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

CSC212 Data Structures

Assignment Documentation
Simple Search Engine

Name	ID
Layal Saeed Alghamdi	444201202
Raghad Fares Almutairi	443200793

Table of Contents

Target Audience

Search engines are one of the most helpful programs that targets all individuals who need to search through a collection of documents for specific information in general, however in our case this “simple search engine” has been targeted to more specific individuals such as:

- Academic Researchers: Individuals conducting research that involves text analysis, who may use the search engine to quickly locate relevant literature or data.
- Data Scientists: Professionals who may utilize the search engine to analyze text data and extract valuable insights from large document collections.
- Students: Learners who need to search through course materials, papers, and other academic resources for specific terms or topics.

Purpose of the Documentation

As any program goes, we documented everything for many purposes, some of them are:

- Usage Explanation: Provide clear instructions on how to set up and use the search engine, including how to perform searches and interpret results.
- Design Overview: Describe the architecture of the search engine, detailing how the index and inverted index are structured and how linked lists and BSTs are utilized.
- Implementation Details: Offer insights into the code structure, including main classes and methods, and how they interact with one another.
- Performance Considerations: Discuss efficiency, scalability, and provide recommendations for optimization.

- Encourage Developer Contributions: Provide clear guidelines so that developers can contribute to the project, enhance it more and make it an official program the can be utilized on a daily basis.

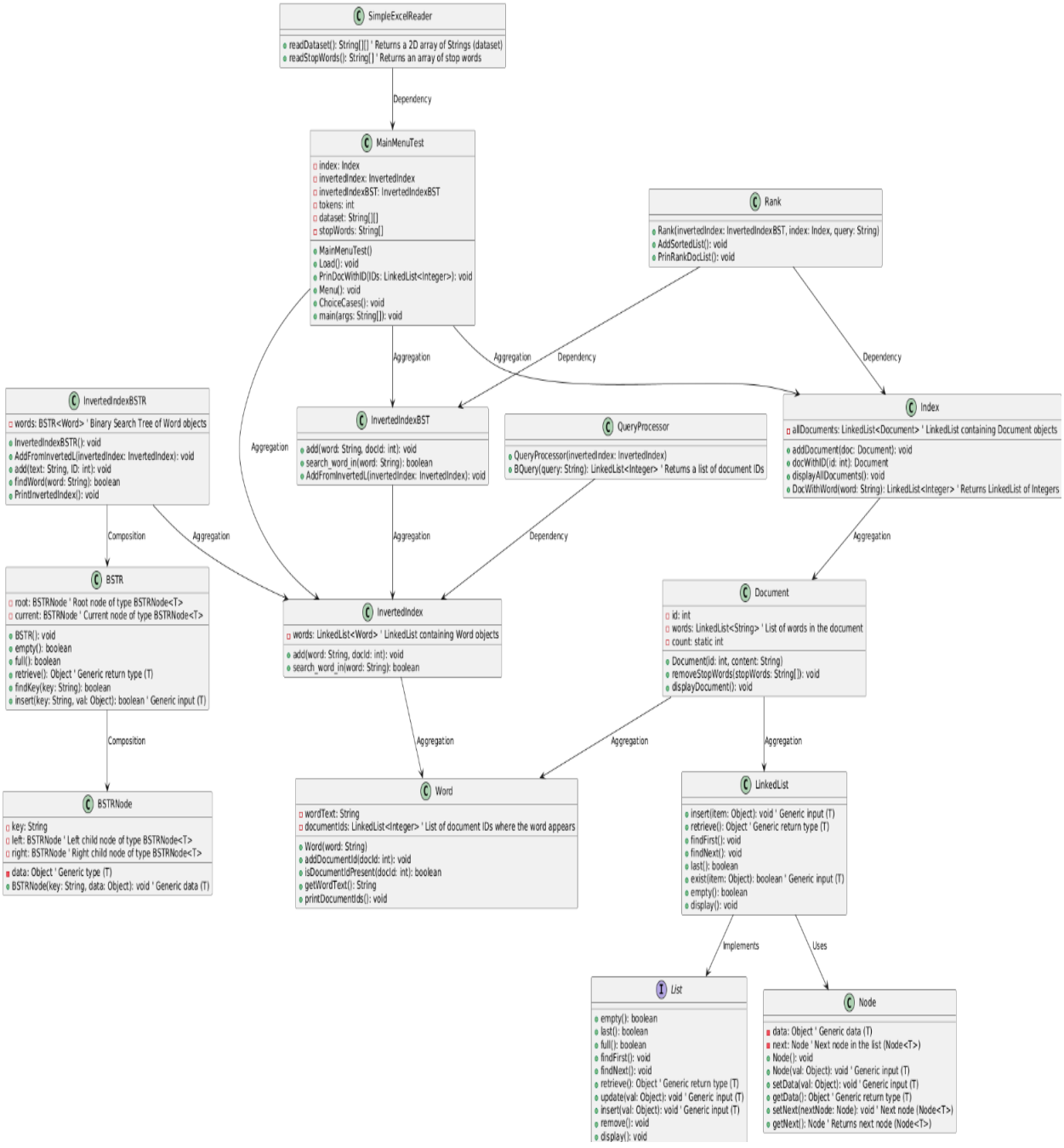
Introduction

In today's age, the ability to efficiently retrieve information is very easy. However, we conducted a small simple yet powerful search engine. This program allowed us to apply fundamental data structures and algorithms, focusing on linked lists and binary search trees (BSTs), to create an effective tool for searching through a collection of documents.

