Lab 5: Trees last modified by Aaron Bloomfield on October 9, 2013 2:09:48 PM EDT

Laboratory 5: Trees

Week of Monday, October 7, 2013

Objective:

1. Learn about binary trees
2. See tree traversals in the context of a useful application
3. Evaluate the performance of binary search trees and AVL trees

Background:

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

Reading(s):

Read Chapter 4, Sections 4.1 – 4.4.2 and Section 4.6 in Weiss' Data Structures textbook. Be sure you understand the part on AVL trees! Weiss section 4.2.2 is available online as well.

Procedure

Pre-lab

1. Complete the makefile tutorial (on Collab) prior to coming to lab. While you will not have to write a makefile for this pre-lab, you will need to know how to write on in the future – all the following labs will be compiled via Makefiles. There is one file that needs to be submitted from the tutorial – you should name this file Makefile-pizza. You can compile your code for the pre-lab using 'clang++ *.cpp'.
2. Your file **MUST** be named Makefile-pizza - not Makefile-pizza.txt, not Makefile-Pizza, not Makefilepizza. If it is not named correctly, it will break our grading scripts, and you will not get credit for that part.
3. Come to lab with a fully functional tree calculator, as described below.
4. Read Chapter 4, Sections 4.1 – 4.4.2 and 4.6 in Weiss' Data Structures textbook, or the alternate readings on the Readings page. Be sure you understand the part on AVL trees!
5. Review the binary search tree and AVL tree code from Weiss – it will be used during lab.
6. Complete the AVL tree worksheet, which is a separate file. The TAs will collect it at the beginning of lab. **You need to turn in that sheet** (you can't write your answer on notebook paper, for example) and bubble in your ID at the bottom of both sides of the page. Feel free to print it single sided or double sided. We will provide copies of this in lecture for you as well.
7. Files to download: TreeCalc.h/cpp, TreeCalcTest.cpp, TreeNode.h/cpp. These files are contained in the prelab/ directory of the code.zip file.
8. Files to submit: TreeCalc.h/cpp, TreeCalcTest.cpp, TreeNode.h/cpp, Makefile-pizza

In-lab

1. Turn in the AVL worksheet at the *beginning* of lab.
2. You should modify the code available on the CS 2150 Collab site as described in the in-lab description on the following pages (avltree.cpp and binarytree.cpp).
3. 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 for you to experiment with. Your new experiment should be in a testfile4.txt file, which you will need to submit.
4. Examine the Makefile for this project. You should understand everything in the Makefile! The grading compile command for the inlab will be performed using 'make'.
5. Files to download: avlnode.h, binarynode.h, avltree.h/cpp, binarysearchtree.h/cpp, tree_test.cpp, testfile[1-3].txt. These files are contained in the inlab/ directory of the code.zip file.

6. Files to submit: avlnode.h, binarynode.h, avltree.h/cpp, binarysearchtree.h/cpp, tree_test.cpp, testfile4.txt, Makefile

Post-lab

1. For this lab you will be a brief lab report electronically via the submission system.
2. Your report must be in PDF format! See the How to convert a file to PDF page for details.
3. Files to download: (none, beyond what was done during the in-lab)
4. Files to submit: analysis.pdf (see the post-lab section for formatting details)

Pre-lab

For this lab you will be using a stack to help you read in a postfix expression into an expression tree. While this is similar to lab 3, you will instead be ultimately creating a expression tree for the postfix expression, rather than leaving it on the stack. You should use the STL stack class for this lab.

Your tree calculator should read in expressions in postfix notation – you can assume that these will be well-formed expressions as we did in lab 3. You will need to build an expression tree using the algorithm described in section 4.2.2 on p. 121 in Weiss. Trees similar to this are used extensively in compilers.

Code base

You must use the skeleton source files on the Collab site as a basis for your prelab. The following guidelines apply to your implementation:

- Do NOT alter TreeCalcTest.cpp. This is the testing program that we will use to run automated tests on your implementations. Do not change it.
- In TreeCalc.h/TreeCalc.cpp:
    - Do NOT alter the readInput() method. Points will be deducted if you do so.
    - The only modification allowed in the printOutput() method is to add calls to your implemented printPrefix(), printPostfix(), and printInorder() methods
- You should implement all the methods as listed in the class definitions for TreeCalc
- You may add additional supporting methods and data members to TreeCalc to complete your implementation.
    - Don't modify TreeNode – note that TreeCalc is a friend of TreeNode, so you can put all your code in TreeCalc.

