CS 216

Laboratory 4:

Trees

Objectives:

To understand the workings of binary trees as well as tree traversals in the context of a useful application. To evaluate the performance of binary search trees and AVL trees.

Background:

A binary tree is a tree with a maximum of two children. Three traversals are commonly associated with binary trees. A preorder traversal processes the given node first and then processes its left and then right subtrees. In an inorder traversal, first the node's left subtree is processed, followed by the node itself, and finally its right subtree. A postorder traversal operates on the left subtree, followed by the right subtree, and finally on the node itself.

Preparation / Readings:

- Read pages 121—155 and page 170 in Weiss' *Data Structures* textbook.
- Review handouts on AVL trees.
- **Note the pre-lab on page 3:** come to lab with working code and with the worksheet completed.

Pre-Lab Code Description

For this lab you will be modifying your glorious stack calculator application from lab #2. Here are the modifications:

Your stack calculator should read in expressions in postfix notation (you can assume that these will be well-formed expressions as we did in lab #2). Rather than merely pushing the operands onto the stack and evaluating the expression as it is read in, you will instead implement the algorithm described in section 4.2.2 on p. 128 in Weiss to convert a postfix expression to an expression tree. Trees similar to this are used extensively in compilers. You may use any binary tree class you wish. You may use the one from the book or write your own. Please comment and give credit as necessary.

Once you have created an expression tree, you should:

- Print the resulting expression tree as a postfix expression.

- Print the resulting expression tree as an infix expression, complete with parentheses.

- Print the resulting expression tree as a prefix expression.

- Calculate the result of your expression and print that to the screen.

Obviously you should have a test harness, as you did in lab #2, to demonstrate your new and improved stack calculator application.

Useful Information

Postfix notation (also known as reverse Polish notation) involves writing the operators after the operands. Note how parentheses are unnecessary in postfix notation.

```
Infix:      ( ( 34 + 6 ) - ( -8 / 4 ) )
Postfix:    34 6 + -8 4 / -
```

A simple postfix stack calculator for integers is described on pages 103-105 in Weiss.

To end standard input on Windows, type CTRL+Z, ENTER.

Procedure

Pre-lab

1. Come to lab with a fully functional modified stack calculator as described on the previous page. Note that check-offs for the stack calculator will only be allowed during the first 15 minutes of class (i.e. you will not have time to modify it during lab, so be sure to test it thoroughly before coming to class).
2. Complete the AVL tree worksheet included in this lab packet before coming to lab.
3. Review the binary search tree and AVL tree code from Weiss. You will be using it during lab.

In-lab

One check-off sheet per person. The grad TAs will randomly assign partners for this lab.

1. Introduce yourself to your partner and find two computers to sit down at. Exchange email addresses in case you need to discuss anything after lab.
2. Each partner should demonstrate their modified stack calculator to the TAs *during the first 15 minutes of class*.
3. You and your partner should modify the code available on the cs216 web site as described in the In-lab description on the following pages. *Both partners should SAVE a copy of the modified code. You will need this to complete the postlab.*
4. Devise a reasonably convincing experiment to show that AVL trees are markedly superior to randomly grown trees. We have provided **`testfile1.txt`**, **`testfile2.txt`** and **`testfile3.txt`**, you may create new files if desired.
5. Turn-in the AVL worksheet and your check-off sheet *before* leaving lab.

Post-lab

1. For this lab you will be submitting your code and a brief lab report electronically via the toolkit. Your fully functional code is **due at 11:59 pm Thursday October 13, 2005**. *Your code will be graded for correctness AND style*. Be sure to include: your name, the date, and the name of the file in a banner comment at the beginning of each file you submit. Please turn in:
   - ➔ A listing of all code required to run your **modified stack calculator pre-lab**. If you wrote your own code for trees or stacks, then please turn that in as well. (If you used the code from the book do not turn it in, but be sure to give credit as necessary)
   - ➔ A listing of any **BST or AVL tree code** you modified during the in-lab activity.
   - ➔ A text file describing an analysis of what you learned during the In-Lab activity. Be sure to include: a) actual numerical results for some operations on both AVL trees and BSTs for the three provided test files, b) a characterization of situations where AVL trees are preferable to BSTs, c) a discussion of the costs incurred in an AVL implementation. Please include this as a text, pdf, or postscript file (MS Word is difficult to print out).