Our simple search engine is designed to perform keyword searches, enabling users to quickly identify which documents contain specific terms and to see the context in which those terms appear. By using linked lists, we efficiently manage the storage of document IDs and their associated occurrences, while the BSTs provide a structured and fast way to retrieve and organize this data.

This simple program gives you the efficiency of its ability to deliver quick and accurate results, making it a valuable resource for users seeking specific information within extensive document collections.

Class Diagram



Important Note: In this UML diagram, the type of `Object` is used as a placeholder where generics (e.g., `<T>`) would typically apply in a programming context. This is done to maintain compatibility with PlantUML, as some versions may not support explicit generic type syntax.

Where you see `Object`, it represents a generic type of parameter (`T`) that would be defined in the actual implementation to support type safety. For example:

- `BSTR` and `BSTRNode` would be `BSTR<T>` and `BSTRNode<T>`.
- `LinkedList` and `Node` would be `LinkedList<T>` and `Node<T>`.

Key Points About the Design

1. **Classes and Attributes**:
    a. Each class has attributes (e.g., `- index: Index` in `MainMenuTest`) and methods (e.g., `+ MainMenuTest()`).
    b. Attributes are marked with visibility (`-` for private, `+` for public).
2. **Relationships**:
    a. **Aggregation (^)**: Indicates that one class uses another as a part of its structure. For example:
        i. `MainMenuTest` aggregates `Index`, `InvertedIndex`, and `InvertedIndexBST`.
        ii. `Document` aggregates `Word` and `LinkedList<String>`.
    b. **Dependency (^)**: Shows that one class depends on another for its operation but does not directly own it. For example:
        i. `QueryProcessor` depends on `InvertedIndex`.
        ii. `Rank` depends on `Index` and `InvertedIndexBST`.
    c. **Composition (^)**: A stronger form of aggregation indicating ownership, as seen with `BSTRNode<T>` being part of `BSTR<T>`.
    d. **Implements (^)**: Indicates implementation of an interface, such as `LinkedList<T>` implementing `List<T>`.
    e. **Uses (^)**: Represents loose association or helper class usage, as seen with `LinkedList<T>` using `Node<T>`.

3. **Hierarchical Flow**:
    a. The flow from top (`MainMenuTest`) to bottom (`Node<T>`) shows the program's organization:
        i. `MainMenuTest` is the entry point, connecting all key components.
        ii. Data structures (`LinkedList<T>`, `BSTR<T>`) form the backbone for storage.
        iii. Supporting classes like `SimpleExcelReader` handle external operations like reading files.
4. **Specialized Structures**:

a. `BSTR` and `InvertedIndexBSTR`: Showcases custom implementations of binary search trees for specific operations.
b. `Word`: Stores words and associated document IDs, critical for inverted index functionality.

System Overview

SimpleExcelReader Class

1. Purpose

The SimpleExcelReader class is designed to read data from a CSV file and a stop words text file. It extracts document information from the CSV and a list of stop words from the text file.

2. Attributes
   - private static final String DEFAULT_DATASET_FILE: Constant for the default path to the dataset CSV file.
   - private static final String DEFAULT_STOPWORDS_FILE: Constant for the default path to the stop words text file.
3. Methods
   1. public static String[][] readDataset()
      - Purpose: Reads the dataset from the default CSV file.
      - Return: A 2D array containing document IDs and their corresponding content.
   2. public static String[][] readDataset(String filePath)
      - Purpose: Reads the dataset from a specified CSV file.
      - Parameters: filePath - The path to the CSV file.
      - Return: A 2D array containing document IDs and their corresponding content.
      - Behavior:
        - Counts valid lines (skipping empty lines and the header).
        - Initializes a 2D array to hold the data.
        - Reads the file again to populate the array.
   3. public static String[] readStopWords()
      - Purpose: Reads stop words from the default stop words file.
      - Return: An array of stop words.
   4. public static String[] readStopWords(String filePath)
      - Purpose: Reads stop words from a specified text file.
      - Parameters: filePath - The path to the stop words file.
      - Return: An array of stop words.
      - Behavior:
        - Counts valid stop words (skipping empty lines).
        - Initializes an array to hold the stop words.
        - Reads the file again to populate the array, converting all words to lowercase.

