((بسم الله الرحمن الرحيم)).

Name: Abdulrahman Ali Salem AL Qahtani.

ID: 443100902.

Name: Ibrahim Fahd Al Dawood.

ID: 442101872.

Report: Simple Search Engine Project

1. Introduction

The Simple Search Engine project aims to build a basic search engine capable of indexing, retrieving, and ranking documents based on user queries. This project emphasizes the use of fundamental data structures such as lists and binary search trees (BSTs) for efficient information retrieval.

2. Objectives

- **Document Processing**: Tokenize documents, remove punctuation, convert text to lowercase, and filter out stopwords.
- **Index Construction**: Build an index and an inverted index to facilitate document retrieval.
- **Query Processing**: Implement support for Boolean queries (AND, OR) and ranked retrieval using term frequency (TF).
- **Performance Enhancement**: Use BSTs for optimizing the search in the inverted index.
- **User Interaction**: Provide an interactive interface for querying the search engine.

3. System Overview

3.1 Data Structures Used

- **Map<Integer, String> originalDocuments**: Stores the original content of each document by its ID.
- **Map<Integer, List<String>> documentIndex**: Holds the processed words for each document.
- **Map<String, List<Integer>> invertedIndex**: Maps each term to a list of document IDs in which it appears.
- **Set<String> stopWords**: A set containing stopwords to exclude during document processing.

3.2 Algorithms Implemented

- **Document Tokenization**: Texts are split into words, and non-alphanumeric characters are removed.
- **Boolean Query Processing**: Handles AND and OR operations to retrieve sets of document IDs.
- **Term Frequency Calculation**: Scores documents based on the number of occurrences of query terms.
- **BST for Enhanced Search**: A binary search tree is used to enable more efficient lookups in the inverted index.

4. Implementation Details

4.1 Document Processing

- **Lowercasing**: All documents are converted to lowercase to ensure uniformity.
- **Punctuation Removal**: Regular expressions are used to strip non-alphanumeric characters.
- **Stopword Filtering**: Common words like "the," "is," and "and" are removed based on stop.txt.

4.2 Index Construction

- The inverted index is built by iterating over tokenized document words and mapping them to document IDs.

4.3 Query Processing

- **Boolean Retrieval**:
  - Supports queries using AND and OR.
  - Uses intersection for AND operations and union for OR.
- **Ranked Retrieval**:
  - Calculates a document's relevance score based on the term frequency of query terms.

4.4 BST Implementation

A BST is used to optimize the lookup of terms within the inverted index, providing a logarithmic search time.

5. Code Structure

The project consists of the following key components:

- **SimpleSearchEngine**: Main class containing methods for document processing, building indexes, and handling queries.
- **BSTNode and BSTInvertedIndex**: Inner classes used for managing the BST-based inverted index.

Main Code Features:

```
public static void main(String[] args) {
    SimpleSearchEngine engine = new SimpleSearchEngine();
    engine.loadStopWords("stop.txt");
    engine.readDocuments("dataset.csv");
    engine.buildInvertedIndex();
    Scanner scanner = new Scanner(System.in);


    while (true) {
        System.out.println("\nEnter your search query (type 'exit' to quit):");
        String query = scanner.nextLine();
        if (query.equalsIgnoreCase("exit")) {
            break;
        }


        // Display Boolean retrieval results
        Set<Integer> booleanResult = engine.processBooleanQuery(query);
        System.out.println("Boolean Query Result: " + (booleanResult.isEmpty() ? "No documents found" :
booleanResult));
```

```
        // Display ranked retrieval results
        List<Map.Entry<Integer, Integer>> rankedDocs = engine.rankDocuments(query);
        if (rankedDocs.isEmpty()) {
            System.out.println("No documents found for ranking.");
        } else {
            System.out.println("Ranked Documents:");
            for (Map.Entry<Integer, Integer> entry : rankedDocs) {
                System.out.println("Doc ID: " + entry.getKey() + ", Score: " + entry.getValue());
            }
        }
    }

    scanner.close();
}
```

6. Testing and Results

6.1 Test Cases

1. **Document Processing**:
    - **Input**: CSV files containing various texts.
    - **Expected Output**: Tokenized words, converted to lowercase, with punctuation removed and stopwords filtered.
2. **Boolean Queries**:
    - **Test Case 1**: Query "data AND machine"
        - **Expected Result**: Set of document IDs that contain both "data" and "machine".
    - **Test Case 2**: Query "data OR learning"
        - **Expected Result**: Set of document IDs that contain either "data" or "learning".
3. **Ranked Retrieval**:

- **Input**: Query "data structures"
- **Expected Output**: List of document IDs ranked by the sum of term frequencies.

6.2 Sample Output

Enter your search query (type 'exit' to quit):

data AND machine

Boolean Query Result: [1, 3, 5]

Ranked Documents:

Doc ID: 3, Score: 8

Doc ID: 1, Score: 5

Doc ID: 5, Score: 3

7. Performance Analysis

7.1 Time Complexity

- **Indexing**:
  - Building the index and inverted index has a linear complexity, O(n), where n is the number of words across all documents.
- **Boolean Query Processing**:
  - For list-based retrieval, the complexity is O(m + k), where m is the size of the lists being merged and k is the number of results.
  - For BST-based retrieval, lookup is optimized to O(log n) for each term.
- **Ranked Retrieval**:
  - The complexity for calculating scores is O(m * t), where m is the number of documents and t is the number of query terms.

7.2 Space Complexity

The primary space usage comes from storing the index, inverted index, and BST nodes. The space complexity is proportional to the number of unique words and document IDs.

7.3 Efficiency Observations

- **Performance with Small Datasets**: List-based and BST-based retrieval methods show minimal differences in performance.
- **Performance with Large Datasets**: BST-based search demonstrates improved performance due to reduced lookup times.

8. Conclusion and Recommendations

The Simple Search Engine effectively indexes and retrieves documents based on Boolean and ranked queries. The use of BSTs improves the search efficiency for larger datasets, although additional optimization techniques such as parallel processing or coaching could further enhance performance.