Empirical Study of Binary Search Trees

Jake Hennessy

College of Charleston  
96 George St.  
Charleston, SC 29401  
+1 (843) XXX - XXXX

jbhennes@g.cofc.edu

**ABSTRACT**

In this paper the author will discuss binary search trees and the various methods that can be used to remove nodes within the tree, specifically comparing the CPU system click time and an arbitrary variable ‘count’ that was assigned to compute the operations the CPU had to perform. The basic study used two sets of data composed of ten trials each; one set had twenty nodes to the tree, and the second tree used 1,000. Noting the various differences and their consistency in trends made the author unsure as to whether there was human error creating bias in the tests, or if the actual system itself was creating bias through library-defined routines.

# INTRODUCTION

The subject of this paper concerns the nature of binary search trees and the nature of searching through the trees to remove certain nodes as specified by the user. The idea of the whole experiment was to test various methods of traversing the tree to find the node in question and testing the efficiency of the algorithm in various formats.

The nature of an algorithm is provide a series of steps/instructions that can be used to solve a problem, which in this case involved taking a binary search tree data structure that contained various values in order from greatest (right most node in a graph representation of the data) to least (which would be the left most node in a graphical representation of the tree.) Binary search trees utilize a strategy of creating a tree like structure with a certain value at the root, and values that are less than the root value in question are to the left of the root, and all values to the right are greater than the root. Through this style of structuring, the tree is always ordered, and the tree is always of an arbitrary size because it does not need to be specified.

The first test was to remove the node in question and replace it with the node in questions right child’s left node, the second test was to remove the node in question and replace it with the node in question’s left child’s right node, the third method involved removing nodes in the same manner as the first two mentioned, but each remove alternated the method between the first and second aforementioned methods to choosing which node to make the new root node, and finally the last method used would randomly calculate which node to remove, using the first two methods mentioned.

The binary search tree is an abstract data construct—meaning that it is a data structure that does not inherently consist of arrays which are considered primitive types in the java language, but instead is made through various means of interfacing, using java comparables, and finally implementing classes that use the pointer method of referencing a variable whose values will be used, without actually copying the variables values to the address spot. The nature of a binary tress is simply a collection of nodes that contain certain data values, and within the class where the nodes are themselves defined, a class of methods there which will implement, manipulate, and alter the behavior and “appearance” in an abstract sense of the tree itself.

The essential setup of a binary tree is simple: it consists of a certain amount of instantiations of the binary node objects that act as an inner class within the main binary search tree class. In this experiment the test code created an array of integers that were randomly selected from 1 through 100, and then these integer values were fed into the binary search tree utilizing the insert method.

By using the test program to create randomly generated trees using the same size, and implementing the various remove methods we could judge the time it took to execute the various methods to remove all of the elements from tree.

# METHODOLOGY

The methods used to test the effienciency of the various algorithms in question involved using code that was provided to the students participating in the students by the author of the course’s textbook, Mark Allen Weiss. The code contained all of the methods that were to be used in the study, including inser, remove, removeL, removeAlternating, removeRandomly, findMin, findMax, and the students were in charge of creating some method(s) that could calculate the time it took to complete the entire removal of the tree that was created at the beginning of the test.

To populate the tree, the initial structure used was an array that was limited to twenty elements, which were instantiated and then populated with values that were to be randomly generated. The randomly generated values ranged from 0 – 100, and were created through using the java round function alongside the random number generator, whose results were then multiplied by 100 to obtain a natural number result. The array would be created first, and then a for-loop would be used to generate the values and subsequently put them into the array, filling it entirely. After the array was populated, the code would then next create an instance of the binary search tree class that would use the java comparable data type Integer. Afterward would use another for-loop to insert each of the values of the array into the tree, and the tree would remain naturally ordered due to the nature of the basic class. In this setup all of the values were inserted, with lesser values becoming the left child of the parent node, and greater values becoming the right child of the parent node.

When the test is begun, the code will run and create the aforementioned objects, and perform the functions mentioned on them, so that we are presented with four identical trees that will be used for the test of the various removal algorithms. Next the trees will be printed, to ensure that they are indeed identical, and then we begin the first removal method that removes the node and replaces with the right child of the child of the node in question. After doing this, each additional removal method is tested, and the results are noted.

To determine how to place the elements in the tree, the values that are calculated are then shifted within the array itself, using the Fisher-Yates shuffle.[[1]](#footnote-1) After the randomization of the array itself, the values are inserted into the array via another for loop that runs until all of the array values have been placed into the array. The randomization of the order of array values allows for the minimum user bias in placing values within the tree, so that it will present the most realistic implementation of the data into the tree. In the end, what is created is a randomly generated tree in somewhat controlled sense, so that every time the test is run it is not only random values that are inserted into the tree, but also uses a randomized order each time for the removal process, so we dismantle the tree in a different order than we created it. Finally, all of the trees get the same order of values to remove from the tree, so that the algorithms used to remove nodes can be accurately judged against each other.