LinkedList Class

1. Purpose

The LinkedList class is a generic implementation of a singly linked list data structure.

2. Attributes

- Node<T> head: The first node in the linked list, representing the start of the list.
- Node<T> current: A pointer to the current node during traversal, allowing operations like retrieval and updating of the current element.

3. Constructor

- public LinkedList(): Initializes an empty linked list by setting both head and current to null.

4. Methods

- public boolean empty():
  - Checks if the list is empty by evaluating if the head is null.
- public boolean last():
  - Determines if the current element is the last element in the list by checking if current.next is null.
- public boolean full():
  - Always returns false since a linked list can grow dynamically and is never "full."
- public void findFirst():
  - Resets the current pointer to the head of the list.
- public void findNext():
  - Advances the current pointer to the next node in the list.
- public T retrieve():
  - Returns the data contained in the current node. If the current node is null, it returns null.
- public void update(T val):
  - Updates the data of the current node with the provided value if the current node is not null.
- public boolean exist(T x):
  - Checks if a given element exists in the list by traversing from the head to the end.
- public void insert(T data):
  - Inserts a new node with the specified data at the end of the list.
  - If the list is empty, the new node becomes both the head and current. If not, it traverses to the end and adds the new node.
- public void remove():
  - Removes the current node from the list.
  - Handles two cases: removing the head node or removing a non-head node by traversing to find the previous node.
- public void display():
  - Prints all elements in the list, starting from the head to the end. Each element is printed followed by a space.

List<T> Interface

1. Purpose

The List<T> interface defines a generic contract for implementing list data structures in Java.

2. Methods

The List<T> interface outlines the following methods:

- boolean empty():
  - Purpose: Checks if the list is empty.
  - Return: true if the list contains no elements; otherwise, false.
- boolean last():
  - Purpose: Checks if the current element is the last element in the list.
  - Return: true if the current element is the last; otherwise, false.
- boolean full():
  - Purpose: Determines whether the list is full.
  - Return: Always returns false for linked list implementations, indicating that the list can grow dynamically.
- void findFirst():
  - Purpose: Sets the current pointer to the first element in the list.
- void findNext():
  - Purpose: Moves the current pointer to the next element in the list.
- T retrieve():
  - Purpose: Retrieves the data of the current node.
  - Return: The data of the current node, or null if the current node is not set.
- void update(T val):
  - Purpose: Updates the current node's data with the provided value.
- void insert(T val):
  - Purpose: Inserts a new element into the list.
- void remove():
  - Purpose: Removes the current node from the list.
- void display():
  - Purpose: Displays all elements in the list, typically in a formatted output.

Node<T> Class

1. Purpose

The Node<T> class represents a single element in a linked data structure, such as a linked list. Each node contains data of a generic type and a reference to the next node in the sequence.

2. Attributes

- public T data: This attribute holds the value stored in the node, allowing it to be of any type due to the use of generics.
- public Node<T> next: This attribute is a reference to the next node in the linked structure. If it's the last node, this reference will be null.

3. Constructors

- public Node():
    - Default constructor that initializes data to null and next to null. This constructor is useful for creating empty nodes.
- public Node(T val):
    - Constructor that initializes the node with a specified value (val) and sets next to null. This constructor allows for easy creation of nodes with predefined data.

4. Methods

- public void setData(T val):
    - Sets the data of the node to the specified value (val).
- public T getData():
    - Returns the data stored in the node.
- public void setNext(Node<T> nextNode):
    - Sets the next reference of the node to point to another node (nextNode).
- public Node<T> getNext():
    - Returns the reference to the next node.

Document Class

1. Purpose

The Document class is designed to represent a textual document, encapsulating its content and providing functionality to process, filter, and display the words contained within it.

2. Attributes

- int id: Unique identifier for the document.
- LinkedList<String> words: A linked list that stores the words extracted from the document content.
- static int count: A static counter that keeps track of the total number of tokens.

3. Constructor

- Document(int id, String content):
    - Initializes the document's ID.
    - Creates an empty LinkedList for storing words.
    - Processes the provided content to populate the words list with individual words.

4. Methods

- private void processDocumentContent(String content):
    - Cleans the input content by:
        - Replacing apostrophes and hyphens with spaces.
        - Converting the content to lowercase.
        - Removing punctuation and non-alphanumeric characters using the removePunctuation method.
        - Splitting the cleaned content into individual words and adding them to the words linked list.
        - Updates the static count with the number of words processed.
- private String removePunctuation(String text):
    - Uses a regex to remove all characters that are not alphanumeric or whitespace, effectively cleaning up the text for further processing.
- public void removeStopWords(String[] stopWords):
    - Iterates through the words linked list and filters out any words that are present in the provided stopWords array.
    - Creates a new linked list (filteredWords) to store only the non-stop words, effectively updating the words list.
- public void displayDocument():
    - Prints the document ID and the processed content (i.e., the words in the linked list) to the console.