Note that the code will not compile out of the box – you need to add code so that the printOutput() method in TreeCalc works.

You should only submit the following five files:

- TreeCalc.cpp and TreeCalc.h
- The unmodified TreeNode.cpp and TreeNode.h
- The unmodified TreeCalcTest.cpp

Note that the last three files should not be modified at all! If you have additional code to include, put it in TreeCalc.cpp or TreeCalc.h (you should not need to use additional classes). Just in case it wasn't clear yet, all your modifications should be in the TreeCalc.h and TreeCalc.cpp files.

Your fully functional tree calculator code should do the following:

1. Read user input in postfix order (assume well-formed expressions) into a stack
2. Build an expression tree using the items in the stack (see Weiss section 4.2.2 on p. 121)
3. Print the resulting expression tree as a *postfix* expression in the EXACT output format as shown in the next section. Spaces matter!
4. Print the resulting expression tree as an *infix* expression, complete with parentheses. See the next section. Your print method must print EXACTLY in this format, including the number of parentheses. Spaces matter!
5. Print the resulting expression tree as a *prefix* expression. (See next section. Your print method must print EXACTLY this format.) Spaces matter!
6. Calculate the result of your expression using the expression tree
7. Print the result to the screen.

Make sure your destructor works! We will be testing that to see if there are any memory leaks.

A few notes:

- We are not dealing with the negation operator (~) that we used in lab 3
- Thus, your code will need to be able to handle the input of negative numbers, as shown in the example below

- Your stack in TreeCalc.h/cpp should be called mystack (or else you will have to change the name in printOutput() – this is the one change you can make to this method)

Print Output Format

Postfix notation (also known as reverse Polish notation) involves writing the operators after the operands. Note how parentheses are unnecessary in prefix or postfix notation. Your print methods must print in the following format. The only spaces are on either side of the operators. Your entire expression must be printed on a single line that terminates with a newline character. You should put parenthesis around each infix operation, regardless if it is needed according to operator precedence.

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

The simple postfix stack calculator for integers you implemented in the last lab is described on pages 97-99 in Weiss.

Sample Execution Run

Below is a sample execution run to show you the input and output format we are looking for.

```
Enter elements one by one in postfix notation
Any non-numeric or non-operator character, e.g. #, will terminate input
Enter first element: 34
Enter next element: 6
Enter next element: +
Enter next element: -8
Enter next element: 4
Enter next element: /
Enter next element: -
Enter next element: #
Expression tree in postfix expression: 34 6 + -8 4 / -
Expression tree in infix expression: ((34 + 6) - (-8 / 4))
Expression tree in prefix expression: - + 34 6 / -8 4
The result of the expression tree is 42
```

In-lab

The objective of the in-lab is 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.

We have discussed both binary search trees and AVL trees in class. AVL trees are a form of a 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 big-theta(log n) in the worst case.

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 search 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.

Summary

You will need to download the following files from Collab: avlnode.h, binarynode.h, avltree.h/cpp, binarysearchtree.h/cpp, tree_test.cpp, testfile[1-3].txt. To complete this lab you will have to make the following code modifications:

1. Edit the AVL and binary search tree code to adjust the counter variables appropriately. These variables are: num_nodes, LeftLinksFollowed, RightLinksFollowed, SingleRotations, and DoubleRotations
    - **NOTE:** The mutable keyword before these variable declarations allows the member functions that are declared as const to modify them (the variables). Basically, it overrides the const modifier.
    - You **should not** reset the LeftLinksFollowed or RightLinksFollowed variables.
    - Those two variables (LeftLinksFollowed and RightLinksFollowed) should only be incremented in the find method, not for the insert or delete method.
    - A double rotation is really two single rotations, but for the purposes of this in-lab we'll only count it as one double

