Name: Henry W. Johnson ID#: 397342

Date: 5/3/2022

**INTRODUCTION:**

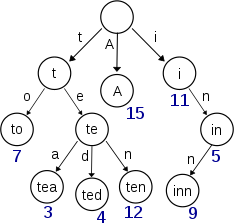
In this task, we will analyze the time taken for executing different algorithms. We have a total of 4 algorithms which finds the substring location in main string. Algorithms: -

1. Trie
2. Hash Map
3. BST
4. Linear Func(vector or list)

# TRIE

a trie, also called digital tree or prefix tree, is a type of [*k*-ary](https://en.wikipedia.org/wiki/M-ary_tree) [search tree](https://en.wikipedia.org/wiki/Search_tree), a [tree](https://en.wikipedia.org/wiki/Tree_(data_structure)) [data structure](https://en.wikipedia.org/wiki/Data_structure) used for locating specific keys from within a set. These keys are most often [strings](https://en.wikipedia.org/wiki/String_(computer_science)), with links between nodes defined not by the entire key, but by individual [characters](https://en.wikipedia.org/wiki/Character_(computing)). In order to access a key (to recover its value, change it, or remove it), the trie is traversed [depth-first](https://en.wikipedia.org/wiki/Depth-first_search), following the links between nodes, which represent each character in the key.

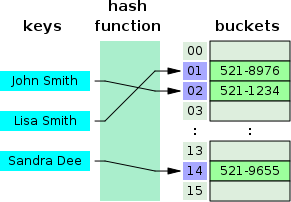
Unlike a [binary search tree](https://en.wikipedia.org/wiki/Binary_search_tree), nodes in the trie do not store their associated key. Instead, a node's position in the trie defines the key with which it is associated. This distributes the value of each key across the data structure, and means that not every node necessarily has an associated value.



# HASH MAP

**hash map**, is a [data structure](https://en.wikipedia.org/wiki/Data_structure) that implements a [set](https://en.wikipedia.org/wiki/Set_(abstract_data_type)) [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type), a structure that can map [keys](https://en.wikipedia.org/wiki/Unique_key) to [values](https://en.wikipedia.org/wiki/Value_(computer_science)). A hash table uses a [hash function](https://en.wikipedia.org/wiki/Hash_function) to compute an *index*, also called a *hash code*, into an array of *buckets* or *slots*, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored.

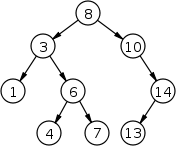
Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash [*collisions*](https://en.wikipedia.org/wiki/Hash_collision) where the hash function generates the same index for more than one key. Such collisions are typically accommodated in some way.



# BST

a binary search tree (BST), also called an ordered or sorted binary tree, is a [rooted](https://en.wikipedia.org/wiki/Rooted_tree) [binary tree](https://en.wikipedia.org/wiki/Binary_tree) [data structure](https://en.wikipedia.org/wiki/Data_structure) with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. The [time complexity](https://en.wikipedia.org/wiki/Time_complexity) of operations on the binary search tree is [directly proportional](https://en.wikipedia.org/wiki/Directly_proportional) to the height of the tree.

Binary search trees allow [binary search](https://en.wikipedia.org/wiki/Binary_search_algorithm) for fast lookup, addition, and removal of data items. Since the nodes in a BST are laid out in such a way that each comparison skips about half of the remaining tree, the lookup performance is proportional to that of [binary logarithm](https://en.wikipedia.org/wiki/Binary_logarithm).



# LINEAL FUNCTION

A linear function approximator is a function *y=f(x,w)* that is linear in the weights, though not necessarily linear in the input *x*:

y = w\_1 \* f\_1(x) + w\_2 \* f\_2(x) + ... + w\_n \* f\_n(x)

where *x*, *y*, and *w* can be vectors, the *f\_i*() functions can be linear or nonlinear, and *w\_i* is the *i*th element of the *w* vector. Examples of linear function approximators include:

### Lookup table

There is a separate weight for each possible value of *x*. There are

only *n* possible values for *x*, and *f\_i(x)=1* when *x=i* and *f\_i(x)*=0 otherwise.

### Linear

The output is just the dot product of *w* and *x*. The individual functions are just *f\_i(x)=x\_i*, where *x\_i* is the *i*th element of vector *x*.

### Radial Basis Functions

Each *f\_i(x)* function looks like a smooth bump. Each *f\_i()* function has a "center" location, and *f\_i(x)* is a monotonic function of the distance from *x* to the center. The "distance" may be Euclidean distance (circular

bumps), or there may be a diagonal covariance matrix (ellipsoidal bumps parallel to the axes), or there may be a full covariance matrix (general ellipsoidal bumps). To be a linear function approximator, the bumps must not move or change shape.

### Wavelets

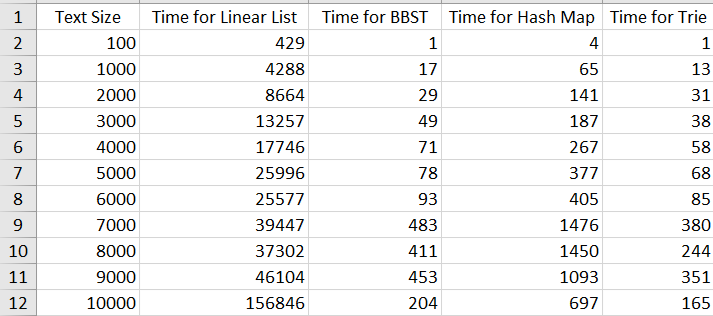
Each f\_i(x) is a wavelet, typically the product of a cosine and a Gaussian. This is particularly useful in image applications, because the human visual system seems to use similar functions.

