King Saud University
College of Computer and Information Sciences
Department of Information Technology

CSC212: Data Structure

Search Engine

Section #	NAME	ID
#41196	*Layan Khalid Aldbays*	*444200653*
#41196	*Rahaf Mohammed Alshatiri*	*444200987*
#41196	*Jana Fahd Alzhrani*	*444200619*

Search Engine Description:

Our project focuses on developing a search engine that offers multiple functionalities to enhance document search and retrieval. The system supports efficient term retrieval using various algorithmic approaches, such as indexing, which maps terms to the documents they appear in, and inverted indexing, which uses data structures like linked lists or binary search trees to link each term to a list of relevant documents. It also facilitates Boolean retrieval by providing document IDs based on user queries using logical operators like AND and OR. Additionally, the search engine implements ranked retrieval, where documents are ranked by counting the occurrences of user-specified terms and sorted in descending order of relevance. Furthermore, the system provides document analysis by displaying term frequencies and listing the IDs of documents containing the terms, offering a comprehensive and efficient solution for document search and analysis.

Search Engine Usage:

The search engine menu offers a comprehensive and user-friendly interface with several powerful features. The "Retrieve a word (Index)" option allows you to search for a specific word using the Index, enabling you to quickly identify the documents containing that word. Similarly, the "Retrieve a word (Inverted Index)" utilizes the Inverted Index, which maps words to the document IDs where they appear, providing efficient word-based searches. For enhanced performance, the "Retrieve a word (BST Inverted)" uses a Binary Search Tree (BST) version of the Inverted Index, allowing faster and more efficient searches.

The "Boolean Retrieval" feature supports logical queries using operators like `AND` and `OR`, enabling complex and precise searches to suit diverse needs. For instance, you can combine multiple conditions to refine your search results further. The "Ranked Retrieval" ranks documents based on their relevance to the query, calculated using Term Frequency (TF). This ensures the most relevant documents appear first, making it easier to find the most useful results.

To view more details about the document, the "Display all documents with the number of words (Tokens)" option shows all documents and their word counts, giving insights into the size of each document. Additionally, the "Display all the words (Tokens) with the number of documents they appear in" provides a comprehensive view of word distribution across documents. For an overall summary, the "Display total number of tokens & unique words" option displays the total word count and the number of unique words in the dataset.

Search Engine Details:

- **Document Processing:**
  Document processing was the first step in building our search engine. Its goal is to prepare raw textual data for indexing and querying by cleaning. This ensures consistency and enhances search accuracy.

  Text cleaning involved removing unnecessary characters such as punctuation, hyphens, and special symbols, while converting all text to lowercase for case-insensitive comparisons. Next, the text was tokenized by splitting each sentence into individual words (tokens) based on spaces. Stop word removal was performed using the list of stop words provided (e.g., "and," "the") and excluding them from further processing.

  Finally, unique words were extracted and stored in a separate list for statistical analysis.

- **Indexing:**

  In Indexing we organized the processed data into structures that enable efficient querying. We implemented Two types of indices : index and inverted index.

  We implemented the Index in the Index class, where each document is mapped to the list of words it contains, allowing us to easily retrieve a document's content by its ID.

  For the Inverted Index, we used the InvertedIndex class, which maps each word to the list of document IDs where it appears. If a word already exists in the index, we simply append the document ID, otherwise, we create a new entry.

  Lastly, we implemented the Inverted Index (using BST) in the InvertedIndex_BST class, utilizing a binary search tree (BST) to store words and their associated document IDs. This approach enables faster lookups and maintains the words in sorted order.

- **Query Processing:**

Through query processing we were able to interpret user queries to retrieve relevant documents efficiently. Our system supports both Boolean logic and mixed queries, providing a flexible and robust search experience.

Implemented Features:

1. Boolean Queries:

   We handled AND and OR operations to combine or filter query terms effectively. Using the index and inverted index, we retrieved document IDs that matched the query criteria.

2. Mixed Queries:

   Our system supported queries combining AND and OR operators, where OR-separated subqueries were processed individually using AND logic, and the results were merged using OR.

3. Search Mechanism:

   We utilized the index and inverted index to locate document IDs for specific terms. Queries were processed (e.g., converting operators like AND and OR to uppercase) to maintain consistency and improve accuracy.

- **Ranking:**

  Ranking assigns scores to documents based on their relevance to the query, ensuring that the most relevant documents are presented to the user first.

