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

CSC212 : *Data Structures*

Simple Search Engine

Section #	NAME	ID
78994	*Nawaf abdullah alqahtani*	*443102239*
78994	*Abdulaziz Alamro*	*444102968*
78994	*Mohammad Alfuraiji*	*444100742*

Supervised By: *Dr. Mohammed AlDalhan*

Class Index

```
Public void addDocument(int docID, LinkedList<String> words)
```

- This method is essential for updating the basic index structure by mapping document identifiers to their respective word lists. It enables the search engine to store document data for later retrieval during queries.

Total Time Complexity: O(n)

```
public LinkedList<DocumentNode> getDocuments()
```

- This method allows external access to the indexed documents for various operations, such as search queries or performance analysis. It ensures encapsulation by providing controlled access to the underlying data structure (documents).

Total Time Complexity: O(1)

```
public LinkedList<Integer> search(String word)
```

- This method enables keyword-based searching within the indexed documents.
- It is a core feature for the search engine, allowing it to identify relevant documents for a given query word.

Total Time Complexity: **O(n * m), where:**
n is the number of documents in the documents list.
m is the average number of words in each document.

Class InvertedIndex

```
public LinkedList<WordNode> getDocuments()
```

- This method allows external access to the inverted index, which is critical for searching and retrieving documents efficiently.
- The returned invertedIndex can be used to perform various operations like searching for words, analyzing the index, or debugging.

Total Time Complexity: O(1)

```
public void addDocument(int docID, String word)
```

- **Core Functionality:** This method is essential for building the inverted index, a critical structure for efficient search operations in a search engine.
- **Prevents Duplicates:** Ensures that the same document ID is not added multiple times for a word.
- **Dynamic Growth:** Handles both updating existing words and adding new words dynamically.

Total Time Complexity:

- **Worst-Case Complexity: O(n * m).**
- **Best-Case Complexity: O(m).**

n the size of invertedIndex
m the size of docIDs list

```
public LinkedList<Integer> search(String word)
```

- **Core Search Functionality:** This method is critical for retrieving all documents that contain a specific word, which is fundamental to any search engine.
- **Efficient Retrieval:** Ensures that only relevant document IDs are returned based on the word searched.
- **Integration:** Works seamlessly with the inverted index to enable fast and efficient word-based search.

Worst-Case Complexity: O(n) (searching through all nodes in invertedIndex).

Best-Case Complexity: O(1) (finding the word at the start of the list).

Class InvertedIndexBST

```
public void addDocument(int docID, String word)
```

This method is a key part of the inverted index using BST. It allows efficient updating of the index:

- **Balanced BSTs:** Ensure search and insertion operations are fast (**O(log n)**).
- **Dynamic Growth:** Handles both updating existing words and inserting new words dynamically.
- **Efficient Document Management:** Ensures no duplicate document IDs in the list.

Total Time Complexity:

- **find:** O(log n)
- **contains:** O(m)
- **insert (into document list):** O(1)
- **update (in BST):** O(1)

  Total: O(log n + m)

```
public LinkedList<Integer> search(String word)
```

- **Efficient Search: Utilizes the BST structure to achieve fast lookups for words in the inverted index.**
- **Dynamic Handling: Returns an empty list if the word is not found, ensuring the method handles missing entries gracefully.**

Total Time Complexity:

- **Best-Case Complexity: O(log n)**
- **Worst-Case Complexity: O(n)**

Class DocumentProcessor

```
private void loadStopWords(String stopWordsFile)
```

- **This method is critical for initializing the list of stop words, which will later be used to filter irrelevant words during text processing.**
- **Efficiently handles large files by processing one line at a time using a BufferedReader.**

Total Time Complexity:

- **O(n): Where n is the total number of characters in the file.**

```
public LinkedList<String> processDocument(String document)
```

- **Text Cleaning: Ensures consistent and meaningful tokens are extracted.**
- **stop Words Removal: Enhances the quality of the tokens by eliminating irrelevant words.**
- **Efficiency: Processes large documents efficiently, especially if the stop words list is kept small.**

Total Time Complexity:

- **O(n * s), where:**

n: Number of tokens after tokenization.

s: Size of the stop words list.

Class QueryProcessor

```
public LinkedList<Integer> andQuery(String word1, String word2)
```

- **Finds documents containing both word1 and word2**

Time Complexity Analysis :

1 - Search for Each Word and its Depends on the indexing structure:

- index.search(word): O(n), where n is the total number of documents (linear search).
- invertedIndex.search(word): O(n), where n is the size of the inverted index.
- invertedIndexBST.search(word): O(log n) for a balanced BST, where n is the number of unique words.

2- Intersection (retainAll):

Compares two lists of document IDs to find the intersection:

- Time Complexity: O(n), where n is the size of the document ID list.

Total Time Complexity :

- **For index: O(n).**
- **For invertedIndex: O(n).**
- **For invertedIndexBST: O(log n).**

```
public LinkedList<Integer> orQuery(String word1, String word2)
```

- **Finds documents that contain either word1 or word2.**
- **Complexity: The complexity depends on the size of the indexing structure and the size of the result lists being merged:**

Time Complexity Analysis :

Iterating Through other:

- Iterates through the list other, which has a size of m (number of document IDs in the word2 search result) and for each document ID and Checks if it exists in result (using contains), which is O(r), where r is the size of the result list.

Total Time Complexity :

- **For index and invertedIndex and invertedIndexBST: O(m * r).**

```
public LinkedList<Integer> andQuery(LinkedList<Integer> list,
String word)
```

- **Purpose: Finds documents that are present in both list and the documents associated with word.**
- **Complexity: Dependent on the size of the input list, the size of the document list for the word, and the indexing structure used.**
- **Best Use Case: Effective for refining search results dynamically.**

Time Complexity Analysis :

1 - Search for Each Word and its Depends on the indexing structure:

- index.search(word): O(n), where n is the total number of documents .
- invertedIndex.search(word): O(n), where n is the size of the inverted index .
- invertedIndexBST.search(word): O(log n) for a balanced BST, where n is the number of unique words.

2 - Iterating Through list:

- Loops through all elements in list (size = l).
- For each document ID in list, checks if it exists in wordDocs (size m):

Total Time Complexity :

- **For index and invertedIndex and invertedIndexBST: O(m * l).**

```
public LinkedList<Integer> orQuery(LinkedList<Integer> list,
String word)
```

- **Purpose: Finds documents that are present in either the input list or the documents associated with word.**
- **Complexity: Depends on the indexing structure, the size of the input list (l), the size of the document list for the word (m), and the size of the resulting list (r).**
- **Best Use Case: Useful for expanding search results dynamically.**

Time Complexity Analysis :

1 - Search for Each Word and its Depends on the indexing structure:

- index.search(word): O(n), where n is the total number of documents (linear search).
- invertedIndex.search(word): O(n), where n is the size of the inverted index.
- invertedIndexBST.search(word): O(log n) for a balanced BST, where n is the number of unique words.

2- Iterating Through list:

- Loops through all elements in list (size = l).

3- Iterating Through wordDocs:

- **Loops through all elements in wordDocs (size m).**
- **For each document ID in wordDocs:**
  - Checks if it exists in result (using contains), which is **O(r)**, where r is the size of the result list.
- **Total Complexity for this step:** O(m * r).

Total Time Complexity :

- **For index and invertedIndex and invertedIndexBST: O(m * r).**

```
public void processBooleanQuery(String query, QueryProcessor
queryProcessor)
```

- **Purpose: Evaluates Boolean queries with AND and OR operators.**
- **Error Handling: Validates query format and operators.**
- **Complexity: Depends on the sizes of the input lists and the number of operators.**
- **Output: Prints the result to the console.**

- **Time Complexity Analysis and Total Time Complexity :**
- **andQuery: It depends rather it is andQuery(word1, word2) for index: O(n) and for invertedIndex: O(n) and for invertedIndexBST: O(log n) or andQuery(inreResult, word1) O(m * l).**
- **orQuery: It O(m * r) for all instance.**
- **printFormattedResult O(n^2) for a linked list, where n is the size of the result**

Total Time Complexity :

- **O(n^2) for a linked list, where n is the size of the result**

```
public void processRankedQuery(String query, Index index)
```

- **Purpose: Calculates and ranks documents based on their relevance to the query.**
- **Output: Displays the ranked documents in descending order of relevance.**
- **Use Case: Provides a foundational method for search engines to return results ordered by importance.**

Time Complexity Analysis :

1 - Iterating Through Documents:

- Iterates through all documents in the index: O(d), where d is the number of documents.

2- Processing Words in Each Document:

- For each document:
    - Iterates through all words in the query: O(w), where `w` is the number of words in the query.
    - Counts occurrences of each word in the document: O(l), where `l` is the average number of words in a document.
- Total for each document: O(w * l).
- Total for all documents: O(d * w * l).

