

**King Saud University**

**College of Computer and Information Sciences**

**Department of Information Technology**

**Data Structure Project**

**1st Semester 1446 H**

**Report**

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 444200814 | Munira Alsaleem |
| 444200523 | Deem Aljarba |
| 444200749 | Maha Alruwais |

**Supervised By: Dr.Ghada Alnifie**

**Project idea:** **Search Engine**

Table of Contents

[Project introduction: 3](#_Toc183291114)

[Classess: 4](#_Toc183291115)

[LinkedList Class: 4](#_Toc183291116)

[Node Class: 4](#_Toc183291117)

[Word Class: 5](#_Toc183291118)

[Document Class: 5](#_Toc183291119)

[Index Class: 6](#_Toc183291120)

[InvertedIndex Class: 6](#_Toc183291121)

[BST Class: 7](#_Toc183291122)

[InvertedIndexBST Class: 7](#_Toc183291123)

[QueryProcessingBST class: 8](#_Toc183291124)

[DocRank class: 8](#_Toc183291125)

[Ranking2 class: 9](#_Toc183291126)

[DocumentManager Class: 9](#_Toc183291127)

[Performance Analysis 11](#_Toc183291128)

[UML 13](#_Toc183291129)

[Conclusion: 13](#_Toc183291130)

[GIThub link: 13](#_Toc183291131)

**Project introduction:** The search engine utilizes a combination of efficient data structures, such as the **inverted index**, **binary search tree (BST)**, and **index**, to optimize search performance and accuracy. These structures form the backbone of the search engine, enabling it to handle search queries quickly and effectively.

* **Inverted Index**: The inverted index is the core structure used for efficient searching. It maps each unique word to a list of document IDs that contain that word. This makes it easy to retrieve all documents associated with a specific term, reducing the time needed to search through many documents.
* **Binary Search Tree (BST)**: The binary search tree is used to store terms in a sorted order. BSTs allow for fast lookups and help organize the terms more efficiently, speeding up the search process. This is particularly important when dealing with large datasets, as it reduces the time complexity of retrieving documents that contain certain terms.
* **Index**: The index is responsible for managing and organizing the collection of documents. It tracks document IDs and helps in accessing the content quickly. The index ensures that the search engine can efficiently handle queries, scaling with increasing amounts of data.

**Search Query Types:**

* **Boolean Queries (AND & OR)**: The search engine supports **Boolean queries** using **AND** and **OR** operators. These operators allow users to refine their searches by combining multiple terms in a structured way:
* **AND**: When the **AND** operator is used, the search results will include only documents that contain **all** the specified terms.
* **OR**: The **OR** operator expands the search results, returning documents that contain **at least one** of the specified terms.
* **Mixed Queries (AND & OR Combined)**: The search engine also supports **mixed queries**, which allow for the combination of both **AND** and **OR** operators within the same query. This enables users to create more complex queries that provide greater control over the search results.
* **Ranking**: In addition to Boolean queries and mixed queries, the search engine ranks results based on the relevance of documents to the search terms. The ranking system considers **factors** such as the **frequency of the search term in each document**, ensuring that the most relevant results **appear first**. This helps users quickly find the documents that best match their search criteria.

# **Classes Implementation:**

## LinkedList Class:

A generic singly linked list implementation. The list supports operations like insertion, removal, searching, updating, and traversal. It maintains a reference to the head of the list and a current node for operations like iteration.

Attributes:

* Head
* Current

Methods:

**Constructor:** Initializes an empty linked list with both head and current set to null.

**Empty():** Checks if the linked list is empty or not.

**Last():** Checks if the current node is the last node.

**findFirst():** Sets current to the head of the list.

**insert(T val):** Inserts a new node with the given value after the current node. If the list is empty, the new node becomes both the head and current.

**Display():** Prints the data of each node in the linked list, starting from the head.

**retrieve():** Retrieves the data from the current node.

**contains(T val**): Checks if the list contains a node with the given value.

**size():** Counts and returns the total number of nodes in the list.

**update(T newValue):** Updates the data of the current node with the specified newValue.

# Node Class:

The Node class represents a single node in the linked list, containing the data and a reference to the next node.

Attributes:

* Data
* Next

# Word Class:

The Word class represents a word in a collection, where each word can be associated with a list of document IDs that contain the word. The document IDs are stored in a linked list, and the class provides methods to manage and display these document IDs.

Attributes:

* word
* docID

methods:

**Constructor:** Initializes a new Word object with the given word and an empty linked list of document IDs.

**addID(int id)**: Adds the document ID to the docID linked list, if the ID does not already exist in the list. Calls isIDExist() to check for duplicates before inserting the ID.

**isIDExist(Integer id):** Checks if a document ID already exists in the docID list.

**display():** Prints the word and its associated document IDs. If no document IDs are associated with the word, it prints a message saying no documents contain that word.

