King Saud University
College of Computer and Information Sciences
CSC212 Data Structures

Group #99

Name	ID
Ghadeer Alnuwaysir	444200420
Norah Almadhi	444200890
Lamees Alghamdi	444201177

Overview

The implemented search engine is a flexible and scalable system designed to efficiently index, query, and rank documents using two data structures: Linked Lists and Binary Search Trees (BSTs). It supports Boolean queries (AND, OR) and Ranked Retrieval, providing a versatile tool for document search and management. While both data structures were utilized, the BST was assumed to be the better option for searching due to its logarithmic search time complexity in balanced cases, making it more efficient than the linear search required by a Linked List for handling larger datasets.

1. Core Components

1.1 Document Preprocessing

- Documents are preprocessed to remove punctuation, convert text to lowercase, and filter out stop words.
- Each document is represented as a unique ID paired with its processed content.

1.2 Indexing

The search engine builds and manages indices to enable efficient query processing:

- **Index**:
  - Manages a collection of documents using a linked list.
  - Provides methods to add documents, display indexed documents, and find specific words across documents.
- **Inverted Index**:
  - Maps terms (words) to the document IDs where they occur.
  - Implemented using a list-based structure (WordEntry) to store document mappings.
- **Inverted Index with Binary Search Tree (BST)**:
  - Stores terms in a binary search tree for faster lookups.
  - Reduces the complexity of term retrieval, especially for large datasets.

1.3 Query Processing

The Query and BSTQuery classes handle user search queries. They support:

- **Boolean Queries**:
  - **ANDQuery**: Finds documents containing all specified terms.
  - **ORQuery**: Finds documents containing at least one of the specified terms.
- **Mixed Queries**:
  - Combines AND, OR operators in a single query.
- **Ranked Retrieval**:
  - Uses term frequency (TF) to score and rank documents by relevance.

1.4 Ranking

- Documents are ranked based on their term frequency (TF).
- The Rank class computes relevance scores and provides sorted results for ranked queries.

2. Data Structures

2.1 LinkedList

- Used extensively for managing list of words in class Document, list of Documents in Index, list of document IDs in the InvertedIndex and Rank classes.
- Supports operations like insertion, traversal, and removal for efficient query processing.

2.2 Binary Search Tree (BST)

- Utilized in the InvertedIndexBST for faster term lookups.
- Each node in the BST represents a term and its associated document IDs (WordEntry).

2.3 Node

- Forms the backbone of the LinkedList and BST implementations.
- Stores data (e.g., document IDs or terms) and references to the next node (LinkedList) or children (BSTNode).

3. Query Execution

Boolean Queries

- **ANDQuery**:
    - Retrieves document IDs for each term in the query.
    - Performs an intersection of the document ID lists to find common documents.

- **ORQuery**:
    - Retrieves document IDs for each term in the query.
    - Performs a union of the document ID lists to find all matching documents.

Ranked Queries

- Ranks documents based on the frequency of query terms in each document.
- Documents with higher term frequency for the query terms are given higher relevance scores.

4. User Interaction

The user interacts with the search engine via a menu-driven interface (Menu class).

- **Retrieve Term**:

Users can enter a term to search for and choose from three methods of retrieval:

  - **Using Index with Linked List**: Searches the document collection using a linked list structure.
  - **Using Inverted Index with Linked List**: Retrieves document IDs based on the term's presence in the inverted index implemented with linked lists.
  - **Using Inverted Index with BST**: Leverages the binary search tree for efficient term lookups.

- **Boolean Retrieval**:
  - Users can input a Boolean query (using AND/OR operators) to find documents that match the criteria specified in the query.
  - The system retrieves and displays documents that meet the query conditions.

- **Ranked Retrieval**:
  - Users can enter a query for ranked retrieval. The system scores documents based on term frequency and relevance, presenting the results in order of relevance.

- **Show Number of Indexed Documents**:
  - Displays a list of all indexed documents along with their details, such as document IDs and token counts, helping users to understand the indexed content.

- **Show Number of Indexed Tokens**:
  - Users can choose to view indexed tokens using either a linked list or a binary search tree. This option allows for insights into the terms stored in the system.

- **Exit**:
  - Users can exit the program gracefully, concluding their session with a goodbye message.

Classes And Methods

1. Class LinkedList

- **isEmpty()**
  Checks if the linked list is empty by verifying if the Head pointer is null.
- **last()**
  Determines if the current node is the last node in the list by checking if its next pointer is null.
- **full()**
  Always returns false as a linked list does not have a fixed size and cannot be "full."
- **findFirst()**
  Moves the current pointer to the head of the linked list, marking the first node as the starting point for traversal.
- **findNext()**
  Advances the current pointer to the next node in the linked list.
- **update(T d)**
  Replaces the data in the current node with the new value provided (d).
- **retrieve()**
  Returns the data stored in the current node. If the current pointer is null, it returns null.
- **clear()**
  Clears the linked list by setting both the Head and current pointers to null, effectively removing all nodes.
- **insert(T d)**
  Inserts a new node containing the specified data (d) into the linked list:
  - If the list is empty, the new node becomes the head.
  - Otherwise, the new node is inserted after the current node, and the current pointer is updated to the newly added node.
- **remove()**
  Removes the current node from the linked list:
  If the current node is the head, the Head pointer is updated to the next node.
  Otherwise, the node preceding current is linked to the node following current.
  The current pointer is then updated to the next node or reset to Head if the list ends.
- **display()**
  Prints all nodes in the linked list, displaying their data sequentially.
- **displayWordsInline()**
  Prints the data of all nodes in a single line, separated by spaces. Useful for compact display of elements.

- **exists(String value)**
  checks whether a specific value (value) exists in the linked list.

2. Class Node

- **T data**
  Stores the actual value or data of the node. This is a generic type (<T>), allowing the linked list to handle data of any type.
- **Node <T> next**
  A reference to the next node in the linked list. This ensures the continuity of the chain.
- **Node (T d)**
  A constructor that initializes a node with the given data (d) and sets the next pointer to null, indicating the end of the list by default.

3. Class Document

- **LinkedList<String> Words**
  Stores a dynamic collection of words associated with the document, allowing insertion, removal, and traversal of the list for managing word data.
- **int docuID**
  Represents the unique identifier for the document. This field is essential for distinguishing between different documents in a collection.
- **Document(int docuID, LinkedList<String> Words)**
  The constructor initializes the document with:
  - docuID: A unique identifier for the document.
  - Words: A linked list containing the words that belong to the document.
- **getTokenCountPerDoc ()**
  Returns the total number of tokens currently stored in the document.

4. Class Index

- **LinkedList documents**
  A linked list that stores the collection of Document objects.
- **Index()**
  Initializes the index by creating an empty LinkedList for documents.
- **addDocument(Document doc)**
  Adds a new document to the documents linked list if it doesn’t exist already.
- **displayDocs()**
  Traverses the documents linked list and prints the details of each document, including the docuID and the words in the document, displayed in a single line. Prints "empty documents" if the list is empty.
- **returnDocument(int docId)**
  Searches for a document with the specified docId in the documents linked list. Returns the corresponding Document object if found or null if not found or if the list is empty.
- **findWordIndex(String word)**
  Searches the documents linked list for occurrences of a specific word. If the word exists in any document, prints the IDs of the documents where the word is found. If the word is not found, prints a message indicating that the word was not found. If the list is empty, prints "No documents available."

