Project 2, Augmented Binary Search Trees

Due: Please check due date on Blackboard

Objectives

This programming assignment will have you practice working with and implementing Binary Search Trees (BSTs).  You will be augmenting an existing data structure, and will be writing non-trivial recursive code.  As the base for the BST code will be provided to you, this assignment will also help you practice reading and modifying code that has been written by someone else.

Introduction

Most of the time a data structure will include support for all of the operations that we need.  When that’s not the case, we can often ***enhance*** the data structure by adding a small number of member variables and additional operations.  In this project, you will enhance the author’s Binary Search Tree (BST) code to support a number of new operations without modifying any of the author’s original code.  By creating an Augmented BST as an interface, developers still having access to the generic (public) methods common in all BSTs. The following methods MUST be implemented

1. **AnyType NthElement (int n)**

Returns the n-th element (starting from 1) of the in-order traversal of the BST.

1. **int Rank (AnyType x)**

Returns the “rank” of x. The rank of an element is its position (starting with 1) in   
an in-order traversal.

1. **AnyType Median ()**

Returns the median (middle) element in the BST.  If the BST contains an even   
number of elements, returns the smaller of the two medians.

1. **bool IsPerfect()**

Returns true if the BST is a perfect binary tree i.e. is it balanced?

1. **bool IsComplete()**

Returns true if the BST is a complete binary tree.  (This command is extra credit!)

6)             **string PrintLevels(int numLevels)**

Generates and returns the level-order output described in the sample output below. If a child of the parent node is NULL, instantiate an empty BinaryNode and set as the NULL child so that a perfect tree is printed for numLevels levels.

7)     **int RemoveResidue()**

Returns the number of BinaryNodes that were instantiated by the printLevels function.  If a BinaryNode is an empty node, delete it and count it as residue that was removed. This function will “normalize” the BST after a call to PrintLevels.

Most of these operations could easily be implemented by performing an in-order traversal inside the BST and perhaps placing the results in an ArrayList. However, such a method is extremely inefficient. Instead, we are going to achieve faster performance by “augmenting” the BST nodes. You will add a new private integer data member to the **BinaryNode** (“**m\_size**”) which stores the size of the tree rooted at that node i.e. the subtree size (including the root itself).

You must develop your own algorithms for these operations, but obviously you will use the new tree size data member to guide your search. Think before you code! Think *recursively*!

List of valid error strings:

Invalid Argument Exception

Invalid Command Exception

Node Not Found Exception

Item Not Found Exception

NULL Argument Exception

Apply these where you see best fit. They should be followed by some meaningful explanation of what went wrong. i.e. “NULL Argument Exception: No median value for a tree of zero size.”

Command Line Arguments

The command line for this project has two command line arguments (in this order):

1. The name of a data file, which contains integers separated by whitespace.
2. The name of a command file, which contains commands (explained below).

Your program will read the integers from the data file and insert them into the BST. It will then read the command file, perform the requested operations, and produce the specified output. The data file may contain duplicates. If it does contain a duplicate, **do not** insert the duplicate.

The Command File

Each line of the command file represents a single command. Blank lines may occur anywhere in the file and should be ignored. Lines that begin with the pound (‘**#**’) symbol are comments and should also be ignored. You may assume that valid commands are well-formed, but not that all commands are valid. If you encounter an invalid command you should ignore that line in the file, print an appropriate error and continue to the next command. An appropriate message should be printed for any valid command that produces no results (*i.e.,* finding the rank of an integer that does not exist in the tree). The following commands will be found in the command file.

**PRINT <number of levels>**

Prints the specified number of levels of the BST according to a level-order traversal. See the sample output below for required formatting and data.

**RESIDUALS**

This command cleans up the excess nodes that were created in order to print a perfect tree. Returns the number of BinaryNodes that were instantiated by the printLevels function.  If the tree is NULL, return 0. Otherwise return a non-negative integer.

**RANK <N>**

Prints the rank of the integer N. A node’s “rank” is its position within an in-order traversal of the tree.  If N is not found in the tree, the program should print an appropriate message.

In the tree used for the sample output the rank of 30 is 4 and the rank of 70 is 18.

**NTH <N>**

Prints the Nth integer if an in-order traversal were performed. If there are less than N integers in the tree, an appropriate error message should be produced.  In the tree used for the sample output the command NTH 6 prints the value 37.