*If you did not get elements of the in-lab test sequence checked off during lab, you may attempt to receive partial credit for these by successfully implementing those elements after lab and submitting your corrected code at turn-in. In order to receive these points you must submit in addition to your code, a paragraph (**text file only** please) describing what difficulties you encountered getting your code working and what you did to solve then.*

CS 216

Laboratory 4:

Performance of Binary Search Trees and AVL Trees

In-Lab Activity

Objectives:

To begin developing an empirical approach to the analysis of data structure performance. In this lab you will compare the performance of BSTs and AVL trees on several data files.

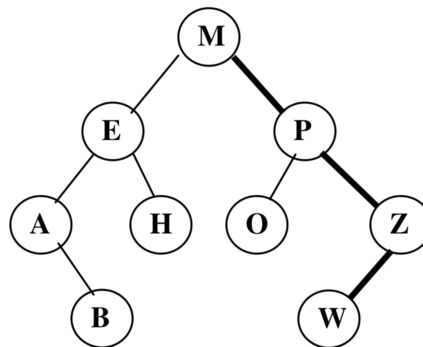
Background:

We have discussed both Binary Search Trees and AVL Trees in class. AVL trees are a form of balanced search tree. For a little extra computational cost, AVL trees ensure that the heights of any two subtrees of a parent node are within 1 in value; so retrieval performance is O(log n) in the worst case.

Code Description

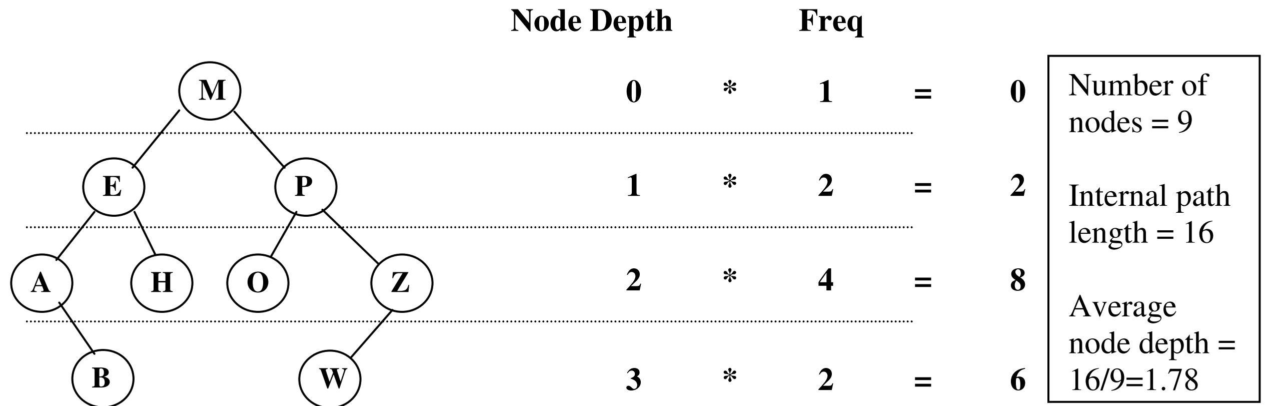
Since a goal of the lab is to compare binary and AVL trees, we need to establish common metrics on which to compare the trees created by each algorithm. One metric for comparing the data structures is retrieval time. For both binary and AVL trees, the retrieval time of a value is dependent on the depth of the node storing the value. For example, it is much quicker to find a value stored in the root node than to find a value stored in a leaf.

We will use two measures of retrieval time. The first measurement we will use is the number of links followed. For example, finding element **W** in the tree below requires following 3 links- 2 right links and 1 left link. Note that we will keep track of all left and right links traversed *for the life of the tree*.



The bold lines indicate the path taken for locating element w. This path contains 3 links.

While measuring path length provides a good indication of the retrieval time of individual elements, we are also interested in the average retrieval time for the elements within the tree. Therefore the other measurement of interest is the expected path length to a given node from the root. The expected path length to a given node is equivalent to the number of links between a node and the root node, or in other words, the depth of the node. The expected path length is the same as what Weiss defines on p. 140 as being the average internal path length. The internal path length is defined as the sum of the depths of *all* nodes in a tree. The tree in the example below has internal path length of 16, and average internal path length (expected path length or average node depth) is 1.78. You will need to modify AvlTree.h, BinarySearchTree.h and tree_test.cpp to determine and display the expected path length for each type of tree.



<u>What you and your partner need to do:</u> We have already declared the necessary private data members and public inspector functions in BinarySearchTree.h and AvlTree.h to record this information. Your job is to modify the relevant routines to record the information correctly.