Index Class

1. Purpose

The Index class is designed to manage a collection of Document instances. It facilitates the addition, retrieval, and display of documents while providing functionality to search for documents containing specific words.

2. Attributes

- LinkedList<Document> allDocuments: A linked list that stores all Document objects.

3. Constructor

- Index(): Initializes the Index instance by creating a new empty LinkedList that will hold the documents.

4. Methods

- public void addDocument(Document document):
  - Accepts a Document object as a parameter and adds it to the allDocuments linked list. This method allows the index to grow as new documents are added.
- public Document docWithID(int ID):
  - Searches for a document by its unique ID.
  - If the linked list is empty, it prints a message indicating that the document does not exist and returns null.
  - Iterates through the allDocuments list to find a document with the specified ID, returning the document if found. If not found after traversing the list, it returns null.
- public void displayAllDocuments():
  - Prints a header indicating the content of the index.
  - Iterates through the allDocuments linked list and calls the displayDocument() method of each Document to show its details.
- public LinkedList<Integer> DocWithWord(String word):
  - Searches for all documents that contain a specific word.
  - Initializes an empty LinkedList<Integer> to store the IDs of matching documents.
  - If the allDocuments list is empty, it returns the empty linked list.
  - Iterates through each document, checking if the word exists in the document's words linked list (after converting it to lowercase and trimming whitespace).
  - If the word exists, it adds the document's ID to the Ans linked list.
  - Returns the linked list of document IDs that contain the specified word.

Word Class

1. Purpose

The Word class is designed to represent a single word and track its occurrences across different documents.

2. Attributes

- String wordText: Stores the actual word.
- LinkedList<Integer> documentIds: A linked list that holds the IDs of documents where the word appears. This enables efficient tracking of word occurrences across multiple documents.

3. Constructor

- Word(String word):
    - Initializes the wordText attribute with the provided word.
    - Creates an empty LinkedList for storing document IDs.

4. Methods

- public void addDocumentId(int docId):
    - Adds a document ID to the documentIds list if it is not already present. This helps maintain a unique list of document IDs associated with the word.
- public boolean isDocumentIdPresent(int docId):
    - Checks if a given document ID is already in the documentIds list.
    - Iterates through the linked list of document IDs to determine presence, returning true if found and false otherwise.
- public String getWordText():
    - Returns the text of the word stored in wordText.
- public void printDocumentIds():
    - Prints the word and the list of document IDs where it appears. This method provides a simple way to visualize the association between the word and its corresponding documents.
- public boolean isWordPresentInDocument(String content):
    - Searches for the word in a given document content by splitting the content into words and checking for an exact match.
    - Returns true if the word is found in the document, and false otherwise.
- public void display():
    - Displays the word along with its associated document IDs in a formatted output.

InvertedIndex Class

1. Purpose

The InvertedIndex class is designed to implement an inverted index data structure, which allows for efficient retrieval of documents based on keywords. It maps words to their corresponding document IDs.

2. Attributes

- LinkedList<Word> words: A linked list that stores Word objects, each representing a unique word along with the IDs of documents that contain that word. This structure allows for dynamic storage and retrieval of words and their associations.

3. Constructor

- InvertedIndex():
    - Initializes a new empty LinkedList to hold the words and their associated document IDs.

4. Methods

- public void add(String text, int docId):
    - Adds a word and its associated document ID to the inverted index.
    - First, it checks if the word already exists using the search_word_in method.
        - If the word exists, it retrieves the existing Word object and adds the document ID to it.
        - If the word does not exist, it creates a new Word object, adds the document ID, and inserts the new word into the linked list.
- public boolean search_word_in(String w):
    - Searches for a specified word in the inverted index.
    - Iterates through the words linked list, comparing each word's text to the input word.
    - Returns true if the word is found, else it returns false.
- public Word retrieve():
    - Returns the current Word object from the linked list allowing access to the word data currently being processed.
- public void displayInvertedIndex():
    - Displays all words stored in the inverted index along with their associated document IDs.
    - Iterates through the words linked list and calls the printDocumentIds method of each Word to show which documents contain the word.

InvertedIndexBST Class

1. Purpose

The InvertedIndexBST class implements an inverted index using a binary search tree (BST) to efficiently map words to their corresponding document IDs. This structure allows for quick insertion and retrieval of words and supports efficient searching.

2. Attributes

- **private BST<Word> words**: A binary search tree that stores Word objects, each representing a unique word and its associated document IDs.

3. Constructor

1. **InvertedIndexBST()**: Initializes an empty BST for storing words.