  Here’s what we implemented:

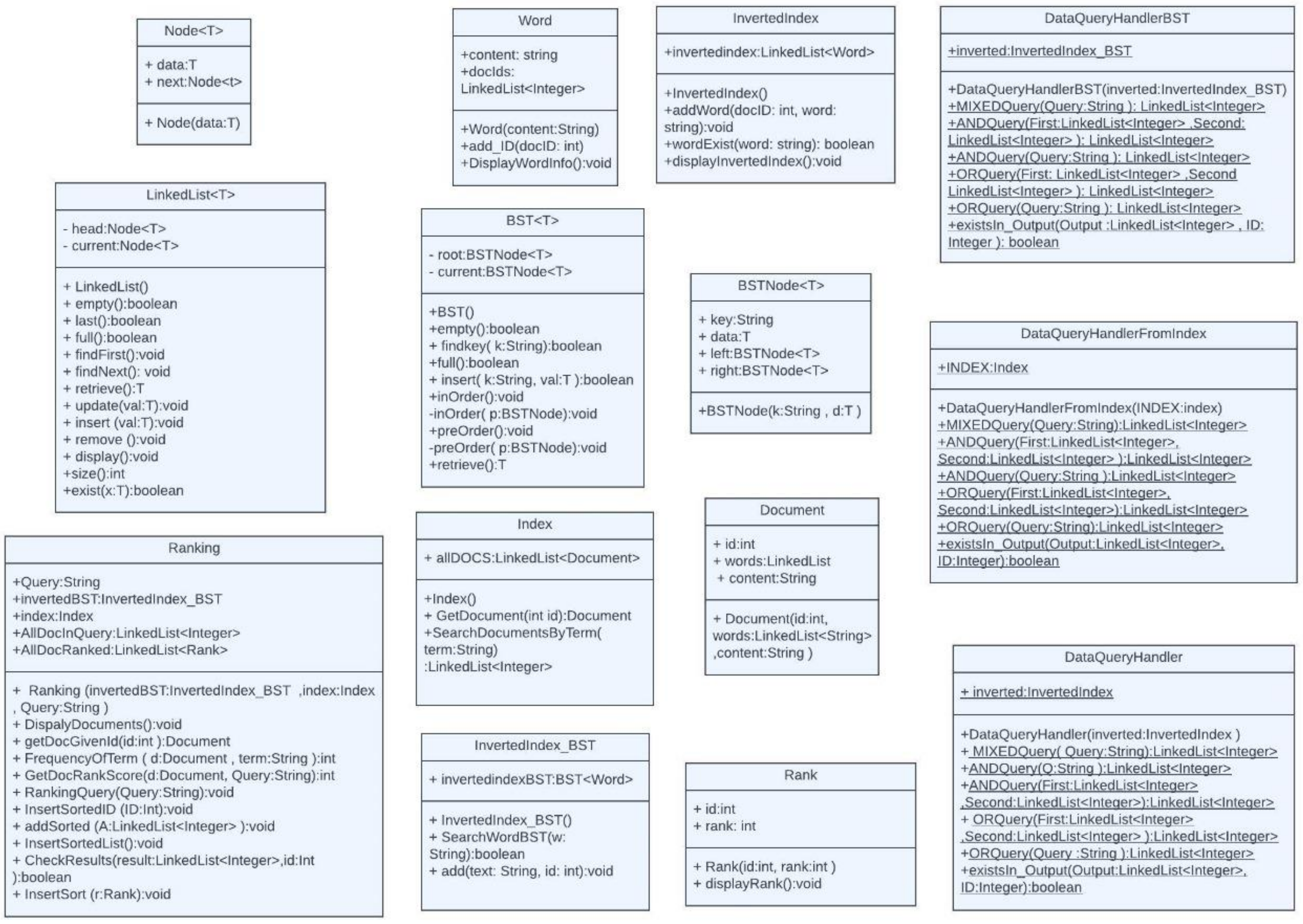
  1. Term Frequency (TF):
     We calculated how many times each query term appears in a document and summed up the frequencies of all query terms in each document to determine its relevance score.

2. Document Scoring:
   We ranked documents based on their scores, with higher scores indicating greater relevance to the query.

3. Rank Query:
   We retrieved all documents containing the query terms and calculated their scores. The scores were stored in a sorted list to enable efficient retrieval of top-ranked documents.

4. Sorted Ranked List:
   We maintained a list of ranked documents in descending order of their scores, ensuring that the most relevant results were presented to the user first.

Class Diagram:


