```
class DocumentParser
{
private:
    int pageCount;
    std::ifstream xmlStream;
    std::string xmlName;
    StemAndStopRemoval * stemStopRemoval;
    Index * index;
public:
    DocumentParser();
    StemAndStopRemoval * getStemmer();
    void Parse();
    void ReadInXML(std::string fileName);
    void ParsePage(std::string text, int pageID);
    Index * GetIndex();
    std::string retrievePage(int targetID);
    std::string retrieveAttribute(int targetID, int attributeID);
    std::string getFileName(int);
    void renamePageFiles();
};
```

- Parses the XML file using rapid XML
- Stems and deletes stop words from going into the Index
- Sends the words to the index of choice
- Is able to retrieve pages back to output to the user
    - It does this by creating files of each individual XML document and then when we search for that ID, it finds the page with that ID

```
class HashTable
{
public:
    HashTable();
    void createHashTable();

    void insert(const std::string & term,const int ID);
    long hash(std::string);
    AVL_Tree * getBucket(int keyValue);
    void hashToFrequencyVector();
    std::vector<std::pair<std::string,int>> * GetFreqVector();

private:
    int buckets;
    std::vector<std::pair<std::string,int>> frequencyVector;
    AVL_Tree ** indexHashTable;

};
```

- This is the non-STL Hash Table that will be one of our index data structures
- It works by creating an array of bucket size
- A pointer to an AVL Tree will be contained in each element of our hash table
- In order to insert into the hash table we first need to hash the string in order to decide where it goes
- Once we have decided where it goes, we go to that location in the hash table array and insert it into an AVL tree
- It will be able to calculate the top 50 words in the index

```
class AVL_Tree
{
private:

    class AVL_Node
    {
    public:
        std::string element;
        std::vector<int>logCount;
        AVL_Node * left;
        AVL_Node * right;
        int height;
        AVL_Node();
        AVL_Node(const std::string & theElement, const int id, AVL_Node * lt, AVL_Node
* rt, int h=0);
    };
    std::vector<std::pair<std::string,int>> frequencyVector;
    void treeToFrequencyVector(AVL_Tree ::AVL_Node *& rootNode);
    void insert(const std::string &term, const int id, AVL_Node * & t);
    void rotateWithLeftChild(AVL_Node * & k2);
    void rotateWithRightChild(AVL_Node * & k1);
    void doubleWithLeftChild(AVL_Node * & k3);
    void doubleWithRightChild(AVL_Node * & k1);

    AVL_Node * getNode(std::string nodeName);
    int height(AVL_Node * t) const;
    int max(int lhs, int rhs);
    void PrintTree(AVL_Node *&rootNode);
    AVL_Node * root;
public:
    AVL_Node * getRoot();
    AVL_Tree(): root(NULL){}
    void printTree();
    void TreeToFrequencyVector();
    AVL_Node * GetNode(std::string nodeName);
    void Insert(const std::string & x, const int ID);
    std::vector<std::pair<std::string,int>> * GetFreqVector();
};
```

- This is our AVL Tree index
- Our AVL Tree consists of AVL nodes that you can see created in the class above
- Our AVL nodes consist of an integer that is going to be the ID number, as well as a vector of integers that is going to be where that word appears in terms of page ID's. So say the word computer appears on pages 11 and 13, except it appears on 13 twice. The vector will look like <11, 13, 13>.
- The AVL node will also have a pointer to its left child, as well as pointer to its right child, all while keeping track of its height as well.
- The actual AVL Tree class will also be keeping track of the top 50 words that are getting inserted, as words are being inserted into the index, the vector is dynamically changing itself so that the top 50 words are at the top of the vector.
- The AVL tree also has an insert that is passed a term and an ID and from that information it is able to insert it into the correct position in the AVL tree.

- An AVL tree is also in charge or dynamically altering itself if the tree is found to be unbalanced.
- We also created a method that will be used to get the pointer to a node, once we have that information we are able to extract the word as well as the pages that it appears on
- A print tree method was created in order to show that we were creating an actual AVL tree.

```
class Index
{
private:
    AVL_Tree * masterAVL;
    HashTable * masterHash;
    int dataStructure;
    int totalPageCount;
    int totalWordCount;
    std::vector<std::pair<std::string,int>> masterVector;
    std::deque<std::pair<std::string,int>> masterDeque;
public:
    Index(int i);
    int getDataStructureID();
    void incrementPageCount();
    void incrementWordCount();
    void insertItem(std::string term, int pageID);
    void ClearIndex();
    void writeIndexToDisc();
    void writeStatsToDisc();
    void readInSaveFile(int structureID);
    void readInStatsFile();
    int getPageCount();
    int getWordCount();
    AVL_Tree *& getMasterTree();
    HashTable *&getMasterHash();
    std::vector<int>errorVec;
    std::vector<int> * findWord(std::string wordToFind);
    std::vector<std::pair<std::string,int>> *
sortByFrequency(std::vector<std::pair<std::string,int>> * frequencyVector);
};
```

