

CSC 115: Fundamentals of Programming II

Assignment #4: Binary Trees

Due date:

Sunday, November 25, 2018 at 23:55 pm via submission to connex.

Learning outcomes:

Upon completion of this assignment, you need to be able to:

- Implement a binary tree using a reference based structure.
- Become more familiar with the use of generics in Java.
- Implement iterators that traverse a binary tree in pre, in, and post order.
- Implement a binary search tree that extends a regular binary tree.
- Continue to apply good programming practice in
 - designing programs,
 - proper documentation,
 - testing and debugging problems with code (**the marking tester will not be provided this time**).

References:

- CSC115 Java [coding conventions](#).
- Textbook Chapter 11 Sections 11.1, 11.2 and 11.3.

Introduction

In this course, we create data structures that are containers for *items* as a means to store, insert, retrieve and remove these items. Each of the ADTs we've studied have been linear (*linear*) data structures, in that they store items one after the other. In this assignment, we implement a basic binary tree that stores and allows access to any object in a non-linear ADT. Extending the basic binary tree, we also create a binary tree that stores data types, taking advantage of their *natural ordering*.

Traversing a linear data structure is intuitive; we start at one end and continue through the line one item at a time. A binary tree, however, has four different traversals: three that are implemented using recursion, and another (*level order*) that is implemented using a queue. In this assignment, you will implement the three recursive traversals by completing a linear data structure that stores the ordered items, and uses an Iterator object to serve the items.

To illustrate the uses of the basic binary tree, we dust off the arithmetic expression from

Assignment 3 and discover that converting infix to postfix can be performed by returning a post-order iterator. To illustrate the uses of a binary *search* tree (BST), we insert items and note that an in-order iterator provides us with a sorted list of items.

Quick Start

- (1) Create a fresh directory to contain this assignment: CSC115/assn4 is a recommended name. Download this pdf file and the a4_files.zip file to this directory.
- (2) Extract the a4_files.zip file. A directory will be created to store both the Java files and the HTML files which contain the specifications for the public classes.
- (3) The following files are complete:
 - `TreeNode.java`
 - `InvalidExpressionException.java`
 - `TreeException.java`
 - `DrawableBTree.java`
 - `Tools.java`
 - `Expression.java`
- (4) The following files are partially done and need to be completed:
 - `BinaryTree.java`
 - `BinaryTreeIterator.java`
 - `BinarySearchTree.java`

[Hint]

This assignment may seem complicated on your first read-through. There are a lot of concepts, and many of them are set in the assignment to guide you through the process of creating programs that are modular and interact well with other programs. Please take our advice and go through this assignment in steps. These steps have been set out with the intention of helping you develop understanding incrementally as you progress from one step to another. As a bonus, we hope you note the intricacies of the provided code and use these to develop your own personal style of programming.

Descriptions of the Helper Classes

The completed java classes are used to support and test the Binary trees. You must NOT change these, however, it is highly recommended that you examine and understand them.

- `InvalidExpressionException` and `TreeException` need no explanations.
- `TreeNode` is a package-protected class and therefore has no specification document. The textbook has a description of a `TreeNode` that contains a reference to both a left and right child. The authors' implementation appears less complicated, however, all paths in their trees are one-way. The `TreeNode` in this

assignment also contains a reference to the parent node, which may or may not be used. Using a parent data field requires some extra *linking* statements during insert and remove, but it allows two-way directional access. You may choose whatever you like: if you choose not to use the parent data field, then just ignore it in the tree implementations.

- **Expression** is a slightly modified ArithExpression from Assignment 3. You may use this class as the tester for the BinaryTree class once it is working. The tokenize method is unchanged, except that the infix tokens are stored in an array. To access the postfix expression, the infix expression is inserted into the BinaryTree and if everything works as expected, the postfix expression is created by the postfix traversal of the binary tree. You do not need to do anything except run this class.
- **Tools** is exactly the same as in Assignment 3. It acts as a helper class for the Expression class.
- **DrawableBTree** is provided as a means to **render** the BinaryTrees. The rendering statements are already included in the Expression class so you can test the BinaryTree class **visually**. Once you start inserting a few items into the BinarySearchTree, simply add the following two lines to the main method to render the tree:

```
DrawableBTree dbt = new DrawableBTree<classname>(theTree);  
dbt.showFrame();
```

where classname is the actual type of items stored (e.g. String) and theTree is the variable name of the BinarySearchTree object. The second statement causes a frame to become visible on the desktop. You can resize the frame as desired and the tree will expand to fit the frame. While the frame is visible, the Java Virtual Machine (JVM) will keep running, and will quit when you close the frame.