4. Methods

2. **public void add(String text, int docId)**:
    - Adds a word and its associated document ID to the inverted index.
    - If the word already exists, it retrieves the Word object and adds the document ID. If not, it creates a new Word, adds the ID, and inserts it into the BST.
3. **public boolean search_word_in(String w)**:
    - Searches for a specified word in the BST. Returns true if found and false else.
4. **public Word retrieve(String text)**:
    - Retrieves the Word object associated with the specified text.
5. **public void displayInvertedIndex()**:
    - Displays all words in the inverted index along with their associated document IDs. It first checks if the index is empty and then iterates through the words in sorted order.

QueryProcessor Class

1. Purpose

The QueryProcessor class is designed to handle and evaluate user queries containing logical operators (AND, OR, NOT) to return a linked list of document IDs that match the specified conditions.

2. Attributes

- static InvertedIndex invert: A static reference to an InvertedIndex object, which stores the mapping of words to document IDs. This allows the query processor to access the underlying data structure for document retrieval.

3. Constructor

- public QueryProcessor(InvertedIndex invert): Initializes the QueryProcessor with a provided InvertedIndex instance, allowing it to perform queries on that specific index.

4. Methods

- public static LinkedList<Integer> BQuery(String Query):
    - Evaluates the user's input query.
    - Checks for the presence of logical operators (AND, OR) and delegates to the appropriate method (AndQ, OrQ, or MixedQ).
- public static LinkedList<Integer> MixedQ(String Query):
    - Handles queries that contain both AND and OR operators.
    - Splits the query by the OR operator, processes each segment with AND, and then combines the results using OR.
- public static LinkedList<Integer> AndQ(String Query):
    - Processes a query that uses the AND operator.
    - Splits the query into individual words, retrieves document IDs for each word, and finds the intersection of document IDs across all words.
- public static LinkedList<Integer> AndQ(LinkedList<Integer> FirstIntersection, LinkedList<Integer> SecondIntersection):
    - Finds the intersection of two linked lists of document IDs, returning only those that are present in both lists.
- public static boolean foundElement(LinkedList<Integer> FullIntersection, Integer docID):
    - Checks if a document ID exists in a given linked list, returning true if found, this makes sure there are no duplicates.
- public static LinkedList<Integer> OrQ(String Query):
    - Processes a query using the OR operator.
    - Similar to AndQ, it retrieves document IDs for each word and combines them into a single list, ensuring no duplicates.
- public static LinkedList<Integer> OrQ(LinkedList<Integer> FirstIntersection, LinkedList<Integer> SecondIntersection):
    - Combines two linked lists of document IDs into a single list, ensuring no duplicates.

Extra for NOT opperator, but not included in the main test.

- public static LinkedList<Integer> NotQ(String Query):
    - Processes a query using the NOT operator.
    - Retrieves document IDs for the included term and excludes those for the excluded term.
- public static LinkedList<Integer> Subtract(LinkedList<Integer> IncludedDocs, LinkedList<Integer> ExcludedDocs):
    - Returns a list of document IDs from IncludedDocs that are not in ExcludedDocs.
- public static boolean foundElementForNot(LinkedList<Integer> list, Integer docID):
    - Checks if a document ID exists in a list, specifically for use in NOT operations.

InvertedIndexBSTR Class

1. Purpose

The InvertedIndexBSTR class serves as an inverted index **specifically tailored** for the **Rank class**..

2. Attributes

- **BSTR<Word> words**: A binary search tree (BSTR) that stores Word objects, each associated with a unique word and a list of document IDs where the word appears.

3. Constructor

- **public InvertedIndexBSTR()**: Initializes the words attribute as an empty binary search tree.

4. Methods

1. **public void AddFromInvertedL(InvertedIndex inverted)**
    - **Purpose**: Transfers entries from another InvertedIndex instance to this inverted index.
    - **Parameters**: inverted - The source inverted index from which entries are copied.
    - **Behavior**: Iterates through the words in the provided inverted index, inserting them into the words BSTR.
2. **public void add(String text, int ID)**
    - **Purpose**: Adds a new word and its associated document ID to the index.
    - **Parameters**: text - The word to add; ID - The document ID where the word is found.
    - **Behavior**: If the word does not already exist, it creates a new Word object and inserts it into the BSTR. If it does exist, it adds the document ID to the existing Word.
3. **public boolean findWord(String word)**
    - **Purpose**: Checks if a specific word exists in the index.
    - **Parameters**: word - The word to search for.
    - **Return**: true if the word is found; false otherwise.
4. **public void PrintInvertedIndex()**
    - **Purpose**: Prints information about the inverted index.
    - **Behavior**: Checks if the index is null or empty and prints corresponding messages.

DocumentRank and Rank Classes

1. DocumentRank Class

Purpose:

The DocumentRank class is designed to represent a document with its associated rank. This class allows for easy storage and manipulation of document IDs and their corresponding ranks, which later will be used for sorting and displaying results.