5. Class InvertedIndex

- **LinkedList<WordEntry> invertedList**
  A linked list that stores the inverted index, where each WordEntry represents a word and its associated document IDs.
- **InvertedIndex()**
  Constructor that initializes an empty LinkedList for invertedList.
- **addWord(int ID, String WORD)**
  Adds a word to the inverted index. If the word already exists, its associated WordEntry is updated by adding the document ID then returns False. Otherwise, a new WordEntry is created and added to the inverted list and True is returned.
- **indexWordEntry(String w)**
  Searches for the specified word (w) in the invertedList. Returns true if the word exists, otherwise returns false.
- **displayInvertedIndex()**
  Traverses the invertedList and prints each word and its associated document IDs. If the list is empty, it prints "empty documents".
- **findWordInvertedIndex(String word)**
  Searches for a word in the invertedList and displays its associated document IDs.
    - If the invertedList is empty, prints "The inverted index is empty. "
    - If the word exists, retrieves the corresponding WordEntry and displays the document IDs.
    - If the word is not found, prints "The word [word] was not found."

6. Class Processor

- **LinkedList<String> stopW**
  A linked list storing the stop words for filtering during preprocessing.
- **Index index**
  An Index object that stores documents with their associated words.
- **InvertedIndex invertedind**
  An InvertedIndex object for managing an inverted index.
- **InvertedIndexBST invertedindBST**
  An InvertedIndexBST object for managing an inverted index using a binary search tree.
- **int TotalTokens**
  An integer representing the total number of words (includes stop words).
- **int TotalVocabulary**
  An integer representing the total number of unique words (vocabulary size) with no duplicates or stop words.

- **Processor()**
  Initializes the Processor by creating empty LinkedList for stop words and initializing the Index, InvertedIndex, and InvertedIndexBST objects.
- **loadStopWords(String stopWordsFile)**
  Reads a file containing stop words, processes them, and stores them in the stopW linked list. Each word is converted to lowercase and trimmed before insertion.
- **preprocessContent(String content, LinkedList<String> WordsLL, int id)**
  Processes the text content of a document by: Splitting the content into words, Removing punctuation from each word, Filtering out stop words and adding the valid words to WordsLL, InvertedIndex, and InvertedIndexBST.
- **isStopWord(String word)**
  Checks if a given word is present in the stopW linked list. Returns true if the word is a stop word and false otherwise.
- **WordsLLMethod(String content, int id)**
  Creates a new LinkedList of words for a document by preprocessing its content and returning the linked list.
- **readDocuments(String fileName)**
  Reads a document file line by line, where each line contains a document ID and its content, separated by a comma. For each document:
  - Extracts the document ID and content.
  - Creates a LinkedList of words using WordsLLMethod.
  - Adds the document to the Index.
- **LoadF(String StopW, String Doc)**
  Loads stop words from a specified file and processes the documents from another specified file by calling loadStopWords and readDocuments.
- **displayDocsByIds(LinkedList<Integer> ids)**
  Displays the documents with the specified IDs. For each ID:
  - Retrieves the document from the Index.
  - Displays the document's ID and its associated words.
  - Prints an error message if a document with the given ID is not found.

7. Class BSTNode

- **String key**
  The key associated with the node, used for comparisons and ordering in the binary search tree.
- **T data**
  The data stored in the node, represented as a generic type.
- **BSTNode<T> left, right:** References to the left and right child nodes.
- **BSTNode(String key, T data)**
  Constructor that initializes a new BSTNode with the specified key and data:
  - Sets the key to the provided string.
  - Sets the data to the provided generic value.
  - Initializes left and right as null references.

8. Class BST

- **BSTNode<T> root**
  A reference to the root of the binary search tree.
- **BSTNode<T> current**
  A reference to the current node used during traversal or insertion.
- **clear()**
- **BST()**
  Constructor that initializes an empty binary search tree, Sets both root and current to null.
- **empty()**
  Returns true if the tree is empty (i.e., root is null), otherwise returns false.
- **full()**
  Always returns false since the binary search tree does not have a fixed size limit.
- **retrieve()**
  Returns the data stored in the current node. If current is null, it returns null.
- **FindKey(String k)**
  Searches for a node with the given key (k) in the binary search tree.
  Starts from the root and traverses the tree based on key comparisons.
  Updates current to the found node and returns true if the key exists.
  Returns false if the key is not found.
- **preOrder()**
  Performs a preorder traversal of the tree (root, left, right) and prints the keys and associated data:
  - If the tree is empty (root is null), prints "empty tree".

