PROJECT 1 :

An Empirical Study of Three Search Structures:

Binary Search Tree, Trie, AVL Tree, Hash Table

The Eminent: Emily Peguero Marmolejos

Jeffrey Torres

Tashi Dolma

Class: CSCI 323 - Analysis of Algorithms or 700

Semester: Spring 2020

Introduction

In this project we attempt to explore the varieties of advanced search structures on real word data. We aim to improve our understanding of, space complexity, time complexity, sorting, and various structures and algorithms. This project specifically tests a variety of data on AVL tree structure, Binary Tree Structure, Trie structure, and a hash table with collision resolution and quadratic probing. In this project we also want to hold theory up to action and compare the empirical data we collected and compare them to known runtimes.

This project focus on Data structures, algorithms, design techniques, and the comparison of their empirical performance with the theoretical understandings. Our group studied on following Data Structures and algorithms for search/lookup:

- Hash Table, with collision resolution Quadratic Probing
- Binary Search Tree
- An advanced search structure such as AVL Tree
- TRIE

We get opportunity to explore different sets of Data. We took an article and parse out the words to get a word list while studying binary search tree. While we study Hash table, we extracted data into text file from a an online site(https://opendata.cityofnewyork.us/). In AVL, randomly generated the data and build a tree. We run the code for different values of n and simultaneously track and collect the empirical runtimes for respective data structures with different value of data.

Algorithm Overview:

Binary Search Tree

Binary Search Trees (BST), sometimes called ordered or sorted binary trees, are a particular type of container: a data structure that stores "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its *key* (e.g., finding the phone number of a person by name).

Parser Class:

I used the JSoup library in order to get an array of strings that excluded any numbers or symbols, I then fed that array into a hash table for easy access.

Building the project:

The code was originally provided from geekforgeek.com. The data I decided to focus on was the frequency of words parsed from a webpage as well as lexicographical sorting. I built an object named Parser that would return a hash table with a String as a key and an Integer as the value, that represents the frequency of that String in an article.

The url I parsed was article: "https://www.tutorialspoint.com/java/lang/string_split.htm"

Specific Trial Runs: “Pineapple”; “the”; 30; 9; Max Values; Min Values

The Binary Search tree I created creates a BST that can be sorted by value (frequency) or lexicographical order of the keys.

In the trial I ran, put the same hash table into 2 trees who were sorted differently.

freq_tree1 // BST sorted wordlist by frequency value

lex_tree1 // BST sorted wordlist by Lexicographical order

I decided to have two types of sort to accommodate a wider range of data to be fetched. If the user inputs a String, then a lexicographical BST would be used to find that word. If the user inputs an integer then the Frequency BST would be iterated through until a node of that frequency is found. The string “Quit!”(WITH Exclamation) ends the run.

On the specific trial run I searched for words “Pineapple” (because it is NOT a part of the wordlist), “the” (because it is a part of the wordlist), an integer 30, another integer 9 as well as Max and Min values for both frequency and lexicographical BSTs.

OUTPUT FROM TRIALS:

The outputs were edited to reduce redundancy, the actual output would look much like the output for just the ‘find the min and max values run’. I left only the data that was important for that search specifically.

“Pineapple”

```
What word would you like to search in the BinarySearch Tree? Enter a
number to find frequency, a string to search for a word.
 Enter : 'QUIT!' to end search.
Pineapple
That word is not in the search tree
What word would you like to search in the BinarySearch Tree? Enter a
number to find frequency, a string to search for a word.
 Enter : 'QUIT!' to end search.
Quit!
```

```
Lexicographical Binary Search Tree Data :

Number of comparisons to create the frequency BST : 0
Number of comparisons to create lexicographical BST : 1061
Number of comparisons for frequency search : 0
Number of comparisons for lexographical search : 8
Number of iterations for max search : 5
Number of iterations for min search : 5
Total comparisons: 1069
Total iterations: 10

Time taken: 8520 milliseconds
N: size of wordlist 248 nodes including root.
Number of comparisons in this run in total : 3621069
Number of iterations in this run in total : 18810
```

“the”

```
What word would you like to search in the BinarySearch Tree? Enter a
number to find frequency, a string to search for a word.
 Enter : 'QUIT!' to end search.
the
Keyword : the
What word would you like to search in the BinarySearch Tree? Enter a
number to find frequency, a string to search for a word.
 Enter : 'QUIT!' to end search.
quit!

Lexicographical Binary Search Tree Data :

Number of comparisons to create the frequency BST : 0
Number of comparisons to create lexicographical BST : 1061
Number of comparisons for frequency search : 0
Number of comparisons for lexicographical search : 9
Number of iterations for max search : 5
Number of iterations for min search : 5
Total comparisons: 1070
Total iterations: 10

Time taken: 6260 milliseconds
N: size of wordlist 248 nodes including root.
Number of comparisons in this run in total : 3621070
Number of iterations in this run in total : 18810
```

3

```
What word would you like to search in the BinarySearch Tree? Enter a
number to find frequency, a string to search for a word.
```

```
 Enter : 'QUIT!' to end search.
30
That frequency is not in the search tree
What word would you like to search in the BinarySearch Tree? Enter a
number to find frequency, a string to search for a word.
 Enter : 'QUIT!' to end search.
quit!

Frequency Binary Search  Tree Data :

Number of comparisons to create the frequency BST : 362
Number of comparisons to create lexicographical BST : 0
Number of comparisons for frequency search : 7
Number of comparisons for lexicographical search : 0
Number of iterations for max search : 6
Number of iterations for min search : 182
Total comparisons: 376
Total iterations: 188


Time taken: 4984 milliseconds
N: size of wordlist 248 nodes including root.
Number of comparisons in this run in total : 3761061
Number of iterations in this run in total : 18810

9

What word would you like to search in the BinarySearch Tree? Enter a
number to find frequency, a string to search for a word.
 Enter : 'QUIT!' to end search.
9
Keyword : of Frequency : 9
What word would you like to search in the BinarySearch Tree? Enter a
number to find frequency, a string to search for a word.
 Enter : 'QUIT!' to end search.
quit!
Frequency Binary Search  Tree Data :

Number of comparisons to create the frequency BST : 362
Number of comparisons to create lexographical BST : 0
Number of comparisons for frequency search : 7
Number of comparisons for lexographical search : 0
Number of iterations for max search : 6
Number of iterations for min search : 182
Total comparisons: 376
Total iterations: 188

Time taken: 6150 milliseconds
N: size of wordlist 248 nodes including root.
```

```
Number of comparisons in this run in total : 3761061
Number of iterations in this run in total : 18810
```

Just Min and Max values ONLY

```
What word would you like to search in the BinarySearch Tree? Enter a
number to find frequency, a string to search for a word.
 Enter : 'QUIT!' to end search.
quit!
__________Random Search terms BST : _____________
Finding the max freq :
Word with Max Value is : lang : 51
0Finding the lex max :
Word with Max Value is : zero : 1
Finding the min freq :
'Float' with frequency : 1
Finding the lex min :
'a' with frequency : 3

__________Empirical Evidence BST : _____________

Parser data :

Number of iterations to get data table : 498
Number of comparisons to get data table : 250

Frequency Binary Search  Tree Data :

Number of comparisons to create the frequency BST : 362
Number of comparisons to create lexographical BST : 0
Number of comparisons for frequency search : 0
Number of comparisons for lexographical search : 0
Number of iterations for max search : 6
Number of iterations for min search : 182
Total comparisons: 362
Total iterations: 188

Lexicographical Binary Search Tree Data :

Number of comparisons to create the frequency BST : 0
Number of comparisons to create lexicographical BST : 1061
Number of comparisons for frequency search : 0
Number of comparisons for lexicographical search : 0
Number of iterations for max search : 5
Number of iterations for min search : 5
Total comparisons: 1061
Total iterations: 10

Time taken: 4296 milliseconds
```

```
N: size of wordlist 248 nodes including root.
Number of comparisons in this run in total : 3621061
Number of iterations in this run in total : 18810
```

Analysis:

Known Runtime for BST search:

Best O(1) -Where the root is the target searched for

Average O(lgn) – lgn represents the ideal height of the tree , this tree is not self balancing and lg(n) height is not garunteed

Worst case O(n) – this occurs when n is the height of the tree, which means the BST is just a deteriorated tree, or a linked list. This can occur if the root is the smallest value or the largest value of the given data

Known Runtimes for BST insertion:

Average – O(h), h =lg(n), where h represents height of the tree

Worst case – O(n) – happens when tree is

CHART FOR BINARY SEARCH TREE:

Height Freq Tree: 183 & Height Lex Tree : 14

Action	Expected value	Actual value	Time for entire run (ms)

Search for data not in tree:
'Pineapple'
	lg(n) // n = 248
	lg(248) =
	7.95419631039
		8 comparisons	8520

Search for freq not in tree:
30
	lg(n) // n = 248
	lg(248) =
	7.95419631039
		7 comparisons	4984

Search for data in tree:
'the'
	lg(n) // n = 248
	lg(248) =
	7.95419631039
		9 comparisons	6260

Search for frequency in tree:
9
	lg(n) // n = 248
	lg(248) =
	7.95419631039
		7 comparisons	6150

Max Search for frequency	O(h)// h is height
	O(183)
		6 iterations	4296

Min Search for frequency	O(h)// h is height
	O(183)
		182 iterations	4296

Max Search for String	O(h)//h is height
	O(183)
		5 iterations	4296

Min Search for String	O(h)// h is height
	O(14)
		5 iterations	4296

Insertion Frequency	O(h)//h is height
	O(183)
		362 Comparisons	4296

Insertion Lexicographical	O(h)// h is height
	O(14)
		1061 comparisons	4296

Conclusion:

The data is interesting some values, were right on par, as far as the runtime for searching for data, which is amazing, even if the frequency BST is quite unbalanced. I expected the number of comparisons for insertion of the lexicographical tree to be at least twice that of insertion for the frequency BST, because comparing Strings using the 'compareTo()' method is one comparison and comparing the value THAT function returns is another comparison.

Ways that the data could be improved is by finding a value or string that is the most median, and using that as the root. A self balancing tree would of course be more optimal. All in all the Binary search tree has great search times .

Trie:

A trie is a tree-like data structure whose nodes store the letters of an alphabet. By structuring the nodes in a particular way, words and strings can be retrieved from the structure by traversing down a branch path of the tree. Tries in the context of computer science are a relatively new thing.

Building the project:

The original code was from geeksforgeeks. The code was specific for strings that were only lowercase and no symbols or numbers; this led to a change n the parser class. A change was made to the original regex that was used as a delimiter. The function 'toLowerCase()' is used in this code in order to accommodate an array of size 26 for its alphabet.

Specific Trial Runs: "love"; "the"; 3; "Th"

I chose these trials because one word is not in the trie ("love"), one word is in the trie ("the") , an integer would do no search, and part of a word that is in the trie ("Th") is a typical search, because it proves that even if a successive collection of characters can be a limb on the trie, it is not a part of the trie.

OUTPUT FROM TRIALS:

The program asks for a user input using a while loop and searches the input only if that input is a string. If the input is a digit no search occurs, or the while loop gets exited.

"Love"

```
What keyword would you like to search?
Love

What keyword would you like to search? enter an integer to quit.
love --- Not present in trie
7
__________Empirical Evidence BST : _____________
Parser Data :
Number of iterations to get data table : 498
Number of comparisons to get data table : 250
Trie Data
Number of comparisons to create trie 1019
Number of iterations to create trie 28135
Number of comparisons to search the trie 3
Number of iterations to search the trie 4

Time taken: 4372 milliseconds

N :  248 words were added to the Trie.
Number of comparisons in this run in total : 1022
```

```
Number of iterations in this run in total : 28139
```

“The”

```
What keyword would you like to search?
The

What keyword would you like to search? enter an integer to quit.
the --- Present in trie
6
__________Empirical Evidence BST : _______________
Parser Data :
Number of iterations to get data table : 498
Number of comparisons to get data table : 250
Trie Data
Number of comparisons to create trie 1019
Number of iterations to create trie 28135
Number of comparisons to search the trie 5
Number of iterations to search the trie 3

Time taken: 3742 milliseconds

N :  248 words were added to the Trie.
Number of comparisons in this run in total : 1024
Number of iterations in this run in total : 28138
```

3

```
What keyword would you like to search?
3
__________Empirical Evidence BST : _______________
Parser Data :
Number of iterations to get data table : 498
Number of comparisons to get data table : 250
Trie Data
Number of comparisons to create trie 1019
Number of iterations to create trie 28135
Number of comparisons to search the trie 0
Number of iterations to search the trie 0

Time taken: 1928 milliseconds

N :  248 words were added to the Trie.
Number of comparisons in this run in total : 1019
Number of iterations in this run in total : 28135
```

“th”

```
What keyword would you like to search?
th
```

```
What keyword would you like to search? enter an integer to quit.
th --- Not present in trie
3
__________Empirical Evidence BST : _______________
Parser Data :
Number of iterations to get data table : 498
Number of comparisons to get data table : 250
Trie Data
Number of comparisons to create trie 1019
Number of iterations to create trie 28135
Number of comparisons to search the trie 3
Number of iterations to search the trie 2

Time taken: 3539 milliseconds

N :  248 words were added to the Trie.
Number of comparisons in this run in total : 1022
Number of iterations in this run in total : 28137
```

Analysis:

Known Runtime for Trie Search:

Worst: O(26*k) = O(k)-where k is the length of the string

Average: O(k)

Best: O(1) where the string is only 1 in length

Known Runtime for Trie insertion:

Worst: O(26*k) = O(k)

Average: O(k)

Best: O(1)

CHART FOR TRIE:

Action	Expected Value	Actual Value	Time Elasped during run (ms)

Search for word not in trie: ‘Love’	O(k)//k is length of string
	O(4)
		4 iterations + 3 comparisons	4372

Search for word found in trie: ‘the’	O(k)// k is the length
	O(3)
		3 iterations + 4 comparisons	3742

Creation of the trie	O(Mk)// k is the size of the length of the strings and M is the size of the dictionary of the trie O(243k)//k is variable value	28135 iteration + 1019 comparisons	1928

Search for a string not found in trie: ‘Th’	O(k)//k is length of string
	O(2)
		2 iterations + 3 comparisons	3539

Conclusion:

The trie may be one of the most efficient structures for searching for strings. The search runtime holds true, there are always k iterations (k is the length of a given string). One downside to Trie is that it takes up a lot of space. The space complexity is O (size of alphabet*k*dictionary), which is extremely large. The values that surprised me were that of the values of creating the Trie. The given word list was relatively large, the large number of comparisons upon the creation of the trie is due to comparing variables to the alphabet array each node holds.

AVL Tree

The **AVL tree** is a self-balancing binary search tree. As such, it adheres to the same rules as a normal binary search tree, where nodes in the left subtree are less than the root and nodes in the right subtree are greater than the root.

However, the structure of the binary search tree is dependent on the order of the inserted data. Suppose a set of data is inserted into the binary search tree in increasing order. Each node in the tree is assigned a balance factor value, ranging between -1 and +1. This value is determined by the following formula:

height of left subtree - height of right subtree = balance factor

Inserting a node into the tree may cause a previously inserted node to have a balance factor outside of the -1 to +1 range. If this occurs, the tree will restructure the subtree(s) of the unbalanced node, so that the unbalanced node and each node in the restructured subtree(s) has a balance factor between -1 and +1. The AVL tree is able to maintain a balanced structure that allows for more efficient operations. More detailed explanations of each type of rotation are given below.

Generate random integers in range 0 to 999999

The number of comparisons in the Search operation is the level of the tree from the root to its node. In the worst case, the number of comparisons in the Search operations is equal to the depth of deepest leaf of the tree (and the total number of steps required for any of these operations is proportional to the number of comparisons)

CHART FOR AVL TREE

Empirical Runtime

Size(n)	Avl	Number of Comparison
10	4milliseconds	297
100	10milliseconds	701
1000	14milliseconds	856
10000	21milliseconds	888

Theoretical Runtime:

Time	O(logn)	O(logn)
Space	O(n)	O(n)
Search	O(logn)	O(logn)
Insert	O(logn)	O(logn)

Algorithm Overview: Hash Table

A hash table is a data structure which stores data in an associative manner. The data is stored on the table inside of an array, where each data value has its own unique index value. To create an index where the desired element is to be inserted or searched for, a technique known as hashing is employed.

Hashing may lead to mapping to an index that is already being used, known as a collision. To resolve collisions, the method of open addressing was employed. Furthermore, quadratic probing was used. In quadratic probing, to find the next available spot for insertion, we look for the i^2-th slot during the i-th iteration. This is applied to the hash function, which employs the mod operator in most cases. This will lead to less severe clustering of data and will make searching and inserting faster.

Implementation Considerations

Java was used to create the code for this project. The types of operations that were kept track of were the ones such as comparisons (if-else statements), the operations that were inside a while or for loop, and somewhat costly operations such as computing the modulo. Two counters (one in the hash table class and the other in the main class) were created to keep track of the total number of these operations, taking the sum of both of these counters. To get the data to use on the hash table, we used the NYC OpenData website using random search terms. This data was then exported to a CSV file, converted into an Excel file, and converted again into a text file. The files were named patient_categories (Source 1), foodscrap_dropoff_locations (Source 2, and rent (Source 3). Two charts were made. The first chart kept track of the number of operations given the size of the hash table at n = 5000, 10000, and 15000 on the three files. The second chart kept track of the execution time for size of the hash table on the three files.

Raw Output

Taken from running patient_categories.txt on Eclipse

n = 5000

```
Hash Table:
0, Resident  154    20
1, Local Worker     378    50
2, Resident + Local Worker77     10
3, Clean-Up Worker  51     7
4, Clean-Up Worker + Local Worker       20     3
5, Rescue/Recovery  4      1
6, "Passerby, Commuter"    32     4
7, Student   25     3
8, Other     13     2

Number of operations: 27
Execution time: 1 milliseconds
```

n = 10000

```
Hash Table:
0, Resident  154    20
1, Local Worker     378    50
2, Resident + Local Worker77     10
3, Clean-Up Worker  51     7
4, Clean-Up Worker + Local Worker       20     3
5, Rescue/Recovery  4      1
6, "Passerby, Commuter"    32     4
7, Student   25     3
8, Other     13     2

Number of operations: 27
Execution time: 2 milliseconds
```

n = 15,000

```
Hash Table:
0, Resident  154    20
1, Local Worker     378    50
```

```
2, Resident + Local Worker77     10
3, Clean-Up Worker  51     7
4, Clean-Up Worker + Local Worker       20     3
5, Rescue/Recovery  4      1
6, "Passerby, Commuter"    32     4
7, Student    25     3
8, Other      13     2

Number of operations: 27
Execution time: 2 milliseconds
```

Size of Hash Table vs. Number of Operations

Size(n)	Source 1	Source 2	Source 3
5000	27	816	29798
10000	27	816	17950
15000	27	816	16106

Size of Hash Table vs. Execution Time using Eclipse

Size(n)	Source 1	Source 2	Source 3
5000	1ms	15ms	51ms
10000	2ms	17ms	62ms
15000	2ms	17ms	63ms

Conclusion

The theoretical runtime for operations in a hash table (such as searching and inserting) is usually $\Theta(1)$. However, depending on the severity of the clustering of data, the available slots in the hash table, and its size, the runtime can become as long as the size of the hash table. For the patient_categories file, because it only has 9 rows, given the lengths of the hash table, execution time was very fast and few operations were done. For the rent file, however, because it has 4,935 rows, fitting that many elements into an array of size 5000 will guarantee collisions which will increase the number of operations and increase execution time. However, as the size of the array grows, the possibility for collisions is greatly reduced, so the number of operations will decrease, eventually to a constant factor. Also, it's important to point out how the code is being executed. Perhaps we can decrease the execution time if the code is run via the command line instead of using Eclipse or any other IDE.

Sources

Code for Hash Table Quadratic Probing

Manish. “Java Program to Implement Hash Tables with Quadratic Probing.” *Sanfoundry*, 6 Dec. 2013, www.sanfoundry.com/java-program-implement-hash-tables-quadratic-probing/.

Hash Table description
“Data Structure and Algorithms - Hash Table.” *Tutorialspoint*, www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm.