Meaningful variable names, private instance variables
  - [0.5] Code readability and indentation
  - [1] Code comments