Team Chevron Chevron
Fidelia Nawar & Annalise Sumpon
CSE2341: Data Structures

Final Project Analysis

9th December 2019

Introduction

For our final project, we were assigned to build a search engine that searches through corpuses of Supreme Court opinions based upon the query of the user. The engine was created with four major parts including the document parser/processor, query processor, index handler, and UI. We also had to implement the use of an inverted file index in order to keep track of each word's corresponding list of documents. This was performed through the implementation of two custom data structures: an AVL tree and a hash table with chaining without collisions. Within the UI, the user is given the option to choose whether to index into an AVL or a hash, and the program parses the document and inserts words into the respective structure. This report serves to analyze which structure is more efficient for the purposes of our program and for the operations of inserting/searching in general.

AVL Trees

AVL trees are an implementation of an overarching family of data structures called trees which are a hierarchical structure composed of nodes pointing to other nodes. A root node has pointers to its child nodes that each have subtrees and each child node has a parent node. AVL trees fall in the subclass of binary trees where each node has no more than two child nodes. An AVL can be compared to a binary search tree in which the left and right subtrees of the node are less than/greater than respectively to the original node. Essentially, an AVL tree is a BST but one that is self balancing, and the heights of the two child subtrees cannot differ by more than one. If this is the case, then rebalancing needs to occur which is performed by rotating the nodes one of four ways.

These types of trees have a logarithmic time complexity where searching, inserting, and removing all take O(log n) in average case and O(n) worst cases. The operations for this data structure are more complex than the operations of the hash table simple due to the need to continuously rebalance the tree with added data. The benefit of using the AVL tree was that we were able to have persistency when inserting/removing an element from the tree in O(log N) space-and-time complexity. This allowed us not only to obtaining a new tree, but also get to retain the old tree.

Hash Tables

Hash tables are a data structure that consists of an associative array that maps keys to values using a hashing function. This function is used to determine which index of the array a certain value should be mapped to. In an ideal situation, the hashing function assigns each key to unique indices, or else collisions occur, but we were able to overcome this using separate chaining and map multiple values to the same key. This process entails each index having a linked list of keys and their corresponding list of values mapped to that specific index. This prevented the issue of collisions and allowed us to have faster performance due to the inverse relationship between the number of collisions and size of table (larger size of table means fewer collisions).

The efficiency for operations of hash tables are constant time close to O(1) for inserting/searching due to the instant lookup of each value by implementing the use of the hash function. This was beneficial for our program because by being able to quickly insert into the hash without needing to be concerned about rebalancing factors, there was a much greater time efficiency compared to that of the AVL tree.

Analysis

In order to test the efficiency and performance between the AVL Tree and Hash Table, we collected data across varying data sizes of 100 documents, 1500 documents, and 20000 documents on the operations of insertion and searching. We decided to cap our maximum document size to 20000 due to the fact that the AVL Tree was taking a significant amount of time to insert and collecting multiple samples would have required a sufficient amount of time. As depicted in the graphs, we can see how hash tables performed much faster overall in each scenario for both operations.

Looking at the insertion graphs from Figures 1.1, 1.2, 1.3, we can see how the two structures performed almost similarly when inserting words from 100 documents, but this trend was drastically changed as larger number of documents were added. AVL Trees proved to be progressively slower, with an average insertion time of 21.96 seconds for 1,500 documents and 2070 seconds for 20,000 documents. On the other hand, the average insertion time for the hash table from 1,500 documents was 2.89 seconds and for the 20,000 documents data set 4.93 seconds. This proves the drastic difference in inserting words from larger data sets and how the hash table is performing much more efficiently.

When looking at the searching capabilities between the two data structures as depicted in Figures 2.0-2.2, we once again see how hash tables are more efficient. We chose to search the same query for each trial of each data set size for purposes of consistency, with the phrase being: “and court unit not trial.” As we can see, the searching functions are only slightly different in speed for the smaller data set but the average searching time for a medium/large data set for the AVL tree was 1.822 seconds and 2.825 seconds respectively, whereas for the hash table, those values were 0.403 seconds and 0.963 seconds. This proves how fast hash tables are not only able to insert, but also search due to the fast access using the hashing functionality. Overall, we noticed a trend of the hash table outperforming the AVL tree.

