CSC 212 Project - Fall 2024

A Simple Search Engine

Name	Student ID	Section	Division of work
Noura Alamro	444200941	444126	Everything related to: -ArrayList class -Document class -DocList class -LinkedList class -VocabList class -AVL classes - project report: performance analysis
Shooq Alawdah	444201083	444126	Everything related to: -Boolean Retrieval types methods -Term Retrieval types methods -BST classes -Main class - Project report
Dalal AlOtaibi	443201040	444126	Everything related to: -Ranked retrieval methods -Ranked Retrieval class -Ranked Document class

Table of contents:

Project Overview

Our project aims to build up a simple search engine that can index, retrieve, and rank documents from given queries. The engine also supports simple Boolean queries (AND, OR) and includes term frequency in the relevance ranking, as our project uses ADT List to create an index and inverted index for building the search engine and implements a BST and AVL to optimize the performance of the inverted index.

Key Features:

1. Data Handling

Data handling begins in the “main” class first, as it sends CSV file, TXT file, vocabList object, vocabAVL object, and vocabBST object to the “DocList” class to be managed.

In the “DocList” class, documents from the CSV file are read then each are processed to extract the words and documents IDs filtering out stop words, punctuation, and numbers. The “DocList” class then creates an object of the “Document” class and adds the string linked list document object to the ArrayList of documents.

The “Document” class receives the processed data and initializes it to a string array so the constructor can loop through the array to index a string list, the vocabList object, the vocabAVL object, and the vocabBST object. Furthermore, counting the tokens too.

2. Vocabulary Management

3. Scalability through Diverse Data Structures (ex: linked list, BST, AVL)

The “VocabList”, “VocabAVL”, and “VocabBST” classes all collectively manage the vocabulary words with their associated document IDs. As each class uses a different data structure for this goal:

- “VocabList” uses a linked list to maintain the vocabulary (inverted index).
- “VocabAVL” uses a self-balancing AVL Tree.
- “VocabBST” uses a binary search tree.

4. User-Friendly Interface (Menu) & Search Options

The “main” class provides a simple menu. It allows users to choose between various retrieval methods:

- Term Retrieval: using the methods “RetrieveIndex “, “RetrieveInvertedIndexe”, “RetrieveBST” as each method utilities a different approach such as lists, inverted lists, and BST.

- Boolean Retrieval: using logical operations (AND, OR, AND-OR) through the “BooleanAVL”, “BooleanBST”, “BooleanLinkedList”, and “BooleanIndex” methods to filter documents based on user queries.

- Ranked Retrieval: retrieves and ranks documents based on the user’s query using the methods “rankedRetrieval”, “sortRankedDocuments”, and “sortedInsert”.

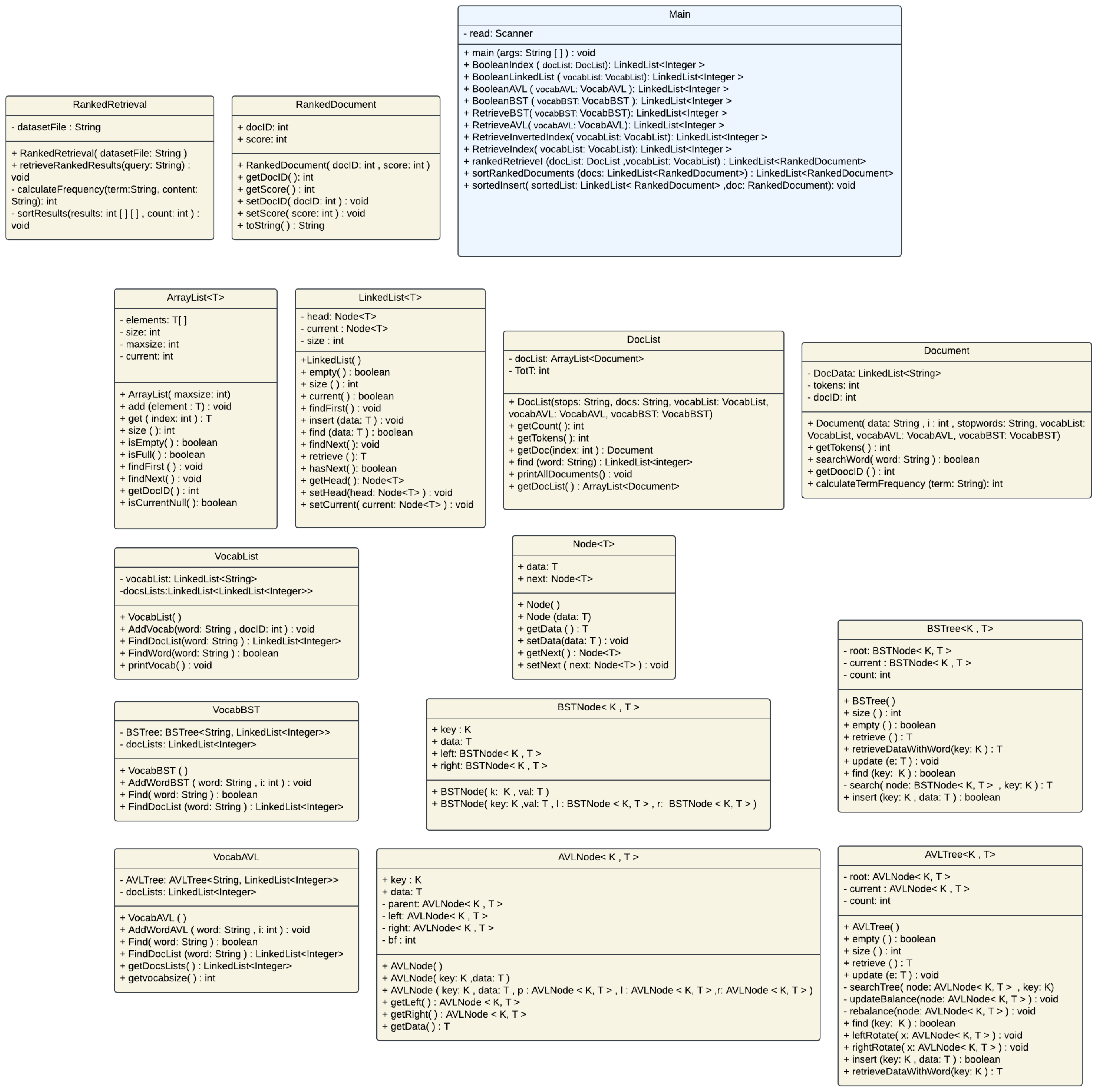
- Indexed Documents: prints all documents with the number of words in them.

-Indexed tokens: prints all the words with the number of documents they appear in.
- Retrieval of each of the number of tokens and vocabulary.

5. Robust Input Handling

Our engine also Implements exception handling to manage potential issues (e.g., file not found, parsing errors) when the user inputs for service selection and during the reading and processing of stop words and documents to ensure the program responds appropriately to unexpected inputs.

Design Diagram:


Main
- read: Scanner
+ main (args: String [ ]) : void
+ BooleanIndex (docList: DocList): LinkedList<Integer >
+ BooleanLinkedList (vocabList: VocabList): LinkedList<Integer >
+ BooleanAVL (vocabAVL: VocabAVL): LinkedList<Integer >
+ BooleanBST (vocabBST: VocabBST): LinkedList<Integer >
+ RetrieveBST(vocabBST: VocabBST): LinkedList<Integer >
+ RetrieveAVL(vocabAVL: VocabAVL): LinkedList<Integer >
+ RetrieveInvertedIndex(vocabList: VocabList): LinkedList<Integer >
+ RetrieveIndex(vocabList: VocabList): LinkedList<Integer >
+ rankedRetrievel (docList: DocList ,vocabList: VocabList) : LinkedList<RankedDocument>
+ sortRankedDocuments (docs: LinkedList<RankedDocument>) : LinkedList<RankedDocument>
+ sortedInsert(sortedList: LinkedList< RankedDocument> ,doc: RankedDocument): void
RankedRetrieval
- datasetFile : String
+ RankedRetrieval(datasetFile: String)
+ retrieveRankedResults(query: String) : void
- calculateFrequency(term:String, content: String): int
- sortResults(results: int [ ] [ ] , count: int) : void
RankedDocument
+ docID: int
+ score: int
+ RankedDocument(docID: int , score: int)
+ getDocID( ): int
+ getScore( ) : int
+ setDocID(docID: int) : void
+ setScore(score: int) : void
+ toString( ) : String
ArrayList<T>
- elements: T[ ]
- size: int
- maxsize: int
- current: int
+ ArrayList(maxsize: int)
+ add (element : T) : void
+ get (index: int) : T
+ size ( ): int
+ isEmpty( ) : boolean
+ isFull( ) : boolean
+ findFirst ( ) : void
+ findNext( ) : void
+ getDocID( ) : int
+ isCurrentNull( ): boolean
LinkedList<T>
- head: Node<T>
- current : Node<T>
- size : int
+LinkedList( )
+ empty( ) : boolean
+ size ( ) : int
+ current( ) : boolean
+ findFirst( ) : void
+ insert (data: T) : void
+ find (data: T) : boolean
+ findNext( ): void
+ retrieve ( ): T
+ hasNext( ): boolean
+ getHead( ): Node<T>
+ setHead(head: Node<T>) : void
+ setCurrent(current: Node<T>) : void
DocList
- docList: ArrayList<Document>
- TotT: int
+ DocList(stops: String, docs: String, vocabList: VocabList, vocabAVL: VocabAVL, vocabBST: VocabBST)
+ getCount( ): int
+ getTokens( ): int
+ getDoc(index: int) : Document
+ find (word: String) : LinkedList<integer>
+ printAllDocuments() : void
+ getDocList( ) : ArrayList<Document>
Document
- DocData: LinkedList<String>
- tokens: int
- docID: int
+ Document(data: String , i : int , stopwords: String, vocabList: VocabList, vocabAVL: VocabAVL, vocabBST: VocabBST)
+ getTokens( ) : int
+ searchWord(word: String) : boolean
+ getDoocID ( ) : int
+ calculateTermFrequency (term: String): int
VocabList
- vocabList: LinkedList<String>
-docsLists:LinkedList<LinkedList<Integer>>
+ VocabList( )
+ AddVocab(word: String , docID: int) : void
+ FindDocList(word: String) : LinkedList<Integer>
+ FindWord(word: String) : boolean
+ printVocab( ) : void
Node<T>
+ data: T
+ next: Node<T>
+ Node( )
+ Node (data: T)
+ getData ( ) : T
+ setData(data: T) : void
+ getNext( ) : Node<T>
+ setNext (next: Node<T>) : void
BSTree<K , T>
- root: BSTNode< K, T >
- current : BSTNode< K , T >
- count: int
+ BSTree( )
+ size ( ) : int
+ empty ( ) : boolean
+ retrieve ( ) : T
+ retrieveDataWithWord(key: K) : T
+ update (e: T) : void
+ find (key: K) : boolean
- search(node: BSTNode< K, T > , key: K) : T
+ insert (key: K , data: T) : boolean
VocabBST
- BSTree: BSTree<String, LinkedList<Integer>>
- docLists: LinkedList<Integer>
+ VocabBST ( )
+ AddWordBST (word: String , i: int) : void
+ Find(word: String) : boolean
+ FindDocList (word: String) : LinkedList<Integer>
BSTNode< K , T >
+ key : K
+ data: T
+ left: BSTNode< K , T >
+ right: BSTNode< K , T >
+ BSTNode(k: K , val: T)
+ BSTNode(key: K ,val: T , l : BSTNode < K, T > , r: BSTNode < K, T >)
VocabAVL
- AVLTree: AVLTree<String, LinkedList<Integer>>
- docLists: LinkedList<Integer>
+ VocabAVL ( )
+ AddWordAVL (word: String , i: int) : void
+ Find(word: String) : boolean
+ FindDocList (word: String) : LinkedList<Integer>
+ getDocsLists( ) : LinkedList<Integer>
+ getvocabsize( ) : int
AVLNode< K , T >
+ key : K
+ data: T
- parent: AVLNode< K , T >
- left: AVLNode< K , T >
- right: AVLNode< K , T >
- bf : int
+ AVLNode( )
+ AVLNode(key: K ,data: T)
+ AVLNode (key: K , data: T , p : AVLNode < K, T > , l : AVLNode < K, T > ,r: AVLNode < K, T >)
+ getLeft( ) : AVLNode < K, T >
+ getRight( ) : AVLNode < K, T >
+ getData( ) : T
AVLTree<K , T>
- root: AVLNode< K, T >
- current : AVLNode< K , T >
- count: int
+ AVLTree( )
+ empty ( ) : boolean
+ size ( ) : int
+ retrieve ( ) : T
+ update (e: T) : void
- searchTree(node: AVLNode< K, T > , key: K)
- updateBalance(node: AVLNode< K, T >) : void
- rebalance(node: AVLNode< K, T >) : void
+ find (key: K) : boolean
+ leftRotate(x: AVLNode< K, T >) : void
+ rightRotate(x: AVLNode< K, T >) : void
+ insert (key: K , data: T) : boolean
+ retrieveDataWithWord(key: K) : T