Attributes:

- **int id**: The unique identifier for the document.
- **int rank**: The rank or score of the document.

Constructor:

- **public DocumentRank(int id, int rank)**: Initializes a DocumentRank object with a specified document ID and rank.

Methods:

- **public void display()**: Outputs the document ID and its rank to the console.

2. Rank Class

Purpose:

The Rank class processes user queries to rank documents based on the frequency of query terms. It utilizes an inverted index to efficiently retrieve documents and calculate their ranks.

Attributes:

- **static String Query**: Stores the user's query.
- **static InvertedIndexBSTR inverted**: Represents the inverted index for document retrieval.
- **static Index index**: Represents the index that contains documents.
- **static LinkedList<Integer> AllDoc**: Stores all document IDs that match the query.
- **static LinkedList<DocumentRank> AllRankDoc**: Stores the final sorted list of documents based on their ranks.

Constructor:

- **public Rank(InvertedIndexBSTR inverted, Index index, String Query)**: Initializes the Rank class with the inverted index, document index, and user query.

Methods:

- **public static void PrinRankDocList()**: Prints the list of ranked documents. If no documents are present, it indicates that there are no results.
- **public static Document DocWithID(int ID)**: Retrieves the complete document object given its ID.
- **public static int WordOccurrenceInDoc(Document Doc, String Word)**: Counts the frequency of a specific word in a given document.
- **public static int GetDocRank(Document Doc, String Query)**: Calculates the rank of a document based on the occurrences of words in the user's query (more than 1 word).
- **public static void RankQuery(String Query)**: Processes the query to retrieve document IDs from the inverted index, adding them to the AllDoc list.
- **public static void AddFirstList(LinkedList<Integer> A)**: Adds document IDs from the linked list A to AllDoc, ensuring no duplicates and maintaining order.
- **public static boolean FoundR(LinkedList<Integer> Result, Integer ID)**: Checks if a document ID already exists in the results list.

- public static void AddSortedIDList(Integer ID): Inserts a document ID into AllDoc while maintaining sorted order.
- public static void AddSortedList(): Fills AllRankDoc with documents and their ranks based on the query.
- public static void AddSortedList(DocumentRank DR): Inserts a DocumentRank object into AllRankDoc while maintaining sorted order based on rank.

MainMenuTest Class

1. Purpose

The MainMenuTest class serves as the main entry point for the application, providing a user interface to interact with an inverted index and perform various document retrieval operations. It facilitates loading datasets, managing stop words, and executing different types of search queries.

2. Attributes

- static Index index: An instance of the Index class, used to manage and retrieve documents.
- InvertedIndex invertedIndex: An instance of the InvertedIndex class, used for storing and searching words associated with document IDs.
- InvertedIndexBST invertedIndexBST: An instance of the InvertedIndexBST class, providing an alternative structure for storing the inverted index.
- Static int DocNum: A counter to track the number of documents processed.
- String[][] dataset: An array to hold the dataset read from a CSV file.
- String[] stopWords: An array of stop words read from a text file.

3. Constructor

- public MainMenuTest(): Initializes the class, setting up the index and inverted indexes, and loading the dataset and stop words using the SimpleExcelReader.

4. Methods

1. public void Load()
    - Purpose: Populates the index and inverted indexes with the data from the dataset.
    - Behavior: Iterates through the dataset, creates Document objects, removes stop words, and adds words to the inverted indexes.
2. public void PrinDocWithID(LinkedList<Integer> IDs)
    - Purpose: Displays documents corresponding to a list of IDs.
    - Parameters: IDs - A linked list of document IDs.
    - Behavior: If the list is empty, it prints a message. Otherwise, it retrieves and prints the document IDs.
3. public static void Menu()
    - Purpose: Displays the main menu options for user interaction.
    - Behavior: Prints available options for retrieving documents, boolean retrieval, ranked retrieval, and more.
4. public static void ChoiceCases()
    - Purpose: Handles user choices from the main menu.
    - Behavior:
        - Initializes the MainMenuTest instance and loads the data.
        - Uses a loop to display the menu and respond to user input.
        - Depending on the user's choice, it executes different functionalities, such as word retrieval, boolean queries, and ranked retrieval.
5. public static void main(String[] args)
    - Purpose: The main starter of the application.
    - Behavior: Calls the ChoiceCases() method to start the menu-driven interface.

Performance analysis

Index Class

1. Constructor (Index())
    - Time Complexity: O(1)
2. addDocument(Document document)
    - Time Complexity: O(1) for adding to the end of the linked list.
3. docWithID(int ID)
    - Time Complexity: O(n), Traverses the linked list to find a document by its ID. In the worst case, it may need to check each document, making it linear with respect to the number of documents.
4. displayAllDocuments()
    - Time Complexity: O(n), Traverses the entire linked list to display all documents, resulting in a linear time complexity.
5. DocWithWord(String word)
    - Time Complexity: O(n * m), where m is the average time complexity of the **exist** method in the words class, because this method has to also traverses the linked list, but for each document, it checks if the word exists, If exist is O(m).

Summary of Performance

- Constructor: O(1)
- Adding a Document: O(1)
- Finding a Document by ID: O(n)
- Displaying Documents: O(n)
- Finding Documents with a Word: O(n * m)

InvertedIndex Class

1. Constructor (InvertedIndex())
    - Time Complexity: O(1)
2. add(String text, int docId)
    - Time Complexity: O(n) for searching the word, O(1) for inserting, because this method first checks if the word exists using search_word_in(), which has a linear time complexity due to traversing the linked list. If the word is found, it adds the document ID in constant time; otherwise, it creates a new Word object and inserts it, also in constant time.
3. search_word_in(String w)
    - Time Complexity: O(n), Traverses the linked list to find the word. In the worst case, it may need to check every word, leading to linear complexity
4. retrieve()
    - Time Complexity: O(1)
5. displayInvertedIndex()
    - Time Complexity: O(n), Iterates through the linked list to display all words and their associated document IDs, resulting in linear time complexity.

Summary of Performance

- Constructor: O(1)
- Adding a Word: O(n)
- Searching for a Word: O(n)
- Retrieving Current Word: O(1)
- Displaying Words: O(n)

Comparison of Index and InvertedIndex

1. Adding Documents
    - Index: O(1) for adding documents to the allDocuments list.
    - InvertedIndex: O(n) for adding a document because it needs to check if the word already exists.
2. Searching for Documents
    - Index: O(n) when searching for a document by ID.
    - InvertedIndex: O(n) when searching for a word, (finding words instead of documents).
3. Displaying Contents
    - Both classes have O(n) complexity for displaying their contents.

InvertedIndexBST Class

1. Constructor (InvertedIndexBST())
    - Time Complexity: O(1)
2. add(String text, int docId)
    - Time Complexity: O(log n) for average case, O(n) for worst case (unbalanced tree), this method first checks if the word exists using search_word_in(), which is O(log n) on average for a balanced BST. If the word is found, it retrieves it in O(log n) as well. Inserting a new word takes O(log n) on average due to the BST properties.
3. search_word_in(String w)
    - Time Complexity: O(log n) for average case, O(n) for worst case.
4. retrieve(String text)
    - Time Complexity: O(log n) for average case, O(n) for worst case.
5. displayInvertedIndex()
    - Time Complexity: O(n), because this method performs an in-order traversal of the BST to display all words. It requires visiting each node, leading to linear complexity.

Summary of Performance

- Constructor: O(1)
- Adding a Word: O(log n) average case, O(n) worst case
- Searching for a Word: O(log n) average case, O(n) worst case
- Retrieving a Word: O(log n) average case, O(n) worst case
- Displaying Words: O(n)

Comparison of InvertedIndex and InvertedIndexBST

1. Adding Word
    - InvertedIndex: O(n), for checking if it exists by searching through the linked list.
    - InvertedIndexBST: O(log n) average, O(n) in the worst case (if the tree becomes unbalanced), searching for the word.
2. Search for Word
    - InvertedIndex: O(n).
    - InvertedIndexBST: O(log n) average, O(n) in the worst case.
3. Retrieve Documents by Word
    - InvertedIndex: O(n), retrieving the list of document IDs associated with a word requires searching through the linked list.
    - InvertedIndexBST: O(log n) average, O(n) in the worst case , retrieving the document IDs from a balanced BST or an unbalanced BST.
4. Displaying Contents
    - Both classes have O(n) complexity for displaying their contents.

Comparison of All Three Classes

Now let's compare the three classes based on their performance:

Class	Add Complexity	Search Complexity	Retrieve Complexity	Display Complexity
Index	O(1)	O(n)	O(n * m)	O(n)
InvertedIndex	O(n)	O(n)	O(n * m)	O(n)
InvertedIndexBST	O(log n) average, O(n) worst	O(log n) average, O(n) worst	O(log n) average, O(n) worst	O(n)

Conclusion

1. **Efficiency in Adding Documents**:
    - The **Index class** is the most efficient for adding documents due to its O(1) complexity.
    - The **InvertedIndexBST** is also efficient when balanced, while the **InvertedIndex** is less efficient due to its O(n) complexity.
2. **Searching and Retrieving**:
    - The **InvertedIndexBST** offers efficient searching and retrieval due to its BST structure, making it preferable for applications requiring frequent word lookups.
    - The **InvertedIndex** has linear search complexity, making it less optimal for quick searches.
3. **Displaying Contents**:
    - All three classes have O(n) complexity for displaying their contents.

