INFO1105/1905 Data Structures

Assignment 1: Trees

This assignment can be completed individually or in pairs.

Submission Details

This assignment is due by **1pm on Monday 14th September**. **No late submissions accepted** (unless covered by an approved Special Consideration request).

If your assignment is incomplete, please submit your work before the deadline anyway - even if you do not expect it to pass any of the automatic tests. Please do this even if you intend to apply for Special Consideration. If you know in advance that you will not be able to complete the assignment on time, then ask your tutor for advice, and try to focus on completing a subset of the tasks.

You must submit:

- To PASTA:
  - Your source code (a zip file containing the "src" folder that contains your ".java" files, just like the weekly assessments.)
  - Note: you will need to add yourself to a group in PASTA for this assignment. There is a button for this below the "submit" button for the assignment. Groups can have at most two students (you can be in a group on your own if you wish to do the assignment individually.) You should do this before or during your next tutorial.
- To eLearning:
  - Your report, which should be a typed document (.pdf or .txt preferred)
  - A signed and completed cover sheet

Overview of assignment:

You are provided with an ADT and implementation of an arbitrary tree. For each part of this assignment you will need to implement some methods which act on Trees.

This assignment is worth 8% of your final grade.

Each of the four parts to this assignment will be marked automatically, on PASTA. The automatic tests make up 70% of the mark for this assignment. The remaining 30% of the marks will be allocated by hand, based on our assessment of the overall quality of your code and of your report.

Skeleton code is available for download on Piazza and eLearning. This code contains the required interfaces for each part of the assignment and a base tree implementation that your code will extend. Do not modify any of the files in the "interfaces" or "simpletree" packages. None of your added code should be in a package (i.e. in (default), just like all the weekly tasks.

Some basic unit tests to get your testing started will be provided at a later date. These will correspond to no more than half the available automatic marks. The remaining tests will be hidden, it is up to you to make sure that your tests are sufficiently broad!

Hand Marking [20%]

Part of your mark is based on the overall quality of your code. Your code will be assessed on the following criteria. This is not an exhaustive list, but a guideline to give an idea of the sort of qualities we will be looking for:

- Commenting
    - all non-trivial methods should have a description
    - each important block of code should have a short comment explaining it
    - do not excessively comment (it is not necessary to comment every line)
- Code quality
    - variable names should be informative
    - use of whitespace (blank lines breaking blocks of code into logical sections)
    - consistent indentation
    - try to avoid very long lines (>100 characters)
- Efficiency and Maintainability
    - for example, avoid looping over much more information than necessary
    - avoid excessive code repetition (use helper methods when appropriate)

Part 1: Tree traversals [15%]

Implement the following methods, which output some useful traversals of the trees.
In all cases, the children of a node should be visited in the same order in which they appear in the underlying data structure (do not consider the *value* contained in the node when deciding the order.)

public List<E> preOrder()
// Output the values of a pre-order traversal of the tree

public List<E> postOrder()
// Output the values of a post-order traversal of the tree

public List<E> inOrder()
// Output the values of a in-order traversal of the tree
// This operation should only be performed if the tree is a *proper binary tree*.
// If it is not, then throw an UnsupportedOperationException instead of returning a value
// Otherwise, perform the traversal with child 0 on the left, and child 1 on the right.

Part 2: Tree properties [25%]

Implement the following methods, which test or output certain properties that the tree might have.

public int height()
// calculate the height of the tree (the maximum depth of any position in the tree.)
// a tree with only one position has height 0.
// a tree where the root has children, but no grandchildren has height 1.
// a tree where the root has grandchildren, but no great-grandchildren has height 2.
// we will define an empty tree to have height -1.

public int height(int maxDepth)

// calculate the height of the tree, but do not descend deeper than ‘depth’ edges into the tree
// do not visit any nodes deeper than maxDepth while calculating this
// do not call your height() method
// (some trees are very, very, very big!)

public int numLeaves()
// calculate the number of leaves of the tree (i.e. positions with no children)

public int numLeaves(int depth)
// calculate the number of leaves of the tree at exactly *depth* depth.
// the root is at depth 0. The children of the root are at depth 1.

public int numPositions(int depth)
// calculate the number of positions at exactly *depth* depth.