Performance Analysis:

In the “main” method of the class “main “, it calls the “BooleanIndex (DocList docList)“ method to retrieve results in the form of an integer linked list using Boolean retrieval. The time complexity of the “BooleanIndex(DocList docList) “method is (O (n * m)), where (n) is the number of terms in the user's search query and (m) is the total number of documents in the “DocList “object. As the method iterates through the array of terms provided by the user, it searches through every document by calling the “find (String word)” method from the “DocList“ class for each term. The “find (String word)” method, in turn, includes a while loop that retrieves document IDs using the “search Word (String word) “method from the “Document “class. Additionally, there is a nested loop in the “BooleanIndex (DocList docList)“ method to manage logical combinations of terms, such as "and" and "or."

Likewise, for the “BooleanAVL(VocabAVL vocabAVL)”, “BooleanBST(VocabBST vocabBST)”, and “BooleanLinkedList(VocabList vocabList)” methods. To compare the Boolean Retrieval process of the different data structures of the index, inverted index, BST, and AVL. We calculated the Big-ohs of the methods used in each of the data structures that retrieve data (find (String word) and Find DocList (String word)). The “DocList “class (representing indexed data structure) had the highest complexity, making it the least efficient with a time complexity of O (n * m), especially as the number of documents and stop words increases.

The Inverted Index provides a better solution, with retrieval methods like “FindDocList(String word)” and “FindWord(String word)” running in time O(n), suitable for the Boolean Retrieval process but less efficient for larger vocabularies.

On the other hand, BST and AVL Trees had the logarithmic complexities of O (log n) for searching and finding associated document lists, making them highly efficient for large amounts of data. Whereas AVL Trees have an advantage in insertions due to its balancing properties and its time complexity always being O (log n). In contrast, a BST can sometimes become unbalanced, leading to a worst-case performance of O(n), which isn’t ideal, but BST is a better option compared to the inverted index because of its average case while the inverted index time complexity is always O(n).

In conclusion, AVL and BST structures are the best options for quick searches while DocLists (index) may be sufficient for simpler document tasks, and Inverted Indices serve well in document retrieval systems but with certain limitations.

Note: below are the time complexities of all the methods in each of the data structures.

Tables of Time Complexities:

Index: DocList

n = the number of documents read from the file, m = the number of stop words

Method	**Big(O)**
getCount( )	**O (1)**
getTokens( )	**O (1)**
getDoc(int index)	**O (1)**
find (String word)	**O(n*m)**
printAllDocuments ( )	**O(n)**
getDocList( )	**O (1)**

Inverted index: VocabList

n = the number of words in vocabulary, m = the number of average number of documents associated with a word in the vocabulary

Method	Big(O)
AddVocab (String word, int docID)	O(n+m)
FindDocList (String word)	O(n)
FindWord(String word)	O(n)
PrintVocab( )	O(n)

BST: VocabBST

m = the length of the linked list for the document IDs

Method	**Big(O) Average Case**	**Big(O) Worst Case**
AddWordBST(String word, int i)	**O (log n + m)**	**O (n + m)**
Find (String word)	**O (log n)**	**O(n)**
FindDocList(String word)	**O (log n)**	**O(n)**
getDocList( )	**O (1)**	**O (1)**

AVL: VocabAVL

n = the number of unique words in the AVL tree, d = the length of the linked list for the word

Method	Big(O)
AddWordAVL(String word, int i)	**O (log n + d)**
Find (String word)	**O (log n)**
FindDocList(String word)	**O (log n)**
getDocsLists( )	**O (1)**
getvocasize( )	**O (1)**