The method used for calculating the results consisted of a twofold approach: the first method used java’s own built in system clock (System.nanoTime()) that would act as a stopwatch by creating a variable that would contain the time at the instruction previous to the beginning of the invocation of the method loop, and another variable that would contain the time value at the time after the last instruction of the method loop. By taking the difference of these two times, the test could obtain a time that illustrated how long it took for the entire algorithm to run enough times to remove all of the nodes that were inserted into the tree. During the removal loop, each iteration of the removal instruction would be individually timed and accumulated so that we could have an average time per each removal and printing of the tree, and a final total that would show the entire time for the algorithm to finish.

The second method used for clocking the efficiency of the algorithms was to create a static variable called count, which would be used as an accumulator to denote the number of instructions used at the end of the entire removal of the tree. After each assignment, comparison, or outside method call, the count variable was incremented by 1, and then was printed out for each removal within the tree, clocking the total number of assignments, braches, and method calls for the removal of the entire tree.

To collect data, the experiment used a spreadsheet to collect the data from ten trials in which the program would be run and all of the trees would contribute their own data to the final printout with data

# RESULTS

## Initial Testing

The initial tests done were on smaller sample sizes of 20 elements per tree, the table below shows the averages after 10 trials in terms of the system’s clock time measured in nanoseconds and terms of the count variable that stores the number of instructions the CPU must perform.

**Table 1. Data collected from sample sizes of 20, expressed in average time for the removal of the entire set of nodes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method:** | Remove() | removeL() | removeAlt() | removeRan() |
| **Average Time** | 18209958.2  nanosec. | 15335395.5  ns | 12383051.3  ns | 11131765  ns |
| **Average**  **Count** | 147  instructions | 145.4 | 146.7 | 145.6 |

**Figure 1. Chart plotting the data points for total number of commands / average time elapsed**

## Secondary Testing

The second round of tests saw the same methods being used, but as opposed to the original sample size of 20, this round of tests used sample sizes of 1,000.

**Table 2. Data collected from sample sizes of 1000, expressed in average time for the removal of the entire set of nodes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method:** | Remove() | removeL() | removeAlt() | removeRan() |
| **Average Time** | 971006639.6  nanosec. | 811488201  ns | 765245960.7  ns | 729711372.5  ns |
| **Average**  **Count** | 4704.6 instructions | 4667.5 | 4695 | 4677 |

**Figure 2. Chart plotting the data points for total number of commands / average time elapsed**

# Discussion

The study was conducted 10 times for each dataset, and in both series of tests, the data was very close in terms of instruction count and CPU time for each respective set of tests, with the range of counts for each series of tests around only one percent. Upon closer inspection of the data, there were certain trends that were noted across several runs of the program.

Across both series of tests, the remove() method fared the worst in both clock times and average number of instructions, which was surprising given that the removeL() method is analogous to a mirror image of the remove routine, so any improvement upon the original code is not inherently apparent. In the first tests, the removeL() method fared much better than the orginal remove(), and following the same trend the removeL() method had the lowest number of instructions, and lowest amount of time spent on the instructions in the second round of testing.

The two methods removeAlt() and removeRand() were the interesting cases in this study. These particular methods often would come in as the fastest algorithm in terms of time on the system’s CPU clock, but in terms of instructions the removeAlt() algorithm was significantly higher because of the various condition checking that determined which child to replace with. As opposed to this, the removeRand() algorithm was better in processing speed and time, despite having to generate another variable to do its calculations. Occasionally, the removeAlt() algorithm would take a significantly longer period of time, and as a result its average is somewhat higher that what was often observed.

# CONCLUSIONS

Upon the tabulation of data, we could potentially set a trend curve that appeared proportionally similar in both graphs that illustrated the relative processing cost of each method when used to remove the entire tree’s data. When all of the information was compiled, there were a few lingering points that would influence further study.

The first of these was the idea that the randomization routine that the study used in the source code was something that was inherently built into the java library, and as result did not accurately reflect the instruction count, despite offering a true data point for CPU time in nanoseconds. Subsequent tests would hammer out any inconsistencies between the two values by looking at the utilities (java.Math.Random) routine and evaluate the actual influence on count as established by the guidelines in the methodologies section.

The most interesting question that was raised by the research was the huge discrepancy between the remove() and the removeL() routines in terms of both processing times and instruction count. In each of the series of tests done the removeL() and remove() routines were the respective minimum and maximum average times. The reason for the discrepancies isn’t exactly clear as to whether the system uses the CPU in a certain fashion as to distribute “randomly” generated integers in a biased fashion that would only become more and more apparent with larger and larger datasets.

# REFERENCES

1. Random shuffling of an array. stackoverflow.com/questions/1519736/random-shuffling-of-an-array

[2] Executing a time method in java. stackoverflow.com/questions/180158/how-do-i-time-a-methods-execution-in-java

1. The Fisher-Yates shuffle is an algorithm perfected by Fisher, Yates, and was based on the Knuth shuffle to generate a random permutation of a finite set. [↑](#footnote-ref-1)