Implementing A Binary Search Tree for an Address Book File

Assignment 1

Zach Nudelman

CSC2001F

UCT

Cape Town

NDLZAC001

# Problem Description

This report shows how the Binary Search Tree has a faster speed than a Linear Array List when searching for data entries using a key. It was required to write three applications:

1. PrintIt – which loaded the data into a Binary Search Tree and printed the data out in alphabetial order of name;
2. SearchIt – which searched for a full entry based on a full name in a Binary Search Tree; and,
3. SearchItLinear – which had the same funtionality as SearchIt but using a linear List Array to load the data. The speeds of SearchIt and SearchItLinear were recorded using the Unix ‘time’ application and the data was compared.

# Application Design

## Record Class

The class constructed a record object. A String was accpeted as a parameter. The constructor instantiated two attributes:

1. Name, and
2. Line.

Name was found by taking a substring of the line from the last ‘|’ character – as this is how the data entry is seperated.

## BinaryTreeNode Class

Thisclass constructed a binarytreenode object which had a Record and two BinaryTreeNodes as attributes. It’s methods are the typical accessor and mutator methods.

## BinarySearchTree Class

This class constructed a BST with one BinaryTreeNode as an object. The two methods of interest are:

### insert(Record data, BinaryTreeNode node) – This method (which is called by insert(Record data) recursively inserted a record object into the tree. It would compare the record with the data of the node. If the alphanumeric value was lower, it would check to see if the node’s left child was null. If it was null, the record would be inserted as a new BinaryTreeNode. If not, the method was called again with the child node as the new parameter. The opposite would happen if the value of the record was greater than the data of the node (with right children instead of left ones)

### find(String name, BinaryTreeNode node) – This was called by find(String data). In a similar fashion to the algorithm of insert(), find() recursively found a record. I.e., if a node’s data’s name was equal to that of the name inserted into the method, that node was returned. If the node was not found, the method returned null

## Querygen Class

This class had a single method: makeInput(int n). The testdata file was read into a linear ArrayList. Another ArrayList ‘queries’ was constructed. A for loop iterated n times creating a random number each time. The random number was then used as the index of an element of the ArrayList to be accessed. This element’s (of class Record) name was accessed and the name was added to the queries ArrayList. This ArrayList which now had n random names in it was returned.

## QuerygenMain Class

This class was used to generate 20 random names using the Querygen class to run the SearchIt and SearchItLinear classes. (See image 1)

## /Users/Zach/Desktop/Screen Shot 2017-04-09 at 9.56.45 PM.pngSearchItLinear Class

*Image 1: Query file produced upon running the QuerygenMain application. The 11th name was replaced with my own to test the SearchIt and SearchItLinear classes’ ability to cope with names that do not exist on the “testdata” file*

SearchItLinear found 20 data entries of given names/ It created an ArrayList populated with the details. It iterated through the list to find the given names and if they were found, printed the details out, if not, printed out “Not Found”. (See Image 2)

## SearchIt Class

This class initialised a bst and read in the “testdata” file to it – each line of data was added to the tree using the [insert(Record data)](#_insert(Record_data,_BinaryTreeNode) method. An ArrayList was constructed and populated with the names for “testdata”. It iterated over the bst and invoked the [find(String data)](#_find(String_name,_BinaryTreeNode) method to find each line of details for the given names. (See image 3)

## PrintIt Class

PrintIt had two main algorithms:

1. Reading in the data – A binary search tree was constructed. The “testdata” file was open and read using java’s BufferedReader. A loop iterated through each line of data and inserted (again using the [insert(Record data)](#_insert(Record_data,_BinaryTreeNode_1) method) them into a binary search tree.
2. inOrder() –This method is invoked by inOrder(BinarySearchTree bst). It accepts a node. If the node is not null, it traverses down the left branches of the tree (by recursively calling the inOrder method on the node’s left child). Once it reaches the smallest node (i.e., its child is null) then it prints that node. It climbes back up the tree, printing each node as it traverses. It then goes to the root’s right child and repeats this process. Eventually, all the nodes will have been printed in order from smallest to biggest.