**MEDIAN**

Prints the median value in the tree. In the tree used for the sample output the median value is 47. In the event of an even tree, the lesser of the two elements is selected (This should be clear but I am sure a question about this will show up on Piazza. I digress…)

**REMOVE <N>**

Removes the specified integer, N, from the tree. If N is not in the tree print “FAILED” otherwise print “SUCCESS”

**PERFECT**

This command checks whether or not the BST is a perfect binary tree, and prints YES if true, and NO if the BST is not.  (See the notes for a definition of perfect trees)

**COMPLETE** Implementing this command is extra credit!

If implemented, this command prints YES if true, and NO if the BST is not. If it is not implemented, you must ignore the command as if it were a comment.

Example Command File

A command file might look this:

|  |
| --- |
| **# test the RANK function**  **RANK 33**  **RANK 12365**  **# find the 33rd element**  **NTH 33**  **PRINT 4**  **REMOVE 67890**  **MEDIAN**  **# is the tree perfect or complete?**  **PERFECT**  **COMPLETE**  **#end of command file** |

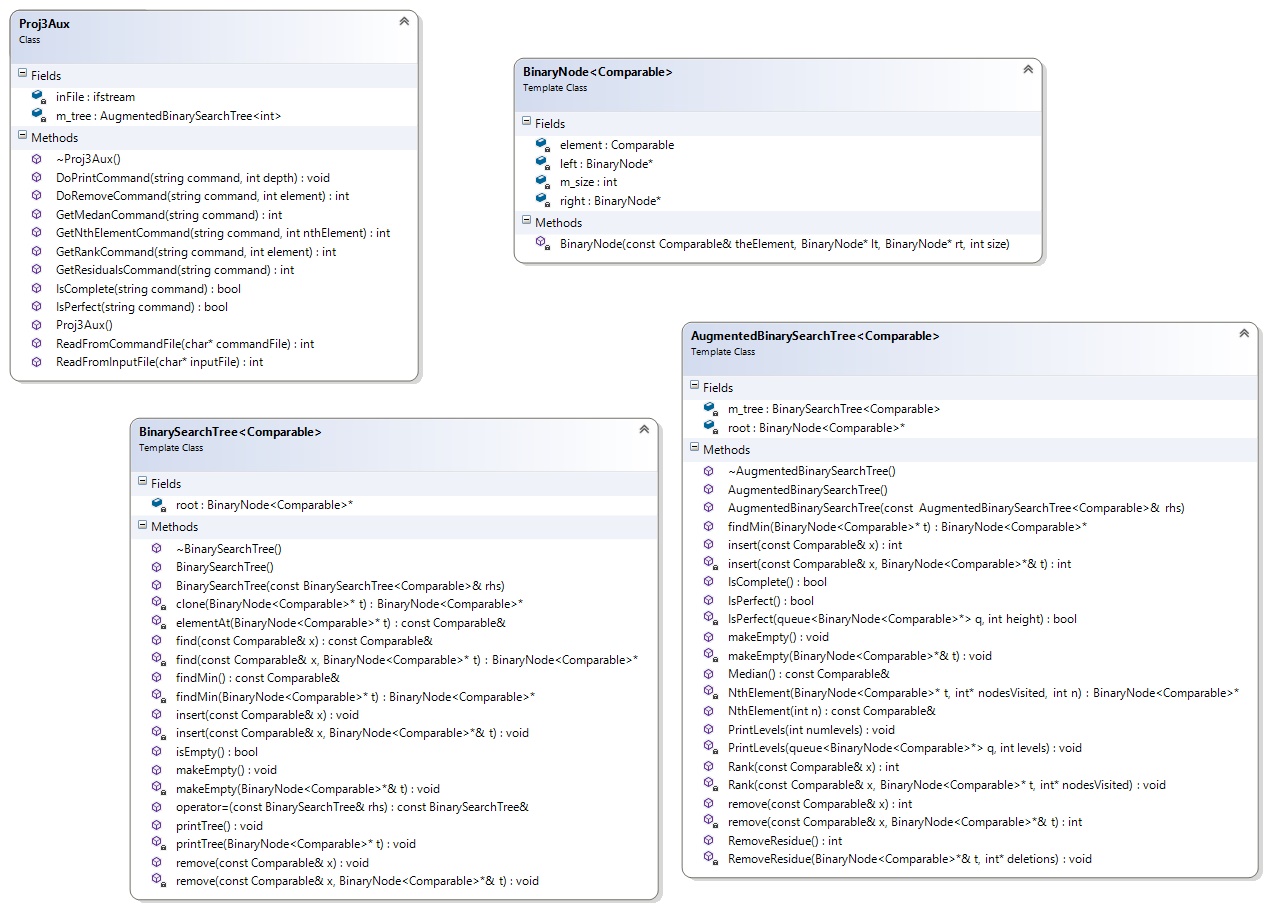
Getting Started and Your Tasks

To get started:

* Download the **BinarySearchTree.cpp**, **BinarySearchTree.h, AugmentedBinarySearchTree.h** and the driver.cpp file
* Augment the **BinaryNode** class by adding an integer (“**m\_size**”) to store the tree's size.  
  (The size includes the root – so a node with no children is a tree of size 1.)

