This document is a detailed explanation of why we used some techniques over others and also the time complexity of the main methods.

Queries

1. AND Query:

- The method uses two pointers, one for each list, and compares the docId values at each step. If the IDs match, the corresponding element is added to the result. If the IDs don't match, the pointer for the list with the smaller ID moves forward.
- **Time Complexity:** Each pointer moves through its respective list at most once, so the overall complexity is **O(n + m)**, where n is the size of the first list and m is the size of the second list.

2. OR Query:

- Similar to the AND query, the OR query uses two pointers to traverse the two lists. It adds elements to the result based on whether the docIds match or not, ensuring no duplicates. If the docIds are equal, the frequencies are combined, and both pointers advance. Otherwise, the pointer of the list with the smaller ID moves forward.
- **Time Complexity:** As with the AND query, each pointer moves through its respective list at most once. The

complexity is **O(n + m)**, where n is the size of the first list and m is the size of the second list.

3. NOT Query:

- The method iterates through the first list, adding elements to the result only if they are not present in the second list. It uses a second pointer to traverse the second list as needed.
- **Time Complexity:** For each element in the first list, the method may need to traverse some or all of the second list. Therefore, the worst-case complexity is **O(n + m)**, where n is the size of the first list and m is the size of the second list.

Overall Time Complexity for All Three Methods:

For each of the methods (AND, OR, and NOT queries), the time complexity is **O(n + m)**, where n and m are the sizes of the two input lists. Each method processes the lists using a two-pointer technique, ensuring that each element in both lists is considered once.

The reason for using the **two-pointer technique** in each of these query methods is to efficiently process and compare elements from the two input lists (list1 and list2) without unnecessary repetitions or complexity. Here's a breakdown of why this approach was chosen:

1. Efficiency in Comparing Sorted Lists:

- The two-pointer technique works particularly well when the lists are **sorted** (which is typically the case in many applications involving frequency lists or inverted indexes).
- By keeping pointers to the current nodes in both lists, we can compare the elements in constant time (just comparing their docIds).
- This allows us to process the lists in a linear manner (i.e., move the pointers forward as we process), instead of having to compare every element from one list to every element in the other list, which would be less efficient.

2. Why O(n + m) is Optimal:

- The time complexity of **O(n + m)** is optimal because:
    - Each element in both lists needs to be examined at least once. This is unavoidable because we need to ensure that we've checked all elements in both lists for the AND, OR, and NOT queries.

- If we were to use a more naive approach (like nested loops), the time complexity would be **O(n * m)**, which would be inefficient, especially if the lists are large.
- The two-pointer technique ensures that both lists are only traversed once, leading to the more efficient **O(n + m)** complexity.

3. Avoiding Redundant Operations:

- **AND Query:** We only add a docId to the result when both lists contain the same ID. The two pointers allow us to skip over elements in either list that do not match, which avoids checking unnecessary pairs.
- **OR Query:** We ensure no duplicates by advancing the pointer in the list that has the smaller docId, while adding elements from both lists as necessary. This prevents redundant checks and additions.
- **NOT Query:** We add an element from list1 to the result if it is not present in list2. The second pointer moves forward only when necessary, preventing excessive traversal of list2.

DocumentProcessing

Load

1. Reading the File:

- **Time Complexity: O(n)**, where n is the number of lines in the file.
- The file is read line by line, which takes linear time relative to the number of lines.

2. Processing Each Line:

For each line, the following steps are performed:

- **Skipping Metadata Lines:**
  - If a line contains metadata like "tokens", "vocab", or "topics", it is skipped. This check is **O(k)**, where k is the average length of the line. However, this operation is relatively fast and does not dominate the overall complexity.
- **Extracting Content After the First Comma:**
  - Finding the first comma in the line takes **O(k)**, where k is the length of the line.
  - Extracting content after the first comma and trimming takes **O(m)**, where m is the length of the content after the comma.

Since each line is processed individually, this part contributes **O(k)** for each line, which simplifies to **O(n)** when considering all lines (assuming the file has n lines).