- This is the class that is going to be in charge of indexing the information passed to it by the document parser.
- It will decide what data structure to use then create the associated index
- It is going to increment both the page count, as well as the word count so that we can access that information when asked for the statistics.
- It's find word method is going to go into the associated index and find a node and then it will output a vector of integers that will be used by the query processor
- It's going to be able to sort the frequency vector that holds the top 50 words
- It will be able to create a persisted index in terms of both the index and the stats

```
class QueryProcessor
{
public:
    QueryProcessor();
    void query(DocumentParser* docParser);
    bool isCommand(std::string);
    std::vector<std::pair<int, double> > AND(std::vector<int>*, std::vector<int>*);
    std::vector<std::pair<int, double> > OR(std::vector<int>*, std::vector<int>*);
    std::vector<std::pair<int, double> > NOT(std::vector<int>*, std::vector<int>*);
    std::vector<std::pair<int, double> > AND2(std::vector<int>*, std::vector<int>*,
std::vector<int>*);
    std::map<int,int> ANDnot(std::vector<int>*, std::vector<int>*);
    std::vector<std::pair<int, double> > NOTand(std::map<int, int> ,
std::vector<int>*);
    std::vector<std::pair<int, double> > rank(std::map<int, int>);
    void PrintSearchResults(std::vector<std::pair<int, double>> finalVector,
DocumentParser *docParser);

};
```

- This is the query processor, it is going to be able to take in a word or phrase sent to the program by the user and output the page numbers, as well as a little bit of information of what is on that page
- It does this by breaking the query up into a vector so that it can look at its size, then decide which path to take
- After it knows the queries size, it looks to see if there are Boolean commands that need to be dealt with, if there are, then it sends the vector of pages to a method that will alter them depending on the Boolean command.
- It needs to be able to rank these pages numbers depending on how many times that word appears on that page using the TFIDF statistic
- Finally, it takes that final ranked vector and outputs it to the console, after it is outputted; the user has options of what they want to do with their query.

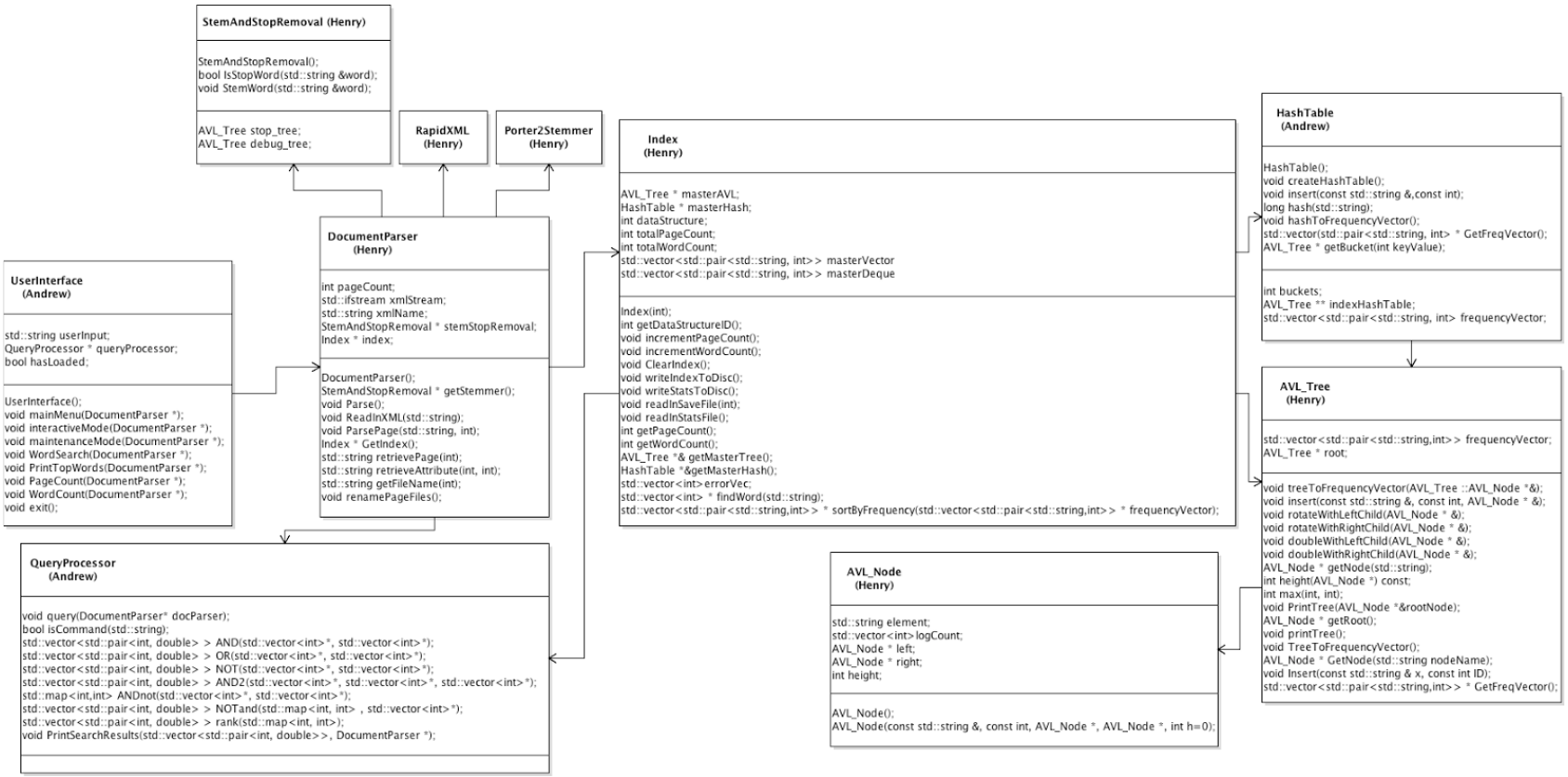
```
class StemAndStopRemoval
{
private:
    AVL_Tree stop_tree;
    AVL_Tree debug_tree;
public:
    StemAndStopRemoval();
    bool IsStopWord(std::string
&word);
    void StemWord(std::string &word);
};
```

