COMP 1406 Fall 2015

Assignment #1: Recursion in BSTrees

Due Date: Thu. Oct. 22 by 10:00pm in culearn

Assignment Revisions and Corrections will be noted here.

Recursion in Binary Search Trees

This assignment is intended to give you practice writing recursive code that traverses a common computer programing structure: a binary search tree. In this assignment you will implement a simple binary search tree using recursive methods: insert, find and remove. These trees are very common to computer science and are typically handled with recursive methods to help attain their intended efficiency.

It is intended that you write this assignment by modifying the answer code from assignment #2. You can use your own answer code or the code we post as answer code as your starting point. We will describe the requirements as though you are

modifying the answer code from assignment #2 but you are of course free to start from scratch as well.

Assignment Marking:

This assignment is based on 21 specific design requirements numbered R1.1... R5.3. Two marks each for a total of 42 marks. You get 2 marks if the requirement is satified and well implemented, 1 mark if only partly satisfied or poorly implemented, and 0 if the requirement is not satisfied.

In addition there are some general programming and good practice requirements that you must satisfy. These requirements are numbered R0.1, R0.2 .... You will lose marks from your assignment score if these requirements are not satisfied.

Good Programming and Good Practice Requirements

The following requirements will pertain to all your assignments regardless of what your application is supposed to do (i.e. regardless of the design requirements). These requirements are to ensure that your code is readable and maintainable by other programmers (or readable by TA's in our case), and that your program is robust and follows good software engineering practrices You will lose 2 marks from your total assignment mark for each of the

following requirements that is not satisfied. If you do not satisfy requirement R0.0 or R0.1 you will get nothing for the assignment.

R0.0) IMPORTANT Uniqueness Requirement. The solution and code you submit MUST be unique. That is, it cannot be a copy of, or be too similar to, someone else's code, or other code found elsewhere. A mark of 0 will be assigned to any assignment that is judged by the instructors or the TA's not to be unique. (You are free to use any code posted on our course web site as part of the course notes or example code.)

R0.1) CODE ORGANIZATION, SUBMISSION AND COMPILATION: Your code should be submitted to culearn as an IntelliJ IDEA project. You can zip up the files is you like but you must use .zip format (not other compression formats.) The markers must be able to compile your code using the Java 1.8 or later compiler from within the course IDE (IntelliJ IDEA in this case). (Demonstration code will be provided as IntelliJ projects and your work will be assessed by opening your code as an IntelliJ IDEA project, compiling the code and then running it. For the purposes of assignments only have one file with a main(){...} routine in it and use the word "main" as part of the class name. The following process will be used to mark your assignment code:

1)The TA will open IntelliJ and open your code as an IntellliJ project. (If you compress your code then provide a .zip file only (we will not support .rar, .tar, ... etc.) Only .zip files.

2)The TA will then rebuild (compile) your project and look for a source file among your code with "main" in its file name and then run that as a java application.

3)Once your code launches it will be evaluated against the assignment requirements.

If any of steps 1,2 or 3 fail the assignment gets a mark of 0. So be aware no partial marks will be awarded to assignments that don't compile and run. Assignments are intentionally broken down into many small requirements. It is better have running code that satisfies some of them than to have code that won't compile and run but claims to be address more requirements.

The following good practice requirements will be in effect for all assignments.

R0.2) All of your variables, methods and classes should have meaningful names that reflect their purpose. Do not follow the convention common in math courses where they say things like: "let x be the number of customers and let y be the number of products...". Instead call your variables `numberOfCustomers` or `numberOfProducts`. Your

program should not have any variables called "x" unless there is a good reason for them to be called "x". (It's OK to call simple for-loop counters i,j and k etc. when the context is clear and very localized.)

R0.3) All variables in your classes should be `private`, unless a specific design requirements asks for them to be public (which is unlikely). It is good programming practice to design objects that provide services to others through their `public` methods. How they store their variables is their own private business.

R0.4) Robustness Requirements: Your program should never crash when is is being run for marking. Make sure you have no null pointer exceptions or attempt to access an array or data structure out of bounds. (We get especially annoyed by out of bounds errors since they still seem to be the number one bug in programming and have been for a long long time!)

R0.5) Code Comment Requirements: Comments in your code must coincide with what the code actually does. It is a very common bug in industry for people to modify code and forget to modify the comments and so you end up with comments that say one thing and code that actually does another. By the way, try not to over-comment your code; instead choose good variable names and method names which make the code more "self commenting".

R0.6) Hard Coded Constants: Your code should not have hard coded constants used in places like if statements or function parameters. Your constants should have meaningful names. Don't have if statements like `if(ball.getLocationX() + 40 < 100)` ...; instead your code should look like `if(ball.getLocationX() + ballRadius < rightBoundaryX) ...;` If necssary create local variables that reflect the use of the constant. e.g. `double rightBoundaryX = 100;` then you can refer to that in your program logic.

VERY IMPORTANT: Any demo code or sample code fragments provided may have bugs (although none are put there intentionally). It is part of your task to identify errors in the code and in the requirements. Please report errors so they can be fixed and discussed in class.

Design Requirements

Binary Search Tree
File
Run
John
Andy
Frank
Bob
Fred
desc:

Pictured above is an application showing a binary search tree, or BST. A BST is a binary tree meaning each node has at most two children. The children are referred to as the left child and right child. Also it is a search tree meaning that items, or labels, in the left subtree of a node are less than, or "come before" the label of the node. Similarly labels in the right sub-tree of a node are great than or "come after" the label of the node. Here greater than and less than refer to alphabetical order of the node labels. This order can be determined for strings using the String class' `compareTo()` method.