3. Text Processing:

- **Converting Text to Lowercase:**
  - The toLowerCase() operation takes **O(m)**, where m is the length of the content after the first comma.
- **Splitting the Content into Words:**
  - Splitting the content into words using whitespace requires **O(m)** time, where m is the length of the content.
  - This operation depends on the number of words in the line, but overall it is proportional to the length of the content.

4. Word Cleaning and Filtering:

- **Cleaning Non-Alphanumeric Characters:**
  - For each word, the cleaning process (removing non-alphanumeric characters) takes **O(w)**, where w is the length of the word. Since each word is processed independently, this is **O(m)** for the entire line (where m is the total number of characters in the content).

- **Filtering Stop Words:**
    - Checking if a word is in the filter (stop words list) is an **O(1)** operation if the filter is implemented as a hash set (which is typically the case). Thus, for each word, this operation is constant time **O(1)**.

5. Building the Inverted Index:

- **Checking if a Word Exists in the Inverted Index:**
    - The method iterates through the inverted index to check if the word exists. If the index is implemented as a **binary search tree (BST)**, the search time for each word is **O(log k)**, where k is the number of unique words in the index so far.
    - If the word exists, adding the document ID takes **O(1)**.
    - If the word does not exist, inserting a new entry in the inverted index takes **O(log k)** for the search and **O(1)** for insertion.
- **Inserting a Word into the Index:**
    - Inserting a word involves searching for the word and possibly adding it to the index. If the index is implemented as a **BST**, the insertion time is **O(log k)**, where k is the number of distinct words in the index.

6. Adding Documents to the Index:

- Adding a document to the index involves checking if its list of words is non-empty and then storing it. This is a

relatively fast operation compared to the word processing and index building, with time complexity of **O(1)** per document.

Overall Time Complexity:

Let's summarize the overall time complexity considering the above steps:

- **File Reading: O(n)** where n is the number of lines.
- **Line Processing (for each line):**
  - **Text extraction and cleaning** takes **O(m)** where m is the length of the line.
  - **Splitting into words and processing each word** takes **O(m)**, with additional operations (like cleaning and filtering) contributing **O(m)** in total per line.
- **Inverted Index Operations (for each word):**
  - Searching for a word in the **BST** and inserting takes **O(log k)**, where k is the number of distinct words in the index. Since k grows as more words are added, in the worst case, this can approach **O(log n)** if every word is unique.

Assuming the document has n lines and each line has m characters, and there are p total words across all lines:

- **Processing each line**: **O(n * m)**.

- **Processing each word** (for each line and each word): **O(p * log p)** for checking and inserting into the inverted index.

Thus, the overall time complexity is:

- **O(n * m + p * log p)**.

Where:

- n is the number of lines in the file.
- m is the average length of each line.
- p is the total number of words across all documents.
- log p is the time to perform operations on the binary search tree (BST) for each word.

In the worst case, if every word is unique and there are a large number of lines, the complexity can approach **O(n * m + p * log p)**. However, this is quite efficient given the typical operations needed to build an inverted index.

Ranking

This Java code is focused on sorting a linked list of Frequency objects and ranking them based on their frequency values. Here's a detailed breakdown:

Methods:

1. **rankedRetrival(Node<Frequency> list)**:
    - This method takes a linked list list as input, which consists of Frequency objects.
    - It sorts the list using the mergeSort function and then prints the sorted nodes along with their rankings.
    - The sorted nodes are added to a new LinkedList<Frequency>, which is returned as the result.
2. **mergeSort(Node<Frequency> head)**:
    - This is a classic implementation of the **Merge Sort** algorithm for linked lists.
    - It divides the list into two halves, recursively sorts both halves, and then merges them back into a single sorted list.
    - The list is split by finding the middle node using the getMiddle() method.

3. **getMiddle(Node<Frequency> head)**:
    - This method uses the slow and fast pointer technique to find the middle node of the linked list.
    - The slow pointer moves one node at a time, while the fast pointer moves two nodes at a time. When the fast pointer reaches the end of the list, the slow pointer will be at the middle node.
4. **merge(Node<Frequency> left, Node<Frequency> right)**:
    - This function merges two sorted linked lists (left and right) into one sorted linked list.
    - It compares the Frequency values from both lists, appends the smaller (or equal) value to the resulting list, and moves to the next node in that list.
    - Once one of the lists is exhausted, the remaining nodes from the other list are appended to the result.