Overall, if your primary goal is efficient word management with fast searches and insertions, the **InvertedIndexBST** is the best option. However, if you need to quickly add documents without searching for words, the **Index class** is better suited. The **InvertedIndex** is less efficient overall and might be less advantageous for larger datasets.

Query Processor Performance Analysis

AND Queries

- **Operation**: The AndQ method processes queries containing the "AND" operator.
- **Time Complexity**:
  - Searching for each word in the inverted index (via invert.search_word_in) takes O(n) in the worst case for InvertedIndex, and O(log n) in average case for InvertedIndexBST.
  - The intersection operation (AndQ(FirstIntersection, SecondIntersection)) involves comparing elements from two linked lists. In the worst case, this takes O(m * k) where m is the size of the first list and k is the size of the second.
- **Overall**:
  - **Index**: O(n * m) (for search and intersection)
  - **InvertedIndex**: O(n * m) (for search) + O(m * k) (for intersection)
  - **InvertedIndexBST**: O(log n) (for search) + O(m * k) (for intersection)

OR Queries

- **Operation**: The OrQ method processes queries containing the "OR" operator.
- **Time Complexity**:
  - Similar to AND queries, searching for each word has the same complexities.
  - The union operation (OrQ) also involves checking for duplicates while combining two linked lists.
- **Overall**:
  - **Index**: O(n * m)
  - **InvertedIndex**: O(n * m)
  - **InvertedIndexBST**: O(log n) + O(m + k) for unioning.

Comparison of Performance

Operation	Index	InvertedIndex	InvertedIndexBST
AND Query	O(n * m)	O(n * m)	O(log n) + O(m * k)
OR Query	O(n * m)	O(n * m)	O(log n) + O(m + k)

Conclusion: Which Index is More Efficient?

1. **Adding Documents**:
   - The **Index class** is the most efficient for adding documents because it operates in O(1).
2. **Boolean Operations**:
   - For **AND** and **OR queries**, the **InvertedIndexBST** is significantly more efficient due to its logarithmic search times. This advantage grows as the number of documents increases.
3. **Overall Efficiency**:
   - The **InvertedIndexBST** is the best choice for Boolean operations due to its efficient searching capabilities, which dramatically reduce the time required for processing queries compared to the Index and InvertedIndex.

Usage Instructions for the Search Engine

Our simple search engine is designed to be user-friendly, by already providing a menu that provides the main functions, allowing for both simple and complex queries. However, there is a certain syntax that is preferable to be followed in order to get the wanted redults.
This section provides detailed instructions on how to effectively use the search engine, including how to input queries and what to expect in search results.

1. Inputting Queries

Boolean Queries

The search engine supports Boolean queries that allow users to combine terms using logical operators. Here's how to construct and input these queries:

- Operators:
  - AND: Returns documents containing all specified terms.
  - OR: Returns documents containing at least one of the specified terms.
  - Mixed: Returns documents containing the logical concination of both the specified terms and one of them based on the way it is written.
- Syntax:
  - Use logical operators between the terms in any case wanted(e.g., AND, OR).

Ranked Queries

The search engine can also handle ranked queries, which return results based on relevance.

- Input Format: Simply input the search terms separated by spaces (e.g., market sports).
- Ranking: Documents will be ranked based on the frequency of the search terms, with more relevant documents appearing at the top of the results, for documents that are on the same level of relevance, the document IDs are ordered ascendingly.

2. What to Expect in Search Results

When you execute a search query, the following information will typically be included in the results:

- Document List: A list of document IDs that match the query criteria.
- Ranking Information: For ranked queries, documents will be displayed in order of relevance, based on the scoring system used (e.g., term frequency).

Testing Process for the Search Engine

Testing is a critical phase in the development of the search engine to ensure its functionality, performance, and reliability. After each class we have made, we made sure to associate it with test class dedicated for that class that we made, in order to make sure that everything works well and coroporeates perfectly in the end result.

Test Classes

- InvertedIndexTest class
- InvertedIndexBSTTest class
- QueryProcessingTest
- RankTest

Conclusion

In the end, we tried to describe in detail all the key points of our search engine, this documentation has provided a comprehensive overview of the search engine's architecture, functionality, and testing processes. So, in conclusion this simple search engine architecture is designed to provide efficient indexing and retrieval of documents, enabling users to quickly access relevant information through both simple and complex queries. Key components such as the inverted index, document ingestion processes, and robust query handling ensure that the system operates effectively, even as data volumes grow.

Key Takeaways:

1. Efficiency: The use of an inverted index and binary search tree structure allows for rapid document lookups, significantly enhancing performance.

2. User-Centric Design: Support for Boolean queries and ranked results improves the overall user experience, catering to diverse search needs and preferences.

3. Scalability: The architecture is built to handle increasing data loads without compromising performance, making it suitable for a variety of applications.

- repository link: Our GitHub link :)

It was a private one as what we asked to provide but we made it now public so u can have the access to it.