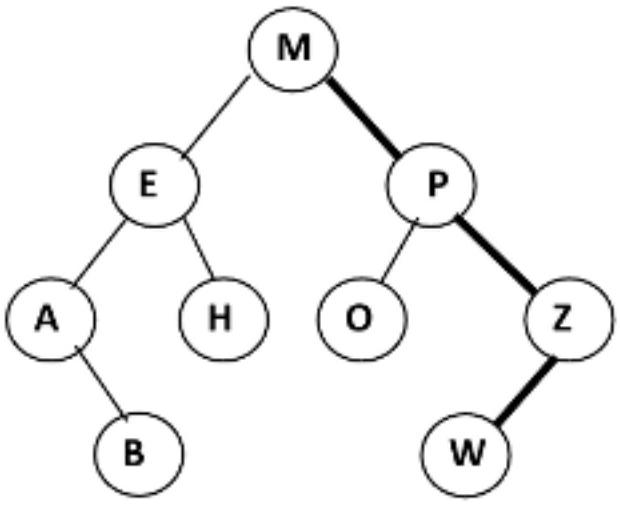
rotation.

2. Implement the following methods in the AVL and binary search tree code. See below for details as to how these methods work.
    - double AvlTree::exp_path_length();
    - double BinarySearchTree::exp_path_length();
    - int AvlTree::int_path_length(AvlNode *t, int depth);
    - int BinarySearchTree::int_path_length(BinaryNode *t, int depth);

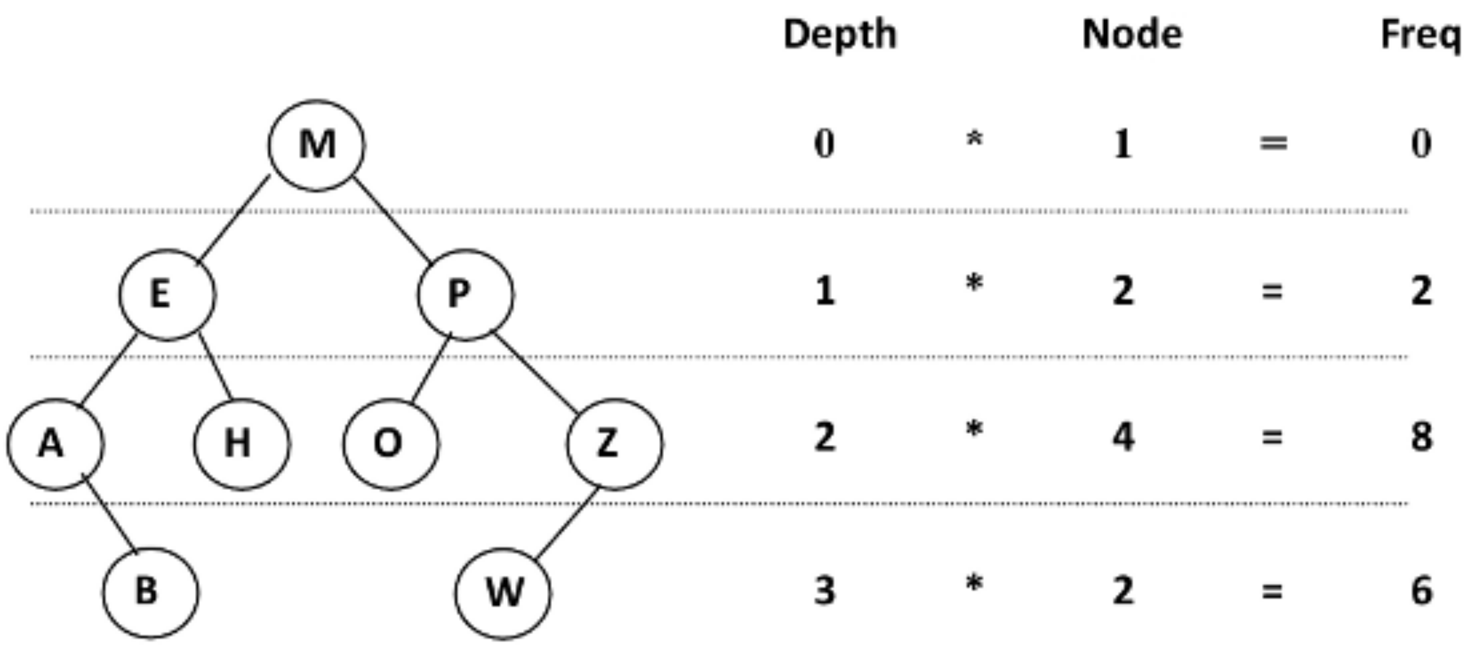
Explanation

We will use two measures of retrieval time: expected path length and internal path length. The base counting unit 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***.