Otherwise, calls a helper method preOrder(BSTNode<T> p) to recursively traverse the tree.

- **preOrder(BSTNode<T> p)**
  A private helper method for preorder traversal:

- If the tree is empty (root is null), prints "empty tree".

Otherwise:

- Prints the data of the current node (assumes the data can be cast to WordEntry and calls toString() on it).
- Recursively traverses the left child.
- Recursively traverses the right child.

- **insert(String k, T val)**
  Inserts a new node with the given key (k) and value (val) into the binary search tree.
  If the tree is empty, initializes the root and current with the new node.
  If a node with the key already exists, does not insert and returns false.
  Otherwise, determines the correct position for the new node based on key comparisons and inserts it as a left or right child.
  Updates current to the newly inserted node and returns true to indicate a successful insertion.

9. InvertedIndexBST

- **BST<WordEntry> invertedList**
  word and its associated document IDs.
- **InvertedIndexBST()**
  Constructor that initializes the invertedList as an empty binary search tree (BST).
  A binary search tree that stores WordEntry objects, with each node representing a
- **addWord(int ID, String WORD)**
  Adds a word to the binary search tree:
  - If the word is not already in the tree (!searchWord(WORD)), creates a new WordEntry for the word, inserts the document ID, and adds the new entry to the tree, returns True.
  - If the word already exists, retrieves the corresponding WordEntry and adds the document ID to it, returns False.
- **searchWord(String WORD)**
  Searches for a word in the binary search tree by calling the FindKey method on the BST, returns true if the word is found and false otherwise.
- **displayInvertedIndex()**
  Displays the inverted index by performing a preorder traversal of the binary search tree.
  - Prints each word and its associated document IDs.
  - If the tree is empty or null, prints "Empty".
- **findWordInvertedIndexBST(String word)**
  Searches for a word in the invertedList and displays its associated document IDs:
  - If the invertedList is empty or null, prints "The inverted index is empty."

- If the word exists, retrieves the corresponding WordEntry and displays its document IDs.
- If the word is not found, prints "The word [word] was not found."

10.Class WordEntry

- **String word**
  The word represented by this entry.
- **datastructurep.LinkedList<Integer> docIds**
  A linked list containing the IDs of documents where the word appears.
- **WordEntry(String word)**
  Constructor that initializes a new WordEntry object with the specified word.
  - The word field is set to the provided string.
  - The docIds field is initialized as an empty linked list.
- **getWord()**
  Returns the word associated with this entry.
- **addID(int docId)**
  Adds a document ID to the list of docIds:
  - Checks if the document ID already exists in the list using existsInDocIds.
  - If not, inserts the new ID into the list.
- **existsInDocIds(int docId)**
  Checks if a given document ID is already present in the docIds linked list:
  - Iterates through the list to search for the specified ID.
  - Returns true if the ID is found, otherwise returns false.
- **toString()**
  Overrides the toString() method to provide a string representation of the word and its associated document IDs.
- **getWordDocCount()**
  Returns the number of documents associated with the word by returning the size of the docIds linked list.

