AVL\_Tree:

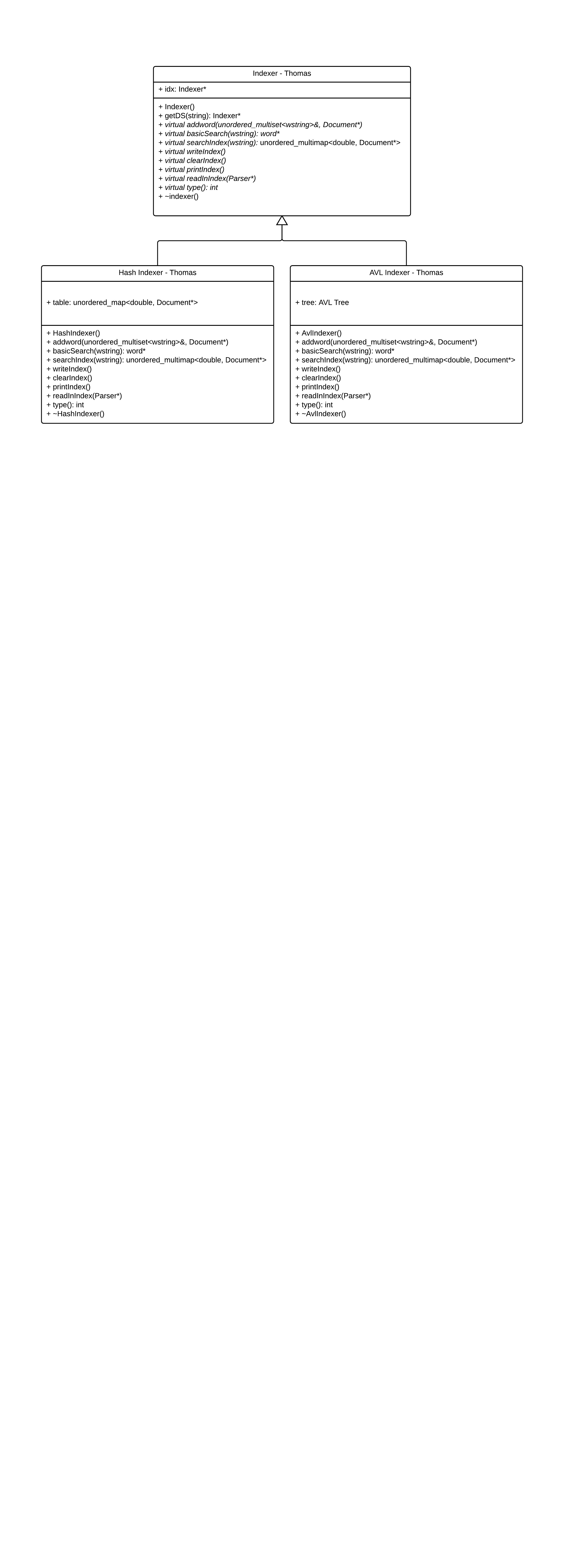
This class is needed to create the AVL Tree data structure. We needed to create an AVL Tree data structure because it was a requirement of the project. One of the major advantages to an AVL tree is that it is self balancing so when searching the index, worst case is a O(log n) time complexity. This tree is implemented specifically around our word objects. It is technically a templated tree, but we went into the tree structure and modified the functions so that they worked specifically for our needs during the project. One of the most needed changes was a good way to write the index to Disk, we accomplished this with a recursive write function inside of the tree itself. 99% of this tree is not our own implementation and we cannot take credit for its implementation.

Stemming & associated classes:

For this project we used the Oleander stemming library, which is a porter based stemmer. Stemming is a fundamental function of the parser. We had to have a reliable stemmer so that the same word would be stemmed and inputted into the index the same way no matter what ending it has. We chose to go with this stemmer because it was one of the simplest to implement into our own code and decently quick while parsing the data. This stemmer is generic and will work with many languages but we only needed it to work with the English language so we simply included the functions necessary to stem English words.

Indexer:

The Indexer class is responsible for creating and accessing the search index. The search index was required to be created as either an AVL Tree or as a Hash Table, so both implementations of the index had to be accounted for in the Indexer class. To accomplish this, the Indexer class was a purely virtual function that had the two classes HashIndexer and AvlIndexer inherit from it. This use of polymorphism allowed for other functions to only call a generic instance of an Indexer, but have the correct indexing function (HashIndexer vs AvlIndexer) be called during runtime. The fist major task of the Indexer is to create the search index, which it accomplishes though the addWord function. Depending on which type of index is being created, the addWord function will create word objects out of the words in the input files, and store those objects in either a Hash Table or an AVL Tree. In addition to creating a search index from scratch, the Indexer class was responsible for writing the search index to disk, and then recreating the index from that file at a later time. The ability to write the index out to a file that is saved on the hard drive of the computer is extremely important because it allows for the index to be quickly re-built at any time after the index was built from scratch once. This saves time both when testing and implementing the index on a regular basis, but also for the search index to be quickly re-built if the search engine crashes during use. The Indexer function interacts heavily with the Word class and Document class since those two classes make up the objects that are stored in the search index.