Node<T>
+ data:T
+ next:Node<t>
+ Node(data:T)
Word
+content: string
+docIds:
LinkedList<Integer>
+Word(content:String)
+add_ID(docID: int)
+DisplayWordInfo():void
InvertedIndex
+invertedindex:LinkedList<Word>
+InvertedIndex()
+addWord(docID: int, word:
string):void
+wordExist(word: string): boolean
+displayInvertedIndex():void
DataQueryHandlerBST
+inverted:InvertedIndex_BST
+DataQueryHandlerBST(inverted:InvertedIndex_BST)
+MIXEDQuery(Query:String): LinkedList<Integer>
+ANDQuery(First:LinkedList<Integer> ,Second:
LinkedList<Integer>): LinkedList<Integer>
+ANDQuery(Query:String): LinkedList<Integer>
+ORQuery(First: LinkedList<Integer> ,Second
LinkedList<Integer>): LinkedList<Integer>
+ORQuery(Query:String): LinkedList<Integer>
+existsIn_Output(Output :LinkedList<Integer> , ID:
Integer): boolean
LinkedList<T>
- head:Node<T>
- current:Node<T>
+ LinkedList()
+ empty():boolean
+ last():boolean
+ full():boolean
+ findFirst():void
+ findNext(): void
+ retrieve():T
+ update(val:T):void
+ insert (val:T):void
+ remove ():void
+ display():void
+size():int
+exist(x:T):boolean
BST<T>
- root:BSTNode<T>
- current:BSTNode<T>
+BST()
+empty():boolean
+ findkey(k:String):boolean
+full():boolean
+ insert(k:String, val:T):boolean
+inOrder():void
-inOrder(p:BSTNode):void
+preOrder():void
-preOrder(p:BSTNode):void
+retrieve():T
BSTNode<T>
+ key:String
+ data:T
+ left:BSTNode<T>
+ right:BSTNode<T>
+BSTNode(k:String , d:T)
DataQueryHandlerFromIndex
+INDEX:Index
+DataQueryHandlerFromIndex(INDEX:index)
+MIXEDQuery(Query:String):LinkedList<Integer>
+ANDQuery(First:LinkedList<Integer>,
Second:LinkedList<Integer>):LinkedList<Integer>
+ANDQuery(Query:String):LinkedList<Integer>
+ORQuery(First:LinkedList<Integer>,
Second:LinkedList<Integer>):LinkedList<Integer>
+ORQuery(Query:String):LinkedList<Integer>
+existsIn_Output(Output:LinkedList<Integer>,
ID:Integer):boolean
Ranking
+Query:String
+invertedBST:InvertedIndex_BST
+index:Index
+AllDocInQuery:LinkedList<Integer>
+AllDocRanked:LinkedList<Rank>
+ Ranking (invertedBST:InvertedIndex_BST ,index:Index
, Query:String)
+ DispalyDocuments():void
+ getDocGivenId(id:int):Document
+ FrequencyOfTerm (d:Document , term:String):int
+ GetDocRankScore(d:Document, Query:String):int
+ RankingQuery(Query:String):void
+ InsertSortedID (ID:Int):void
+ addSorted (A:LinkedList<Integer>):void
+ InsertSortedList():void
+ CheckResults(result:LinkedList<Integer>,id:Int
):boolean
+ InsertSort (r:Rank):void
Index
+ allDOCS:LinkedList<Document>
+Index()
+ GetDocument(int id):Document
+SearchDocumentsByTerm(
term:String)
:LinkedList<Integer>
Document
+ id:int
+ words:LinkedList
+ content:String
+ Document(id:int,
words:LinkedList<String>
,content:String)
DataQueryHandler
+ inverted:InvertedIndex
+DataQueryHandler(inverted:InvertedIndex)
+ MIXEDQuery(Query:String):LinkedList<Integer>
+ANDQuery(Q:String):LinkedList<Integer>
+ANDQuery(First:LinkedList<Integer>
,Second:LinkedList<Integer>):LinkedList<Integer>
+ ORQuery(First:LinkedList<Integer>
,Second:LinkedList<Integer>):LinkedList<Integer>
+ORQuery(Query :String):LinkedList<Integer>
+existsIn_Output(Output:LinkedList<Integer>,
ID:Integer):boolean
InvertedIndex_BST
+ invertedindexBST:BST<Word>
+ InvertedIndex_BST()
+ SearchWordBST(w:
String):boolean
+ add(text: String, id: int):void
Rank
+ id:int
+ rank: int
+ Rank(id:int, rank:int)
+ displayRank():void

Performance Analysis:

Class	Method	Big O Notation
Index retrieval (index class)	GetDocument(int id)	O(n)
	SearchDocumentsByTerm(String term)	$O(n^2)$
Inverted Index retrieval using lists of lists (Inverted Index class)	addWord(int docID, String word)	O(n)
	wordExist(String word)	O(n)
	displayInvertedIndex()	$O(n^2)$
inverted index retrieval using BST (InvertedIndex_BST class)	add(String text, int id)	O(Logn)
	SearchWordBST(String w)	O(Logn)

Performance Analysis Of Boolean Retrieval:

Approach	Time Complexity (worst Case)	Query processing
Index With Lists	$O(n^2)$	Iterates through all documents and words sequentially, making it unsuitable for large datasets.
Inverted Index With Lists	O(n)	Maps each word directly to its associated documents, enabling faster execution of Boolean operations like AND and OR.
Inverted Index With BST	O(logn)	Utilizes tree traversal for efficient word lookups. Boolean operations are optimized through quick merging and intersections.

Conclusion Regarding Performance Analysis:

The Inverted Index with BST delivers the best performance due to its logarithmic query time and optimized handling of Boolean operations, making it the ideal solution for large and complex datasets. However, the Index with Lists demonstrates the worst performance, with a time complexity of $O(n^2)$, as it sequentially iterates through all documents and words. This makes it inefficient and unsuitable for large datasets.

Link To GitHub:

https://github.com/ijanao/Data-Structere-Project