**getIDs():** return LinkedList<Integer>

# Document Class:

The Document class represents a document that contains an ID and a list of words. The document can display its ID and the words it contains using the methods provided.

Attributes:

* id
* words

methods:

**Constructor:** Initializes a new Document with the specified ID and a linked list of words.

**display():** Prints the document's ID followed by the list of words contained in the document. The words are printed one by one, separated by commas.

**containsWord(String w):**return boolean

**getId():**int

# Index Class:

The Index class represents a collection of Document objects. It provides methods to add, retrieve, and display documents by their ID, as well as to get the total count of documents in the index.

Attributes:

* documents

methods:

**Constructor**: Initializes an empty Index by creating a new linked list to hold Document objects.

**addDocument(Document document):** Adds the given document to the Documents linked list.

**display():** Prints the details of all documents in the index. If no documents are present, it prints a message saying there are no documents to display.

**getDocumentByID(Integer ID):** Searches for a document by its ID in the Documents list. If found, the document is returned; otherwise, null is returned.

**getDocumentCount**(): Counts the number of documents in the Documents linked list and return it.

**SearchTermInDocuments(String word):**return LinkedList<Integer>

# InvertedIndex Class:

The InvertedIndex class represents an inverted index structure that maps words to document IDs. It allows for adding words to the index, checking for word existence, and displaying the index's contents.

Attributes:

* Inverted\_Index

Methods:

**Constructor**: Initializes an empty InvertedIndex by creating a new linked list to hold Word objects.

**add(String word, int id):** Adds a document ID to the word's entry in the inverted index. If the word already exists in the index, the document ID is added to its list. Otherwise, a new Word object is created and inserted into the index.

**isWordExist(String w):** Checks whether the specified word exists in the inverted index.

**display():** Displays the entire inverted index by printing each word along with its associated document IDs. If the index is empty, it prints "Inverted\_Index is empty".

**searchWordInDocumentInverted(String word):** LinkedList<Integer>

# BST Class:

The BST (Binary Search Tree) class is a generic data structure that stores key-value pairs. Each node in the tree holds a key (of type String) and a value (of a generic type T). The tree supports standard operations like insertion, search, and in-order traversal.

Attributes:

* Root
* Current

Methods:

**Constructor:** Initializes an empty binary search tree with root and current set to null.

**empty():** Checks if the tree is empty by verifying if the root is null.

**full():** always returns false.

**retrieve():** Retrieves the data stored in the current node.

**findkey(String tkey):** Searches for the node with the given key. If the key is found, the current node is updated to the found node, and the method returns true. If the key is not found, current is set to the last node visited during the search, and the method returns false.

**insert(String k, T val):** Inserts a new key-value pair into the binary search tree. If the key already exists, the method returns false. Otherwise, it inserts the node in the correct position based on the binary search tree property.

**displayInOrder():** Displays the elements of the tree in in-order traversal order. If the tree is empty, it prints a message indicating that the tree is empty.

**traverseAndDisplay(BSTNode node):** A private recursive method that performs an in-order traversal of the binary search tree, displaying each node's value.

# InvertedIndexBST Class:

The InvertedIndexBST class represents an inverted index implemented using a Binary Search Tree (BST). The index maps words to the document IDs that contain those words.

Attributes:

* InvertedIndex

Methods:

**Constructor:** Initializes the inverted index using a BST<Word>. The tree is initially empty.

**add(String word, int id)**: This method adds a word to the inverted index. If the word is already in the index, it adds the document ID to the list of documents that contain the word. If the word is not already in the index, it creates a new entry for the word with the provided document ID.

**display()**: Displays the contents of the inverted index in in-order traversal order. It prints the words in lexicographical order along with the list of document IDs where each word appears. If the index is empty, it prints a message indicating that the inverted index is empty.

**isWordExist(String w)**: This method checks whether a specific word exists in the inverted index by searching for the word in the InvertedIndex BST. It returns true if the word is found, and false if the word does not exist in the index.

**searchWordInDocumentInvertedBST(String word):** LinkedList<Integer>

# QueryProcessingBST class:

The QueryProcessingBST class is designed to handle Boolean queries using an inverted index implemented with a Binary Search Tree (BST). It supports operations for AND, OR, and mixed queries, providing methods to evaluate the presence of terms in the index and combine results efficiently.

Attributes:

* Inverted

Methods:

**Constructor**: Initializes the QueryProcessingBST object with the given InvertedIndexBST for query processing.

**andQ(String query)**: Processes an AND query by splitting the query string on "AND" and combining the document IDs for each term using the andQ(LinkedList<Integer>, LinkedList<Integer>) helper method.

