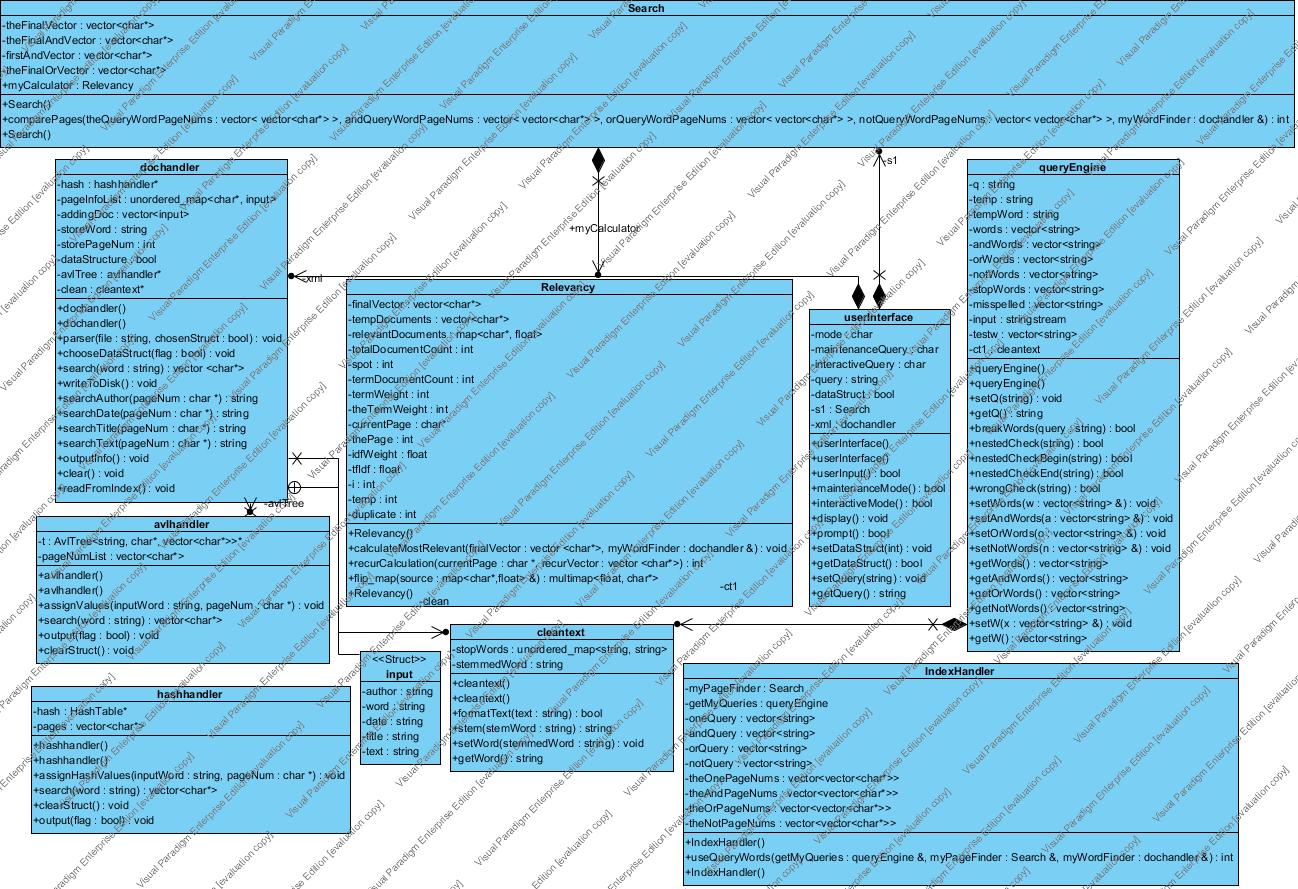
**UPDATED UML DIAGRAM**



**CLASS DOCUMENTATION/DESCRIPTION**

**avlhandler**

This class handles all functions that go in and out of the avl tree.  This primarily includes inserting values into the AVL Tree, outputting the contents,writing the contents to disk, and searching through the data structure.  This make it easier for the avl tree to be accessed from multiple sources including dochandler and our search function.

**avltree**

The avltree class implements what is called in avlhandler and is templated in order to make it easier to implement.  It also can be easily used multiple times thanks to our templating!

**cleantext**

This class formats words so that they are simplified and will match the query words that are also sent through cleantext.  This include removing stop words(words that are common, but unneeded) removing punctuation marks at the end of words, and making all letter lower case.  It also throws out any unnecessary characters like chinese and (#,%,@...).  The stop words are all loaded into an unordered map in order to increase search speed and prevent linearly searching through an array.

**dochandler**

This is one of the most important classes due to the fact that it parses the data using rapid XML, calls cleantext to format the words, and passes the data off to the data structure handlers to be indexed. It also keeps track of other data pulled from the document like author, date, title, and a copy of the unparsed text. It acts as the main hub for all information coming and going within our program  and helps in consolidating the multiple variables and functions that are passing and storing data.

**hashhandler**

Hashhandler is important because it controls everything that happens within the hashtable.  This includes searching, storing data, clearing the index, and storing the data structure to disk.

**hashtable**

Hashtable essentially implements what is called in hashhandler, but also sets the rehash size for the hashtable.  This is important because if we did not have this the program would constantly need to rehash thus slowing down the indexing process.

**indexHandler**

indexHandler receives vectors of words, and depending on the query searched, some of them are populated and others are not. It then passes these words to the search function in dochandler and stores the vectors of document numbers the words are contained on in another vector. Upon receiving and storing all of this information, the vectors containing vectors of page numbers are passed off to be compared, contrasted, and combined.

**search**

search uses conditional statements to determine which block of code to execute. It then takes whichever vector of vectors are populated and combines them based off of the query entered in order to end with one final vector containing only the documents that are relevant to the search that can be used to determine relevancy. If they are vectors in the OR vector, all documents are used because all are relevant. If they are vectors in the AND vector, only documents that are in all the vectors within the big AND one are included. If they are vectors in the NOT vector, if the final vector has any documents also included in the not vectors then the document number is deleted from the final vector. The final vector made from these conditions is then sent to relevancy.

**relevancy**

relevancy takes the final vector and performs a term frequency – inverse document frequency calculation on each document number one time in order to determine the top 15 documents to output to the user. Term frequency is the number of times a word occurs in a specific document. Inverse document frequency is an inverse function of the number of documents in which it occurs. The higher the tf-idf number, the more relevant the document is.

**userinterface**

userinterface combines all classes in an order reliant on user’s input. It provides the user with an option to choose between Maintenance and Interactive mode. Maintenance mode gives user the ability to add documents and/or to clear the index, while Interactive mode asks a user to select which data structure he wants to access and gives him an option to input his query. Interactive mode binds together all other crucial components of a search engine: parser, index handler, and query engine. Switch statements and while loops are primary control statements used to guide a user through interface’s options.

**queryengine**

queryengine’s primary function is to separate the words entered by user and sort them in groups with respect to Boolean prefixes. Apart from normal queries, this query engine has an ability to handle nested queries by putting a Boolean operator, followed by words in parentheses. It also stems the words and checks if there are any stop words present in the query and automatically eliminates them. Since there are three boolean operators (“AND”, “OR”, “NOT”) used by user to search for words, query engine acts according to them. If it finds “AND” operator it stores words that follows into andWords vector, if it hits “OR” the words go into orWords vector, and if a “NOT” operator is used, the words following it are stored in a notWords vector. In case there are no Boolean prefixed operators, the words are stored in word vector. In order to stem the words, it has to communicate with “cleantext.h” class and use its “formatText()” function. Once query engine is done sorting words into vector, it enables passing their content to other functions by using getters and setters.

**AVL & Hash Table Comparison**

For this project, we implemented both an AVL Tree and a Hash Table.  What we first noticed was that the AVL Tree was much slower than the Hash Table for large files, but for small files it is actually more efficient. The AVL Tree has a Big O time complexity of 0(log(n)), this is because the AVL tree stores everything in the tree and each tree node can have up to two children. When we read in the smaller files there was no difference in time between the two.  However, once we read in the 786 MB file we saw huge time gaps of almost 3 minutes between the Hash Table and AVL Tree!  The AVL Tree grows in size as we store more data and thus it must compare itself with more and more nodes to determine where it must be stored. Furthermore, if the tree becomes unbalanced(which occurs commonly) then the computer must rebalance the tree in order to maintain proper sorting functionality.  The Hash Table on the other hand is preferred especially for large data sets. It has a time complexity of 0(1) at its best and a worst case of 0(n) giving it a constant  lookup, insert, and delete time. Granted, the downside to it is that it requires more memory and as you get more data you increase your odds of collisions, but what you lose in memory you make up for in speed. One of the key things we discovered with the hash table was that it required us to resize it constantly.  We realized that the speed dramatically increased when we set the rehash size to a higher number thus preventing the table from constantly needing to be resized. Overall, both of these data structures are good for certain tasks.  However, for what we are doing, I would choose the Hash Table over the AVL Tree any day!

ML DIAGRAM GOES HERE