# ../../../../../../../Desktop/Screen%20Shot%202017-04-09%20at%../../../../../../../Desktop/Screen%20Shot%202017-04-09%20at%

*Image3: Output from SearchIt.java*

*Image 2: Output from SearchItLinear.java*

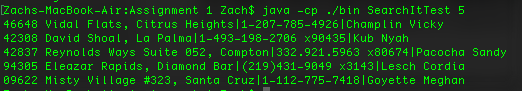
# Experiment

## Design

To run this experiment, I tried to account for as many confounding variables as possible. Consequently, there are a number of elements to the design of it.

### The SearchItTest and SearchItLinearTest Classes

Two special classes were developed to help run the tests for the experiment. Instead of reading data from a query file, these two classes accepted an argument when executing from the terminal. (See image 4)



*Image 4: Running SearchItTest from the terminal*

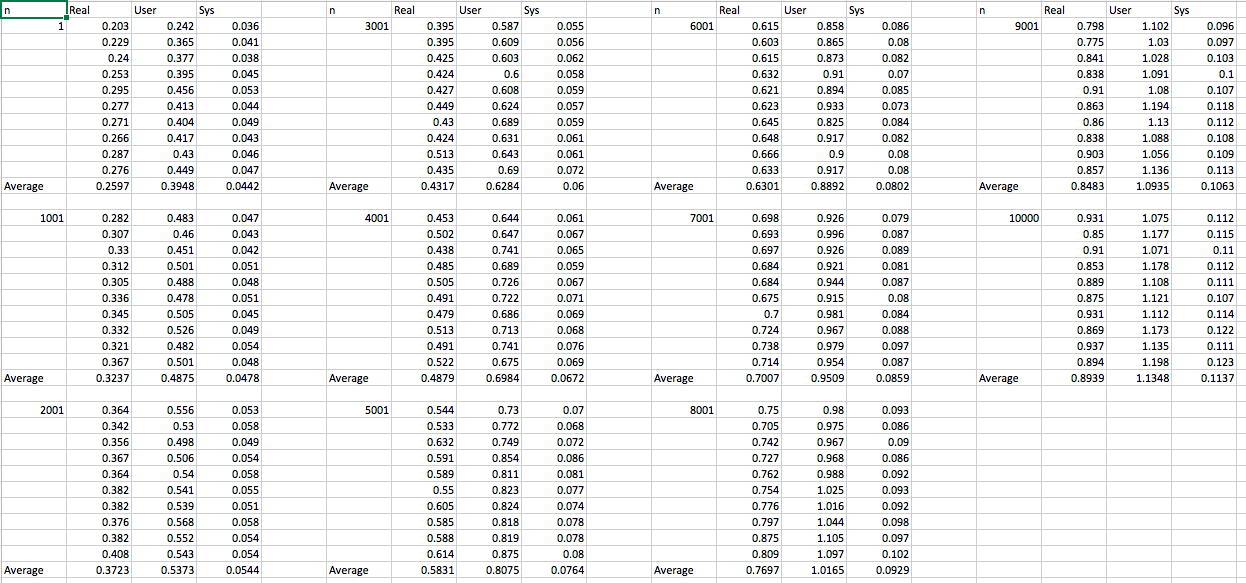
The argument was converted to an int by the class and this int was passed into Querygen to generate a random list of names to search for. The motivation behind this was to have a random sample for each experiment. Furthermore, the tests were run 10 times for each value of n. Thus the two programs were the same in all signifcant ways, reducing any confounds.

### The SearchItRunScript and SearchItLinearRunScript bash scripts

Another factor/variable considered was the time it would take the experimenter(me) to enter the different values of n. To solve this problem I developed two bash scripts, one for each testing class. The scripts executed the time method on 10 iterations of each testing class for different values of n. n began at 1 and increased by 1000 after 10 iterations up to 10 000. This gave me many data points to work with. The scripts then sent the time results (slicing the applications’ actual output) to an excel sheet.