Implementation

The following sections describe the classes that you must complete, in the order you create them.

BinaryTree

The **BinaryTree** class is a basic binary tree that contains generic elements. Implement the methods as instructed, following the specifications. Most of the methods follow the descriptions from the textbook, where the authors provide details on the implementation. The only added method is the public height() method, which calls a private height method: you must use recursion to implement this method.^[*]

*One can never get enough recursion practice.

BinaryTreeIterator

The textbook discusses the [BinaryTreeIterator](#) interface in detail. An [Iterator](#) object allows any number of users to traverse a data structure at one time (think about it as a personal bookmark for a book that is shared). Follow the tip notes in this partially completed class to create the iterator object. Note that in the BinaryTree, each iterator method returns a newly created iterator object.

BinarySearchTree

The [BinarySearchTree](#) class extends BinaryTree in that it stores only those items that have a natural ordering. The reason for this is that the BinarySearchTree imposes an ordering on its items. In Java, every class that has a natural ordering implements [Comparable](#) class. In other words, the item class must have a `compareTo()` method that determines which of two items come earlier in an ordered list. Java generics allows us to specify that every item in the BinarySearchTree implements Comparable by requiring that **E extends Comparable<E>**.^[†]

BinarySearchTree *inherits* every non-public data field and method from BinaryTree, except the constructors. We do not want BinarySearchTree to inherit the public methods that allow a user to add whole left or right subtrees, so we must *override* these methods with new statements that prevent a user from implementing the BinaryTree version of these methods. We demonstrate how to do this with one of the methods and you are to implement the second one.

Most methods are described in the textbook. Several methods are easily implemented using recursion, such as `retrieve()` and `insert()`. Although the textbook authors use recursion for each of the remove cases, you do not have to if you prefer to take advantage of the parent data field in the `TreeNode` class.

Submission

Submit the following completed files to the Assignment folder on `conneX`.

- BinaryTree.java
- BinaryTreeIterator.java
- BinarySearchTree.java

Please make sure you have submitted the required file(s) and `conneX` has sent you a confirmation email. Do not send `[.class]` (the byte code) or `[.zip]` (the compressed folder) files. Also, make sure you *submit* your assignment, not just save a draft. Draft copies are not made available to the instructors, so they are not collected with the other submissions. We *can* find your draft submission, but only if we know that it's there.

[†]If you find yourself asking why Java generics require the keyword *extend* instead of *implements* for the *interface* Comparable, you would not be the first.

A Note about Academic Integrity

It is OK to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution.

Grading Scheme

Marks are allocated for the following:

- Proper programming style is demonstrated, as per the [coding conventions](#) on CSC115 connex Resources. All “NOTE TO STUDENT” comments must be removed from the code.
- Source code that follows the specifications and instructions.
- Internal testing:
 - BinaryTree does not need internal testing. The Expression class can be run to test the basics.
 - BinarySearchTree must test ALL its methods, as well as the appropriate public methods that inherit from BinaryTree.
- You will receive no marks for any Java files that do not compile.

Requirement	Marks
Proper programming style as per the code conventions on CSC115 connex Resources	2
Write tests for ALL the methods in BinarySearchTree in its main()	1
Pass tests for BinaryTree class (3 test)	
Constructor (1 test)	1
attachRightSubtree (1 test)	1
height() method (1 test)	1
Pass tests for BinaryTreeIterator class (9 test)	
Constructor (1 test)	1
In-order traversal (4 test)	2
Post-order traversal (4 test)	2
Pass tests for BinarySearchTree class (9 tests)	
attachRightSubtree (1 test)	1
insert() method (1 test)	1
retrieve() method (1 test)	1
In-order traversal (1 test)	1
Post-order traversal (1 test)	1
delete() method (4 test)	4
Total	20