The TA will quiz you and your partner on your code modifications.

Once your code is working correctly, devise a reasonably convincing experiment to show that AVL trees are markedly superior to randomly grown BST trees. What does it cost to achieve such better performance? Can you characterize the situations where AVL trees perform better than BSTs? We have provided **`testfile1.txt, testfile2.txt`** and **`testfile3.txt`**, which in addition to your own test cases may be of value.

For your postlab report you will need to provide evidence comparing the two types of trees. Each partner needs to turn in their own copy of this report, but you may want to get started on this during lab so you can be sure that your code is doing its calculations correctly.

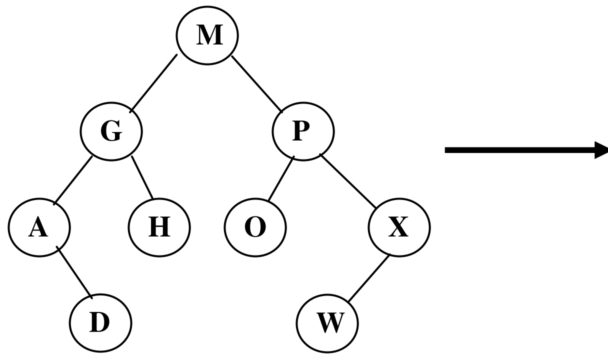
UVa ID (no aliases): ___________________

Name:_________________ Section Number:_______

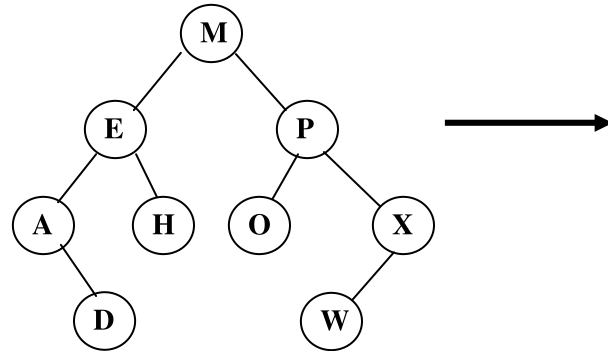
AVL Tree Worksheet

Directions: Insert the following elements into the shown trees. Draw the resulting tree.

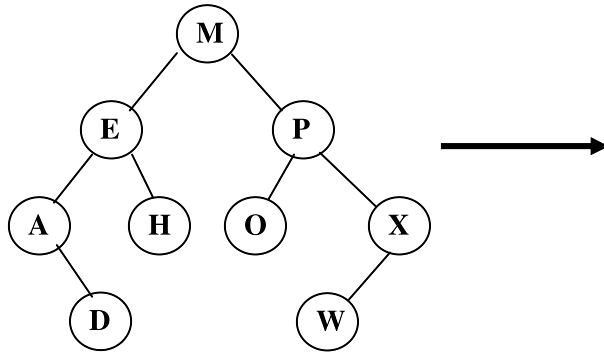
A.) Insert E **Tree after insertion:**



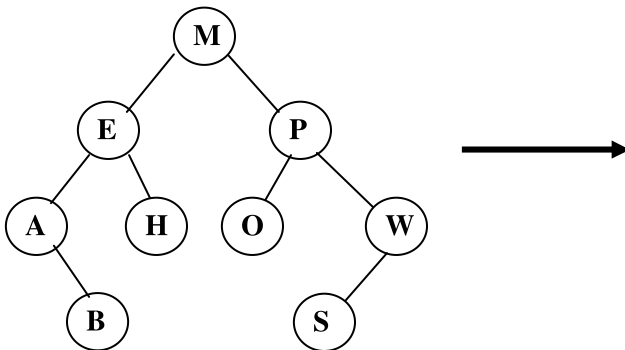
B.) Insert F **Tree after insertion:**



C.) Insert C **Tree after insertion:**



D.) Insert Q **Tree after insertion:**



CS 216: Program and Data Representation

Lab #4: Trees - Check-off Sheet

(One check-off sheet per person)

Date:	**Circle:** **Your** **1 Tue 7:30 – 9:15 PM** **Section:** **2 Tue 12:15 – 2:00 PM**
My Email Address: **(UVa ID please, no aliases)**	**My Name:**
Partner's Email:	**My Partner's Name:**

Modified Stack Calculator Demonstrated	
AVL Tree Worksheet Completed	
Explanation of AVL tree and BST code modifications	
Correct calculations in modifications of AVL tree and BST	

Pledge: