Algorithms Assignment 3

John Higgins

Lane Harrison

Algorithms

Over-View

In this report a comparison between Linear Probing Hash Tables (LPH) and Binary Search Tree Table(BST) is made in application to text processing. The goal for each algorithm is to build a search engine for a given set of documents categorized by a specific word, its frequency and a known metric called tf-idf. The tf-idf of any given word measures how unique it is to the given document. It has been calculated as follows:

```
tf-idf = tf * log2(1 + (number-of-docs / the-words-frequency-across-all-docs)
```

For brevity, each method discussed in the following section is of the Interface Table, which serves to represent the main functions of both the Hash Table and the Binary Search Tree Table.

Hash Table

A hash table similar to that found in our text book on page 470 (pdf.483) was adapted to provide functionality needed. To manage collisions linear probing was used.

Variables

The variables specific to each Hash Table are explained here

Int N: the number of keys

String doc_name: the name the document came from. If the document is a combination of all documents its name is "All"

Int doc_name_i: the index of the doc named doc_name in an array of all documents

Int M = 16: the initial size (unless specified otherwise) of the hash table upon creation

String[] Keys: an array of all keys, where the index of a key is at the keys given hash value

Term[] Vals: an array of all Terms, where the index of a term is its keys hashed value. A term contains each word/keys information such as its frequency tf_idf and other relevant information.

Term[] top_10: an array of terms initiated when the top_10 method is called on a document, so that the top_10 for a given document must only be calculated once.

BST<Double, Term> all_words: a binary search tree (a tree different than the Binary Search Tree table discussed) that contains all key values so that the top_10 function may simply remove the maximum ten times.

Methods

String get_data(): puts all values found in the table into a String to be inserted into an excel document for analysis

v**oid put(String key):** Inserts a given key into the table by first resizing if necessary and then hashing the key into a Int value. This hashed value is then placed and linear probing is used to avoid collisions.

When put if the key already exists in the table the given keys term value is incremented, otherwise a new string and term pair is inserted with the new key and a new term that represents it.

Term get(String key): hashes the key and iterates through the table looping back to the beginning until it has been found or it reaches a null key (property of linear probing)

boolean contains(String key): hashes the key and iterates through the table looping back to the beginning until it has been found (returns true)or it reaches a null key (returns false)(property of linear probing)

void merge(Table d, int max): Once all String term pairs for each document have been inserted using put(String key), a document named all is created and all documents contents are "merged" or combined into a single one

String[] getKeys(): returns the list Keys. This is only used internally and does not violate encapsulation

Object getRoot(): Not Used

int getDNI(): returns doc_name_i

void calculate(Document[] all): calculates and places the tf-idf scores for each word in each document using the formula

```
tf-idf = tf * log2(1 + (number-of-docs / the-words-frequency-across-all-docs)
```

Term[] top10(): calculates the top 10 tf-idf scores by removing the maximum 10 times from all_words

void tf_idf(String key, double idf): a helper function for calculate which places the tf_idf score for a given key

int[] get_search(String key): Used only on "All" document. Hashes the key and iterates through the table looping back to the beginning until it has been found or it reaches a null key (property of linear probing). Returns a list of Int's that are indexes into an array of documents, so that costly null searches are avoided.

String[] sample(): returns a list of Strings one tenth of the total number of unique keys.

Binary Search Table

A binary Search Tree similar to that found in our text book on page 398 (pdf.411) was adapted to provide functionality needed.

Variables

Here the variables specific to each Hash Table are explained

Int N: the number of keys

String doc_name: the name the document came from. If the document is a combination of all documents its name is "All"

Int doc_name_i: the index of the doc named doc_name in the array of all documents

Term[] top_10: an array of terms instantiated when the top_10 method is called on a document, so that the top_10 for a given document must only be calculated once

BST<Double, Term> all_words: a binary search tree (a tree different than the Binary Search Tree table discussed) that

contains all key values so that the top_10 function may simply remove the maximum ten times

Node root: the root of the binary search table

Methods

String get_data(): puts all values found in the table into a String to be inserted into an excel document for analysis

v**oid put(String key):** Inserts a given key into the table by recursively checking the appropriate branch based on comparison to the current Node (comparisons begin at root and works down the tree). If the key already exists in the table the given key's term value is incremented, otherwise a new key and term pair is inserted based on the key being put.

Term get(String key): Recursively checks the appropriate branch based on comparison to the current Node (comparisons begin at root at work down the tree) and returns either the found key's term or null

boolean contains(String key): Recursively checks the appropriate branch based on comparison to the current Node (comparisons begin at root at work down the tree) and returns if the key exists within the tree

void merge(Table d, int max): Once all String term pairs for each document have been inserted using put(String key), a document named all is created and all documents table contents are "merged" or combined into a single one.

String[] getKeys(): Not Used

Object getRoot(): returns root

int getDNI(): returns doc_name_i

void calculate(Document[] all): calculates and places the tf-idf scores for each word in each document using the formula

```
tf-idf = tf * log2(1 + (number-of-docs / the-words-frequency-across-all-docs)
```

Term[] top10(): calculates the top 10 tf-idf scores by removing the maximum 10 times from all_words

void tf_idf(String key, double idf): a helper function for calculate which places the tf_idf score for a given key

int[] get_search(String key): Again recursively iterates through the tree, to find the given key. From the term if found, returns a list of Int's that are indexes into an array of documents, so that costly null searches are avoided.

String[] sample(): returns a list of Strings one tenth of the total number of unique keys.

Comparison

Observation:

To test the functionality of construction and search each was run with 20 test sets. For search one tenth of all the unique keys were randomly chosen for each test set. For construction, the data set provided was used repeatedly.

Search:

In Chart A it is apparent that there exists a single outlier within the data collected. This single outlier impacts the average of the data greatly because it is so large, observable in Chart B. However, when this point is removed LPH (Linear Probe Hash Table) and BST (Binary Search Table) are comparable. The height of both BST and LPH appear to be similar, however under closer examination of Chart C, the spread especially closer to the bottom is larger. This is shown further in Chart B given that BST has larger mean search time than LPH. Therefore, the LPH representation outperforms that of the BST in search.

Chart A

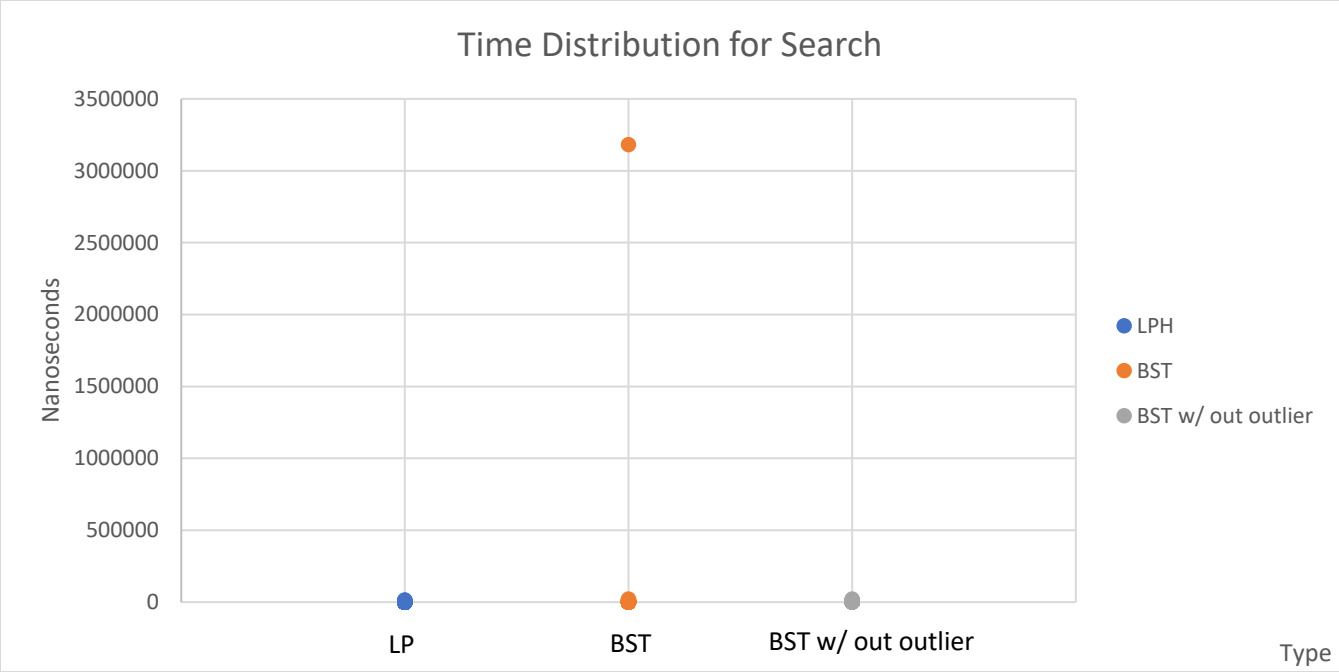


Chart B

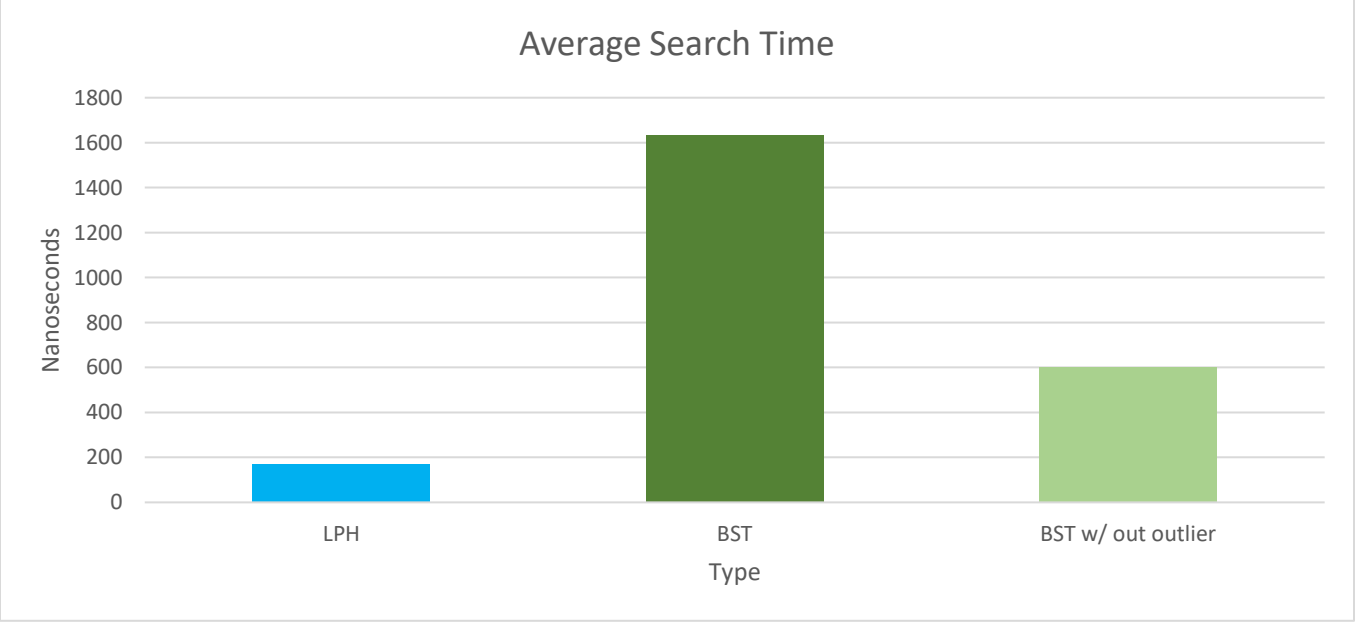
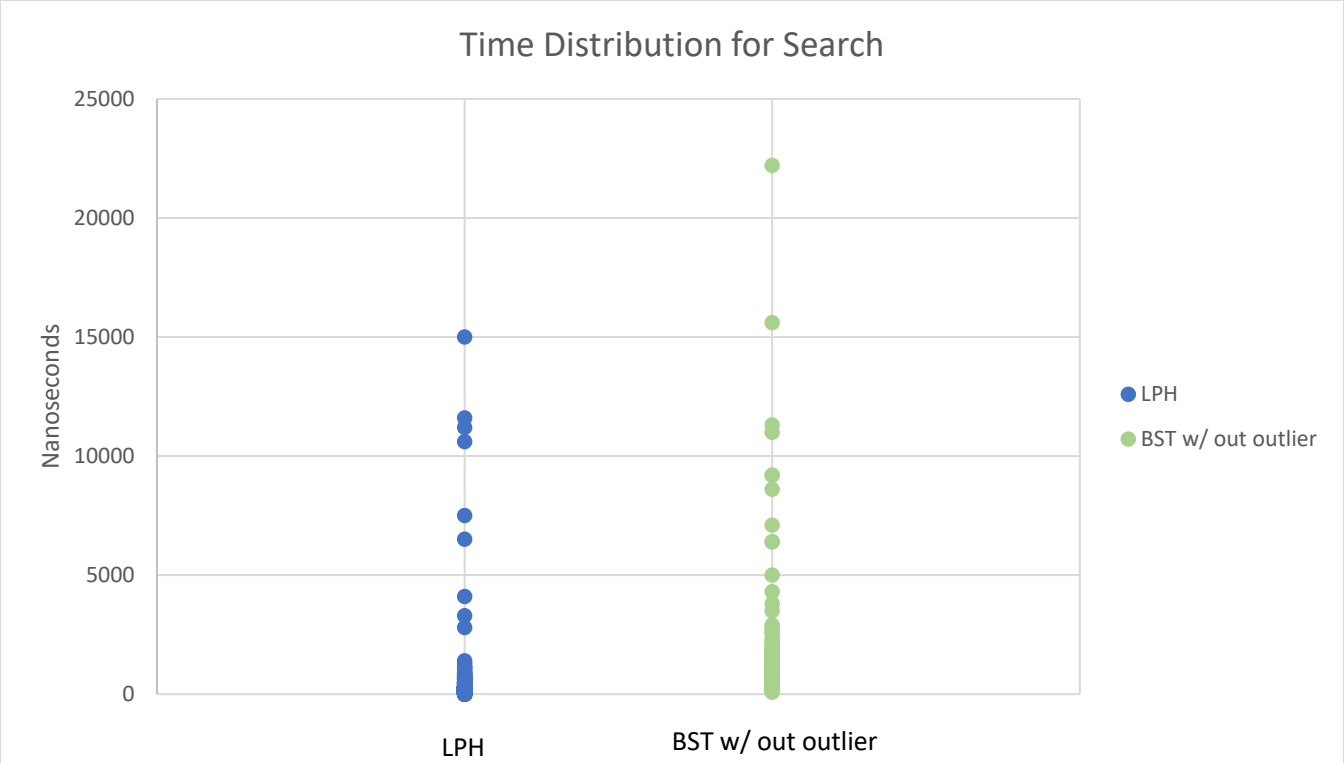