- This class will be in charge of stemming words, as well as checking to see if it is a stop word
- It does this by comparing every word that is passed in to see if it is a stop word, if it is not a stop word then it then stems the word if it is necessary and then the word is ready to be indexed

```
class UserInterface
{
public:
    UserInterface();
    void mainMenu(DocumentParser *docParser);
    void interactiveMode(DocumentParser
*docParser);
    void maintenanceMode(DocumentParser
*docParser);
    void WordSearch(DocumentParser *docParser);
    void PrintTopWords(DocumentParser *docParser);
    void PageCount(DocumentParser *docParser);
    void WordCount(DocumentParser *docParser);
    void exit();


private:
    bool hasLoaded;
    std::string userInput;
    QueryProcessor * queryProcessor;
};
```

- This is the first thing that the user is going to encounter, it will ask them what they want to do
- If a user wants to go into interactive mode they will we able to search for a word, check statistics, or decide what data structure to load into
- If they want to enter maintenance mode they will be able to add a file to the index
- It will be what is in charge of sending the query to the query processor
- Finally, it will be able to exit the program


StemAndStopRemoval (Henry)
StemAndStopRemoval();
bool IsStopWord(std::string &word);
void StemWord(std::string &word);
AVL_Tree stop_tree;
AVL_Tree debug_tree;
RapidXML (Henry)
Porter2Stemmer (Henry)
DocumentParser (Henry)
int pageCount;
std::ifstream xmlStream;
std::string xmlName;
StemAndStopRemoval * stemStopRemoval;
Index * index;
DocumentParser();
StemAndStopRemoval * getStemmer();
void Parse();
void ReadInXML(std::string);
void ParsePage(std::string, int);
Index * GetIndex();
std::string retrievePage(int);
std::string retrieveAttribute(int, int);
std::string getFileName(int);
void renamePageFiles();
UserInterface (Andrew)
std::string userInput;
QueryProcessor * queryProcessor;
bool hasLoaded;
UserInterface();
void mainMenu(DocumentParser *);
void interactiveMode(DocumentParser *);
void maintenanceMode(DocumentParser *);
void WordSearch(DocumentParser *);
void PrintTopWords(DocumentParser *);
void PageCount(DocumentParser *);
void WordCount(DocumentParser *);
void exit();
QueryProcessor (Andrew)
void query(DocumentParser* docParser);
bool isCommand(std::string);
std::vector<std::pair<int, double> > AND(std::vector<int>*, std::vector<int>*);
std::vector<std::pair<int, double> > OR(std::vector<int>*, std::vector<int>*);
std::vector<std::pair<int, double> > NOT(std::vector<int>*, std::vector<int>*);
std::vector<std::pair<int, double> > AND2(std::vector<int>*, std::vector<int>*, std::vector<int>*);
std::map<int,int> ANDnot(std::vector<int>*, std::vector<int>*);
std::vector<std::pair<int, double> > NOTand(std::map<int, int> , std::vector<int>*);
std::vector<std::pair<int, double> > rank(std::map<int, int>);
void PrintSearchResults(std::vector<std::pair<int, double>>, DocumentParser *);
Index (Henry)
AVL_Tree * masterAVL;
HashTable * masterHash;
int dataStructure;
int totalPageCount;
int totalWordCount;
std::vector<std::pair<std::string, int>> masterVector
std::vector<std::pair<std::string, int>> masterDeque
Index(int);
int getDataStructureID();
void incrementPageCount();
void incrementWordCount();
void ClearIndex();
void writeIndexToDisc();
void writeStatsToDisc();
void readInSaveFile(int);