11.Class Query

- **static InvertedIndexBST invertedBST**
  A static instance of InvertedIndexBST used to process queries on a binary search tree-based inverted index.
- **Query(InvertedIndex inverted)**
  Initializes the inverted field with the provided InvertedIndex instance.
- **BooleanQuery(String Query)**
  Determines the type of query (AND, OR, or mixed) and processes it accordingly:
  - Calls andQuery(String) for AND-only queries.
  - Calls ORQuery(String) for OR-only queries.
  - Calls MixedQuery(String) for queries containing both AND and OR.
- **existsInResult(LinkedList<Integer> list, Integer element)**
  Checks if a specific document ID (element) exists in a list:
  - Traverses the list to find a match for the given document ID.
  - Returns true if the document ID exists, otherwise returns false.
- **ORQuery(String query)** Processes an OR query by:
  - Splitting the query string into terms using the "OR" delimiter.
  - For each term, retrieves the document IDs from the binary search tree.
  - Combines the document IDs for all terms using the ORQuery(LinkedList<Integer>, LinkedList<Integer>) method.
  - Returns the list of document IDs that contain any of the terms.
- **ORQuery(LinkedList<Integer> A, LinkedList<Integer> B)**
  Performs a union operation on two lists of document IDs:
  - Adds all unique document IDs from both A and B to the result.
  - Returns the resulting list of document IDs.
- **andQuery(String Query)**
  Processes an AND query by:
  - Splitting the query string into terms using the "AND" delimiter.
  - Retrieves the document IDs for the first term.
  - Iteratively intersects the result with document IDs of subsequent terms using andQuery(LinkedList<Integer>, LinkedList<Integer>).
  - Returns the list of document IDs that contain all the terms.
- **andQuery(LinkedList<Integer> A, LinkedList<Integer> B)**
  Performs an intersection operation on two lists of document IDs:
  - Iterates through A and checks if each document ID exists in B.
  - Adds the common document IDs to the result list.
  - Returns the resulting list of common document IDs.

- **MixedQuery(String Query)**
  Processes a mixed query containing both AND and OR operations:
  - Splits the query into parts using the "OR" delimiter.
  - Processes each part as an AND query using andQuery(String).
  - Combines the results using ORQuery(LinkedList<Integer>, LinkedList<Integer>).
  - Returns the list of document IDs matching the query.

12.Class Rank

- **Rank(InvertedIndexBST invBST, Index indx, String query)**

Initializes a Rank object with the provided inverted index, document index, and query string.
  - invBST: The inverted index for term-to-document mapping.
  - indx: The document index for managing document details.
  - query: The query string to rank documents.
- **String query**
  The query string used for ranking documents.
- **InvertedIndexBST invBST**
  An instance of InvertedIndexBST for accessing the inverted index stored as a binary search tree.
- **Index indx**
  An instance of Index for retrieving document details.
- **LinkedList<Integer> inQueryDocs**
  A linked list of document IDs relevant to the query.
- **LinkedList<DocRank> rankDocs**
  A linked list storing the ranked documents with their IDs and scores.
- **TermFrequency(Document doc, String term)**
  Calculates the term frequency of a given word (term) in a document (doc):
  - Counts the occurrences of the term in the document's word list.
  - Returns the frequency as an integer.
- **RankScore(Document doc, String query)**
  Computes the rank score for a document based on the given query:
  - Splits the query into individual terms.
  - Sums the term frequencies of all query terms in the document.
  - Returns the total rank score.
- **RankQuery(String query)**
  Ranks documents based on their relevance to the query:
  - Retrieves document IDs for all terms in the query using invBST.
  - Calculates the rank score for each document.

    - Inserts each document and its rank score into rankDocs in sorted order.
- **existsInList(LinkedList<Integer> list, int value)**
  Checks if a given document ID exists in a linked list:
    - Traverses the list and returns true if the value is found; otherwise, returns false.
- **insertDR_IntoSortedList(DocRank dr)**
  Inserts a DocRank object (containing a document ID and its rank score) into the rankDocs list in descending order of rank:
    - Finds the correct position based on the rank value.
    - Maintains the sorted order of rankDocs.
- **displayRankedDocs()**
  Displays the ranked documents:
    - Prints the document IDs and their respective scores in descending order.
    - Prints "empty list" if no ranked documents are present.

13.Class DocRank

- **DocRank(int id, int rank)**
  Initializes a DocRank object with the specified document ID and rank score.
    - id: The unique identifier for the document.
    - rank: The calculated rank score of the document.
- **int id:** The ID of the document.
- **int rank:** The rank score of the document.
- **displayIdRank():** Prints the document ID and rank score in a formatted manner.

14.Class Menu

- **public static void main(String[] args)**

The Menu class uses the main method to provide a console-based interface. Each option is implemented with appropriate method calls from the Processor, Query, and Rank classes, with these key functionalities:

1. Initialization

- Load Data: Loads stop words and a dataset using the Processor class.
- Setup Search Objects: Initializes Query and Rank objects for boolean and ranked searches.