**METHODOLOGY:**

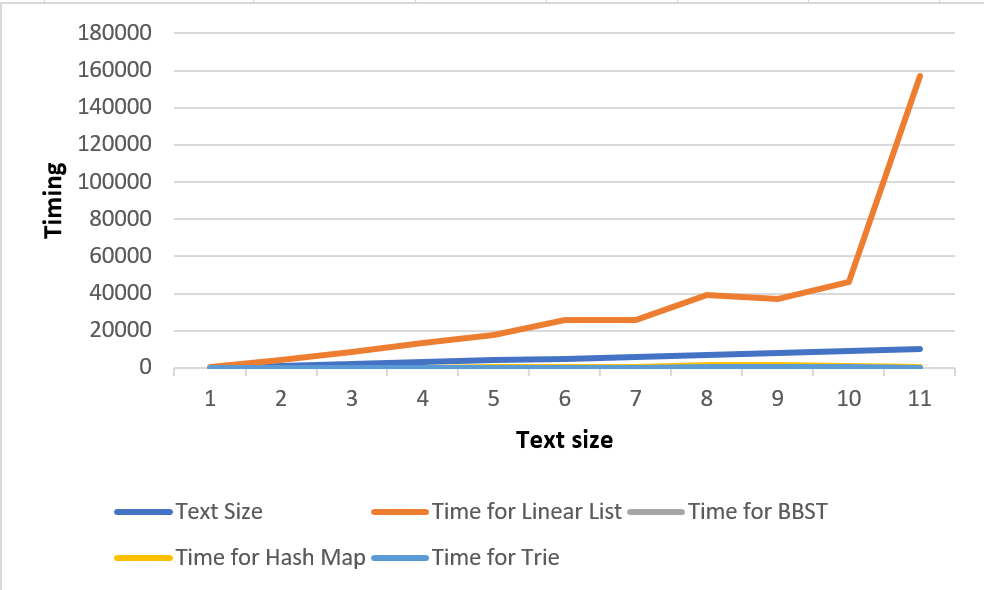
We have created different files with different word count for example 100 words, 1000 words, 2000 words till 10000 words.. We first read English dictionary file in computer memory and then insert the dictionary data into the data structures that are to be used for example trie, hash map, list and bst. After inserting dictonary data each text file is opened one by one automatically and first the file words are stored in computer memory and then each word is searched in each data structure.. Total time for adding the dictonary to data structure and for searching all the words in the text files are noted. Same process is repeated for all the text files. The time is noted using the chrono library in c++.

**RESULT**

**This is the data we have gathered for word searches:**

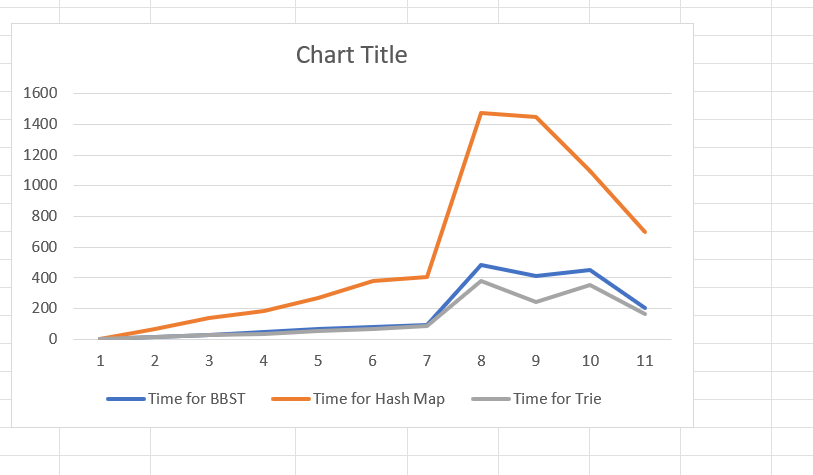


**Timing for searching words for all data structures.**

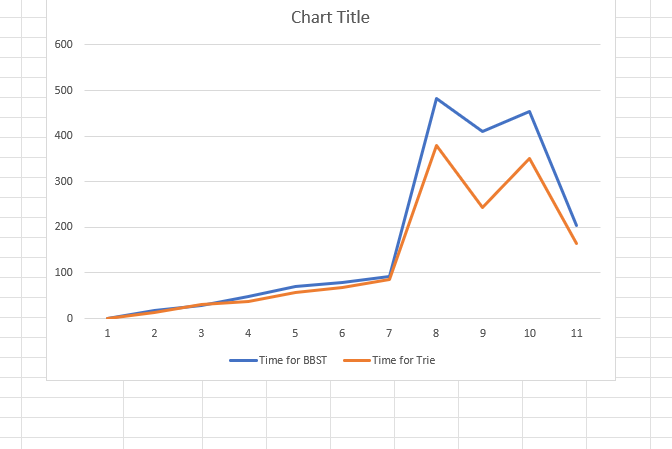


We can see that the linear list is the slowest of all data structures. It is so slow that it makes it look like the other data structures have similar times.

Now let’s exclude linear list and look at the comparison of other three data structures.



Finally, let’s compare trie and bbst.



We can see that trie performs better than the bst.

**CONCLUSION:**

In conclusion trie data structure performs best as its total time searching is less than all other data structures. After trie, bst performed better. On the third place is hashmap and last is linear list whose time as shown in the diagram increases linearly as the number of words in the file are increased.