3- Sorting Results:

- Sorting the rankedResults list: O(n^3)

Total Time Complexity :

O(d * w * l + n^3), where:

- d = Number of documents.
- w = Number of words in the query.
- l= Average number of words in a document.

```
public void processRankedQuery(String query, InvertedIndexBST
index)
```

- **Purpose: Efficiently ranks documents for a search query using an inverted index.**
- **Optimization: For large datasets, consider using a hash map for rankedResults to reduce the search time for existing document IDs.**

Time Complexity Analysis :

1 - Iterating Over Query Words:

- For each word in the query (W words):
    - Retrieve the document IDs using index.search(word):
        - **Time Complexity:** O(d), where d is the average number of documents for a word.
    - For each document ID:
        - Check if it exists in rankedResults (size r).
        - **Time Complexity for each word:** O(d * r).

2- Sorting Results:

- Sorting the rankedResults list: O(n^3)

Total Time Complexity :

O(w * d * r + n^3), where:

- **w: Number of words in the query.**
- **d: Average number of documents for each word.**
- **r: where is the size of rankedResults.**

```
public void processRankedQuery(String query, InvertedIndexBST
index)
```

- **Purpose: Calculates and ranks documents based on their relevance to the query using an inverted index.**
- **Output: Displays the ranked documents in descending order of relevance.**
- **Efficiency: Optimized for scenarios where the inverted index provides a small subset of documents for each word.**

Time Complexity Analysis :

1 - Iterating Over Query Words:

- For each word in the query (W words):
    - Retrieve the document IDs using index.search(word):
        - **Time Complexity:** O(d), where d is the average number of documents for a word.
    - For each document ID:
        - Check if it exists in rankedResults (size r).
        - **Time Complexity for each word:** O(d * r).

2- Sorting Results:

- Sorting the rankedResults list: O(n^3)

Total Time Complexity :

O(w * d * r + n^3), where:

- **w: Number of words in the query.**
- **d: Average number of documents for each word.**
- **r: where is the size of rankedResults.**

```
private int countOccurrences(LinkedList<String> words, String word)
```

- **Purpose: Essential for calculating word frequencies in a document, which can be used in ranking or analysis.**
- **Efficiency: Works efficiently for small to medium-sized lists. For larger datasets, consider more optimized data structures like hash tables for counting occurrences.**

Total Time Complexity :

O(n * l), where:

- **n: The number of words in the words list.**
- **l: The average length of the words.**

```
private void sortRankedResults(LinkedList<DocumentScore> list)
```

- **Purpose: Sorts search results by relevance, ensuring the most relevant documents appear first.**
- **Efficiency Concerns: The method is inefficient for large datasets due to its cubic time complexity with a linked list. Optimizations such as using an ArrayList (for O(1) get and set) or a more efficient sorting algorithm like QuickSort or MergeSort are recommended for large inputs.**

Time Complexity Analysis :

1 - Outer Loop:

- **Runs n times, where n is the size of the list.**

2 - Inner Loop:

- **For each iteration of the outer loop, the inner loop runs (n - i - 1) times on average.**

3 - Comparison and Swap:

- **Each iteration of the inner loop performs a score comparison (if statement) and possibly swaps elements:**
    - **get: O(n) for each call (as LinkedList requires traversal to access an element by index).**
    - **set: O(n) for each call.**

Total Time Complexity :

- **O(n^3) for a linked list.**

```
private void printRankedResults(String query,
LinkedList<DocumentScore> rankedResults)
```

- **Purpose: Provides a clear and readable format for displaying ranked search results**
- **Efficiency Concern: The use of LinkedList can make this method inefficient for large datasets. Consider using ArrayList for better performance (O(n)) when iterating and accessing elements by index.**

Time Complexity Analysis :

1 - Iterating Through the List:

- **The method iterates through all elements in rankedResults.**
- **Time Complexity: O(n), where n is the number of documents in the rankedResults.**

2 - Retrieving Elements (get):

- **Each call to get(i) on a linked list requires O(n) traversal in the worst case.**

- **For N elements, this results in O(n^2) for a linked list.**

Total Time Complexity :

- **O(n^2) for a linked list, where n is the size of rankedResults.**

```
private void printFormattedResult(String query, LinkedList<Integer> result)
```

- **Purpose: Provides a clear and readable format for query results, making it easier for users to interpret the output.**
- **Efficiency Concern: The use of LinkedList can make this method inefficient for large result sets. Using an ArrayList (for O(1) get operations) is recommended for better performance.**

Time Complexity Analysis :

1 - Iterating Through the List:

- **The method iterates through all elements in the result list.**
- **Time Complexity: O(n), where n is the size of the result.**

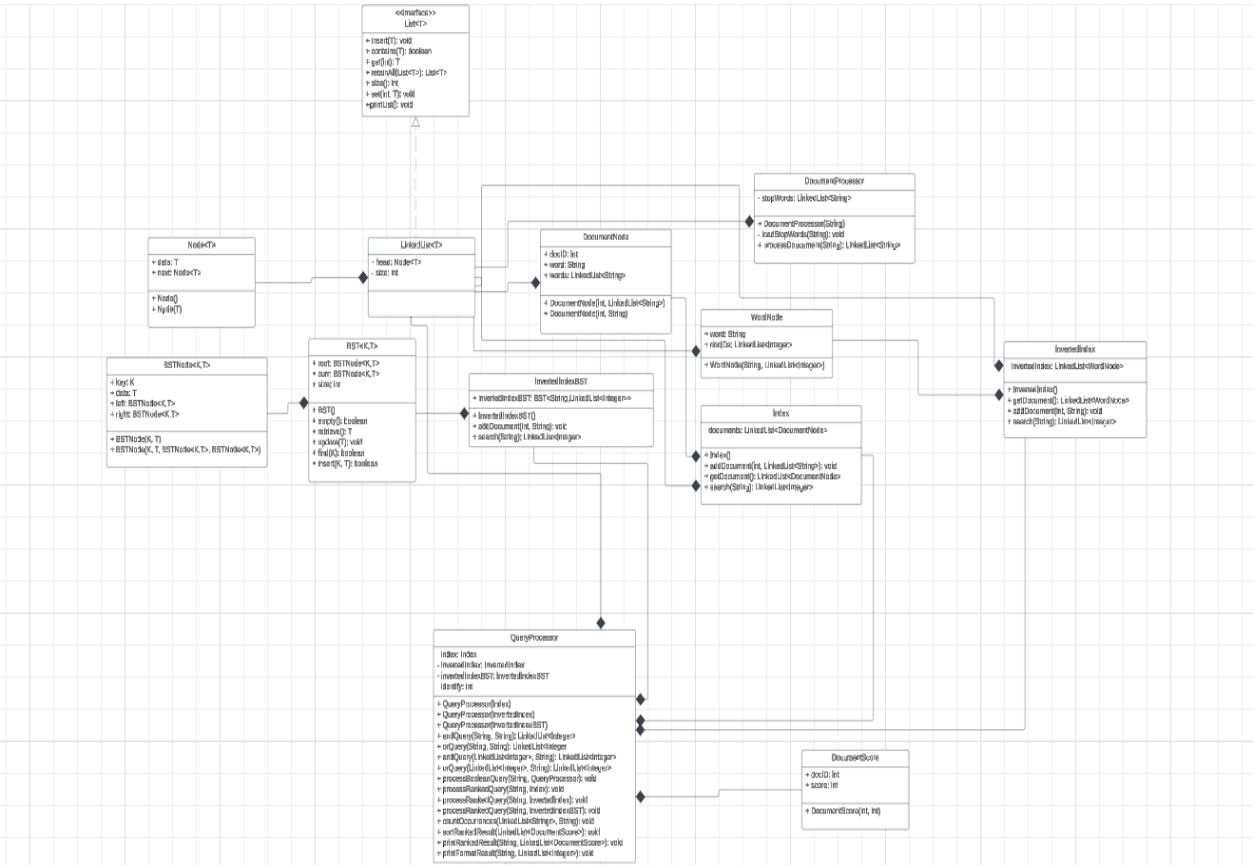
2 - Accessing Elements (get):

- **Each call to get(i) on a linked list requires O(n) traversal in the worst case.**
- **For n elements, this results in O(n^2) for a linked list.**

Total Time Complexity :

- **O(n^2) for a linked list, where n is the size of the result.**

Class Diagram



URL to view the diagram with high quality:
https://lucid.app/users/registerOrLogin/free?showLogin=false&invitationId=inv_524f18bb-78b0-44a0-9241-23e3988892f5&productOpt=chart&invitationType=documentAcceptance&returnUrlOverride=%2Flucidchart%2F056d7cd6-165a-45b8-a1ea-63644063a368%2Fedit%3Fviewport_loc%3D-4000%252C-2598%252C5120%252C2368%252C0_0%26invitationId%3Dinv_524f18bb-78b0-44a0-9241-23e3988892f5

and here is github link that we work on:
https://github.com/abdulazizabdullh/project_CSC_212.git