2. Menu Options

1. Retrieve Term:
   Search for a word and display its documents using one of three data structures:
   - Linked List Index.
   - Inverted Index (Linked List).
   - Inverted Index (BST).
2. Boolean Retrieval:
   Perform logical queries using operators like AND/OR.
3. Ranked Retrieval:
   Rank documents based on their relevance to a given query.
4. Indexed Documents:
   Displays information about indexed documents using display method in class index and shows total tokens and vocabulary using counters in class processor.
5. Indexed Tokens:
   Displays the indexed tokens, allowing the user to choose between Linked List (L) or BST (B) for viewing.
6. Exit:
   Quit the program.

Performance analysis

1. Index Retrieval (Using List of Lists)

- **Overview**:
  The Index class manages a collection of documents, where each document contains a list of words. Retrieving document data involves iterating through the list of documents and performing operations such as word lookup or calculating term frequencies.

- **Search Complexity (Term retrieving)**:
  For a single document, searching for a word in the list of words is linear **(O(n)),** where n is the number of words in the document.
  For all documents, the complexity is O(d * n), where d is the number of documents.

- **Insertion Complexity**:
  Adding a document involves iterating through the linked list to check for duplicates by comparing the ID of the new document with the IDs of existing documents. This process requires examining each element in the list, resulting in a time complexity of **O(n)**.

- **Scalability**:
  As the number of documents and their word counts grow, retrieval performance decreases due to the need for sequential scanning.

Strengths	Weaknesses
Simple implementation and effective for small datasets.	Inefficient for large-scale search operations due to sequential traversal of words and documents.

2. Inverted Index Retrieval

- **Overview**:

  The InvertedIndex class implements a word-to-document mapping using a linked list of WordEntry objects. Each WordEntry contains a word and a list of document IDs where the word appears. Retrieving documents for a query involves searching for the word in the list and retrieving its associated document IDs.

- **Search Complexity(Term retrieving):**
  Searching for a word in the Inverted Index takes **O(n)**, where n is the number of words. Once the word is found, retrieving and displaying its associated document IDs takes O(d), where d is the number of documents per word, O(n+d) in Total.

- **Insertion Complexity**:
  Adding a new word or updating an existing one involves searching for the word in the list, which takes **O(n)**. If the word already exists, its list of document IDs is updated by adding the new document ID, which is a straightforward operation.

- **Scalability**:
  Efficient for moderately large datasets, as word lookup is faster than iterating through all documents.

Strengths	Weaknesses
Improved Retrieval Performance Over Basic Indexing	Not Ideal for Extremely Large Datasets
Supports Boolean Queries	Does not inherently provide order, leading to inefficiencies in word lookups.

3. Inverted Index Retrieval (Using Binary Search Tree)

- **Overview:**

The InvertedIndexBST class organizes the inverted index using a binary search tree (BST). Each node in the BST contains a word and its associated document IDs. Retrieval involves traversing the tree based on the word's key.

- **Search Complexity(Boolean query & Term retrieving):**
  Each term in the query is searched in the BST with a complexity of **O(log n)**, where n is the number of words in the BST. Once found, the associated document list is retrieved, and operations like intersection or union of document lists are performed.

- **Insertion Complexity:**
  Adding a new word or updating an existing one involves inserting or searching in the BST, which is **O(log n)** for a balanced tree.