void readInStatsFile();
int getPageCount();
int getWordCount();
AVL_Tree *& getMasterTree();
HashTable *&getMasterHash();
std::vector<int> errorVec;
std::vector<int> * findWord(std::string);
std::vector<std::pair<std::string,int>> * sortByFrequency(std::vector<std::pair<std::string,int>> * frequencyVector);
HashTable (Andrew)
HashTable();
void createHashTable();
void insert(const std::string &,const int);
long hash(std::string);
void hashToFrequencyVector();
std::vector(std::pair<std::string, int> * GetFreqVector();
AVL_Tree * getBucket(int keyValue);
int buckets;
AVL_Tree ** indexHashTable;
std::vector<std::pair<std::string, int> frequencyVector;
AVL_Tree (Henry)
std::vector<std::pair<std::string,int>> frequencyVector;
AVL_Tree * root;
void treeToFrequencyVector(AVL_Tree ::AVL_Node *&);
void insert(const std::string &, const int, AVL_Node * &);
void rotateWithLeftChild(AVL_Node * &);
void rotateWithRightChild(AVL_Node * &);
void doubleWithLeftChild(AVL_Node * &);
void doubleWithRightChild(AVL_Node * &);
AVL_Node * getNode(std::string);
int height(AVL_Node *) const;
int max(int, int);
void PrintTree(AVL_Node *&rootNode);
AVL_Node * getRoot();
void printTree();
void TreeToFrequencyVector();
AVL_Node * GetNode(std::string nodeName);
void Insert(const std::string & x, const int ID);
std::vector<std::pair<std::string,int>> * GetFreqVector();
AVL_Node (Henry)
std::string element;
std::vector<int> logCount;
AVL_Node * left;
AVL_Node * right;
int height;
AVL_Node();
AVL_Node(const std::string &, const int, AVL_Node *, AVL_Node *, int h=0);

[USER MANUAL]

Welcome to TWSearchEngine, a search engine that works in conjunction with XML documents in order to search for words in the XML documents. There are many "magic wandy stuff" that happens in the background, but as a user you will mostly be working in conjunction with the UserInterface.

[USER INTERFACE]
When the program starts up you will have an option of what mode you want to go into, the modes are **INTERACTIVE** mode, or **MAINTENANCE** mode. Finally, when you are finished with the program you have a way to **EXIT**.

[INTERACTIVE MODE]
Inside of interactive mode you will have your options laid out to you. They are the following: **WORD SEARCH, TOP 50 WORDS, PAGE COUNT, WORD COUNT, AVL TREE, HASH TABLE** or **MAIN MENU**.

[WORD SEARCH]
Word search will allow you to create a properly formatted word or phrase that will then search the index for that word or phrase. The properly formatted boolean queries are the following: [word], AND [word] [word], OR [word] [word], [word] NOT [word], AND [word] [word] [word], and finally AND [word] [word] NOT [word].

[TOP 50 WORDS]
This will output the top 50 words in the index.

[PAGE COUNT]
Page count will output the number of pages indexed.

[WORD COUNT]
Word count will output the number of words indexed.

[AVL TREE]
This will load the index as an AVL Tree.

[HASH TABLE]
This will load the index as a hash table.

[MAIN MENU]
This will return you back to the main menu.

[MAINTENANCE MODE]
Inside of maintenance mode you will have to options, you will be able to **ADD FILE**, **CLEAR INDEX** and **MAIN MENU**.

[ADD FILE]

This will add a file to be indexed. This needs to be a properly formatted XML document

[CLEAR INDEX]
This will clear the current index.

[MAIN MENU]
This will return you back to the main menu.

[EXIT]
Exits the program.