**Putting Data Structures to the Test  
Saad Faruque, Mayank Sharma CSCI 323 - Analysis of Algorithms**

**Introduction**

In computer science there are numerous data structures which allow users to store and retrieve information in different ways and with different search algorithms. Though many data structures may seem similar or simply superior to others, it is important to note that many data structures are better fulfilling specific use cases varying on the used data and utilizing specific search algorithms. This being said, there are many ways to measure the efficiency of a data structure algorithm, one notably being how long it takes for the search algorithm to actually retrieve the information(Time). In this project we are going to utilize a total of 5 data structures, tries, HashTables, binary search trees, AVL trees, and red black binary search trees, as well as their efficiency measured in time for their associated algorithms for search/lookup. The aim is to find and compare the most efficient data structure with look up time from the 5 mentioned data structures.

**Algorithm Overview**

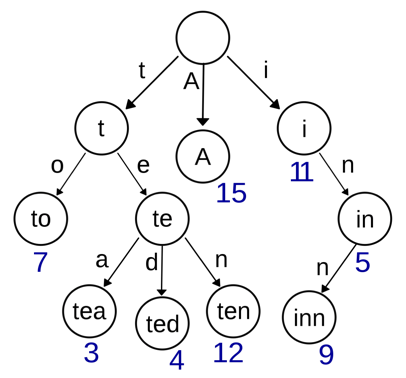
Each data structure chosen(tries, HashTables, binary search trees, AVL trees, and red black binary search trees) have not only different implementations but different ways of creation. In this project not only are we going to measure the efficiency, but we will give a brief way of possible implementation and overview of creation with all algorithms chosen.

**Tries:**

A trie is a type of search tree1 as well as an ordered tree data structure. Unlike a binary search tree, tries are not limited to only have two children nodes per parent node(​**figure 1**​). Each child node of a parent, when looking at strings in our case, is the next possible value(character) the string can have. Because of this, the time complexity for creating a trie is O(W\*L), where W is the number of words, and L is an average length of the word. Because of this, the same goes for searching for words in trie, where L is the steps for each of the W words.

page1image12849856

1 A search tree is a tree data structure used for locating specific keys from within a set. In order for a tree to function as a search tree, ​the key for each node must be greater than any keys in subtrees on the left, and less than any keys in subtrees on the right.



**HashTables:**

**Figure 1: A basic implementation of a trie**

A HashTable is a data structure that implements an associative array abstract data type, a structure that maps keys to values. A HashTable is always given a hash function(a function that decides what ‘encryption’ or function that the key will go through to make itself unique). A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found. One implementation of a hash function can be f(1), where the key is going to be itself.To find a value in a HashTable, the user will need to find the key in which the value was hashed. Since our implementation of the hash table is with chaining, our implementation stores the first record of each chain in the slot array itself(​**Figure 2**​). Because of this, creating a hash table(inserting into the data structure) takes O(n) time, where looking for the data takes O(1) time/constant as long as you know the respective key.

**Figure 2: An implementation of hash tables where collisions are dealt with chaining**

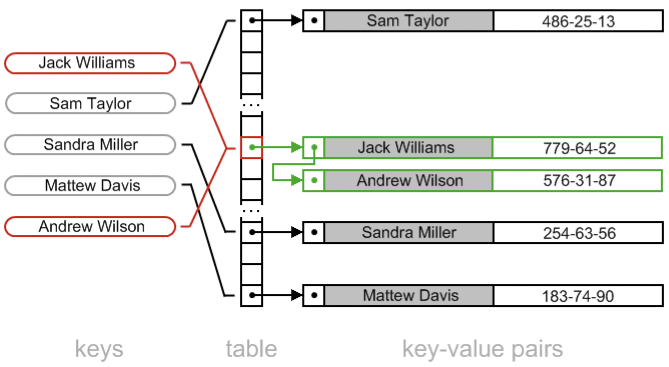
**Binary Search Tree(BST):**

A binary search tree(here called a BST) is a type of search tree which allow for best case fast lookup at O(logn) due to the methods used for node storage. A binary search tree implements a very famous (logn) search, represented in the name, binary search. A BST parent node has max 2 child nodes, where the left child is required to be of less value than the parent, and the right child is required to be of more value.Though due to the non-existing requirement of

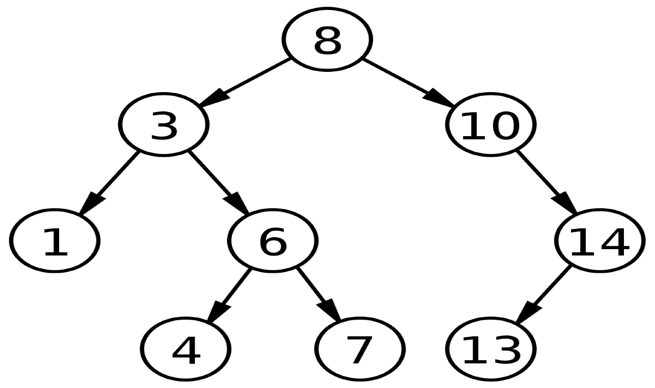
2  
a BST having to be balanced , the worst case of the BST lookup can be O(n) where n is the

height of the BST.

2

page2image12852160

​Trees which do not impose constraint on height difference between leaf nodes



**AVL Trees:**

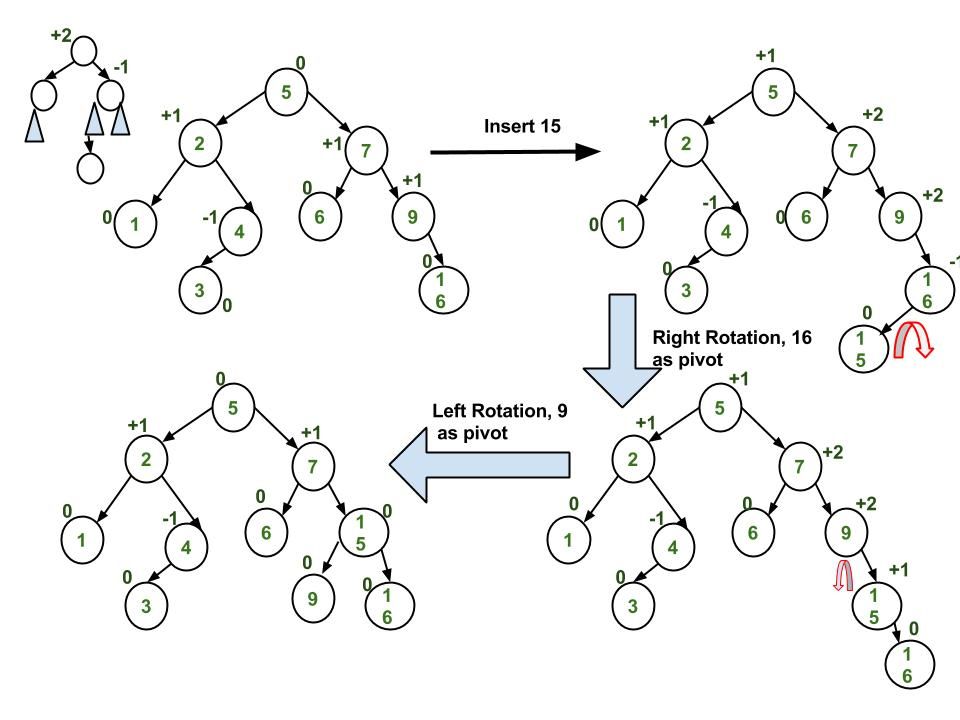
**Figure 3: Showing a balanced Binary Search Tree**

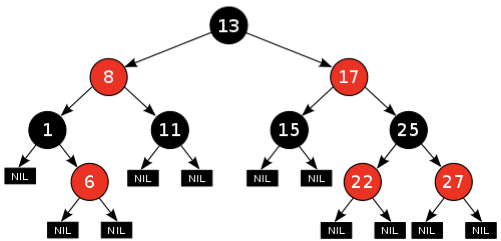
An AVL Tree (Adelson-Valsky and Landis) is a type of binary search tree which is self-balancing. In children node and parent node aspects(referring to which can be placed where), there are no differences, however the root node for the AVL tree is always causing the BST to be balanced. Due to this being the case, the worst search time complexity for the AVL tree is O(logn), where n is the amount of elements inside the tree. Insertion is also O(logn) due to its balancing. Balancing is done through a series of pivots on the node which involve a series of rotations on the nodes.

**Figure 4: We can see a self balancing BST(AVL Tree) where a right and left rotation is performed.**

**Red Black BST:**

A red-black binary search tree is also a type of binary search tree which is self balancing. Each node in the red-black BST has an extra bit, and that bit is often represented as its interpreted as the color of the node(red or black). These colors are used to aid with balancing during insertions and deletions of the tree. Though the balancing of the tree is not as strict as the AVL tree, it is good enough to guarantee a search time and insertion time of O(logn) where n is the amount of elements inside the tree.





**node is represented as either a red or black node**

**Implementation Considerations**

Our data is based off of a real life search relevant data (Popular NYC Baby names obtained from open city data) and a random concatenation of words based off of all the words utilized in all of Charles Dickens work.

Due to debate in group members in measuring, there were two implementations in measuring time in nanoseconds, though both were similar or roughly the same. In looking at efficiency of BST, hash tables, and tries, system time is measured(both start and finish) inside their relevant search functions. However in AVL and red-black BST, a StopWatch class is used to measure system time. Some time for StopWatch initialization should be taken when looking at results.

**Conclusion**

Though many people memorize search and insertion times of their favorite data structures associated with the specific search algorithm, many don’t put them to the test by actually implementing and timing their computer. This project was very informative, putting our knowledge to the test and confirming through implementation what we have learned throughout academia.

Our data structure which retrieved our data the quickest was our hash table. This did not come as a surprise as aforementioned hash tables retrieve at a constant time. This was also not a shock as though chaining was implemented, due to the randomness and variety of our data, chaining was most likely not stressed. Our second fastest data structure was the binary search tree. Looking at its relative search time complexity this also did not come as a shock(it was the one with the second fastest time complexity as well). This can also be due to the fact with the variety and randomness in our data, as random data tends to make for an already balanced binary search tree. Tying in at third and fourth place, the AVL tree and red-black BST seemed to

**Figure 4: A red black BST where every**

have very close and similar search time complexity. Both were self balancing, and the AVL trees more stricter balancing implementation did not come into play because of the data. What did come as a shock was the difference in search time of the AVL, BST, and red-black BST, however this can be due to the fact of implementation consideration(where a class was initialized for AVL and red-black BST). The slowest data structure relative to its search algorithm seemed to be with tries, as more data inputs went, the slower and worse the trie’s search time became.

**Search Run Information**

Using Data Source #1 (In source section)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **size(n)** | **Data Structure** | **Search term** | **Does contain?** | **Runtime for search** | **# compares** |
| **10 HashTable Madeline False 1,100 ns 1** | | | | | |
| **10 Trie Madeline False 66,300 ns 1** | | | | | |
| **10 BST Madeline False 9,100 ns 20** | | | | | |
| **10 AVL Madeline False 12,700 ns 12** | | | | | |
| **10 RB BST Madeline False 13,400 ns 10** | | | | | |
| **100 HashTable Geraldine True 4,900 ns 2** | | | | | |
| **100 Trie Geraldine True 58,000 ns 9** | | | | | |
| **100 BST Geraldine True 700 ns 2** | | | | | |
| **100 AVL Geraldine True 26,900 ns 21** | | | | | |
| **100 RB BST Geraldine True 9,900 ns 21** | | | | | |
| **1000 HashTable Mike False 7,200 ns 1** | | | | | |
| **1000 Trie Mike False 116,000 ns 2** | | | | | |
| **1000 BST Mike False 26,900 ns 268** | | | | | |
| **1000 AVL Mike False 33,800 ns 30** | | | | | |
| **1000 RB BST Mike False 11,900 ns 37** | | | | | |
| **10000 HashTable Brian True 10,400 ns 4** | | | | | |
| **10000 Trie Brian True 196,100 ns 5** | | | | | |
| **10000 BST Brian True 18,200 ns 150** | | | | | |
| **10000 AVL Brian True 24,000 ns 21** | | | | | |
| **10000 RB BST Brian True 1,600 ns 33** | | | | | |

Using Data Source #2( Info in Source section)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **size(n)** | **Data Structure** | **Search term** | **Does contain?** | **Runtime for search** | **# compares** |
| **10 HashTable fence False 900 ns 1** | | | | | |
| **10 Trie fence False 40,700 ns 1** | | | | | |
| **10 BST fence False 5,500 ns 12** | | | | | |
| **10 AVL fence False 13,700 ns 12** | | | | | |
| **10 RB BST fence False 25,600 ns 13** | | | | | |
| **1000 HashTable door True 7,300 ns 2** | | | | | |
| **1000 Trie door True 159,600 ns 4** | | | | | |
| **1000 BST door True 4,700 ns 28** | | | | | |
| **1000 AVL door True 40,500 ns 30** | | | | | |
| **1000 RB BST door True 13,300 ns 33** | | | | | |
| **10000 HashTable saad False 6,500 ns 1** | | | | | |
| **10000 Trie saad False 70,200 ns 3** | | | | | |
| **10000 BST saad False 5,000 ns 32** | | | | | |
| **10000 AVL saad False 23,000 ns 36** | | | | | |
| **10000 RB BST saad False 11,500 ns 31** | | | | | |
| **4.7M HashTable sunshine True 1,800 ns 2** | | | | | |
| **4.7M Trie sunshine True 164,700 ns 8** | | | | | |
| **4.7M BST sunshine True 7,500 ns 44** | | | | | |
| **4.7M AVL sunshine True 16,500 ns 45** | | | | | |
| **4.7M RB BST sunshine True 12,500 ns 45** | | | | | |

**Sources:**

Hash Table with chaining implementation:

https://www.sanfoundry.com/java-program-implement-hash-tables-chaining-list-heads/

Popular Baby Names(Data Source #1):  
NYC Open Data: ​https://data.cityofnewyork.us/Health/Popular-Baby-Names/25th-nujf/

Entire Work of Charles Dickens(Data Source #2): dickens.txt: ​https://introcs.cs.princeton.edu/java/data/

StopWatch Class:  
Code: ​https://algs4.cs.princeton.edu/home/

Knowledge of Algorithms: Algorithms by Robert Sedgewick:

Sedgewick, Robert, and Van Wyk Christopher J. ​*Fundamentals, Data Structures, Sorting, Searching*​. Addison-Wesley, 2009.

It ran slower than brute-force but that may be because of the implementation