Query Engine:

 The Query Engine is responsible for processing the search queries that the user enters and then computing the search results for that specific query. The Query Engine first determines what kind of query has been made (whether it is a single word search, or a search that contains one of the boolean prefixes), and then calls an instance of the Indexer to search the index for the words that are in the query. Next, the Query Engine uses the Relevancy Ranking value for each document (which is calculated by the Word class) to rank documents in order of decreasing relevancy, with the most relevant document to the search being the first document listed in the result list that is returned to the user. The Query Engine then uses the AND, OR, and NOT functions (or a combination of the three) in conjunction with the searchSingle or searchMultiple and searchDouble functions to determine what the final list of results will be. The Query Engine is called from the User Interface to process the query that the user enters to the User Interface. The Query Engine then calls the Indexer to search through the index, and ultimately returns the final list of search results back to the User Interface.

Parser:

The parser is the heart and soul of our program. The parser’s fundamental use is to read in the XML dump from Wikipedia and discern the keywords from the text. Without the parser, all junk words would be put into the index causing a major waste of memory space and a waste of searching time because there would be so many words that would never be searched. We implemented our parser so that all you need to do is create an instance of parser and give it an index type to fill. Then simply call the parse dump function and pass it a string with the file name that you wish to use to either create or augment the index and the parser will do the rest. Our parser accomplishes this by reading in all of the text in the provided file into a string, then it separates the text into individual pages and passes that to the page parser. The page parser then breaks the page down further into specifically the headers that we wish to use. When the headers have been discerned we pick out the important information such as the Document title, author, and date. We then pick out the text and send it into a space delimited parsing algorithm. The algorithm takes the text, breaks it down by character to find words and then compares the word first against an English dictionary, then against a list of stop words, and finally against a set of previously stemmed words to save on time. If it makes it passed all of these levels, the word is stemmed and added to the index with a pointer to the document object that holds all of its relevant information.



SearchEngineControl:

SearchEngineControl is the real brain of the search engine. It is necessary to keep all of the classes in order for execution during the system. The control system allows for the user to execute as many different types of modes as they would like to consecutively. The biggest benefit of the SearchEngineControl system is that it encapsulates all of the classes needed to run the search engine so setting up the engine without having the inner workings of a search engine exposed to the main method is possible. It is very easy to implement and once an instance is created, the search engine control creates its own instances of the parser, indexer, and the user interface.

UserInterface:

The user interface is the pretty face that the program presents to the user. The user interface is necessary because it takes away the necessity to continually change hard coded values in the code itself. A user interface is merely a convenient way for someone who has never used our system before to execute useful commands. Our Interface has 3 different modes: maintenance, interactive, and stress test. Maintenance mode is a basic mode that is simply used by the creator to add or subtract documents from the index. It comes in handy especially when you wish to write the current index to disk so that it can be reopened later. The Interactive mode is the most interesting mode because it allows a user to either change index types, or enter a properly formatted Boolean and get search results that they can open and view the document themselves. Finally we have the stress test mode. This mode is primarily for the developer because it allows for the creation of a simple text document that will control how the index reacts. This mode comes in especially handy for testing the efficiency and power of different types of indexes. The user interface is quite simple in and of itself because it simply requires the entry of a number here and there to accomplish queries and more complex actions with the blink of an eye. It is also fool proof because misplaced keystrokes will simply return a witty rebuttle and the requirement to enter your choice again.



Word:

The Word class is responsible for holding all the data relevant to a word that is stored in the search index, and is used to create the Word objects that are stored in the search index. Each Word object holds a list of the documents in which that specific word has been referenced, as well as the Relevancy Ranking of each of those documents with respect to that word, and the number of times the word was referenced in each document and in all the documents in total. The Word class manages the list of documents that the word is referenced in and is responsible for adding additional documents to the list (via the addDocument function) and handling the occurrences of documents being added more than once (via the duplicate function). The Word class is also responsible for calculating the Relevancy Ranking value for each document. By using the data stored in the Word object, the setRank function is able to set a Relevancy Ranking value for each document held inside the Word object. This value represents the relative importance of the document to the word based on the TFIDF (Term Frequency, Inverse Document Frequency) of that word inside that specific document. The TFIDF value is, approximately, a representation of how often the word appears in a certain document relative to how many times it appears in the document as a whole. This TFIDF value allows for each document stored inside a Word object to be ranked according to its relevancy to that word, which allows for the Search Engine to deliver a list of results that is tailored to the search terms that the user entered in the query.

Document:

We created a custom container for all of the data surrounding a single document in the XML dump. The document objects hold the title, author, date, document number, and file location for the document text that was written to disk while parsing. These containers are used to keep track of all relevant information that a word object would possibly need to reference to either search for Booleans or to print out the document text. The objects themselves are fairly simple to implement, they simply have a constructor and getters for each part of the information held inside of the object itself. They can be implemented by themselves without the need of any other objects being created with them.