**andQ(LinkedList<Integer> a, LinkedList<Integer> b)**:Combines two lists of document IDs, keeping only those that appear in both lists (AND operation).

**isExist(LinkedList<Integer> a, Integer id)**:Checks if a document ID exists in a list of document IDs.

**ORQ(String query)**:Processes an OR query by splitting the query string on "OR" and combining the document IDs for each term using the ORQ(LinkedList<Integer>, LinkedList<Integer>) helper method.

**ORQ(LinkedList<Integer> a, LinkedList<Integer> b)**: Combines two lists of document IDs, keeping all unique IDs from both lists (OR operation).

**BooleanQuery(String query)**: Identifies whether the query contains "AND" or "OR" and directs the query to the appropriate handler (AND, OR, or mixed queries).

**MixQuery(String query)**: Handles mixed queries by first processing the AND operations, then combining the results using OR operations.

# DocRank class:

represents a document and its corresponding rank in the ranking process. It is used to store the document ID and the score (rank) that determines its relevance to a given search query.

Attributes:

* id
* rank

methods:

**Constructor:**): Initializes a DocRank object with the given document ID and rank.

**display()**: Displays the document ID and its rank.

# Ranking2 class:

handles the ranking of documents based on a query. It interacts with an inverted index to retrieve relevant documents, computes their relevance scores based on the query terms, and ranks them accordingly.

Attributes**:**

* index
* invertedIndex
* query
* documentIds
* rankedDocuments

Methods**:**

**Constructor**: Initializes the Ranking object with the given Index, InvertedIndexBST, and query.

**getDocumentById(int id):** Retrieves the Document by its ID from the index.

**countTermFrequency(Document document, String term):** Counts how many times the specified term appears in the given document.

**calculateDocumentScore(Document document, String query):** Calculates the score for a document based on term frequency from the query.

**findRelevantDocuments(String query): Finds** the relevant document IDs based on the query.

**addDocumentIds(LinkedList<Integer> termDocumentIds):** Adds document IDs to the list, ensuring no duplicates.

**displayRankedDocuments():** Displays the ranked documents.

**insertDocumentIdSorted(Integer id):** Inserts a document ID into the sorted list of document IDs.

**generateRankedList():** Generates a ranked list of documents based on the query.

**insertRankedDocument(DocRank docRank**): Inserts a DocRank object into the ranked documents list in sorted order.

**isDocumentIdExists(LinkedList<Integer> list, Integer id):** Checks if a document ID already exists in the provided list.

# DocumentManager Class:

manages the loading of documents, stop words, and the building of an inverted index for a given collection of documents. It provides methods to calculate the vocabulary size, count tokens in a file, load stop words, clean and extract words from documents, and manage the indexing process. This class helps in creating and manipulating an index for document retrieval systems.

Methods:

**Constructor:** Initializes a new DocumentManager object, setting up the necessary structures to handle stop words, documents, and indexes.

**calculateVocabularySize(String fileName**): Calculates the number of unique words (vocabulary size) in the specified file.

**countTokensInFile(String fileName**): Counts the total number of tokens (words) in the specified file.

**LoadingStopWords(String fN):** Loads stop words from the specified file and adds them to the StopWords list.

**Load(String fN):** Loads documents from the specified file, parses them, and adds them to the Index object. Each document is processed by extracting and cleaning the words, and indexing them in both the InvertedIndex and InvertedIndexBST.

**cleanAndExtractWords(String DocLine, int id):** Cleans and extracts words from a given document line and adds them to the inverted index (both InvertedIndex and InvertedIndexBST).

