ELEC 278 – Data Structures and Algorithms

Lab 4: AVL Trees

Start: November 2, 2022 **Due:** Day 7 of Lab Day

Contents

1. Basic Lab Information

You should review the Lab Overview document prior to reading the material for this lab exercise.

If you have already installed a C compiler on your computer (or have one available), then you can continue with the lab. If you have not installed the C programming tools, please stop working on this lab, and review Lab Overview document Appendix A. You may also wish to review Tutorial 1 before you attempt this lab.

You should also review Appendix B of the Lab Overview document. This contains important information about how to submit your assignment.

You may also find it useful to review previous labs, especially Lab 03, as you work on this lab exercise.

1.1 Background

A weakness with binary search trees (BSTs) is that it is possible for their search performance to degrade significantly. If the data inserted into a BST is somewhat ordered, then the tree begins to resemble a long linked list, with a few branches off to the side. Performance on searches degrades significantly as well, moving from O(logN) towards O(N).

2. Lab Resources

Specific to this lab:

1. Lab Instructions (this document)
2. A zip file containing source code (in a folder called Lab04Src). Check the file "PACKING.LIST" (it is a plain text file) for a list of files that are included.

Learner provided:

1. Programming environment selected by the learner. The Lab Overview document provides a list of possible C programming tools that you may use.

This lab has code for you to read. The purpose of the example code is to reinforce what you have seen in the class material. It also provides you with code to cut-and-paste into the code you write to solve the assigned problems. This lab has multiple programming problems to solve.

Note on supplied code:

You will notice that the supplied collection of code contains a version of BST like last lab's code but missing some important components. This is because both BST and AVL trees are Binary trees, and the code to create a tree descriptor and a node are the same for both. The code to do a find for a specified key is also common to both BST and AVL. Where the code differs is in the **Insert** routine and in the **Delete** routine. The supplied BST code does not include a delete because it is not required for the lab. Also note that print routines could have been placed in the generic bintree.c module, but they were not.

You will see why a BST module is included when you look at the second deliverable for this lab.

2.1 List of Files provided with this Lab

(See the file PACKING.LIST included in the Lab04Src.zip file for the most up-to-date list.)

avl.c
avl.h
bintree.c
bintree.h
bst.c
bst.h
lab04data_B.txt
main.c
main_B.c
Makefile
MakeRandom.c
avl_data_A.txt
output.txt
Packing.List

3. Lab Objective

There are two objectives for this lab:

a) Add three import
b) Something else
c) Using code from Lab 3, compare the performance of BST and AVL trees.

3.1 Topics covered in this lab

- AVL Trees (BSTs) – Implementation choices.
- Performance comparison between AVL and BST searches.

4. Lab Steps

4.1 Overview

This lab includes two steps, all of which involve adding code to the code provided to you in the zipped directory **lab04Materials.zip.**

In the first step, you will implement code to calculate the height and balance factor of any node in an AVL tree, and you will also implement functions to rebalance an AVL node after an insertion (rotations).

From the node definition, you can see that each node has a field recording its own height. Start by assuming that the heights of the left and right subtrees are correct. Calculate the height of the node

root using the heights of its subtrees. No recursion is needed here.

The balance factor of a node root is :

$$Balance\ Factor = H_L - H_R$$

After a value is inserted in the tree, the tree may be unbalanced. If the balance factor if a node is less than -1 or greater than 1, then the node is unbalanced and needs rebalancing.

As discussed in class, there are 4 types of rotations:

Name	Rotation
Left of Left (LoL)	Right rotation
Right of Right (RoR)	Left rotation
Right of Left (RoL)	Left rotation on subtree Then Right rotation
Left of Right (LoR)	Right rotation on subtree Then Left rotation

Determine the type of rotation needed and perform it on the tree.

In the second step, you will modify the supplied BST code, so that the routine to implement an insert in the BST has the same interface as the code to insert to the AVL. That is, if you inspect the AVL code you will see the code to do an insert is called ***with a pointer to the AVL Tree***, but the code to do an insert to the BST is called ***with a pointer to a node***. You will create a version of insert for the BST tree that is called with a pointer to the BST (tree). Look carefully at the AVL code, because you will see that it is necessary for recursion purposes, to have an insert function with a parent node pointer as parameter.

The point of the second step is to compare the performance of AVL and BST. There is a file containing a large set of numbers that has been deliberately constructed to exploit a BST weakness – its tendency to become quite imbalanced if its data is inserted somewhat in order. I have provided the

code for the data file generator – it produces a large number of random numbers (which, in theory, are supposed to be evenly distributed through the range) but then takes the 9th decade of the numbers and sorts them. So, about 10 percent of the numbers get inserted in one long chain, with some of the remaining 10 percent of the numbers forming small branches off the chain.

Once the two trees are built, finds are performed on the trees. You will notice in main_B.c that the last 10 numbers read in are saved and used for searches. The code searches for these 10 numbers a million times – a total of 10 million searches. Both the BST and AVL tests completed in a few seconds.

(You may be interested to know that I had a chance to try the code on an old IBM PC-AT with an 80286 processor. The test took almost 2 hours. The computer is not reliable, and I was not sure if it would crash before it completed the test.)

Remember that the generic binary tree code is in **bintree.c**, the AVL-specific code is in **avl.c** and the BST-specific code is in **bst.c**.

4.2 Files you will be submitting

You will be submitting three (3) files: LAB4_AVL.c, LAB4_BST.c and LAB4_MAINB.c.