Our conclusion is that the hash table provides faster insertion and searching access than the AVL tree as shown by the data, and overall more efficient functionality especially for larger data sets.

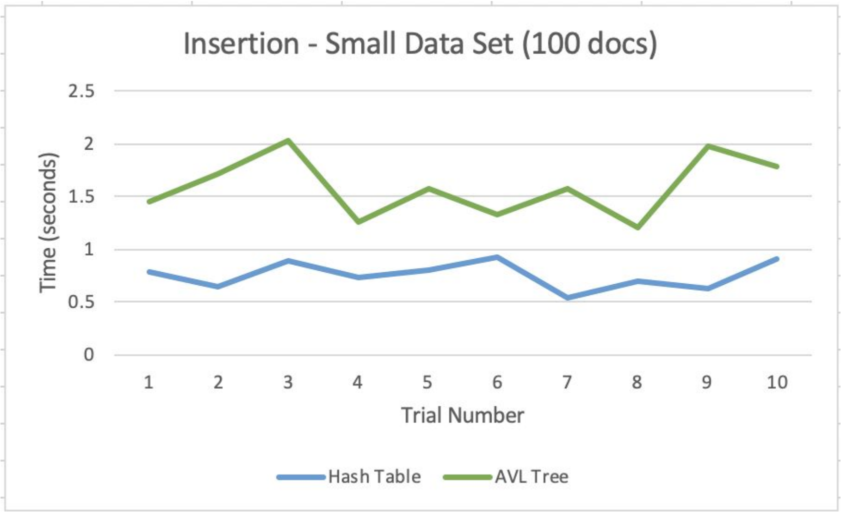


Figure 1.0

Depicting the trends in performance for the two structures for inserting words from a data set size of 100 documents.

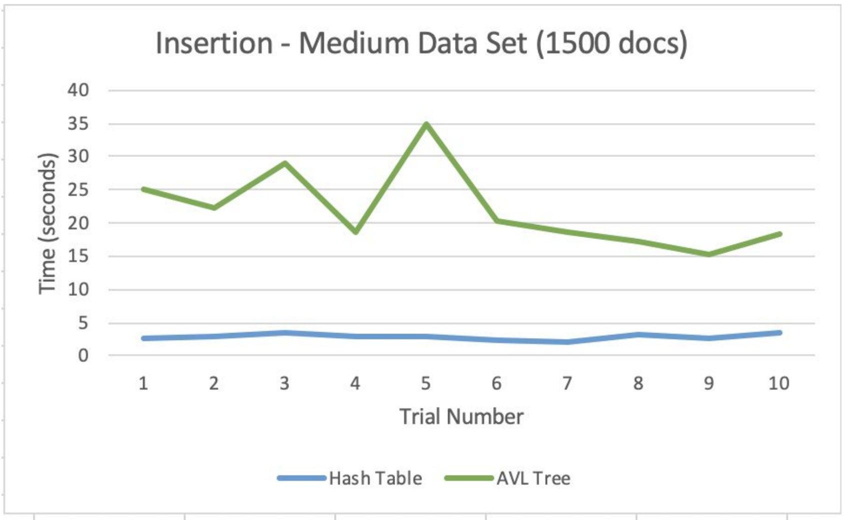


Figure 1.1

Depicting the trends in performance for the two structures for inserting words from a data set size of 1,500 documents.