In the following image, 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 or expected path length for the elements within the tree. To calculate this, we must first calculate the 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 expected path length is 1.78. The expected path length to a given node is equivalent to the average number of links between any node and the root node, or in other words, the internal path length divided by the total number of nodes. The expected path length is the same as what Weiss defines on p. 133 as being the average internal path length (different from internal path length). You will need to modify avltree.cpp and binarysearchtree.cpp to determine and display the expected path length for each type of tree.



Depth		Node		Freq
0	*	1	=	0
1	*	2	=	2
2	*	4	=	8
3	*	2	=	6

- Number of nodes = 9
- Internal path length = 16
- Average node depth = 16/9 = 1.78

The int_path_length() methods take in two parameters – a node (either the AvlNode or BinaryNode, depending on the version) and a depth. Recall that the internal path length is the sum of the depths for each and every node in the tree. The depth is how far down the tree this method is being called. If it is called on the root node, then the depth would be zero (and the function would return the total internal path length of the entire tree). The root node will call it recursively (with depth=1) on it's children. If it is called on a leaf node, then the depth of that particular node is just the value of the depth parameter. If it is called on an internal node, then it is going to call itself on each of the children (in each case yielding the internal path length for the child sub-trees), plus the depth of that particular internal node.

Input format

The input will be provided all on one line. The input in each file has just one line for the paragraph. The provided input files can be run through the program as follows:

```
./a.out < testfile1.txt
```

Note that we are providing the file name as a command-line parameter. The provided code handles this for you already.

What you need to do

We have already declared the necessary private data members and public inspector functions in binarysearchtree.h/cpp and avltree.h/cpp to hold this information. Your job is to modify the relevant routines to record the information correctly.

Consult with the TAs if you have any questions.

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 the files testfile1.txt, testfile2.txt and testfile3.txt, which in addition to your own test cases, may be of value.

Your experiment should be in a testfile4.txt file, which you will have to submit. How and why does your example (that you include in testfile4.txt) show that AVL trees are better than BST trees? A single paragraph is fine, and this should be put into the post-lab report. The point of the testfile4.txt is for you to generate an experiment that shows a large difference in performance between AVL trees and BST trees. Just generating a long piece of text, with no reasoning behind it, is not what we are looking for.

You will need to submit all of your code, along with the testfile4.txt file. Please make sure it compiles!

For your postlab report you will need to provide evidence comparing the two types of trees. You may want to get started on this during lab so you can be sure that your code is doing its calculations correctly.

If you cannot finish this during the in-lab, you can request a 24 hour lab extension.

Post-lab

Note that neither the pre-lab nor the in-lab code is being resubmitted for the post-lab.

The deliverable for this post-lab is a PDF document, the contents of which are described below. It must be submitted in PDF format! See How To Convert A File To PDF for details.

In the report, you should include:

1. Your name, the date, and your CS 2150 lab section.
2. The testfile4.txt file you used in the in-lab. You can just cut-and-paste this into the Word document.
3. Why the testfile4.txt file works (this is the single paragraph that was mentioned in the 'What you need to do' section of the in-lab)
4. Actual numerical results for some operations on both AVL trees and BSTs for the three provided test files and any additional tests you created.
5. A characterization of situations where AVL trees are preferable to BSTs.
6. A discussion of the costs incurred in an AVL implementation.

For parts 5-6, a FULL page (if double spaced) is the minimum length we are looking for (that's a full page for parts 5 & 6 combined). Feel free to single space – the same length (in terms of words, not in terms of page length) is still required.

Be the first to comment

Untitled note

Find a notebook

Add tag

Add comments

Version 6.0.6: c128369/1.0.1.250

Web Clipper tutorial

Options

To:

Saving clip...

Share

Simplified Article

Save

Selection

PDF