The advantage of a search tree is that you can easily find an item by starting at the root and at each node you can decide to visit the left subtree or the right subtree by comparing the the label you are looking for with that of the node you are at. Notice if you follow this strategy you will either find an item by traversing a single path from the root to the item, or reach a leaf where you can conclude the item you are looking for is not in the tree. The strategy lends itself very well to recursion.

If you follow the strategy of starting at the root and then visiting either the left or right subtree until you find what you are looking for, the number of nodes you will visit is proportional to the height of the tree. Conversely, if you had an array, or arraylist, of unsorted nodes and you looped through it looking for

a node the number of nodes you would visit is proportional to the total number of nodes.

There is a famous theorem that says if a tree is sufficiently balanced the height of the tree is proportional to the log of the number of nodes in the tree. Thus a sufficiently balance BST of n nodes can be searched for an item in O(log(n)) time. Whereas an arraylist of unsorted nodes would take O(n) time to search. This is the potential advantage of a binary search tree. To achieve this these methods are typically implemented recursively.

Though it might not be obvious you can insert an item into a binary search tree in O(log(n)) time as well and you can remove an item from a binary search tree in O(log(n)) time. Again in both cases recursive code provides the most straight-forward implmentation.

For this assignment you can start with the answer code from assignment #2 (either that provided by us, or your own code). Your app should provide the "insert", "remove" and "find" buttons and methods and do the animations as in the previous assignment. The difference will be that you will be building a binary search tree instead of an offspring tree. Also, in this assignment the insert, remove and find methods **must** be implemented recursively. Note there is no concept of gender in a BST like there was in our previous "offspring" tree assignment.

BSTreeNode Requirements.

R1.1) Replace the `TreeNode` class from the previous assignment with a `BSTreeNode` class subject to the following requirements.

R1.2)The `BSTreeNode` should allow only two children: a left child and right child.

R1.3 Instead of an arraylist of children the `BSTreeNode` should just have a `leftChild` and `rightChild` variable each of type `BSTreeNode`.

R1.4 Instead of an `addChild()` method the `BSTreeNode` should have an `addLeftChild()` and `addRightChild()` method. You are free to add other needed methods.

Finding Nodes Requirements.

R2.1) You should be able to find and highlight a node in the tree by typing the node label in the application text field and pressing the "Find" button subject to the following requirements.

R2.2)The find operation should be done by an `public BSTreeNode find(String aLabel)` method in the `BSTreeNode` class. It should return the node if found, and hightlight the node. It should return `null` if no node is a found.

R2.3)The `find` method must be implemented recursively. (You are not allowed to have a "helper"

arraylist of all the nodes". Your code must traverse the tree by starting at the root and visiting the appropriate children.

R2.4)Your `find` method should run in time proportional to the height of the tree. This will happen automatically if you visit only the required left subtree or right subtree.

Node Insertion Requirements.

R3.1) You should be able to insert a node into the BSTree by typing a single word in the text field and clicking the "insert" button..

R3.2)The insertion should be done by an `public void insert(String aLabel)` method in the `BSTreeNode` class. It should insert a new node with label aLabel in proper search tree order. If a node with the same label as the one you are trying to insert already exisits your `insert()` method should do nothing and return.

R3.3)The `insert` method must be implemented recursively. (You are not allowed to have a "helper" arraylist of all the nodes". Your code must traverse the tree by starting at the root and visiting the appropriate children. (We will go over the strategy for recursive insertion in class, or you can also google it. It is a well known technique.)

R3.4)The tree resulting from the insert must be a legal binary search tree.

R3.5)Your `insert` method should run in time proportional to the height of the tree. This will happen automatically if you only visit the appropriate left subtree or right subtree during insertion.

Node Removal Requirements.

Note we will only implement a simplified removal: we will only require that you be able to remove leaf nodes. (If you are looking for additional challenge see if you can code your method to also allow removal of non-leaf nodes.)

R4.1) You should be able to remove leaf nodes from the tree by typing the node label in the application text field and pressing the "Remove" button.

R4.2)The removal should be done by an `public remove(String aLabel)` method in the `BSTreeNode` class.

R4.3)The `remove` method must be implemented recursively. (You are not allowed to have a "helper" arraylist of all the nodes". Your code must travese the tree by starting at the root and visiting the appropriate children. (We will go over the strategy for recursive removal in class, or you can also google it. It is a well known technique.)

R4.4) The tree resulting from the removal must be a legal binary search tree. (You need only implement removal of leaf nodes.)

R4.5)Your `remove` method should run in time proportional to the height of the tree. Again this will happen automatically if you only visit the appropriate left subtree or right subtree during removal.

Node Animation Requirements.

If a mutable structure changes suddenly it can be hard to follow what the changes were. But if the structure "slides" or "morphs" into its new configuration then the changes are easy to follow. We want you to implement this like in the last assignment. The main difference here is that a left child should always be drawn to the left of its parent and a right child should be drawn to the right of its parent.

R5.1) When the animation timer is running (controlled by the provided menu items) insertion and removal of nodes should cause the restructuring of the tree to be animated over time (we will show a demo in class). When the animation timer is not running the changes can be abrupt.

R5.2) When animation is running and a new child node is inserted, the child node should start at its parent's location and "slide" to its proper location over

the course of time (a few seconds). Other nodes should also slide to their new intended locations.

R5.3) Left child nodes should be drawn to the left for their parent; right child nodes should be drawn to the right of their parent.