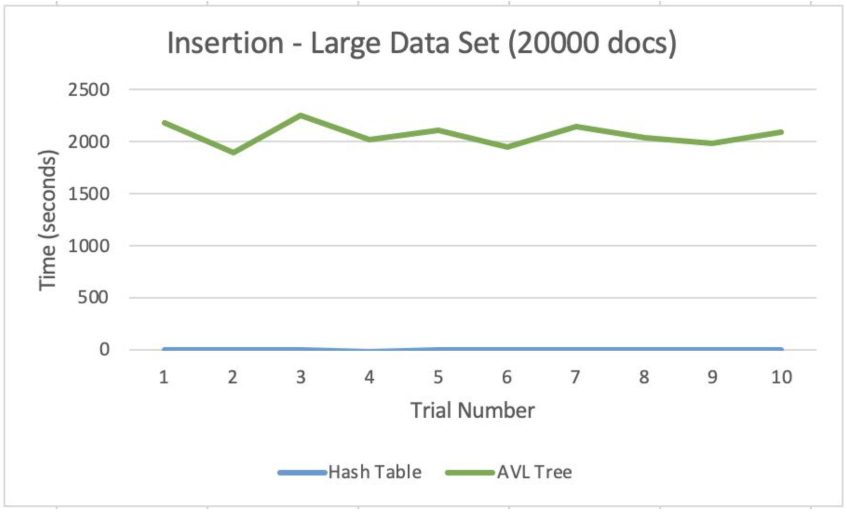


Figure 1.3

Depicting the trends in performance for the two structures for inserting words from a data set size of 20,000 documents.

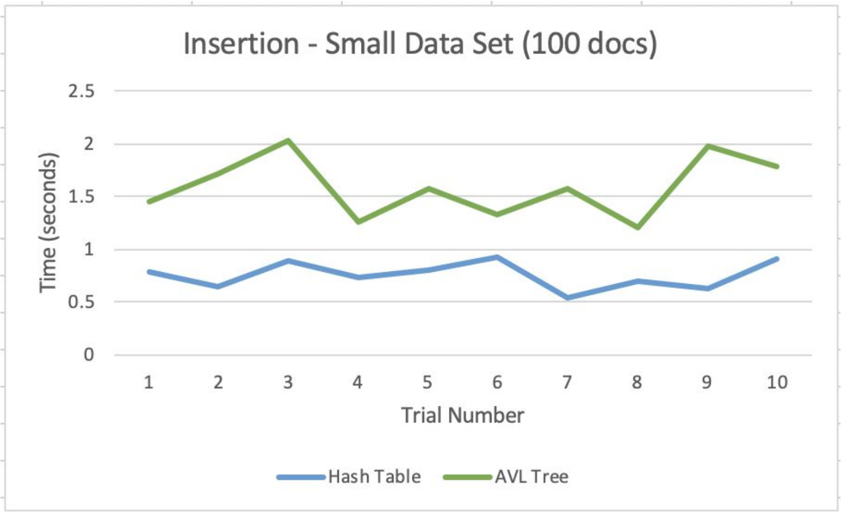


Figure 2.0

Depicting the trends in performance for the two structures for searching words from a data set size of 100 documents.

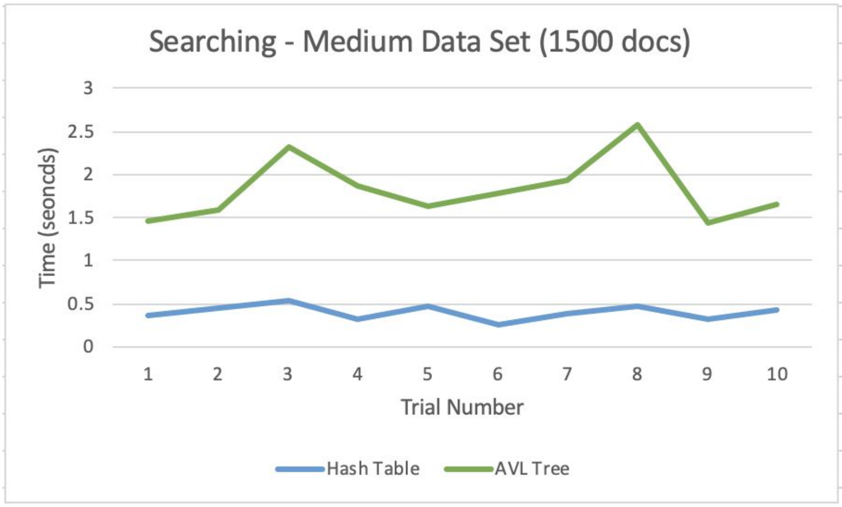


Figure 2.1

Depicting the trends in performance for the two structures for searching words from a data set size of 1500 documents.

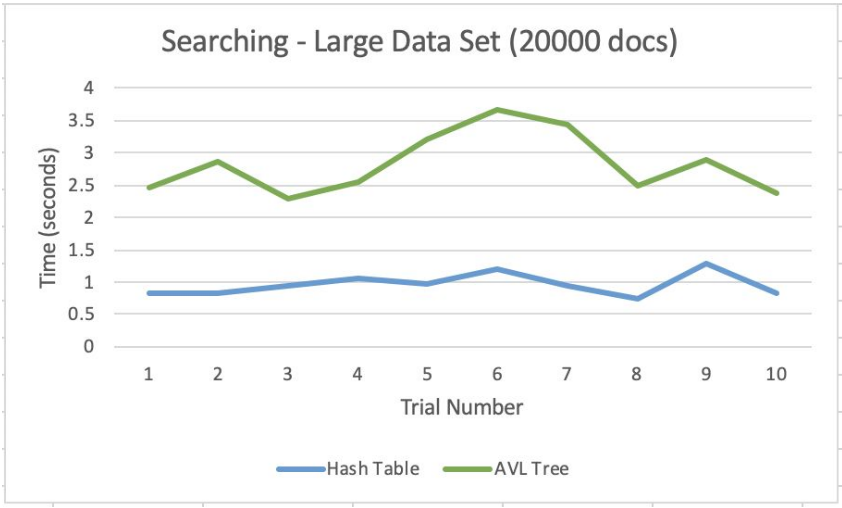
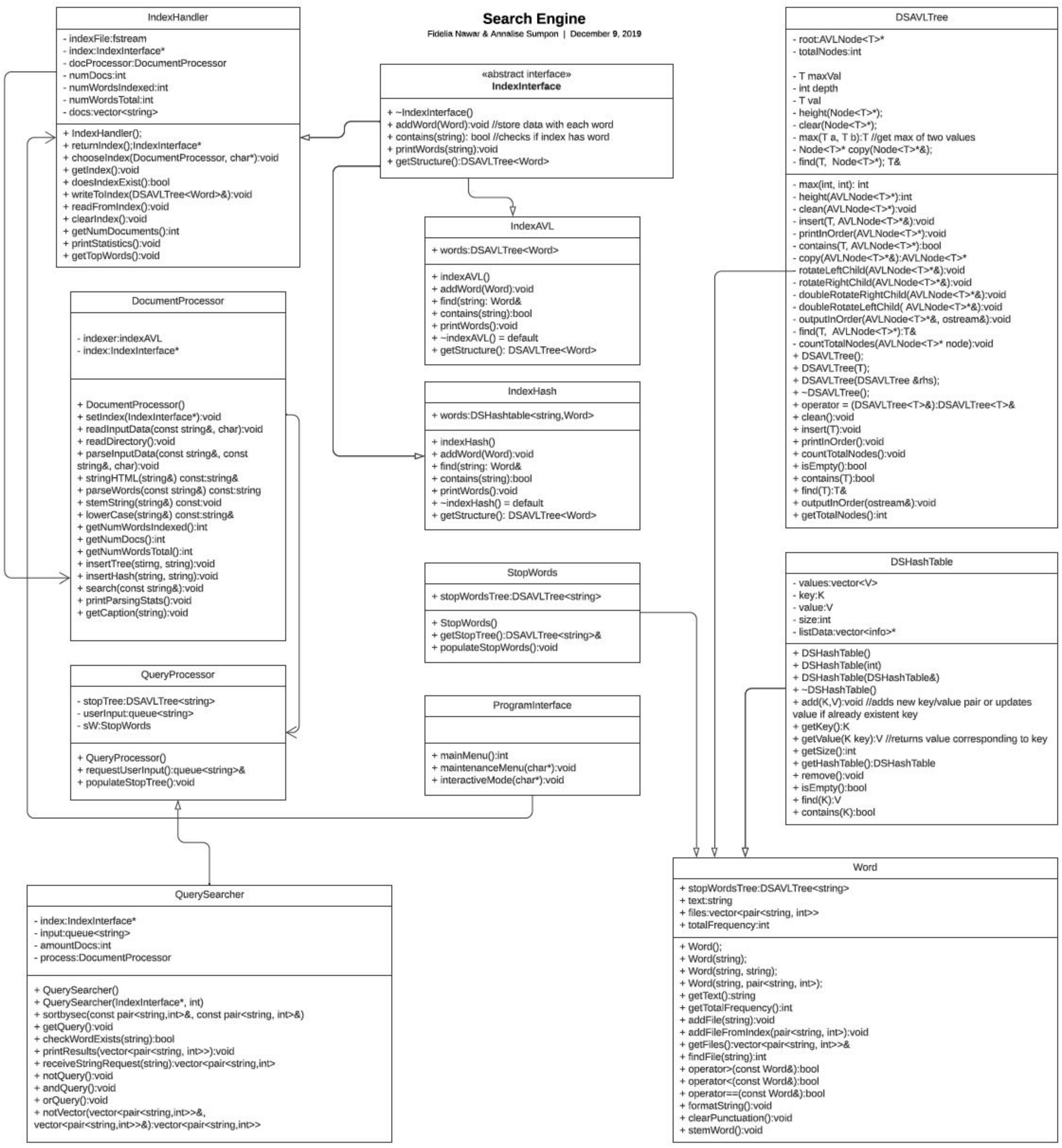


Figure 2.2

Depicting the trends in performance for the two structures for searching words from a data set size of 20000 documents.

Updated UML Class Diagram


Search Engine
Fidelia Nawar & Annalise Sumpon | December 9, 2019
IndexHandler
- indexFile:fstream
- index:IndexInterface*
- docProcessor:DocumentProcessor
- numDocs:int
- numWordsIndexed:int
- numWordsTotal:int
- docs:vector<string>
+ IndexHandler();
+ returnIndex();IndexInterface*
+ chooseIndex(DocumentProcessor, char*):void
+ getIndex():void
+ doesIndexExist():bool
+ writeToIndex(DSAVLTree<Word>&):void
+ readFromIndex():void
+ clearIndex():void
+ getNumDocuments():int
+ printStatistics():void
+ getTopWords():void
«abstract interface»
IndexInterface
+ ~IndexInterface()
+ addWord(Word):void //store data with each word
+ contains(string): bool //checks if index has word
+ printWords(string):void
+ getStructure():DSAVLTree<Word>
IndexAVL
+ words:DSAVLTree<Word>
+ indexAVL()
+ addWord(Word):void
+ find(string: Word&
+ contains(string):bool
+ printWords():void
+ ~indexAVL() = default
+ getStructure(): DSAVLTree<Word>
IndexHash
+ words:DSHashtable<string,Word>
+ indexHash()
+ addWord(Word):void
+ find(string: Word&
+ contains(string):bool
+ printWords():void
+ ~indexHash() = default
+ getStructure(): DSAVLTree<Word>
DocumentProcessor
- indexer:indexAVL
- index:IndexInterface*
+ DocumentProcessor()
+ setIndex(IndexInterface*):void
+ readInputData(const string&, char):void
+ readDirectory():void
+ parseInputData(const string&, const string&, char):void
+ stringHTML(string&) const:string&
+ parseWords(const string&) const:string
+ stemString(string&) const:void
+ lowerCase(string&) const:string&
+ getNumWordsIndexed():int
+ getNumDocs():int
+ getNumWordsTotal():int
+ insertTree(stirng, string):void
+ insertHash(string, string):void
+ search(const string&):void
+ printParsingStats():void
+ getCaption(string):void
StopWords
+ stopWordsTree:DSAVLTree<string>
+ StopWords()
+ getStopTree():DSAVLTree<string>&
+ populateStopWords():void
QueryProcessor
- stopTree:DSAVLTree<string>
- userInput:queue<string>
- sW:StopWords
+ QueryProcessor()
+ requestUserInput():queue<string>&
+ populateStopTree():void
ProgramInterface
+ mainMenu():int
+ maintenanceMenu(char*):void
+ interactiveMode(char*):void
QuerySearcher
- index:IndexInterface*
- input:queue<string>
- amountDocs:int
- process:DocumentProcessor
+ QuerySearcher()
+ QuerySearcher(IndexInterface*, int)
+ sortbysec(const pair<string,int>&, const pair<string, int>&)
+ getQuery():void
+ checkWordExists(string):bool
+ printResults(vector<pair<string, int>>):void
+ receiveStringRequest(string):vector<pair<string,int>
+ notQuery():void
+ andQuery():void
+ orQuery():void
+ notVector(vector<pair<string,int>>&, vector<pair<string,int>>&):vector<pair<string,int>>
DSAVLTree
- root:AVLNode<T>*
- totalNodes:int
- T maxVal
- int depth
- T val
- height(Node<T>*);
- clear(Node<T>*);
- max(T a, T b):T //get max of two values
- Node<T>* copy(Node<T>*&);
- find(T, Node<T>*); T&
- max(int, int): int
- height(AVLNode<T>*):int
- clean(AVLNode<T>*):void
- insert(T, AVLNode<T>*&):void
- printInOrder(AVLNode<T>*):void
- contains(T, AVLNode<T>*):bool
- copy(AVLNode<T>*&):AVLNode<T>*
- rotateLeftChild(AVLNode<T>*&):void
- rotateRightChild(AVLNode<T>*&):void
- doubleRotateRightChild(AVLNode<T>*&):void
- doubleRotateLeftChild(AVLNode<T>*&):void
- outputInOrder(AVLNode<T>*&, ostream&):void
- find(T, AVLNode<T>*):T&
- countTotalNodes(AVLNode<T>* node):void
+ DSAVLTree();
+ DSAVLTree(T);
+ DSAVLTree(DSAVLTree &rhs);
+ ~DSAVLTree();
+ operator = (DSAVLTree<T>&):DSAVLTree<T>&
+ clean():void
+ insert(T):void
+ printInOrder():void
+ countTotalNodes():void
+ isEmpty():bool
+ contains(T):bool
+ find(T):T&
+ outputInOrder(ostream&):void
+ getTotalNodes():int
DSHashTable
- values:vector<V>
- key:K
- value:V
- size:int
- listData:vector<info>*
+ DSHashTable()
+ DSHashTable(int)
+ DSHashTable(DSHashTable&)
+ ~DSHashTable()
+ add(K,V):void //adds new key/value pair or updates value if already existent key
+ getKey():K
+ getValue(K key):V //returns value corresponding to key
+ getSize():int
+ getHashTable():DSHashTable
+ remove():void
+ isEmpty():bool
+ find(K):V
+ contains(K):bool
Word
+ stopWordsTree:DSAVLTree<string>
+ text:string
+ files:vector<pair<string, int>>
+ totalFrequency:int
+ Word();
+ Word(string);
+ Word(string, string);
+ Word(string, pair<string, int>);
+ getText():string
+ getTotalFrequency():int
+ addFile(string):void
+ addFileFromIndex(pair<string, int>):void
+ getFiles():vector<pair<string, int>>&
+ findFile(string):int
+ operator>(const Word&):bool
+ operator<(const Word&):bool
+ operator==(const Word&):bool
+ formatString():void
+ clearPunctuation():void
+ stemWord():void