public boolean isBinary()
// is the tree a binary tree?
// every position in a binary tree has no more than 2 children

public boolean isProperBinary()
// is the tree a *proper binary tree*?
// every position in a proper binary tree has either zero or two children

public boolean isCompleteBinary()
// is the tree a complete binary tree?
// 1) all the levels except the last must be full
// 2) all leaves in the last level are filled from left to right (no gaps)

public boolean isBalancedBinary()
// is the tree a balanced binary tree?
// a balanced tree is one where for every position in the tree, the
// subtrees rooted at each of the children of the that position have
// heights that differ by no more than one.
// NOTE: if a node has only one child, the other child is considered to be a subtree of height -1

public boolean isHeap(boolean min)
// is the tree a min-heap (if min is True), or is the tree a max-heap (if min is False)
// heaps are trees which are both complete and have the heap property:
// in a min-heap, the value of a node is less than or equal to the value of each of its children
// similarly, in a max-heap the value of a node is greater than or equal to the value of each child

public boolean isBinarySearchTree()
// is the tree a *binary search tree*?
// a binary search tree is a binary tree such that for any node with value *v*:
// - if there is a left child (child 0 is not null), it contains a value strictly less than *v*.
// - if there is a right child (child 1 is not null), it contains a value strictly greater than *v*.
// if there is only one child, you may assume that it is a **left** child.

Part 3 (INFO1105 only): Comparing trees [15%]

If you are enrolled in INFO1105, implement the following method, which allows us to compare trees:

public int compareTo(Tree<E> other)
// compare the tree with another tree
// check the structure and values of the trees:
// check the positions left-to-right, top to bottom (i.e. root, then depth 1, then depth 2, etc.)
// - If this tree has a position that the other tree does not, return 1.
// - If the other tree has a position that this one does not, return -1.
// - If the position is in both trees, then compare the values (if they are different, return the difference)
// If the two trees are identical, return 0

Part 3 (INFO1905 only): Self-balancing binary search tree[15%]

If you are enrolled in INFO1905, implement the following methods, which allow **balanced** insertion and deletion to a binary search tree. You may assume that the tree is a balanced binary search tree before either of these methods are called.

public boolean add(E value)
// if *value* is already in the balanced BST, do nothing and return false
// otherwise, add value to the balanced binary search tree (BST) and return true
// use the algorithm shown in the week 6 lecture - the BST must remain balanced

public boolean remove(E value)
// if *value* is in the balanced BST, remove it and return true
// otherwise, do nothing and return false
// implement the algorithm shown in the week 6 lecture to ensure that the BST remains balanced

Part 4: Arithmetic Expressions [15%]

Implement an Arithmetic Expression display and evaluator.
For all methods except *isArithmetic*, you may assume that the input is valid arithmetic.

public boolean isArithmetic()
// is this tree a valid arithmetic tree
// every leaf in an arithmetic tree is a numeric value, for example: “1”, “2.5”, or “-0.234”
// every internal node is a binary operation: “+”, “-”, “*”, “/”
// binary operations must act on exactly two sub-expressions (i.e. two children)
// note: all the values and operations are stored as String objects

public double evaluateArithmetic()
// evaluate an arithmetic tree to get the solution
// if a position is a numeric value, then it has that value
// if a position is a binary operation, then apply that operation on the value of it’s children
// use floating point maths for all calculations, not integer maths
// if a position contains “/”, its left subtree evaluated to 1.0, and the right to 2.0, then it is 0.5

public String getArithmeticString()
// Output a String showing the expression in normal (infix) mathematical notation
// For example: “(1+2)+3”
// You must put brackets around every binary expression
// You may omit the brackets from the outermost binary expression

Background information:
Storing arithmetic expressions in comparable tree structures is one approach used to implement exact mathematics (recall from the first tutorial, where we showed that floating point mathematics is only an approximation of an actual value.) If we store the expression used to actually calculate a number, then it is possible to compare expressions in a way that does not use floating point mathematics. However, checking for equivalence of two mathematical expressions is considerably more complicated than the comparisons made in this assignment. For example, we would also need to be able to do some non-lossy transformations, such as simplifying expressions like (2/1) to 2, or (4/6) to (2/3)).

Part 5: Written report [10%]

Submit a report containing the following:

- Testing report (1 page, plus appendix): Explain succinctly how you tested your code and why your testing is comprehensive. Provide a list of your test cases in the appendix, with expected and observed output.
- Group work (If you submitted as a pair) Explain how you collaborated and the percentage of authorship for the various sections.