IE 170 Laboratory 5: Binary Search Trees

Dr. T.K. Ralphs

Due Feb 27, 2006

1 Laboratory Description and Procedures

1.1 Learning Objectives

You should be able to do the following after completing this laboratory.

1. Understand binary search trees (BSTs) and the BST property.
2. Understand how to implement a binary search tree with a linked list.
3. Understand the importance of the depth of the tree on performance.
4. Understand how to analyze the performance of a BST.

1.2 Key Words

You should be able to define the following key words after completing this laboratory.

1. Symbol table
2. Dictionary
3. Binary search tree
4. Tree depth
5. Balanced tree

1.3 Scenario

You work for a large credit card company affiliated with hundreds of different banks that offer customized versions of its card. When a customer wants to use a card to make a purchase, the card number is entered into a small computer called a point of sale (POS) terminal. This computer in turn queries a database of card numbers to determine whether the sale should be approved or not. One of the main functions the credit card company performs for its bank affiliates is to maintain the POS terminals and service the queries as they arrive. This involves maintaining records for literally hundreds of thousands, possibly millions of credit cards. Each time there is a query from a POS terminal, the database must be searched for the corresponding record to determine if the sale should be approved or not. There are literally thousand of queries a minute and they must be answered very quickly. Furthermore, the database is very dynamic. New card numbers are added and old card numbers deleted minute by minute. Recently, the database has been failing to respond to queries at peak times and sales have been lost. Your job is to implement a new one capable of much better performance.

1.4 Design and Analysis

In this lab, you will insert items into a binary search tree with a linked list as the underlying data structure. As we saw in Lecture 9, one of the main indicators of performance for binary search trees is their depth. The depth of a search tree limits the worst-case performance of the insert and search operations, so keeping the tree as shallow as possible results in better performance. Numerous authors have proposed sophisticated methods for automatically keeping binary search trees balanced. Examples are random BSTs, splay trees, AVL trees, red-black trees, top-down 2-3-4 trees, etc.

The depth of the tree doesn't tell the full story, however. What we may be more interested in is how long it takes to perform each operation "on average." Let's take the search operation as an example. Assume that we are searching for a randomly selected item that is in the tree. In this case, the *average* time to execute the search will be the "average depth" of a node in the tree, which is simply the sum of the depths of the nodes divided by the total number of nodes. This quantity can easily be calculated empirically (see the analysis questions below). It can also be determined theoretically that the average depth of a binary search tree built by inserting n randomly generated elements is approximately $2 \ln n$ (see CLRS 12-3). We do not, however, know much about what the depth of a BST will be after a random sequence of operations including both insertion and deletion. In fact, it is not clear how to even define such a sequence and hence, analyzing this behavior is more complex. However, we will perform an empirical analysis as part of this lab.

Above, we have discussed two different measures that tell us something about performance in a randomly generated BST. Even these measures do not give us the full story. For one thing, the lists of elements to be inserted into a binary search tree are not always random. Patterns in the data can lead to poor performance. If the list of elements to be inserted into the BST is already sorted or almost sorted, this can create a tree that is badly out of balance. In practice, some search paths are traversed more than others. For example, if the tree is balanced except for a single long chain of nodes and those nodes happen to be ones that are never touched after being inserted, performance will not be adversely affected. Obviously, this is difficult to predict. It is possible to dynamically adjust the tree through a technique called *splaying* such that the nodes on paths that are "well traveled" in the tree will be more likely to be near the root. This technique works well in situations where the searches are correlated with each other, i.e., the same search paths tend to be traveled over and over.

Another interesting measure is called the *imbalance*. For any given node in the tree, we will define the absolute difference between the number of nodes in the right and left subtrees to be the *imbalance* of that node. For a balanced tree, all nodes must have an imbalance of either 0 or 1. The *total imbalance* of the tree is then the sum of all imbalances of its nodes. Although knowing the imbalance of the nodes in the tree does not allow us to make a statement about the worst-case running time of operations, it may give us a better feel for the shape of the tree and what overall performance may look like.

The goal of this lab is to study the balance of randomly generated binomial search trees as a measure of performance of the data structure. To study these measures in a dynamic environment, you should implement a basic binary search tree data structure supporting the interface given in the header file `binarySearchTree.hpp`. Using this class, you will implement a client that generates BSTs through a random sequence of insertions and possibly deletions. You will then report on the average depth of the trees generated using the statistics above (see the analysis section below).

1.5 Program Specifications

1.5.1 Program(s) Function

You should implement a BST class that works with the provided client program in order to answer the questions in the analysis section. The client will generate random sequences of items to be inserted and/or deleted and report the statistics required. This will be done repeatedly and results reported. In this simplified case, the "items" will simply be integers.

1.5.2 Algorithms

The algorithms to be implemented in this lab are those supporting the operations on a binary search tree specified by the interface in `binarySearchTree.hpp`.

1.5.3 Data Structures

The basic data structure required for this laboratory is a binary search tree.

2 Laboratory Test Files

The files for this laboratory are in the zip archive `Lab5.zip` available on the course Web site. The archive will unpack into a directory called `Lab5` with `shell` and `data` subdirectories. The `shell` directory contains the file `binarySearchTree.hpp` that contains the interface to be supported for this lab. In the `data` subdirectory, there is a test data file called `input100.txt` that you may use for testing the ability to build a binary tree from a file. A drive program has also been provided for you.

3 Laboratory Assignments

3.1 Programming (50 points)

1. Implement a binary search tree class supporting the given interface.
2. Write a client program that uses your binary search tree class and behaves as specified above.
3. Verify that your program compiles and runs correctly using the test files provided for the lab.

3.2 Analysis (25 points)

1. (10 points) Randomly generate and insert 100 integers between 1 and 100000 into a BST and calculate both the depth and average depth of the final tree. Repeat this procedure 100 times and show the distribution of values in a histogram.
2. (5 points) Compare the "average" average depth to the theoretical value of approximately $2 \ln 100$. Assuming the distribution of values in the histogram to be normal, perform a hypothesis test to determine whether the two values are equal. Assess the results.
3. (10 points) Generate a BST as in part (a) and then search for 100 randomly generated integers between 1 and 100000. Compare the average number of comparisons required to do the searches comapare to the average depth of the tree. Assess the results of this comparison.

3.3 Follow-up Questions (25 points)

1. (5 points) 12.1-1
2. (5 points) CLRS 12.2-1 (explain your answer)
3. (5 points) 12.3-3
4. (10 points) CLRS 12-1