Your tasks include:

* Using the provided BST code, enhance the AugmentedBinarySearchTree by implementing the method stubs found. All function signatures in AugmentedBinarySearchTree.h are locked in, you cannot change the parameters with the exception of IsComplete() should you attempt the extra credit.
* You are NOT directly using anything from the BinarySearchTree.h/.cpp. You are copying the and augmenting it in AugmentedBinarySearchTree
* Writing Proj2Aux functions in a separate file based on the UML diagram provided.



* Creating a Makefile for this project.  At minimum it must contain three rules:
  + The first rule in the file must build (compile and link) your project – typing the command “**make**” should run this rule.
  + The second required rule must run your project – by typing the command   
    “**make run FILE=input.txt COMMANDS=commands.txt**” where the exact filenames can be replaced.  Refer to Prof. Lupoli’s slides on Makefiles for an example of how to set this up. If you haven’t found out by now, makefile variables are declared via the following syntax: dollar sign, paren, variable name, close paren.
    - You will need to **redirect stderr** to the file **errors.txt**.  This implementation separates the errors from standard output which must ***print to the terminal/screen*** NOT to results.txt.  You will need to do the research on your own.  Command line args remain at 3, so there is not an issue with the driver.
  + The third required rule must clean your project of the driver “Proj2”, all .o files, and residual files (~, #, etc) that appear during the development stage – typing the command “**make clean**” should run this rule.

Other Requirements, Notes, and Hints

These items cover those topics not addressed elsewhere in the project description.

Requirements:

* Although we are only using the BST for integers, the BST must remain a generic class.
* No new friendships.
* No new data members (other than the member variable for tree size) in the **BinarySearchTree** or the **BinaryNode** classes.
* The name of your executable must be **Proj2**, and must be created by simply typing “**make**”.
* Duplicate values are not allowed. Attempts to insert a duplicate value should be ignored and ensure that m\_size for each node along the search path is not modified.
* Attempts to remove non-existent values should print FAILED to stdout, not stderr.
* From a coding perspective, the easiest way to avoid duplicate insertions or attempting to remove non-existent elements is to first call **find( )**. However, this technique is NOT permitted for two reasons:
  1. Calling **find( )** before each insert/remove doubles the running time for these operations and is therefore inefficient.
  2. Recursively coding the insert/remove methods to handle these situations is a great learning experience.
* Each command read from the file (and its arguments, if any) must be echoed as part of your program’s output. There is a particular order everything should be printed out, we will run a diff on the output so make sure you do the same and tweak accordingly.
* The level order print (**PRINT** command) prints the value of each node, the size of the tree rooted at that node, and the value of the node’s parent in that order, inside parentheses, separated by commas: (node value, node’s subtree size, parent value).
* Since the root has no parent, print NULL for the root’s parent value.
* For readability, separate the levels with a blank line and print no more than 6 node/size/parent values per line. If a level requires multiple lines, use consecutive lines (without a blank line between them).  See the sample output below for an example.
* Efficiency counts! Use the most efficient algorithm possible. In particular, don’t run multiple searches in the tree when one will do. (Yes, we know it technically doesn’t matter from an asymptotic analysis point of view.)

Hints:

* Almost all of these operations should be written recursively. (But not every single one.)
* A level order traversal uses a queue to store nodes to be printed. The pseudocode is given below. You may use the STL queue or write your own.

**instantiate a Queue**

**enqueue information about the root**

**while the Queue is not empty**

**(1)dequeue the next node's information**

**(2)print the information about the node**

**(3)enqueue information about the node's children (if the**

**node is empty, construct a residual node so that a**

**perfect tree is always printed).**

* You should **not** create a member variable like **m\_parent** that points to the node’s parent.  Refer to the class notes for information on identifying parent nodes in BSTs.
* Make sure that your implementation does not increment nodes until an integer was successfully inserted. Think recursively!
* Reminder: Your program should **not** use the find( ) function – there are more efficient ways to retrieve the location of a specific value when you have an augmented BST.
* If a BST is empty, then it is balanced, i.e. it is perfect. Perhaps counterintuitively, it is also Complete by definition.
* Implementing level order printing of the BST will/may help with developing the logic for IsPerfect.