The code you need to write for these files is described in more detail in the following sections.

4.3 Deliverable 1 – Add Missing Routines to AVL Code

You will notice that the following functions in AVL.C require operational code. Add the code. Test your code using MAIN.C.

```
Node* rotateRight(Node* root)
// Rotate to right.  Returns new root pointer.

Node* rotateLeft(Node* root)
// Rotate to left.  Returns new root pointer.

int getBalanceFactor(Node* root)
// Get balance factor - difference between left height and right height

int calcHeight(Node* root)
// Calculate height of this node by adding 1 to maximum of left, right
// child height.

Node* rebalance(Node* root)
// Check balance factor to see if balancing required (bf > 1 or bf < -1).
// If balancing required, perform necessary rotations.
```

MAIN.C reads data from one of two sources. If the program (called avl.exe if you use the supplied makefile) is run without parameters, it reads a default input file containing 10 numbers. The output from the program should be like this, except for the lines about unbalanced nodes and rotations:

```
Inserting 10
Inorder: 10(h=0,bf=0) -
-------
Inserting 3
Inorder: 3(h=0,bf=0) - 10(h=1,bf=1) -
-------
Inserting 1
Node 10 is unbalanced. Left of Left: Rotate Right
Inorder: 1(h=0,bf=0) - 3(h=1,bf=0) - 10(h=0,bf=0) -
-------
Inserting 7
Inorder: 1(h=0,bf=0) - 3(h=2,bf=-1) - 7(h=0,bf=0) - 10(h=1,bf=1) -
-------
Inserting 20
Inorder: 1(h=0,bf=0) - 3(h=2,bf=-1) - 7(h=0,bf=0) - 10(h=1,bf=0) - 20(h=0,bf=0) -
-------
Inserting 15
Node 3 is unbalanced. Right of Right: Rotate Left
Inorder: 1(h=0,bf=0) - 3(h=1,bf=0) - 7(h=0,bf=0) - 10(h=2,bf=0) - 15(h=0,bf=0) - 20(h=1,bf=1) -
-------
Inserting 18
Node 20 is unbalanced. Right of Left: Rotate Left
Rotate Right
Inorder: 1(h=0,bf=0) - 3(h=1,bf=0) - 7(h=0,bf=0) - 10(h=2,bf=0) - 15(h=0,bf=0) - 18(h=1,bf=0) - 20(h=0,bf=0) -
-------
Inserting 17
Inorder: 1(h=0,bf=0) - 3(h=1,bf=0) - 7(h=0,bf=0) - 10(h=3,bf=-1) - 15(h=1,bf=-1) - 17(h=0,bf=0) - 18(h=2,bf=1) -
20(h=0,bf=0) -
-------
Inserting 16
Node 15 is unbalanced. Left of Right: Rotate Right
Rotate Left
Inorder: 1(h=0,bf=0) - 3(h=1,bf=0) - 7(h=0,bf=0) - 10(h=3,bf=-1) - 15(h=0,bf=0) - 16(h=1,bf=0) - 17(h=0,bf=0) -
18(h=2,bf=1) - 20(h=0,bf=0) -
-------
Inserting 22
Inorder: 1(h=0,bf=0) - 3(h=1,bf=0) - 7(h=0,bf=0) - 10(h=3,bf=-1) - 15(h=0,bf=0) - 16(h=1,bf=0) - 17(h=0,bf=0) -
18(h=2,bf=0) - 20(h=1,bf=-1) - 22(h=0,bf=0) -
-------
```

4.4 Deliverable 1B - Add delete functionality to AVL.

The functions required to implement deleting a node from an AVL are not provided. You will need to add operational code for findParentHelper() and delete(). You will have to build an interface for testing; the markers will use a different version of main.c to test your code.

Detailed specifications:

- The file name that you provide shall be LAB4_AVL.C
- The program takes input from a file - either the default if no file name is provided on the command line – the values to insert are found in the default input file. When testing your program, the values used by the markers may be different.
- The output is determined by the existing code – ***PLEASE DO NOT ALTER THE OUTPUT FORMAT***.
- The first line of your program shall conform to the course standard for headlines, that is, it should be
  // LAB4_AVL.c – Lab 04 – FirstName, LastName

4.5 Deliverable 2 – Compare AVL and BST Performance

As noted above in 4.1 Overview, the second part of the lab involves comparing the performance of AVL and BST on identical data. The module containing main() for this part is called main_B.c.

Review this file (main_B.c) and see how it stores data and how it runs a test. Modify it so that it stores the data in a BST as well as the AVL Tree.

Further modify main_B.c so that it runs a comparable test using the BST as is run on the AVL.

As noted in 4.1 Overview, the BST code carried over from lab 3 does not have the same interface for insert as the AVL code. The BST code only takes a pointer to a node (making it easier to handle the recursion) and the AVL code takes a pointer to a tree – and deals with the recursion by having a second recursive routine. Write code in BST.C to make the BST interface match the AVL interface. (When looking at the AVL version, can you see a reason why getting a pointer to the tree is preferable to getting a pointer to a node?)

Detailed specifications:

- The two source file names shall be LAB4_BST.c and LAB4_MAINB.c.
- The input mechanism and command line interface shall be unchanged from the supplied code.
- The first line of your files shall conform to the course standard for headlines, that is, it should be
  // LAB4_XXXX.c – Lab 04 – FirstName, LastName
  where XXXX is either BST or MAINB