**isExistINStopWords(String w):** Checks if a given word exists in the stop words list.

**isWordInList(LinkedList<String> list, String word):** Checks if a word exists in a given list of words.

**insert(String word):** Inserts a word into the list.

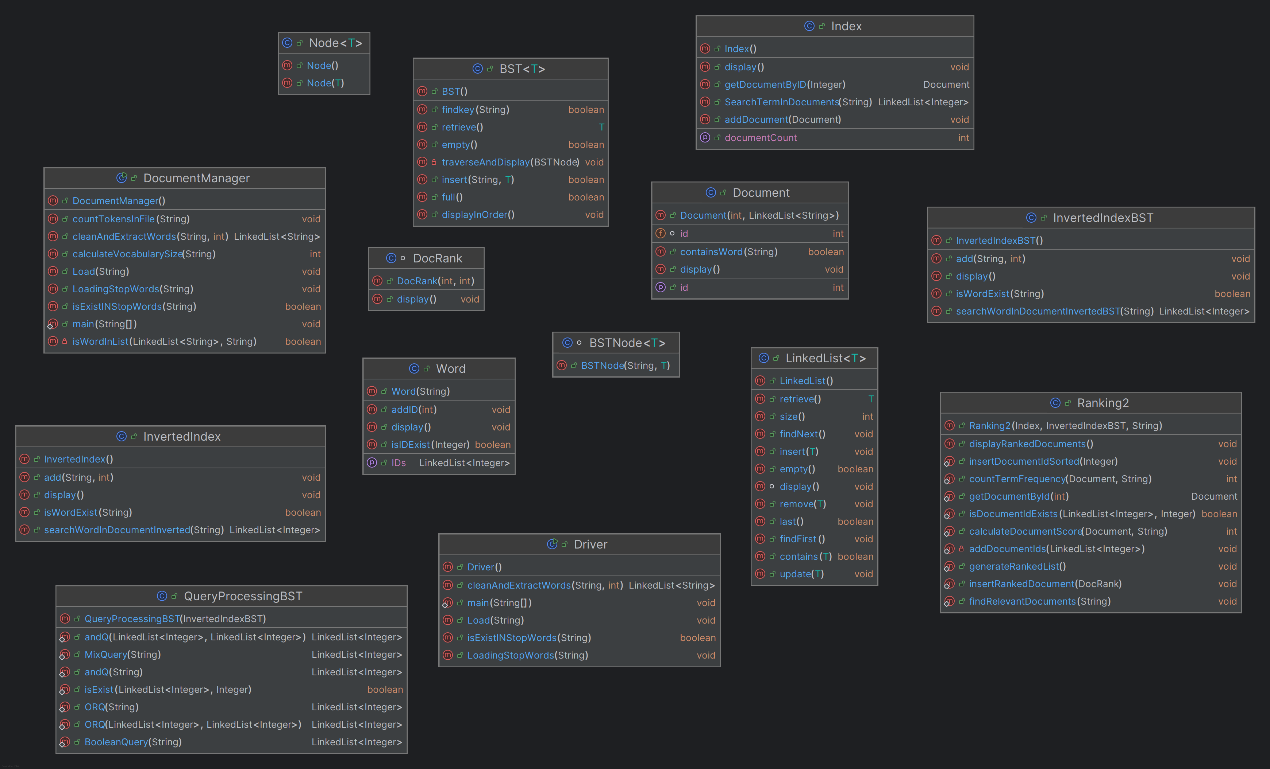
# **Performance Analysis**

|  |  |  |
| --- | --- | --- |
| Method | Code | Big O |
| SearchTermInDocuments (index retrieval(using list of list))) | public LinkedList<Integer> SearchTermInDocuments(String word) {  LinkedList<Integer> docIDs = new LinkedList<>();  if (Documents.empty()) {  return docIDs;  }  Documents.findFirst();  while (!Documents.last()) {    if (Documents.retrieve().containsWord(word)) {  docIDs.insert(Documents.retrieve().id);  }  Documents.findNext();  }    if (Documents.retrieve().containsWord(word)) {  docIDs.insert(Documents.retrieve().id);  }  return docIDs;  } | O(n) |
| searchWordInDocumentInverted (inverted index retrieval(using list of list)) | public LinkedList<Integer> searchWordInDocumentInverted(String word){  LinkedList<Integer> docIDs=new LinkedList<Integer>();    if(Inverted\_Index.empty())  return docIDs;    Inverted\_Index.findFirst();  while(!Inverted\_Index.last()){  if(Inverted\_Index.retrieve().word.equals(word))  return Inverted\_Index.retrieve().docID;  Inverted\_Index.findNext();  }  if(Inverted\_Index.retrieve().word.equals(word))  return Inverted\_Index.retrieve().docID;      return docIDs;  }    } | O(n) |
| searchWordInDocumentInvertedBST (inverted index retrieval(using BST)) | public LinkedList<Integer> searchWordInDocumentInvertedBST(String word){  LinkedList<Integer> docIDs=new LinkedList<Integer> ();    if(!isWordExist(word))  return docIDs;    return InvertedIndex.retrieve().docID;          } | O(Log(n)) |

**searchWordInDocumentInvertedBST** (inverted index retrieval(using BST)) is the best method and the fastest because it uses a Binary Search Tree (BST) to organize and search the inverted index. The time complexity of this method is O(log(n)), which is significantly better than O(n) as seen in the other methods.

In O(n) the search time increases linearly with the number of documents or words, meaning it takes longer as the dataset grows. However, In O(log(n)) the time grows much slower, as the BST divides the data hierarchically, reducing the number of comparisons needed to find the word.

# **UML**



# **Conclusion:**

This project showcases a search engine that utilizes efficient data structures, including the inverted index, binary search tree (BST), and index, to enhance both the speed and accuracy of searches. By integrating these structures, the search engine can deliver fast, precise, and scalable results for a wide range of search queries.

# **GIThub link:**

[**https://github.com/deemkj/DStructure/tree/main**](https://github.com/deemkj/DStructure/tree/main)