Chart C



Construction:

Through observation of Chart D and Chart C LPH also outperforms that of BST in creation.

Chart D

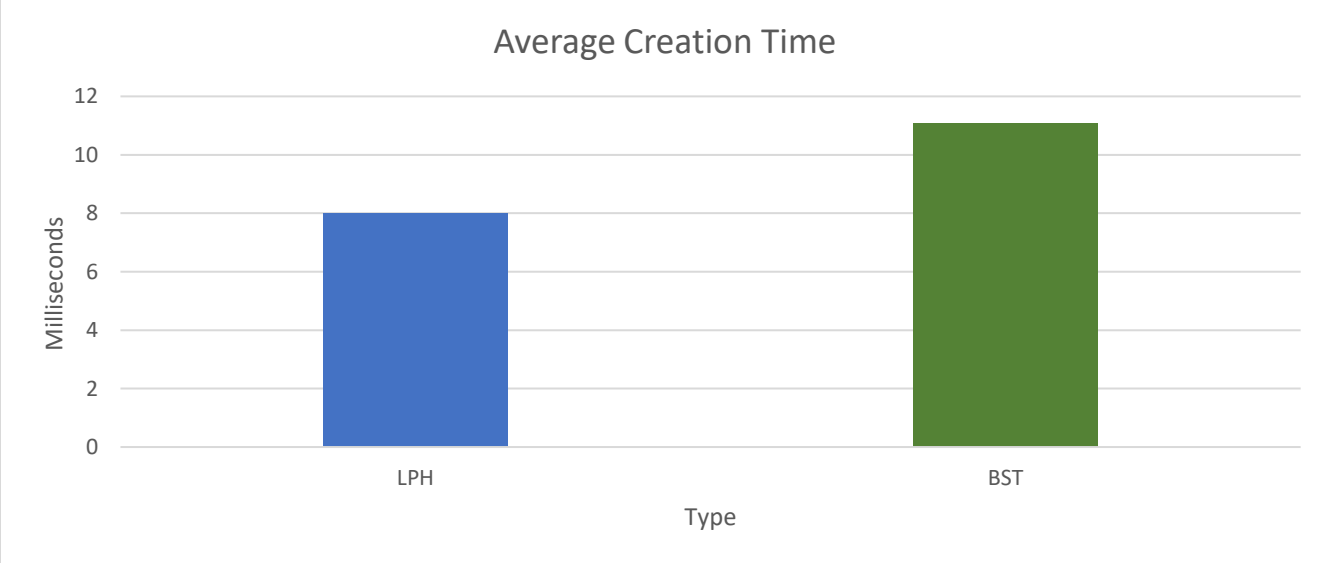
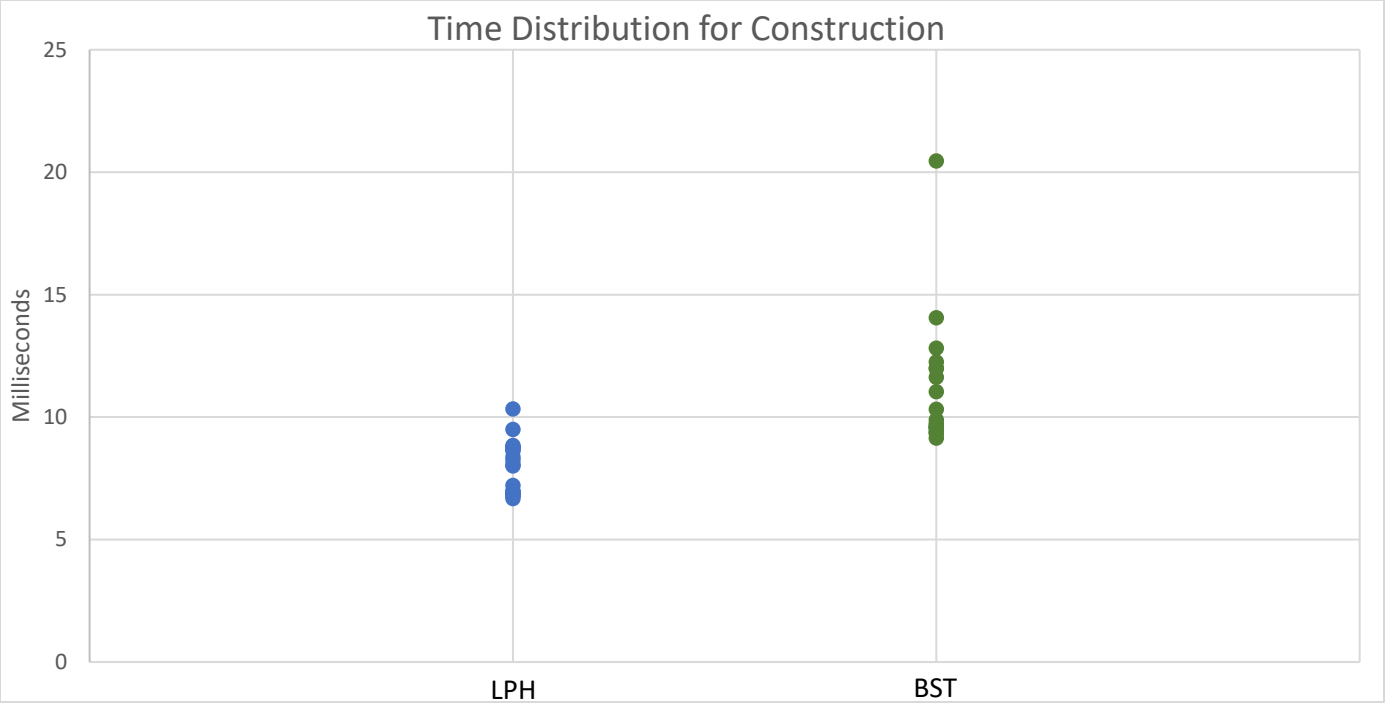


Chart E



Analysis

Through comparison, LPH (Linear Probe Hash Table) is quicker in both creation and sorting for text analysis when compared to a BST (Binary Search Table). When insertion takes place, a key operation in construction, it will take much longer in the BST representation compared to the LPH. This is because in the BST it must be compared with at most the greatest depth of the tree at the point of insertion (between lg2(size-of-tree) and size-of-tree), while the LPH can take anywhere from constant time (no collision occurs) to the size of the largest cluster in the worst case. Since the LPH has been implemented in such a way that it resizes when too small, clusters of large values are prevented and on average each construction will take a shorter amount of time. However because of resizing the gap between the two is smaller than it would be if we knew how many hash values we had.

Similarly, search is slower in a BST when compared with a LPH for the same reason it is slower in search. To find an item in the BST we must again iterate through at most the greatest depth of the tree compared to only the largest cluster in the LPH.

In regard to space, LPH uses more space than the BST for the same amount of keys because it every resize it doubles the amount of storage used. Since the initial size is set to 16 many resizes will occur, especially in documents with many unique words. However, in a BST representation the space used directly corresponds to the number of unique keys, making the BST better with respect to space.