AVL Tree vs. Hash Table

Over the course of the semester and the search engine project itself, we have come to the conclusion that anything standard library is far superior to a data structure that is not a part of the standard library. We went back and forth many times and thought that something designed specifically for a task must be far superior. We also threw in the fact that to become a part of the standard library the data structure had to go through very rigorous testing to prove that it is the best option to suit almost any situation presented. In the end we came to the realization that the Hash table (part of the standard library) was far superior to the AVL Tree. We put lots of thought into why we decided to go primarily with the Hash table, and the most important distinguisher was the placement time to put an item in the index. AVL Trees fall short of the Hash table mostly because of the sheer amount of times that an AVL Tree must rebalance because the words will not be put into the index in the perfect order. The hash table has much more consistent placement because it uses hash functions that are specific enough to place a specific key where very few, if any, other keys already exist in that location. Also the hash table has far more impressive searching times. The AVL Tree has an impressive search time at O(log n), but it does not stand a chance against the Hash table because the Hash table has the ability to search in constant time because of the hashing functions that take specific keys to very specific locations that are easy to identify. We quantified these findings by using the stress test mode and first reading in the dump into a hash indexer and then an AVL tree, our results were quite amazing, it took roughly 4 and a half minutes to read in all of the words into an AVL tree, while it merely took about 3 minutes and 15 seconds to read the same file into a hash table. That is roughly a 33% increase in input speed. The next test was reading in the files that were written to disk. The Hash indexer took roughly 17-20 seconds, and the AVL Tree took roughly 25 – 30 seconds to perform the same action. This somewhat large discrepancy on such a small scale may have to do with the way that the AVL Tree data was output in the first place, it may not have been optimal. The final test was searching. Both Indexes performed about the same in the searching category taking much less than a second to return results, this is largely in part because both have such impressive searching time complexities. Overall, AVL Trees may have an advantage if the input words are put in exactly the way that the tree needs them to be so that it never has to reorder itself. Our final conclusion is that when given a choice; use a hash table to store data that you need to search whether small or big because it has such an impressive ability to input and search for keys.

USERS MANUAL:

The User interface is very simple to access, because the entire library is able to be accessed through one instantiation of a controller for the search engine. To Start any mode of the User interface begin by using these two steps:

1. Declare and instance of search engine control as such:

SearchEngineControl\* controller = new SearchEngineControl();

1. Run the engine by using the following function call:

controller -> runTheEngine();

To then enter Interactive mode:

1. Build and run the program
2. Choose the beginning type of index that you wish to build by either entering 1 for a Hash Table or 2 for and AVL Tree
3. Choose how you would like to fill either index. You can either fill by reparsing the entire XML dump (which is only useful when trying to create an updated persistent index) or by loading a persisting index which was parsed in a prior execution of the program.
4. Then you must enter option 2, which is the option for the Interactive user interface.
5. You are now in Interactive mode and you may enter the number of any of the functions to either change the index type or conduct a query to determine if a word is in the index. From these actions you can view specific documents within the index itself based on your search.

To Enter Maintenance mode:

1. Build and run the program
2. Choose the beginning type of index that you wish to build by either entering 1 for a Hash Table or 2 for and AVL Tree
3. Choose how you would like to fill either index. You can either fill by reparsing the entire XML dump (which is only useful when trying to create an updated persistent index) or by loading a persisting index which was parsed in a prior execution of the program.
4. You must now enter 1 to enter maintenance mode
5. You may now freely explore maintenance mode, which allows you to augment the index, clear the index, and write the existing index to disk so that it can be reread later

DISCLAIMER:

If you intend to implement this search engine into any other library, please include the Stop word text file and the English dictionary text file that are a part of the build folder. These are necessary parts of the engine because they are required to get decent English results quickly.

Stress Test Mode Commands

* Write Index to Disk (WID) – no arguments
  + Takes the created AVL Tree/Hash Table and writes the structure to disk memory in a format that can be easily read to recreate the structures when the program is rerun
* Rebuild Data Structures From Disk (PID) – 1 argument (index type 1- Hash, 2 - Tree
  + Uses a file created when the program terminated to quickly recreate all of the data structures needed
* Recreate Index (RCI) – 1 argument (new filename to build index from)
  + Clears the current data structures and allows for the index to be rebuild using a different input file
* Query using And, Or, and Not (Q) – 1 argument (properly formatted boolean query)
  + Allows the index to be searched for certain keys to determine how long the program takes to search
* Clear Index (CI) – no arguments
  + Deletes all data from the Index so that it can be recreated
* Add to Index (AI) – 1 argument (filename to be added)
  + Allows the use of another input file to append more words to the index from more xml documents
* Build Index (BI) – no arguments
  + Tells the program to build an index from a specific input file, and specifies whether to build a Hash Table or an AVL Tree
* Switch Index Type (SIT) – no arguments
  + Allows for the program to switch from a Hash Table to and AVL Tree or vice versa