## Results

After formatting the data, I had the following:



*Image 5: Results from SearchItLinearTest*

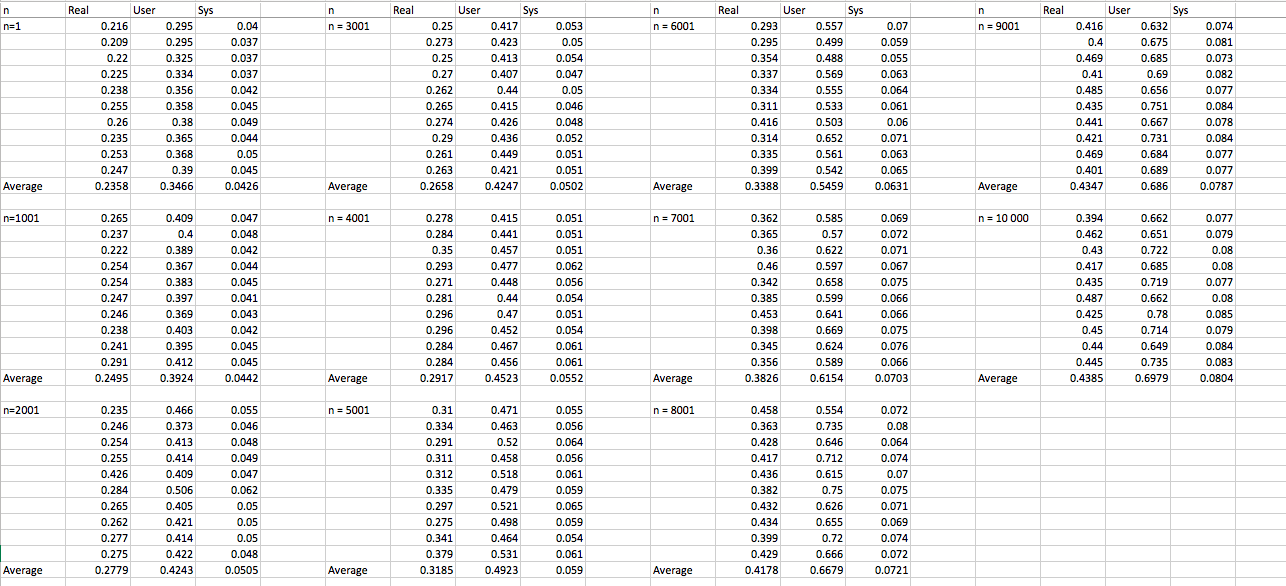
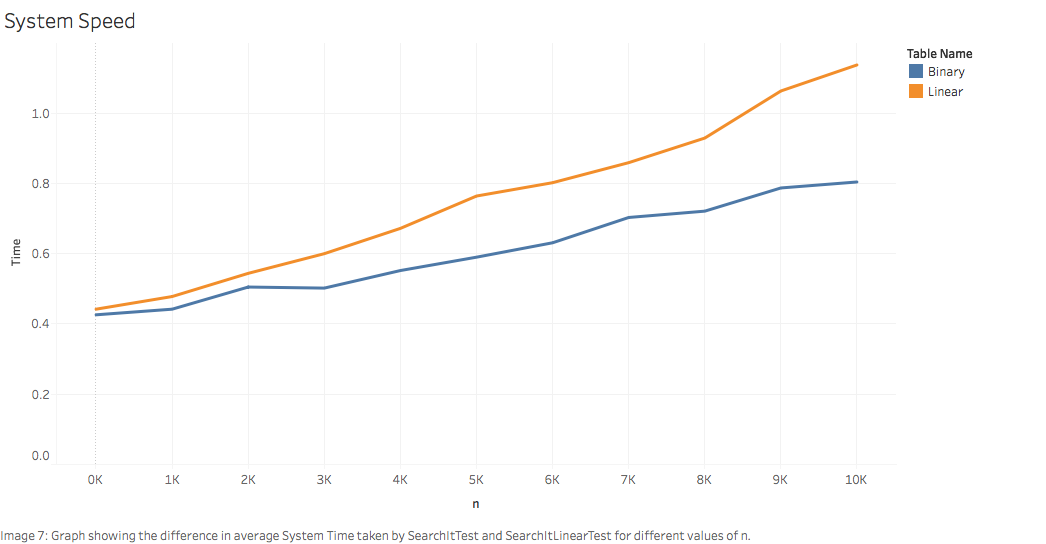
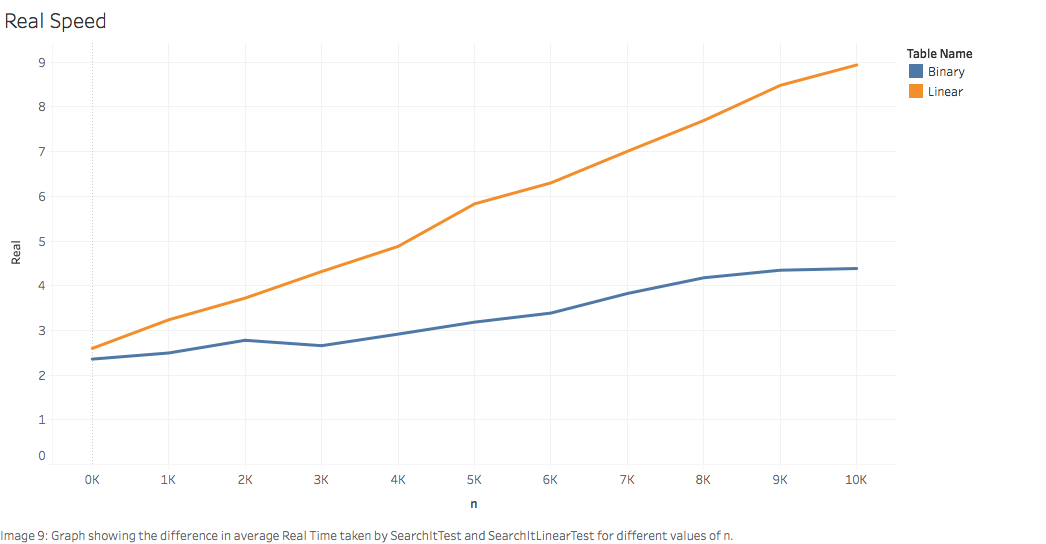
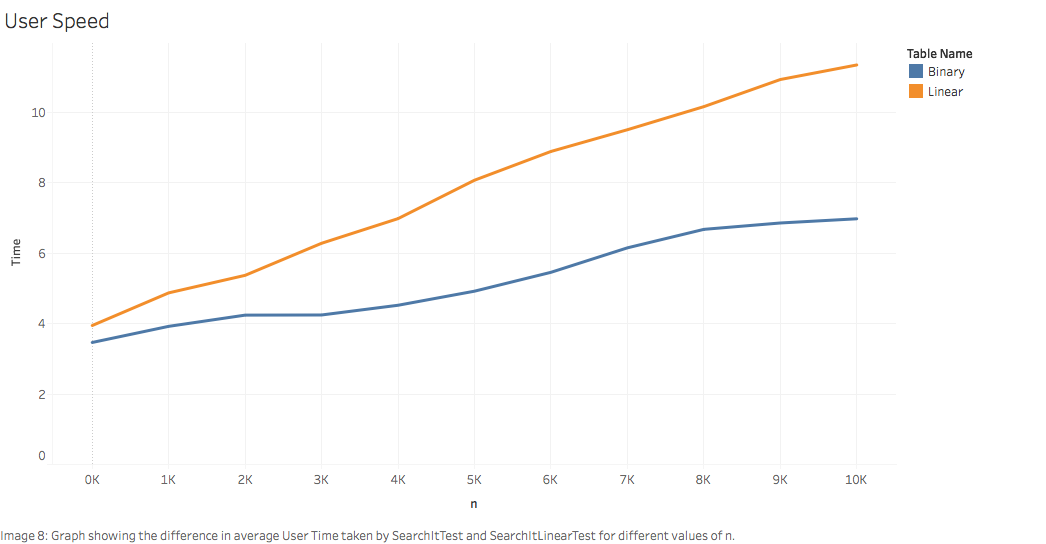


Image 6: Results from SearchItTest

Looking from the graphs, SearchItTest outperformed SearchItLinearTest for Real Time, User Time and System Time. (See images 7-9)





## Discussion

It was already known that a binary search tree is significantly faster than a linear data structure when searching for items. My experiment has proven it once again. From analysing the shape of the graphs, it can be inferred that the speed of a binary search is O(logn) and a linear search is O(n).

## Conclusion

To conclude, this assignment has given me a greater insight into not only the speed of searching in data structures, but also into bash, java, the terminal, makefiles, junit, git and javadocs. From trial and error, a lot of searching online and hours of work, I have started understanding how to develop applications.