- **Scalability:**

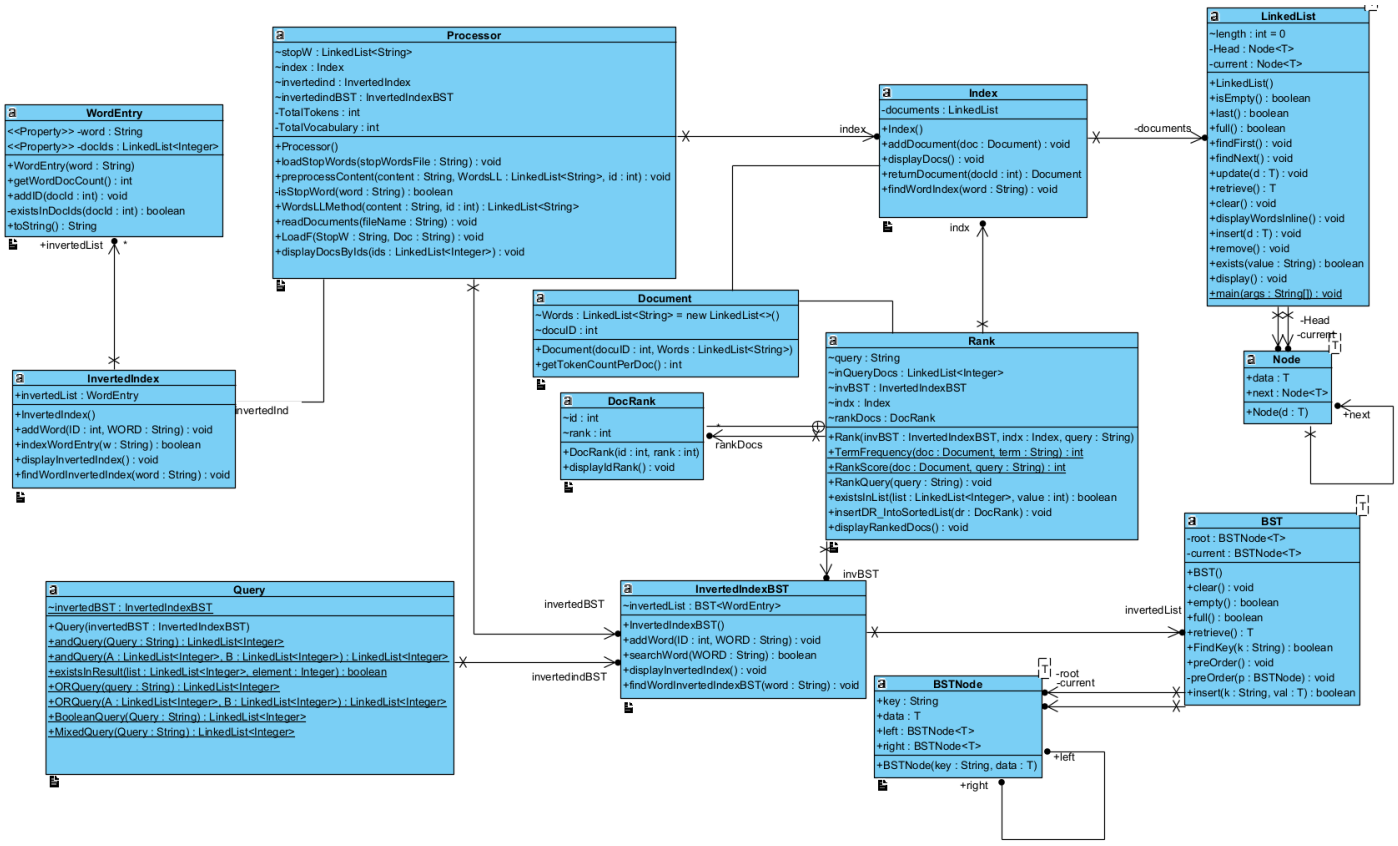
Highly scalable due to the logarithmic time complexity for word searches, making it suitable for large datasets.

Strengths	Weaknesses
Efficient word lookup and document retrieval for large datasets.	Imbalance in Tree Structure (when the height of the tree becomes equal to the number of nodes)
Maintains an ordered structure, enabling fast search operations.	No Direct Access by Index.

Conclusion

The inverted index retrieval using a binary search tree provides the best performance for large-scale datasets due to its logarithmic search complexity. The basic index retrieval is only suitable for small datasets, while the inverted index with a list strikes a balance between simplicity and performance for medium-sized datasets.

Class Diagram



GitHub Private Repository

Link: https://github.com/ghadee3r/DataStructuresProject.git