Notes:

* Be sure to adhere to the definitions of "[perfect binary tree](http://xlinux.nist.gov/dads/HTML/perfectBinaryTree.html)" and "[complete binary tree](http://xlinux.nist.gov/dads/HTML/completeBinaryTree.html)" as used in the text and the course notes. Other material on BSTs may use slightly different terminology.
* The numbers in input.txt will always be positive integers.

Sample Output

This output is provided to show the format of the level-order print (5 levels).

This is the level-order print of the sample tree below.

**Level 0:**

**(55, 21, NULL)**

**Level 1:**

**(40, 12, 55) (60, 8, 55)**

**Level 2:**

**(30, 6, 40) (45, 5, 40) (NULL, NULL, 60) (70, 7, 60)**

**Level 3:**

**(20, 3, 30) (35, 2, 30) (44, 2, 45) (47, 2, 45) (NULL, NULL, NULL) (NULL, NULL, NULL)**

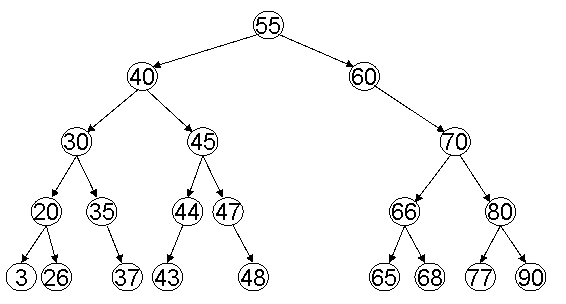
**(66, 3, 70) (80, 3, 70)**

**Level 4:**

**(3, 1, 20) (26, 1, 20) (NULL, NULL, NULL) (37, 1, 35) (43, 1, 44) (NULL, NULL, NULL)**

**(NULL, NULL, NULL) (48, 1, 47) (NULL, NULL, NULL) (NULL, NULL, NULL) (NULL, NULL, NULL) (NULL, NULL, NULL)**

**(65, 1, 66) (68, 1, 66) (77, 1, 80) (90, 1 ,80)**



Extra Credit

For 10 points of extra credit, implement a method to determine if your binary search tree is a complete binary tree. Your program should output an appropriate message in response to the COMPLETE command in the command file.

If you do not wish to implement the extra credit, when the complete command is encountered, output the following message to stderr:

“**COMPLETE command has not been implemented.**”

Project Grading

Project grading is described in the Project Grading Policy. Please re-read the Project Grading Policy for further details on honesty in doing projects for this course.

This project will follow the stated course policy on late projects.  See Blackboard for details.

Project functionality testing will fall into the following categories. The list below does not necessarily include all test possibilities.

* Basic Tests
  + A small tree with an odd number of integers inserted in random order with no duplicates and no deletion.
  + A small tree with an even number of integers inserted in random order with no duplicates and no deletions.
* More Complex Tests
  + A tree created by random insertions and deletions.
  + Attempting to insert duplicate values.
  + Attempting to remove non-existent values.
  + A tree created from integers inserted in sorted order with or without deletions.
* Atypical Tests
  + A tree with a single integer.
  + An empty tree.
  + One or more files which do not exist or cannot be opened.
  + A command file with no valid commands.
* Stress Tests
  + A very large tree built from a large number of random insertions.
  + A very large tree built from a large number of random insertions and deletions.

Project code will be evaluated for the following:

* Appropriate Object-Oriented design and implementation.
* Appropriate use of recursion.
* Efficiency.
* Appropriate Function Comment Headers for all .h functions INCLUDING AugmentedBinarySearchTree.h. Your comments should reflect your understanding of each method, the parameters and their uses.

Files To Be Submitted

Follow the course project submission procedures. You should copy over all of your C++ source code with **.cpp**/**.h** files under the **src** directory and ZIP them together. You must also supply a Makefile build file. In particular, the following Unix commands should work:

**cd /src**

**make**

**make run FILE=input.txt COMMANDS=commands.txt**

**make clean**

The C++ programs must be in directories under the **src** directory. Your submissions will be compiled by a script. The script will go to your **src** directory and run your makefile.  This is required. You will be severely penalized if you do not follow the submission instructions.