Time Complexity:

1. **mergeSort**:
    - **Dividing the list**: The list is repeatedly split in half, so this part of the algorithm takes **O(log n)** time, where n is the number of nodes in the list.
    - **Merging two lists**: For each pair of lists, the merge operation involves traversing all nodes, which takes **O(n)** time.

- Since merge sort recursively divides the list into halves and merges them, the total time complexity of the merge sort is **O(n log n)**.

2. **Overall Time Complexity**:
   - The time complexity of the rankedRetrival function, which involves sorting the list using mergeSort and then iterating through the sorted list to create the ranked retrieval list, is **O(n log n)**, where n is the number of nodes in the linked list.

Space Complexity:

- **Auxiliary Space**: Merge sort uses **O(n)** space due to the recursive calls and the temporary nodes created during the merge operation.
- The overall space complexity is **O(n)**, as new nodes are created in the merge method.

Key Reasons for Using Merge Sort:

1. Stable Sort:

- **Stability** in sorting means that when two elements have equal keys (or frequencies in this case), their relative order remains the same in the sorted output. Merge Sort is a stable sorting algorithm, which is crucial when sorting based on frequencies but needing to preserve the order of elements with the same frequency.

- In this code, the mergeSort ensures that, if two nodes have the same frequency, they will maintain their relative order in the final sorted list, which is important for ranking or tie-breaking logic.

2. Efficient for Linked Lists:

- **Merge Sort is particularly suited for linked lists** because it does not require random access to elements. Linked lists are inherently sequential, however accessing elements in the middle of the list requires traversing from the head node, which is slower than accessing by index (like in an array).
- Merge Sort works by recursively splitting the list in half, which can be done efficiently by adjusting the pointers in the list. No additional space for direct indexing or random access is needed, unlike other sorting algorithms like QuickSort or HeapSort, which are more suited to array-based data structures.

3. O(n log n) Time Complexity:

- **Merge Sort guarantees O(n log n) time complexity** even in the worst case. Other common sorting algorithms like QuickSort can degrade to **O(n^2)** in the worst case if the pivot selection is poor (e.g., in case of already sorted or nearly sorted data). Merge Sort does not have this issue, making it a more predictable and reliable choice for large datasets.

Index

Data Structures:

1. **whole_Docs**:
    - A **linked list** of Document objects representing all the documents in the system.
    - Allows for sequential traversal and insertion at the end.
2. **invertedIndex**:
    - A **linked list** of Word objects representing the inverted index.
    - Each Word maps to the documents in which it appears.
3. **InvertedIndexBST**:
    - A **Binary Search Tree (BST)** used to maintain a secondary inverted index, offering logarithmic search and insertion time for words.

Main methods:

1. Adding Documents:

The add_Document(Document d) method enables adding new documents to the system. The process involves traversing the linked list to the last node and inserting the new document. This ensures that documents are maintained in the order they were added.

- **Time Complexity**:
  O(n), where n is the current number of documents in the list.

2. Counting Tokens Across All Documents:

The getTotalTokens() method calculates the total number of words (tokens) across all documents by iterating through the whole_Docs list and summing the size of the word lists within each document.

- **Time Complexity**:
  O(n * m), where n is the number of documents and m is the average number of words per document.

3. Counting Unique Words:

The getUniqueWords() method counts the total number of unique words in the invertedIndex. Each entry in the inverted index corresponds to a unique word.

- **Time Complexity**:
  O(u), where u is the number of unique words in the inverted index.

4. Retrieving Documents:

The getDocumentByIndex(int id) method retrieves a document by its ID. This is achieved by sequentially traversing the whole_Docs list and comparing each document's ID with the provided value.

- **Time Complexity**:
  O(n), where n is the number of documents.

5. Displaying Documents:

The display_Docs() method iterates through all the documents and prints their details. It skips documents with invalid IDs (-1) or placeholder content ("content").

- **Time Complexity**:
  O(n), where n is the number of documents.

6. Total Document Count:

The getTotalDocuments() method calculates the total number of documents by traversing the whole_Docs list and incrementing a counter.

- **Time Complexity**:
  O(n), where n is the number of documents.